

Table of Contents

TABLE OF CONTENTS	1
OBJECT ORIENTED PROGRAMMING.....	4
OBJECTS	4
CLASSES	4
UML – UNIFIED MODELING LANGUAGE	5
SOFTWARE DEVELOPMENT PROCESS	6
USE CASE MODEL	6
CLASS MODEL	11
EXERCISE – SKI CLUB.....	12
INTRODUCTION TO PROGRAMMING IN JAVA	13
JAVA BASICS	13
SIMPLE (PRIMITIVE) DATA TYPES	13
JAVA OPERATORS	14
MAKING DECISIONS	16
REPETITION	17
ARRAYS.....	17
DISPLAYING DATA	18
USER INPUT	18
YOUR FIRST JAVA APPLICATION	19
EXERCISE – A SIMPLE JAVA APPLICATION	21
CLASSES IN JAVA	22
DEFINING CLASSES IN JAVA	22
ADDING ATTRIBUTES	23
ADDING METHODS	25
CLASS CONSTRUCTORS	28
ACCESSING USER-DEFINED CLASSES	29
EXERCISE –JAVA CLASSES	32
OBJECT ORIENTED PRINCIPLES.....	33
ABSTRACTION	33
ENCAPSULATION	34
INHERITANCE.....	34
POLYMORPHISM.....	35
ADVANTAGES OF OOP	35
APPLYING THE FOUR OBJECT ORIENTED PRINCIPLES	36
ALTERNATE CONSTRUCTORS.....	40
POLYMORPHISM AND OVERRIDING METHODS	41
INHERITANCE MODIFIERS	42
MULTIPLE INHERITANCE	47
EXERCISE – LIBRARY SYSTEM.....	54
JAVA PACKAGES	55
CLASS DOCUMENTATION.....	56
IMPORTING A PACKAGE	60
EXERCISE – NUMBER FORMATS.....	62
JAVA APPLETS.....	63
WRITING A JAVA APPLET	63
RUNNING JAVA APPLETS	67
GRAPHICS CLASS.....	67

EXERCISE – CENTERED IMAGE	78
JAVA COMPONENTS	79
JAVA APPLICATIONS IN A WINDOW	79
LABELS	83
COMMON CODE FOR JAVA APPLETS AND APPLICATIONS	85
BUTTONS	89
TEXTFIELDS	93
EXERCISE – JAVA COMPONENTS	96
TEXTAREAS	97
ABSOLUTE LAYOUT	101
BACKGROUND COLOUR	103
CHECKBOXES	104
RADIO BUTTONS	107
HANDLING EVENTS WITH INNER CLASSES	110
LIST BOXES	113
CHOICE BOXES	116
EXERCISE – COMPONENTS	119
SUMMARY SHEETS	119
JAVA COMPONENTS, LISTENER CLASSES AND LISTENER METHODS SUMMARY	120
EVENT CLASS METHODS	121
GENERAL COMPONENT METHODS	122
SPECIFIC COMPONENT METHODS	123
MOUSE, KEYBOARD AND MENU EVENTS	124
MOUSE EVENTS	124
MOUSE MOTION	125
OTHER MOUSE EVENTS	128
DOUBLE BUFFERING	129
KEYBOARD EVENTS	138
MENUS	142
BUILDING A MENU	143
MENU EVENTS	145
ADAPTERS	145
EXERCISE – MODIFIED MOVING LETTER	155
CONTAINERS AND LAYOUTS	156
CONTAINERS	156
FLOW LAYOUT	157
BORDER LAYOUT	158
GRID LAYOUT	160
CARD LAYOUT	162
GRIDBAG LAYOUT	166
PANELS	170
CLASS EXERCISE – CONTAINERS, PANELS AND LAYOUT MANAGERS	173
HANDLING DATA	174
ARRAYS	174
STRINGS	177
VECTORS	183
CLASS EXERCISE – STRINGS, ARRAYS AND VECTORS	186
EXCEPTIONS	187
EXCEPTION CLASS	187
TRAPPING ERRORS	188
THROWING EXCEPTIONS	189
USING EXCEPTIONS	189

CLASS EXERCISE – ARRAY BOUNDARY EXCEPTIONS	189
INPUT AND OUTPUT	190
FILE HANDLING	190
READING FROM FILES	193
WRITING TO FILES	196
READING AND WRITING DATA FILES.....	199
CLASS EXERCISE – TEXT AND DATA FILES.....	201
THREADS.....	202
CREATING THREADS.....	202
MANAGING THREADS	202
TERMINATING THREADS.....	202
USING THREADS IN JAVA.....	203
CLASS EXERCISE – MULTI-THREADING.....	217
CLASSES IN C++.....	218
DEFINING CLASSES	218
DEFINING CLASS AND INSTANCE FUNCTIONS AND INITIALIZING CONSTANTS	224
USING C++ CLASSES	226
*THIS POINTER.....	231
FRIENDS OF CLASSES	232
MEMBERWISE ASSIGNMENT	237
COPY CONSTRUCTORS	237
THE OUTPUT OF THIS APPLICATION IS	239
OPERATOR OVERLOADING	244
OBJECT ORIENTED C++.....	248
THIS POINTER	248
BASIC INHERITANCE.....	250
STANDARD TEMPLATE LIBRARY (STL)	257
CONTAINERS.....	257
ITERATORS	258
ALGORITHMS.....	259
OPERATIONS FOR SORTED CONTAINERS	260
VECTOR CLASS TEMPLATE	260
DEQUE CLASS TEMPLATE.....	266
LIST CLASS TEMPLATE	270
SET CLASS TEMPLATE	272

Object Oriented Programming

Each application you develop has a purpose in the real world. That purpose may be as complicated as monitoring planes in the air or as simple as controlling a coffee maker. The larger and more complex the system, the more appropriate it is for object oriented programming.

Traditional programming looks at a system in terms of the 'order that things happen'. When you write a program in a sequential programming language such as ANSI C or Quick Basic you have to organize everything so that the instructions are executed in a particular order.

Object Oriented Programming (OOP) takes a different slant. It looks at systems in terms of its components and how they interact with each other. These components, called **objects**, represent actual *people* (customers, employees, managers, patients, etc.) or *things* (on-screen buttons, apples, chairs, etc.).

Objects

Objects form the basis of OOP. Each object has a set of **attributes** that define characteristics like *name*, *age*, and *hire date* for people and *size*, *colour*, *material* and *cost* for things. Each object also has a set of **behaviours** or **operations** associated with it. These are the activities that are associated with the object.

Right now you're familiar with on-screen elements such as labels, buttons, and text boxes. You have defined attributes such as name, caption, visibility, enabled and index for many of these objects in Windows and Visual Basic applications you've written in the past. These attributes define what the object is. You also remember that each of these on-screen elements has a set of methods associated with it such as `mouseover()`, `mousedown()`, and `click()`. These methods define the types of activities and behaviours associated with the object.

Similarly, a sample system might be a grocery store that has an apple as one of its objects. The attributes of the apple might include:

Type:	Granny Smith
Grade:	A
SKU:	4032
Price per kg:	\$3.69

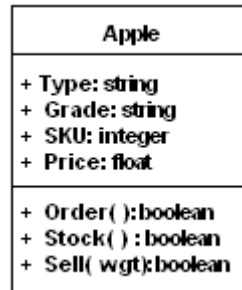
The activities associated with an apple might include ordering, stocking, and selling.

Classes

A **class** is a template that defines what attributes and behaviours an object has. Once a class has been defined and included in an application you can create or **instantiate** as many objects of that class as you need.

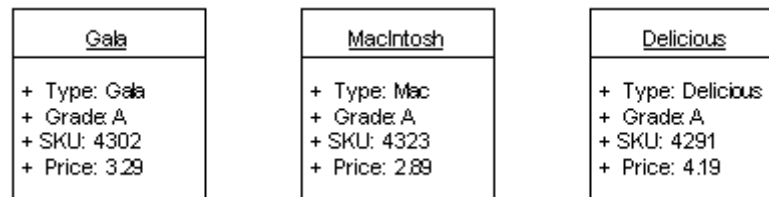
Going back to the grocery store and the apples again, a class called *Apple* would be defined that specifies every apple will be defined according to its *Type*, *Grade*, *SKU* and *Price* (its attributes), and that it will be *ordered*, *stocked* and *sold* (its methods).

Representing this graphically, your *Apple* class would look something like this...



The class is shown in a rectangular box with its name (*Apple*) in a separate section at the top. In the middle section you can see the four attributes (*Type*, *Grade*, *SKU* and *Price*) assigned to the class. The bottom section contains the methods (*order*, *stock* and *sell*) associated with the class.

Once the *Apple* class has been defined and associated with your application you can instantiate as many ‘apples’ as you need. You may have specific objects such as ‘*Gala*’, ‘*MacIntosh*’ and ‘*Delicious*’ each with their own unique *Type*, *Grade*, *SKU* and *Price* values, and each with the ability to be *ordered*, *stocked* and *sold*. Represented graphically, these objects would look like this ...



The object name (*Gala*, *MacIntosh*, or *Delicious*) appears in the top section with the attributes (*Type*, *Grade*, *SKU* and *Price*) and their assigned values in the bottom section. It is assumed that each of these objects has the same methods that are defined by the class.

UML – Unified Modeling Language

The Unified Modeling Language (UML) is the current standard for designing Objected Oriented software. Organization Management Group (OMG), the organization that oversees UML, specifies that ...

“The Unified Modeling Language (UML) is a graphical language for visualizing, specifying, constructing and documenting the artifacts of a software-intensive system. The UML offers a standard way to write a system’s blueprints, including conceptual things such as business processes and system functions as well as concrete things such as programming language statements, database schemas, and reusable software components.”

It is not a process ... simply a way to define various aspects of a given system. There are a number of domains or notations within the UML. These include...

- **User Interaction or Use Case Model** – defines the interactions between the system and its users
- **Interaction or Collaboration Model** – defines how the objects in the system interact with each other to achieve the desired tasks
- **Dynamic Model** – defines the states that classes assume over the life of the system
- **Logical or Class Model** – defines the classes and objects that make up the system
- **Physical Component Model** – defines the software and hardware components that make up the system
- **Physical Deployment Model** – defines the physical architecture and use of components on the hardware

Learning to use UML completely would be a course in itself so for now you'll just use a subset to help analyze and design your object oriented applications.

Software Development Process

Your software development process should start with an analysis of exactly *what* events occur in the specified system. This involves *who* and *what* are involved in all the different scenarios that may occur. This is done using the Use Case Model.

Use Case Model

The Use Case Model describes how the system works. It defines a series of **Use Cases**, each of which describes a unique situation involving a user (the **actor**) and the system. A customer ordering a meal, a library patron signing out a book, a skier signing up for a lesson, and a robot spot welding a joint would all be considered Use Cases ... events that occur in a given system.

There are three basic items included in a Use Case ...

1. Requirements
2. Constraints
3. Scenarios

The **requirements** specify exactly what the Use Case will do. The completed system must provide each of these functional requirements to the end user. This specification includes all the actions that must be performed.

The **constraints** define the rules and limitations for the Use Case. This includes any *pre-conditions* ... things that must have already happened or be in place before this situation can start. It also includes *post-conditions* that state what will be true once the Use Case is complete and *invariant conditions* that define what will be true throughout the time the Use Case is happening

Scenarios define the flow of events that occurs during an instance of the Use Case. Things generally happen in a particular order and the scenario defines this sequence.

The thinking process for developing a Use Case is quite simple ...

- | | |
|--|--------------|
| 1. identify the system goal | REQUIREMENTS |
| 2. identify the key participant(s) or actor(s) | |
| 3. identify the entry conditions | CONSTRAINTS |
| 4. identify any ongoing conditions | |
| 5. identify the exit condition | |
| 6. define the normal flow of events | SCENARIOS |
| 7. define any exceptional cases | |

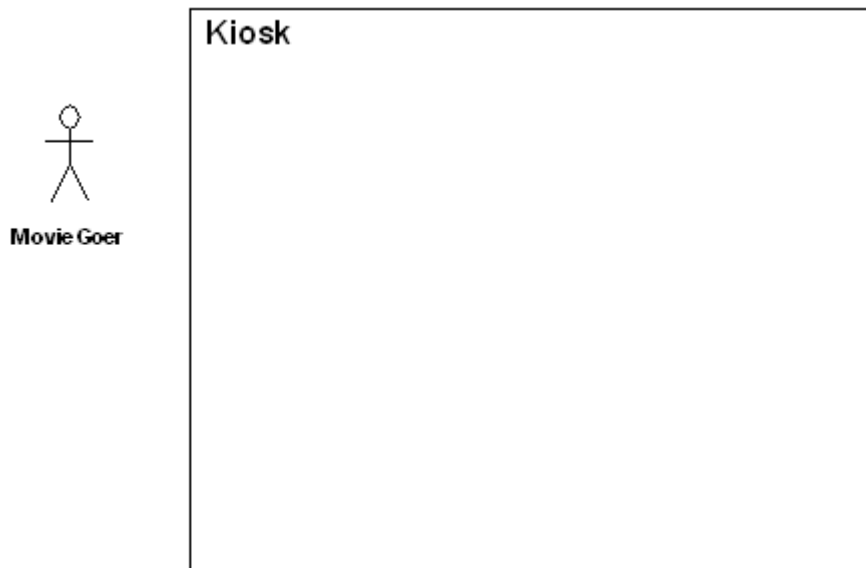
Take, for example, an automated theatre ticket kiosk. First and foremost, its purpose is to sell and dispense theatre tickets. The main participant is a movie goer.

In order to purchase a ticket, the movie goer must be standing at the ticket kiosk and must have enough money to pay for the ticket. There must also be an empty seat in the theatre. These are our entry conditions. There are no particular ongoing conditions that must be met, and the exit condition is that the movie goer has a valid ticket.

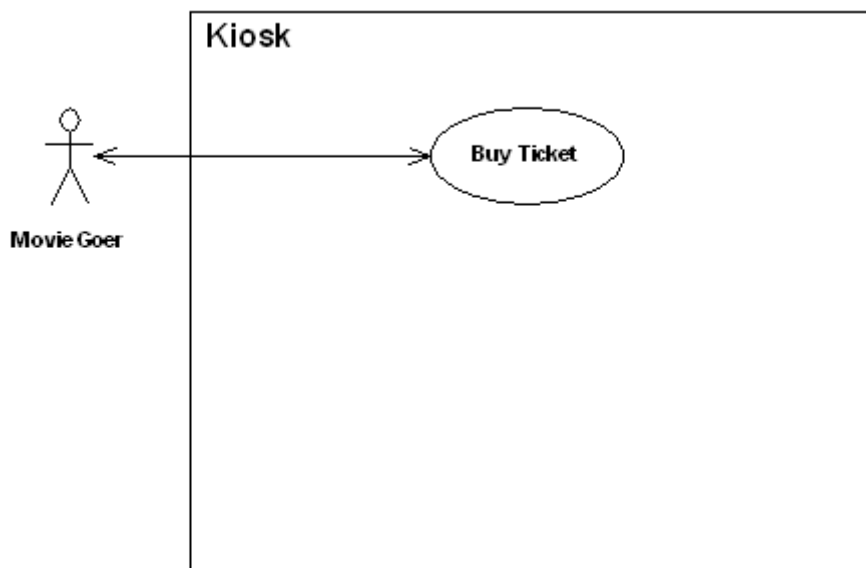
The normal flow of events may go something like this...

Movie Goer:	presses Start button
Kiosk:	displays available movies
Movie Goer:	selects a movie
Kiosk:	displays available dates and times
Movie Goer:	selects a date and time
Kiosk:	display age categories
Movie Goer:	selects senior/adult/student/child ticket type
Kiosk:	displays ticket price
Movie Goer:	enters payment card
Kiosk:	charges ticket price to payment card
Kiosk:	issues receipt
Kiosk:	issues ticket

Now it's time to create an actual **Use Case Model**. Using a graphics package with a UML template (Visio is one that's available to you in the labs) First you start by defining your system and your actors. Actors are shown as *stick people* and the system is shown as a *box*. In the kiosk example, the system is the kiosk and the actor is the movie goer.

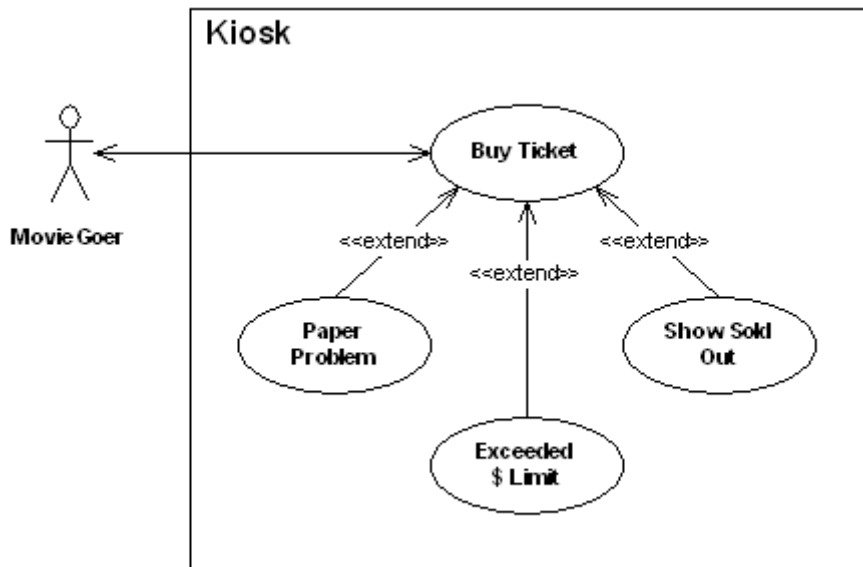


Next you add your normal scenario(s). Scenarios are represented by *ovals*. Each oval contains text that describe the scenario. In the kiosk example there is only one normal scenario ... the purchase of a ticket. The scenario is joined to the actor(s) associated with it by an arrow. Because there is activity from the movie goer to the kiosk and from the kiosk to the movie goer, the arrow has a head at both ends.



So far all that is included in the Use Case Model is the normal scenario. You must also include any exceptions from the norm. Each exception shows up in its own oval joined to the normal scenario with an arrow pointing from the exception to the normal. The arrow is identified with the keyword **<<extend>>** that indicates the scenario might but is not likely to occur.

Now it's time to add the exceptions to the ticket kiosk example.



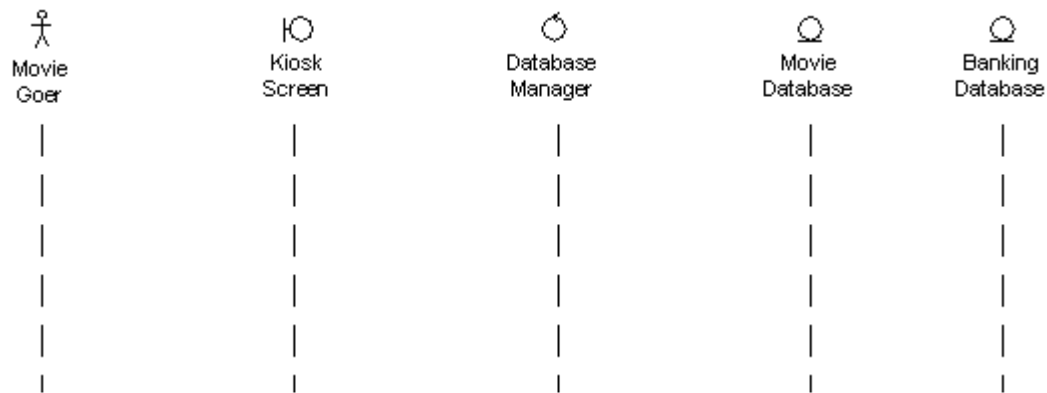
The main activities of the ticket kiosk, both normal and exceptions, are now shown in the Use Case Model. It's now time to formally define the flow of events using a **Sequence Diagram**. This sequence diagram shows the order of events as well as the instigator, the action and the responder for each event.

The first task in setting up a sequence diagram is to identify all the objects involved in the system. So far a movie goer and a kiosk have been identified as elements of the system, but are there more? Certainly if you think inside the kiosk box, you'd realize that as well as the screen used to communicate with the user, there must be some type of data manager, some link to a movie database and some link to a financial database. That gives you five objects to consider when defining your sequence diagram.

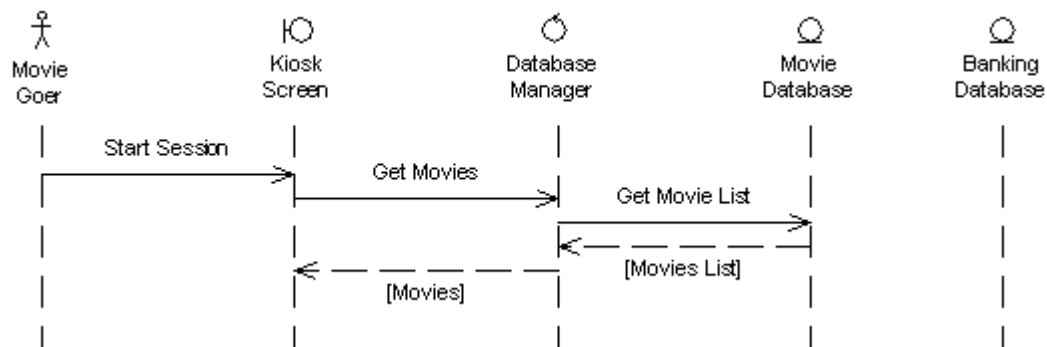
Each type of object has a particular symbol ...

Actor	Stick Person	⋈
Interface Device	Boundary Object	⬢
System Management	Control Object	⦿
Databases	Entity Object	⬢

These objects in a sequence diagram are lined up across the top with the user on the left, the interface next, followed by the system management and any databases. Each object in the system is labeled and has an ‘object line’ that drops down below it. For the ticket kiosk the top of the sequence diagram might look like ...

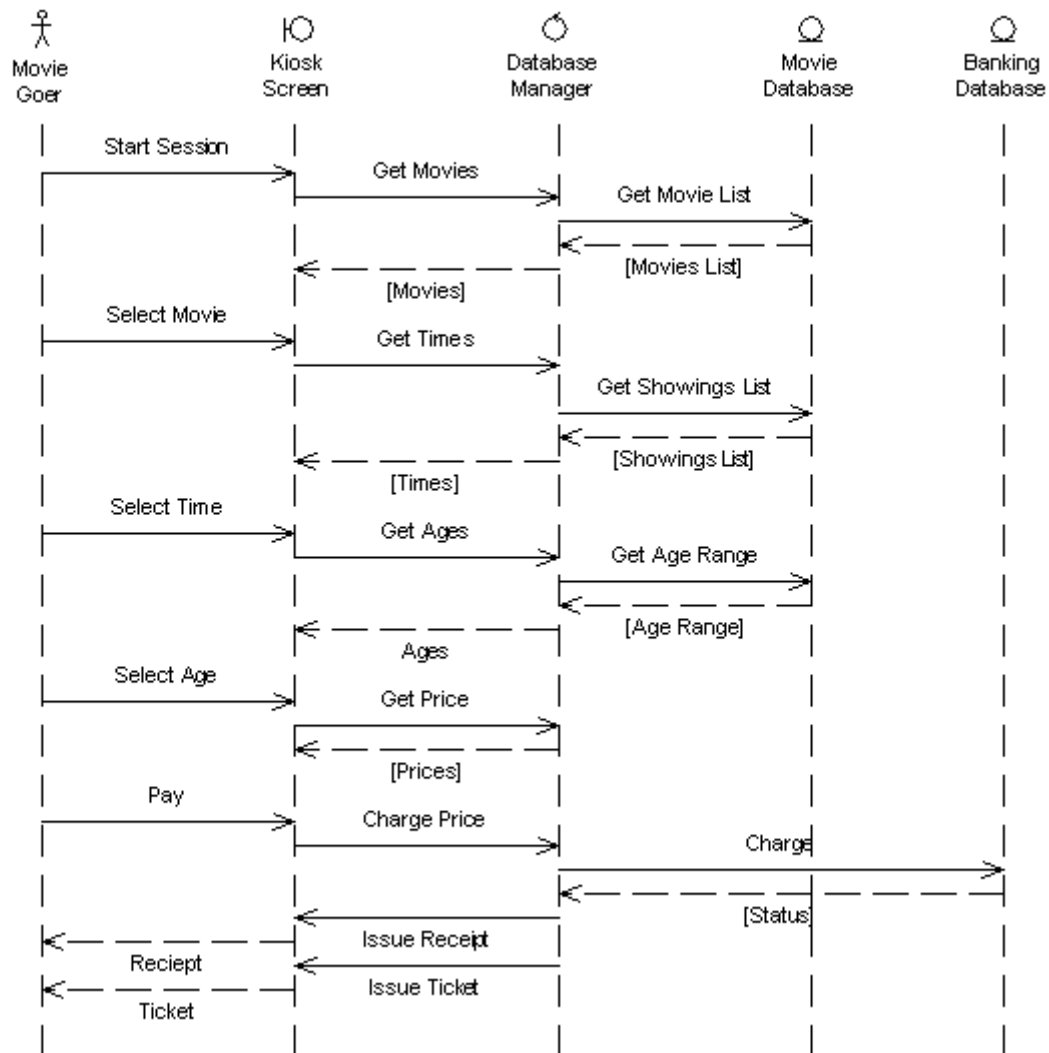


Referring to the flow of events defined in your planning stage, you now fill in the interactions between each of the objects in the order that they will occur. Normally the user/actor triggers the system. In the case of the ticket kiosk, the movie goer presses the Start button on the kiosk...



The interaction is between the *movie goer* and the *kiosk screen* with the user initiating it. Think of it as the movie goer sending a **message** to the kiosk to start. When the *kiosk* receives this **message** it in turn **notifies** the *Database Manager* that it needs to display a list of the movies. The *Database Manager* sends a **message** requesting this list from the *Movie Database*. The requested list of movies is **returned** to the *Database Manager* which **passes** it along to the *kiosk screen*. Notice that messages/actions are solid arrows while returned values are dashed arrows.

This same process is continued until the exit condition you defined in your planning is achieved. In the case of our ticket kiosk, the sequence diagram is complete once the movie goer has a movie ticket and a receipt. The complete sequence diagram might look something like this...

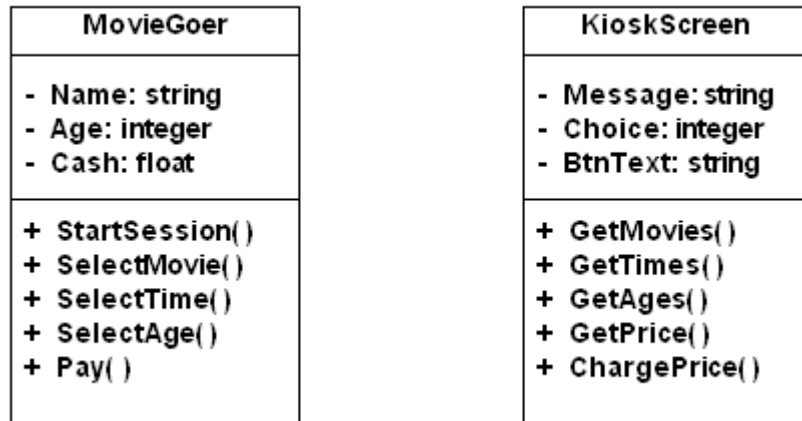


Putting the final Use Case diagram together with the complete Sequence Diagram forms the Use Case Model you'll use to program your system.

Class Model

Once you have defined your Use Case Model it is time to define the details of the classes and objects in the system. You've already had a preview of how to represent classes and objects ... now it's time to take a closer look.

As previously discussed, classes identify both the attributes of the class, as well as its behaviours. For the ticket kiosk example the movie goer and the kiosk screen class definitions might look like ...



Notice that the attributes, defined in the top portion of the rectangle, are all prefixed by minus signs. This identifies them as ‘*private*’ attributes meaning that they can only be accessed from within the class or by using one of the ‘*public*’ methods. In object oriented programming the goal is to keep as many attributes private as possible. This prevents unauthorized and undesired changes from occurring.

The methods, defined in the bottom portion of the rectangle, are all prefixed by plus signs. This identifies them as ‘*public*’ methods meaning that they are accessible to all classes in the system. Any public method can be called by any object.

Once your **class diagram** includes definitions for all your classes you are finished your planning and are ready to start coding!

Exercise – Ski Club

The system you are to analyze is a ski club. The club has members who pay a yearly membership fee as well as single-day skiers who buy individual lift tickets. The club runs a ski school that runs five different classes. Each class is limited to 6 skiers and runs at a different time of the week. Both members and non-members can enrol in ski classes but non-members are charged more. Skiers can drop out of ski classes for a full refund (less an administration charge of course!) as long as the withdrawal is before the first class.

As administrator of the system, you must be able to produce reports on the current membership, class lists for each ski class, the number of day skiers, and cash on hand. Produce a complete Use Case Model (including a Sequence Diagram) and a Class Diagram for this system.

Introduction to Programming in Java

Java Basics

Java is quite similar to C in its structure and its basic instruction set. Some of the more common elements are:

- both Java and C are case sensitive ... '*value*' is not the same as '*Value*' which is not the same as '*VALue*'
- every statement is terminated with a semi-colon (;)
- curly braces ({ }) are used to mark the beginning and end of functional blocks
- comments can be indicated with a double slash (//) or with starting (/*) and ending (*/) comment indicators

As you work through the following pages, you'll realize there are many more similarities between the two languages. Whatever language you're using, make sure you always use spaces and indentation to make your code readable.

Simple (Primitive) Data Types

A number of the basic data types available in Java are the same as those available in C. The following table summarizes the primitive data types. Note the addition of the byte and boolean data types.

Type	Size	Samples/Range
byte	8 bits	
char	16 bits unsigned	'a' '2' '\n' '\x1A'
short	16 bits	-32,768 → 32,767 0x2D
int	32 bits	$-2^{31} \rightarrow 2^{31}-1$ 12456
long	64 bits	$-2^{63} \rightarrow 2^{63}-1$ 12456L
float	32 bits	3.14152 1.60e23 3.42e-12F
double	64 bits	3.14152D
boolean	1 bit	true false

In C the number of bytes reserved for each of these data types is system dependent. That problem is bypassed in Java ... each data type is a fixed size regardless of the system.

Java Operators

The set of operators available in Java is very close to those available in C. As you can see in the following table, the assignment, arithmetic, comparison and basic logical operators are all ones that you have used before.

Category	Operator	Meaning	Example
Assignment	=	assign value	x = 5; ok = true;
Arithmetic	-	Negation	x = -y;
	+	Addition	x = 2 + 3;
	-	Subtraction	x = 12 - y;
	*	Multiplication	x = 5 * y;
	/	Division	F = C / 23;
	%	Modulus	n = x % 5;
	++	Increment	x++; ++y;
Comparison	--	Decrement	x--;
	==	Equality	
	!=	Non-equality	
	<	Less than	
	<=	Less than or equal	
	>	Greater than	
	>=	Greater than or equal	
Logical	!	Not	x = ! ok;
	&	And	x < 3 & y == 0 (both evaluated)
		Or	x < 3 y == 0 (both evaluated)
	^	Xor	x > 3 ^ y == 0 (both evaluated)

There are a few extra logical operators in the Java programming language. These are shortcut operators where the expression on the right side of the operator is evaluated if and only if the expression on the left side of the operator is true.

Category	Operator	Meaning	Example
Logical	&&	Short circuit and	x < 3 && y == 0 (2 nd evaluated iff 1 st true)
		Short circuit or	x < 3 y == 0 (2 nd evaluated iff 1 st true)

Similarly, the Java bitwise operators are almost the same as those in C. There is one new one ... the shift right with a zero fill.

Category	Operator	Meaning	Example
Bitwise	~	Bitwise complement	<code>x = ~x;</code>
	&	Bitwise and	<code>x & 2</code>
		Bitwise or	<code>x = y 5;</code>
	^	Bitwise xor	<code>x = y ^ 7;</code>
	<<	Shift left	<code>x = 1 << 2; (x = 4)</code>
	>>	Shift right (signed)	<code>x = -8 >> 2; (x = -2)</code>
	>>>	Shift right (zero fill)	<code>x = -8 >>> 2; (x = 1073741822)</code>

Look at the examples for the normal shift right and the shift right with a zero fill. If you write the binary equivalent of the integer value -8 you get

```
1111 1111 1111 1111 1111 1111 1111 1000
```

Shifting this twice to the right and filling with ones you get ...

```
1111 1111 1111 1111 1111 1111 1111 1110
```

which is equivalent to -2. Shifting the same binary representation of -8 twice to the right and filling with zeros gives you ...

```
0011 1111 1111 1111 1111 1111 1111 1100
```

which is equivalent to 1,073,741,822 ...a really big difference in the two results. Be careful that you choose the right shift right operator for your application!

The final set of operators includes a few that you've seen before. The assignment shortcuts are commonly used in C so you should already be comfortable using these.

Category	Operator	Meaning	Example
Other	<code>x=</code>	Assignment shortcuts	<code>+=, -=, *=, &=, <<=, etc.</code>
	<code>?</code>	Conditional	<code>(condition) ? (true_value):(false_value)</code>
		Exponent	<code>f = Math.pow(nbr, power);</code>
	<code>()</code>	Casting	<code>n = (int) Math.pow(6.0, 3.0);</code>

The conditional operator is also available in C although some of you may never have used it before. It is very useful in doing the equivalent of one line if/else statements.

There is no specific operator for calculating an exponent or power. You must use one of the methods found in the Math class library. For now think of it as a function that you pass both the base number and the desired exponent.

Typecasting in Java is done the same way as it is in ANSI C. The new data type is enclosed in parentheses (round brackets) followed by the variable to be typecast.

Making Decisions

The decision making structures in Java are pretty much the same as in ANSI C. The keywords if, else, switch, case, break and default are all used the same way. Conditions are defined within parentheses and any blocks of code to be executed given a condition are enclosed in curly braces. The basic structure of each possible type of decision making block is shown here ...

Simple	<pre>if (<i>condition</i>) statement;</pre>
Blocked	<pre>if (<i>condition</i>) { statements; etc. }</pre>
True/False	<pre>if (<i>condition</i>) true statement; else false statement;</pre>
Multiple Tests	<pre>if (<i>condition 1</i>) true statement 1; else if (<i>condition 2</i>) true statement 2; else false statement;</pre>

Multiple values of same expression

```
switch (expression)
{
    case value1:
        statements;
        break;
    case value2:
    case value3:
        statements;
        break;
    ...
    default:
        statements;
}
```


Recall that the `break` statement forces you to break out of the switch block. If you forget to include the `break`, execution will continue on through the rest of the code in the switch block. Notice in the second case (`'value2'` and `'value3'`) the same block of code can be executed for different values of the tested expression. This is similar to C. Finally remember that if the expression is not equal to any of the cases the default block of code is executed. Like in C this default block is optional.

Repetition

The repetition or looping structures in Java are the same as they are in ANSI C. These include the **for**, **while** and **do/while** structures. To loop a specific number of times you would normally choose a **for** loop ...

```
for (initialize; test; modify)
{
    statements;
}
```

where the *looping variable* is assigned a starting value in the **initialize** section, the *condition* is **tested** as a while loop, and the *looping variable* is **modified** in the final section. Use curly braces to begin and end the block of instructions to be executed.

If you want to test your condition before you enter the loop you would choose a **while** loop ...

```
while (condition)
{
    statements;
}
```

This means that if the condition tests true the first time, the loop statements will **NOT** be executed. If you want to guarantee that the code in the loop is executed at least once before you test, you would choose a **do/while** loop ...

```
do
{
    statements;
} while (condition);
```

The loop will be repeated until the condition tests true.

Arrays

Java arrays look and act much the same as arrays in C. Arrays can be one or multi-dimensional and elements are numbered from zero (0). One of the nice things about working with arrays in Java is its boundary checking. No longer can you unknowingly access memory beyond your array ... now you'll get a runtime error!

Java arrays are actually a built in class. Classes will be discussed later, but for now think of them as structures that contain both data and methods. To use an array you have to declare it first. This is done by specifying its data type and name ...

```
int values[];
```

In this case you've defined an integer array called *values*. Next you have to create an instance of the array using the **new** operator. It is here that you define how many elements you want in your array...

```
values = new int[size];
```

Notice that the name of the array 'values' is on the left side of an assignment operator. Like in C, the **new** operator creates a new object, in this case an array of dimension *size*. Unlike C, in Java *size* can be either a fixed number or it can be a variable. As the array is created each element is initialized to zero or null.

Displaying Data

Java applications run like C console applications ... in a DOS screen. This makes your output very simple. A set of **System** methods (functions by any other name) are provided to output Java data types to the screen. The two most commonly used methods are **println()** and **print()** the only difference between the two being the addition of a line feed at the end of the data displayed using **println()**. The format of the two instructions is...

```
System.out.println(argument list)
```

```
System.out.print(argument list)
```

The *argument list* can contain specific text you want displayed as well as the variable names of the values you want printed. Specific text must be enclosed by double quotes (ex. "*this is the text*") and a plus sign (+) is used to join text to data values. For example, assuming the variable *x* has been assigned the value 23 somewhere else in your application, the following instruction will print 'The answer is 23' on the screen.

```
System.out.println("The answer is " + x);
```

After execution of this statement the cursor is left at the beginning of the next line.

User Input

The early Java applications you'll write will get all their user input from the command line. In Java you'll receive two pieces of information ... the number of arguments entered by the user, and an array containing each of the command line arguments.

The array that contains the command line arguments is called **args[]**. It has a length property associated with it that is **args.length**. The individual elements of the array (**args[0]**, **args[1]**, etc.)

contain the actual user input as text. Inputted values can be converted to integers using the following instruction...


```
value = Integer.parseInt (args[x]);
```

Your First Java Application


When you run a Java application you are actually creating an **instance** of a **class**. Recall that a class is nothing more than a collection of information or data grouped together with the methods or functions that allow you to do something to or with that data. Your source code is therefore the definition of a class. It must start with the keyword **class** followed by the name of the class. This **must** be the same as the name of your application.. All the required data and methods are defined inside a set of curly braces ...

```
class className
{
    //data is declared here

    //methods are defined here
}
```

 Create a new file in any text editor. Type in the following...

```
class adding
{
}
```

 Save your file as **adding.java**. Remember, your class name has to be the same as your file name.


In this simple application all you need is a main method. It is going to accept two integer values from the user, add them together and display the result. The method declaration looks like ...

```
public static void main (String arg[])
```

The keyword **public** means that this method is available to external users. The keyword **static** indicates that this method can be called without creating an instance of this method. The main method will always be declared as both public and static.

The rest of the method declaration is the same as it would appear in C. ‘*void*’ represents the return type, ‘*main*’ is the name of the method and the parentheses enclose the argument list. In this case, as it will be anytime you’re using command line input, the argument list contains a **String class** to hold the command line arguments.

The code inside the method looks very similar to a block of C code. Notice that it uses the same blocking and indenting conventions that are used in C.


 Modify your `adding.java` file to include the main method ...


```
class adding
{
    public static void main (String args[])
    {
        int a, b, c;

        //check to see if arguments were entered
        if (args.length == 0)
        {
            //no values entered - assign default values
            a = 1;
            b = 3;
        }
        else
        {
            //convert user-defined values to integers
            a = Integer.parseInt(args[0]);
            b = Integer.parseInt(args[1]);
        }

        c = a + b;

        System.out.println("Sum = " + c);
    }
}
```

 Save your file.

 Compile your **adding.java** file by entering the following command at the DOS prompt...

```
javac adding.java
```

This creates a file called **adding.class**. This is the file contains your Java class, and therefore your Java application.


 Run your application by typing the following command at the DOS prompt...

```
java adding
```

Do not put the `.class` extension in your command ... it will cause an error. You should now see ...

```
Sum = 4
```

since there was no user input and it used its default values for a and b.

 Run your application again. This time enter 2 integer values after the application name...

```
java adding 4 6
```

You should now see ...

Sum = 10

This time there was user input so it added the first two values together and displayed the sum. Any subsequent data values would simply be ignored.

Exercise – A Simple Java Application

Read in x number of integer values from the command line. Display the number of values entered and print them in increasing order.

Design, code and demonstrate your working version. Have source code available at demo time.

Classes in Java

Now that you've had an introduction to the Java programming language and a little practice in generating Use Case Models and Class Diagrams, it's time to put them together.

Defining Classes in Java

Recall that when you write a Java application that it is actually a class. Its structure is ..

```
class className
{
    //data is declared here

    //methods are defined here
}
```

In this class you defined your main method which executed the application...

```
class className
{
    public static void main (String arg[])
    {
        //code goes here
    }
}
```

This class, which must have the same name as your application, is called the *driver* class. It's the one that gets things going.

The process of defining classes for the objects in your system is very similar. Each class has a name, a set of attributes and a set of methods. It also specifies how accessible it is (i.e. public, protected or private). Its basic structure is the same as for the driver class ...

```
public class className
{
    //data is declared here

    //methods are defined here
}
```

Say, for example, your class diagram includes the *Person* class shown here. Its would be defined in its own file called **Person.java** and its basic structure would look like...

```
public class Person
{
    //attributes are declared here

    //methods are defined here
}
```

Person
-\$ nbrPeople:integer
- name:string
- age:integer
- gender:char
+\$ getNbrPeople():integer
+ getlName():string
+ setlName()
+ getAge():integer
+ setAge()
+ getGender():char
+ setGender()

✎ Using any text editor, create a file called **Person.java** and enter the basic shell for your **Person** class (specified on the previous page). For simplicity's sake, put the **Person.java** file in its own folder.

Adding Attributes

A class can have three types of attributes ... **constant values**, **class attributes** and **instance attributes**. As in C, **constants** are fixed values. Those defined in a class are the ones that are required for calculations or data processing within the methods defined in the class. Say, for example, that there is a maximum number of *Person* objects that can be created in the system. That constant could be defined as...

```
public class Person
{
    //attributes are declared here
    static final int MAXPEOPLE = 3;

    //methods are defined here
}
```

Let's take a look at the keywords involved in this declaration...

static	attribute does not need an instance or object to have been created in order to be used ... required for constant attributes
final	attribute will not change ... required for constant attributes
int	data type ... changes depending on what type of data the constant is
MAXPEOPLE	attribute name ... shown in caps for visual clue that it's a constant value

✎ Add the constant **MAXPEOPLE** to the attributes section of your **Person.java** file.

The next type of attribute is a **class attribute**. These are single variables that are shared among all the instances of the class. Looking back at the class diagram on the previous page it is the one identified with a -\$... the minus sign (-) indicates that it is a private attribute and the dollar sign (\$) indicates that it is a class attribute. This attribute keeps track of the number of people objects you have created so that you don't exceed a specified limit. This class attribute is defined like...


```
public class Person
{
    //attributes are declared here
    static final int MAXPEOPLE;

    //class attributes
    private static int nbrPeople;

    //methods are defined here
}
```

Let's take a look at the keywords involved in this declaration...

private	this variable is only accessible to instances of this class ... indicated by minus sign (-) in class diagram
static	single variable shared among all instances of this class ... indicated by dollar sign (\$) in class diagram
int	data type ... changes depending on what type of data the attribute is
nbrPeople	attribute name ... shown in lower case for visual clue that it's a variable

 Add the class attribute called **nbrPeople** to the attributes section of your **Person.java** file. Save your file.

The last type of attribute is an **instance attribute**. These are variables that are created and used by each instance of the class. These are the more common type of attributes since they keep track of the actual characteristics of the object like *size*, *length*, *name*, etc. Instance attributes are usually placed after the class attributes in a class definition. The instance attributes for the *Person* class you've been looking at include...

```
public class Person
{
    //attributes are declared here
    static final int MAXPEOPLE;

    //class attributes
    private static int nbrPeople;

    //instance attributes
}
```



```


        //instance attributes
        private String name;
        private int age;
        private char gender;

        //methods are defined here
    }

```

Let's take a look at the keywords involved in this declaration...

private	this variable is only accessible to this object ... indicated by minus sign (-) in class diagram
String, int, char	data types ... changes depending on what type of data the attribute is (Note: String is a built-in Java class and must be capitalized)
name, age, gender	attribute name ... shown in lower case for visual clue that it's a variable

 Add the instance attributes called **name**, **age** and **gender** to the attributes section of your **Person.java** file. Save your file.

Adding Methods

A class can have two types of methods ... **accessor methods** and **processing methods**. Accessor methods include both **class methods** and **instance methods**.

Class accessor methods are used to access or modify class attributes. They are declared much the same way a function is declared in C and use many of the same keyword modifiers you would when declaring a class attribute. For the *Person* class you need to declare and define a class accessor method to access and return the class attribute *nbrPeople* ...

```

public class Person
{
    //attributes are declared here
    static final int MAXPEOPLE;

    //class attributes
    private static int nbrPeople;

    //instance attributes
    private String name;
    private int age;
    private char gender;
}

```


```

        //methods are defined here
        //class accessor methods
        public static int getNbrPeople()
        {
            return nbrPeople;
        }
    }

```

Looking at the keywords involved in this declaration you see some that are familiar and some are new...

public	this method is accessible from outside this class ... indicated by plus sign (+) in class diagram
static	this method can be called without creating an instance of this class ... indicated by dollar sign (\$) in class diagram
int	return data types ... changes depending on what type of data is returned
getNbrPeople()	method name and parameter list ... in this case no parameters are passed to the method

 Add the class method called **getNbrPeople()** to the methods section of your **Person.java** file. Save your file.

Instance accessor methods are used to access or modify instance attributes. The only difference between declaring class accessor methods and instance accessor methods is the keyword **static**. You need it for class methods, you don't for instance methods. In the *Person* class there are six instance methods, two for each attribute ...one to access the attribute, the other to modify it.

```

public class Person
{
    //attributes are declared here
    static final int MAXPEOPLE;

    //class attributes
    private static int nbrPeople;

    //instance attributes
    private String name;
    private int age;
    private char gender;

```

```

//methods are defined here
//class accessor methods
public static int getNbrPeople()
{
    return nbrPeople;
}

//instance accessor methods
public String getName()
{
    return name;
}

public void setName(String name)
{
    this.name = name;
}

public int getAge()
{
    return age;
}

public void setAge(int age)
{
    this.age = age;
}

public char getGender()
{
    return gender;
}

public void setGender(char g)
{
    gender = g;
}
}

```

There is nothing different about the *getName()*, *getAge()* and *getGender()* methods. They look just like the *getNbrPeople()* method without the **static** keyword. Other than the keyword **public** you can't tell them from C function declarations.

Looking at the parameter list for the *setName()*, *setAge()* and *setGender()* methods you can see that values are passed in to Java methods the same way values are passed into C functions. The instruction **gender = g;** in the *setGender()* method accepts a character from the calling method, stores it in a local variable called *g* and assigns it to the instance attribute called *gender*.

The other two modifying methods *setName()* and *setAge()* use a different technique. They both receive a value through the parameter list and assign that value to a local variable (*name* and *age*) that has the same name as one of the instance attributes. Any time the method refers to that variable it will use the local variable ... not the instance attribute. To access the instance attribute you use the keyword **this** to refer to the current instance of the object ... using *name* or *age* accesses the local variable while using *this.name* or *this.age* accesses the instance attribute.

☞ Add the six instance methods to the methods section of your **Person.java** file. Save your file.

Processing methods are just that ... methods designed to do internal work for you. Use this type of method for things like number crunching, string processing, calculating values, etc. The same rules apply as do in C ... anytime you would extract a block of code to make a C function, do the same to create a Java method. There are no processing methods required for the *Person* class.

Class Constructors

There is just one more thing required when defining a class ... a **class constructor**. This is the block of code that gets executed everytime you create a new instance of the class. It normally contains code to update your class attributes and initialize your instance attributes.

The constructor looks like any other method. It is normally defined right after the attribute declarations and before the class and instance methods. It always has the same name as the class...

```
public class className
{
    //data is declared here

    //class constructor
    className( )
    {
        //initialization code goes here
    }

    //methods are defined here
}
```

Notice that the constructor does not need an accessibility specifier ... it doesn't need to be defined as public, private or protected. This is because it has the same accessibility as the class itself.

With the constructor added, you now have the complete class definition for the *Person* class ready for use by any Java application that is keeping track of people. It includes the attribute declarations, the class constructor (that increments the number of People class attribute) and the method declarations...

```
public class Person
{
    //attributes are declared here
    static final int MAXPEOPLE;

    //class attributes
    private static int nbrPeople;

    //instance attributes
    private String name;
    private int age;
    private char gender;
```

```

//class constructor
Person( )
{
    nbrPeople++;
}

//methods are defined here
//class accessor methods
public static int getNbrPeople()
{
    return nbrPeople;
}

//instance accessor methods
public String getName()
{
    return name;
}

public void setName(String name)
{
    this.name = name;
}

public int getAge()
{
    return age;
}

public void setAge(int age)
{
    this.age = age;
}

public char getGender()
{
    return gender;
}


public void setGender(char g)
{
    gender = g;
}
}

```

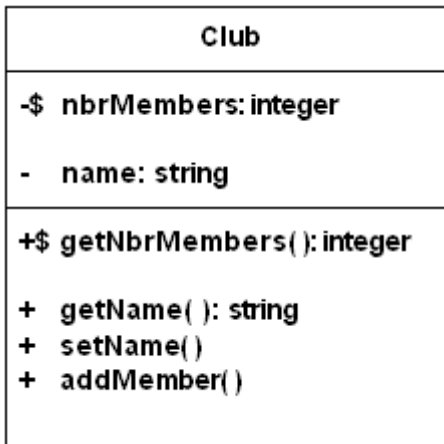
 Add a class constructor to the top of the methods section of your **Person.java** file. Save your file.

Accessing User-Defined Classes


Now that you've defined a class of your own, it's time to use it in a Java application. The process is quite simple ... as long as your user-defined class is in the same folder as your driver class, the application will find and use the class as it is instructed! Let's create a Java application that simulates a club of some kind. This club will use the Person class to keep track of its members.

 Create a driver class file called **Club.java**. Make sure that you put it in the same folder as your **Person.java** file. Save your file.


Just like any other class, the *Club* class needs to define its attributes, a constructor and its methods. Because this is the driver class it also needs a *main()* method to get things going. Assume the class diagram for the *Club* class looks like...



You can see that there are two attributes both defined as private (the minus signs). The first, *nbrMembers*, is specified as a class attribute by the dollar sign. The second attribute, *name*, is an instance variable and will use the built-in Java String class.

 Add these two attributes, *nbrMembers* and *name*, to your **Club.java** file.

You can also see that there are four methods required in this class. The first, *getNbrMembers()*, is a **public class method** used to access the value of the class attribute *nbrMembers*. Remember that the plus sign indicates that the method is public and the dollar sign indicates that it is a class method. The remaining three methods are instance methods. *getName()* and *setName()* are used to access the instance attribute name. *addMember()* is used to add another member to the club.

 Add the methods, *getNbrMembers()*, *getName()* and *setName()* to your **Club.java** file. The method *getNbrMembers()* should simply return the current number of club members. The method *getName()* should return the name of the club and method *setName()* should allow you to pass in a string and set or modify the club name.

Creating Instances

In order to add the method *addMember()* you have to know how to create or instantiate an object of a specified class. This is very similar to creating a block of dynamic memory in C++. The keyword **new** specifies followed by the name of the desired class indicates that a new object is to be created...

```
new ClassName( );
```

This creates the object but doesn't allow you to access to it. To do that you need to assign this new object to a local variable of type *ClassName* ...


```
ClassName var = new ClassName();
```

Calling Instance Methods

Anytime you want to work with the new object you use the local variable name you assigned it ... in this case *var*. To call an instance method for this object you need to specify both the object and the method joined with a period...


```
var.setName("Chris");
```

var is the object name, *setName()* is the method and "Chris" is the data that you are passing to the object.


 Add the final instance method, *addMember()*, to your **Club.java** file. Pass in a name, age and gender. Create a new object of type *Person* in the method and use the accessor methods *setName()*, *setAge()* and *setGender()* to write the passed values to the object's attributes. Define *Person* to be your return type and return the local variable you used when you created your new member. Save your file.

Class Constructors

You only have a few things remaining to complete your *Club* class definition. You must add a class constructor and the *main()* method to get things started.

 Add a class constructor for the *Club* class. Remember that it must have the same name as the class and it does not required an accessibility specifier. Since the constructor contains the code that gets executed when you create an instance, the only thing you need to add is an instruction to initialize the number of members to zero. Save your file.

Main Method

 Add the *main()* method at the end of your class file. It has to be declared the same way in all Java applications so refer back to your last program if you can't remember how to format the method. The first instruction you need is to create a new club. Include the instruction ...

```
new Club();
```


to do this. You don't really need a variable to keep track of this because you're only working with one club. Next you want to create a member for your club so you want a call to the *addMember()* method. Remember that you defined it to expect a name, age and gender and to return an object of type *Person*, so your call must look something like...

```
Person member1 = addMember("Chris", 22, 'M');
```

From this point on you can access this member using the local variable *member1*. For example, if you want to display the age of *member1* you would include an instruction ...

```
System.out.println(member1.getName() + " is " + member1.getAge());
```

The object is always followed by a dot and the desired accessor method.

 Modify the *main()* method so that it creates at least three members.

Accessing Class Methods


Another thing you have to be able to do is access your class attributes using your class methods. The way to do this is a little bit different than calling instance methods. With instance methods you precede the method name with the object name ...


```
var.setName("Chris");
```

For class methods you precede the method name with the class name...

```
Club.getNbrMembers( );
```

This means that you have access to your class attributes anytime ... even if you haven't created any club members yet.

 Modify the *main()* method so that it uses its class attribute *nbrMembers* (through its class accessor method) to loop through and display all the information about each club member. Save your file.

 Compile your Java application. As long as your *Club* class and your *Person* class are in the same folder it will find and compile both classes. Correct any syntax errors you may have. Once you have a successful compile, run your application and make sure your club keeps proper track of its members.

Exercise –Java Classes

Modify your *Club* class to accept *x* number of members specified in the command line. A command line like...

```
java Club Sam 12 F Chris 15 M John 14 M Mary 13 F
```

would add 4 members to the club and display something like...

```
Sam is 12 and female
Chris is 15 and male
John is 14 and male
Mary is 13 and female
```

Remember you can go back and add any processing methods you think will make this type of data handling easier and more modular.

Object Oriented Principles

Now that you can create simple Java applications and their required classes it is time to take a closer look at what Object Oriented Programming (OOP) is all about. Remember that first and foremost OOP is a way to model and solve real world problems in a way that better represents the things or '*objects*' that are involved.

There are four characteristics that identify a program as object oriented...

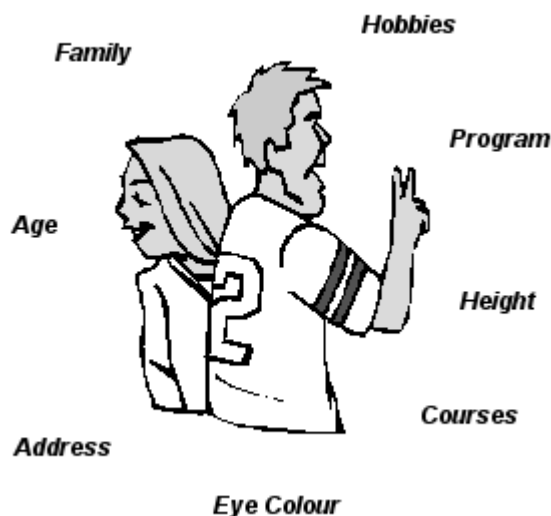
1. Abstraction
2. Encapsulation
3. Inheritance
4. Polymorphism

Abstraction

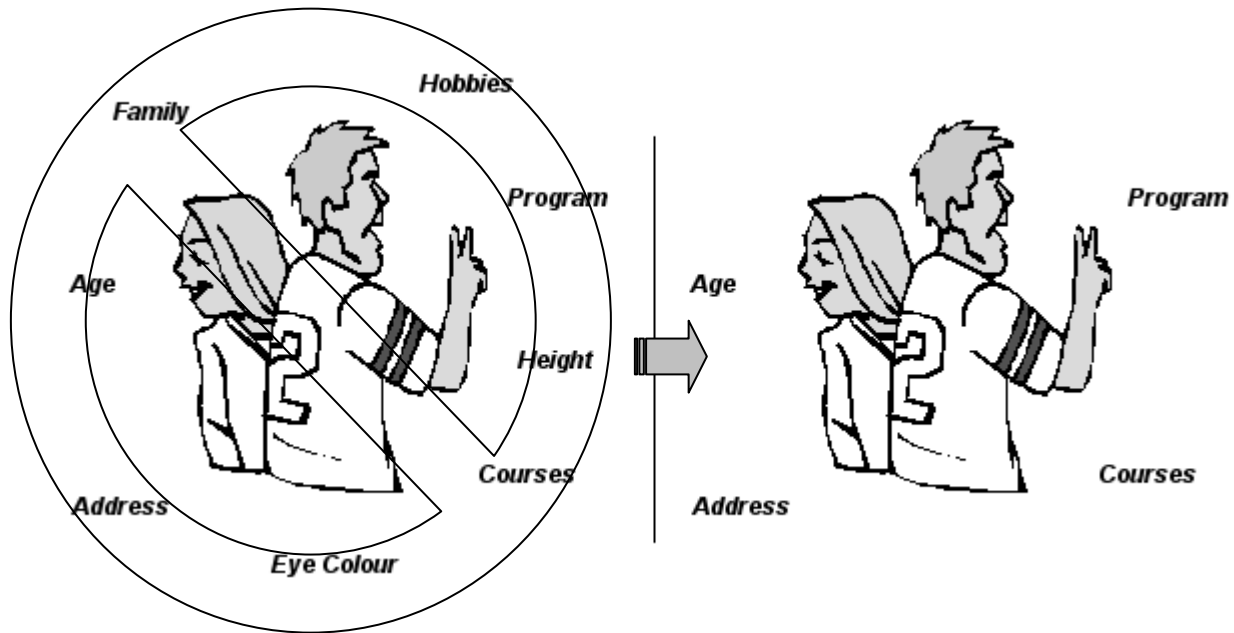
Abstraction can be defined as...

“... the process of reducing an object to its essence so that only the necessary elements are represented. Abstraction defines an object in terms of its properties (attributes), behaviors (functionality), and interface (means of communicating with other objects).” Source: <http://www.atis.org>

Think about yourself for a minute. You have all kinds of attributes ... physical and otherwise. You are a student who may have brown hair, green eyes and be 5' 10" tall. You may play a musical instrument or you may be very good at soccer. You may be a very reliable person and you are most certainly a good programmer! All of these attributes and behaviours make you who you are.



But if your task is to model an object oriented system that tracks student records, a lot of these attributes are not necessary. Now just the things that identify you as a student are required. This process of defining an object just by what is necessary for the system is abstraction.



Encapsulation

Encapsulation can be defined as...

“... the inclusion within a program object of all the resources needed for the object to function – basically, the method and the data.” Source: www.kbcafe.com

As an object oriented system designer it is your job to make sure that all objects within your system contain and protect all relevant data. Information hiding and controlling all access to the information through accessor methods is part of encapsulation. All operations on the data should be done by the object itself ... the rest of the system doesn't care how any of those operations are accomplished, just that they get done.

Inheritance

Inheritance can be defined as...

“... the way to **change or extend an already existing class (parent class)** by making a **child (inherited, derived) class** and writing **only the changed or enhanced** part in that child class.” Source: <http://bepp.8m.com/english/oop/inheritance.htm>

One of the nice things about inheritance is the reusability of code. All parts of the parent class including its data and methods are passed to the child class. That means you code it once and use it as many times as you need it. It also means that if you improve or modify any part of the parent, that

change is immediately transferred to all the children. No more searching through your program to find all the places where you used similar code!

Polymorphism

Polymorphism can be defined as ...

“...the ability of objects to act depending on their **run-time** type.”

Source: <http://bepp.8m.com/english/oop/inheritance.htm>

Polymorphism works with inheritance. Recall that when a child class is derived from a parent class it inherits all the data and methods defined in the parent. But sometimes the way a child must carry out a particular method is different from the way a parent would.

Say for example a parent class called *Shape* defines, among other things, a method to *draw* the shape. A child class called *Circle* is inherited from the *Shape* class. A second child class called *Square* is also derived from the *Shape* class. Both of these child classes inherit the ability to draw but the way it must be carried out in the actual code is quite different. This is where polymorphism comes in. It allows the *draw* method to be modified in the child class to accommodate the special requirements of the child class.

Advantages of OOP

Once you've been using object oriented methods for a while you'll find that there are significant advantages to this approach. They become most apparent when designing in teams or for particularly large systems.

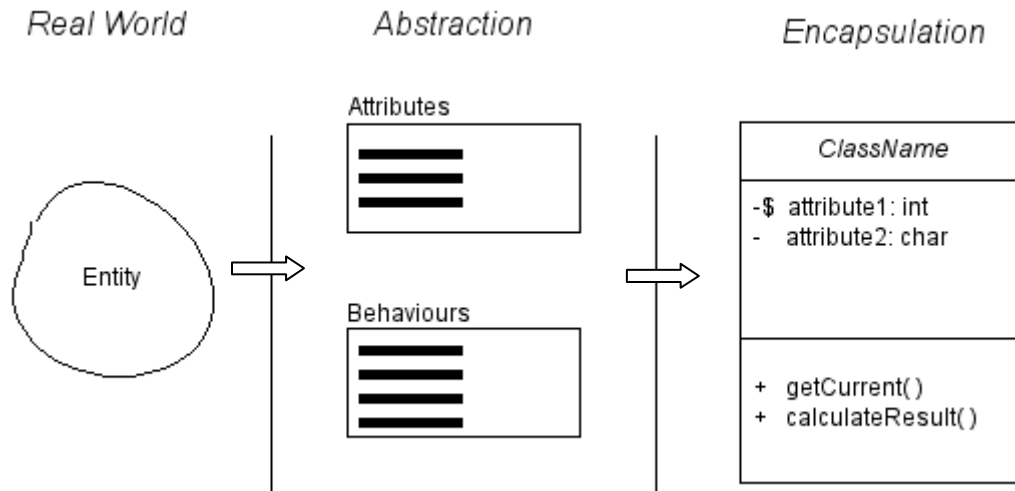
The first advantage is **faster development time**. Applying encapsulation techniques means that you can build discrete, reusable, self contained components. Inheritance allows you to build new classes from existing ones instead of having to start from scratch each time. Finally polymorphism lets you apply the knowledge of existing objects to new ones. This means your code can be more generic.

Object oriented methods also imply **easier debugging**. Because you're reusing objects there is less code to debug. Encapsulation and its information hiding capability prevents accidental overwriting of data values. It also helps localize the source of bugs.

Finally object oriented methods allow for **faster upgrading**. You can quickly make changes to one specific object without having to alter the rest of the application.

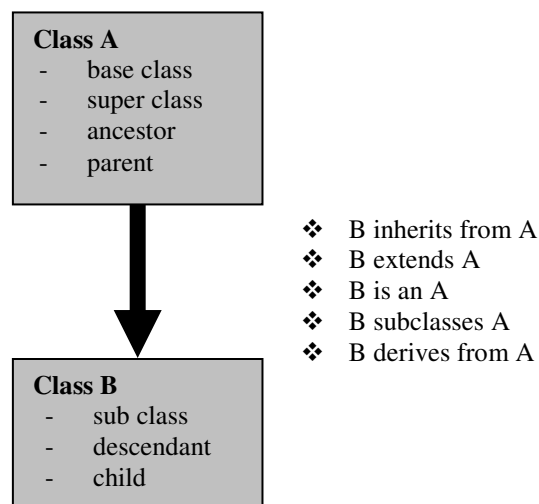
Applying the Four Object Oriented Principles

In your object oriented planning (Use Cases, Sequence Diagrams and Class Diagrams) you have already applied the first two principles ... **abstraction** and **encapsulation**. You start by looking at the system as a series of real objects. You abstract the essential elements of these objects and then set them up as a series of classes ... each with its own attributes and methods...



A Closer Look at Inheritance

Now it's time to take a look at inheritance ... the ability to derive child classes from a parent class. Depending on what resource you happen to be using, there are a number of different terms used to describe the parent and child classes as well as what the relationship is ...




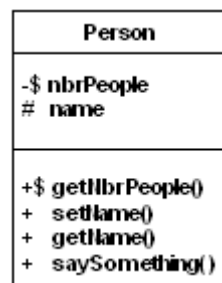
In this diagram **Class A is the parent**. You can see that it may be called the *base class*, the *super class* or the *ancestor*. **Class B is the child**. It may be called the *sub-class* or the *descendant*. Often

these terms are used interchangeably. In this document they will continue to be called the parent and child classes.

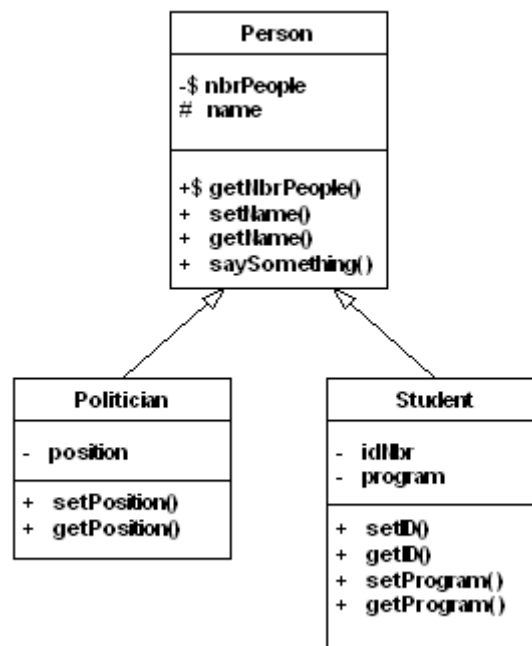
You can also see the different ways of identifying that **class B is the child of class A**. '*B inherits from A*', '*B extends A*', '*B is an A*', '*B subclasses A*' and '*B derives from A*' are all ways of saying that B is A's child. In this document the terms '*inherits from*' or '*derives from*' will be used.

It's now time to try your hand at some inheritance. Before you start take a close look at the **Person** class diagram shown here. In the attributes section you'll see a new symbol (#). So far you've seen private access to attributes (indicated by a minus sign '-') and public access to methods (indicated by a plus sign '+'). The pound symbol (officially called an octothorpe) specifies **protected access**. This protects the attribute like private but allows it to be inherited. That means anytime an attribute is to be inherited it should be declared protected instead of private.

 Create a new folder for this exercise. Use the *Person* class diagram to generate a **Person.java** class file. Put an instruction to increment the number people and one to output 'In the Person constructor' in the class constructor. Have the *saySomething()* method display something like "I'm Chris" substituting in the value of the name attribute.



To show inheritance in a class diagram you use an arrow **from the child to the parent**. There can be as many children of a parent class as you require. In this case there are two child classes. Each child class **inherits all public and protected** elements from the parent.



Each of the two child classes has its own additional elements. The *Politician* class adds one more attribute called ‘*position*’ and two instance accessor methods called *setPosition()* and *getPosition()*. The *Student* class adds two new attributes ‘*idNbr*’ and ‘*program*’. It requires four more accessor methods ... *setID()*, *getID()*, *setProgram()* and *getProgram()*. These elements are in addition to the inherited attribute ‘*name*’ and the inherited methods *setName()*, *getName()* and *saySomething()*. It is important to define all the attributes and methods that are **common** to all people in the parent class.

To convert the child class diagram to Java code you must use the keyword ‘**extends**’. This specifies that the child class will inherit all public and protected elements from the specified parent class. The actual definition of a child class has the following format...

```
public class ChildName extends ParentName
{
    //additional attributes are declared here


    //additional methods are defined here
}
```


Other than the ‘extends’ portion of the declaration, it is pretty much the same as that of a ‘stand-alone’ class. For the *Politician* class the declaration would look like...

```
public class Politician extends Person
{
    //additional attributes are declared here
    String position;

    //constructor
    Politician()
    {
        System.out.println("In the Politician constructor");
    }

    //additional methods are defined here
    void setPosition(String posn)
    {
        ...
    }
}
```

 Create a **Politician.java** class file. Enter the code shown above completing the *setPosition()* method and adding the *getPosition()* method. Save this file in the same folder as the *Person* class file.

 Use the class diagram to create a **Student.java** class file. Save this file in the same folder as the *Person* class file.

 Create a driver class called **Directory.java**. Add the following code to this file...

```
public class Directory
{
    Directory()
    {
        showSize();


        Person entry1 = new Person();
        showSize();
        entry1.setName("Sam");
        entry1.saySomething();

        Politician entry2 = new Politician();
        showSize();
        entry2.setName("Jen");
        entry2.setPosition("Mayor");
        entry2.saySomething();
        System.out.println("and I'm " + entry2.getPosition());

        Student entry3 = new Student();
        showSize();
        entry3.setName("Chris");
        entry3.setProgram("Electronics");
        entry3.saySomething();
        System.out.println("and I'm in" + entry3.getProgram());
    }

    void showSize()
    {
        System.out.print("Current # people: ");
        System.out.println(Person.getNbrPeople());
    }

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

 Compile the **Directory.java** file. Run your application. You can see from your output that you can create instances of the parent class as well as instances of each child class. The methods defined in the parent class are accessible from either the parent or the child classes. The methods defined in the child classes are only accessible by the objects of that particular class.

Alternate Constructors

Up to this point anytime that you create an instance of a class you then have to call accessor methods to assign values to each of the attributes. This seems to be a lot more work that is really necessary. There is another way ... you can set up an alternate constructor that accepts a parameter list containing values to initialize each attribute. An alternate constructor for the *Person* class would be positioned after the default constructor and might look something like...

```
//alternate constructor
Person(String name)
{
    this();
    System.out.println("In alternate Person constructor");
    this.name = name;
}
```

The format of the alternate constructor looks the same as default except for the parameter list which can be as big or small as required. What is new is the statement...

```
this();
```

This statement actually causes the default constructor to be executed. Since you have already defined some initialization code in the default constructor you want to make sure this gets executed in addition to the new code you've just entered in the alternate constructor. **Do not** repeat this code in the alternate constructor ... that would defeat the encapsulation principle discussed earlier!

☞ Add the alternate *Person* constructor to the **Person.java** file. Add a similar alternate constructor to each of the *Politician* and *Student* classes. Make sure you include an output statement identifying where you are and that you initialize all of the attributes available in the class.

☞ Add the following code to the end of the *Directory* constructor of your **Directory.java** class...

```
Person entry4 = new Person("Bill");
showSize();
entry4.saySomething();

Politician entry5 = new Politician("John", "MPP");
showSize();
entry5.saySomething();
System.out.println("and I'm " + entry5.getPosition());

Student entry6 = new Student("Mary", 12345, "Computer");
showSize();
entry6.saySomething();
System.out.print("#" + entry6.getID);
System.out.println(" and I'm in" + entry6.getProgram());
```

You can see from this modified code that you no longer need to use the accessor methods to set each attribute individually.

☞ Recompile the **Directory.java** file. Run your modified application.

Polymorphism and Overriding Methods

If you look at the block of code you entered for adding directory entries *entry5* and *entry6* ...

```
Politician entry5 = new Politician("John", "MPP");
showSize();
entry5.saySomething();
System.out.println("and I'm " + entry5.getPosition());


Student entry6 = new Student("Mary", 12345, "Computer");
showSize();
entry6.saySomething();
System.out.print("#" + entry6.getID);
System.out.println(" and I'm in" + entry6.getProgram());
```


you had to enter extra output statements to display all the required information about the entry. If the entry was a *Politician* you had to display the position. For a *Student* you had to display the ID number and the program. It would be more efficient if you could simply call the `saySomething()` method and it would display all the required information.

This is possible using a technique called **method overriding**. It is very similar to local and global variables that have the same name... the local variable takes precedence over any global variable of the same name. In method overriding a method defined in a child class that has the same name as a method defined in the parent class is the one that gets executed. You can in fact use the identifier **super** to cause the parent method to be executed as part of the child method...


```
super.methodName()
```

where `methodName()` is the method that is being overridden. By placing all the common tasks in the parent method and any specialized tasks in the respective child class methods you can have the best of both worlds.

 Open the *Person* class file. Modify the code in the `saySomething()` method so that it uses a `print` instead of a `println` to output the information. Save your file.

 Open the *Politician* class file. Add the following code to the methods section...

```
public void saySomething()
{
    super.saySomething();
    System.out.println(" and I'm " + position);
}
```

 Open the *Student* class file. Add the following code to the methods section...

```
public void saySomething()
{
    super.saySomething();
    System.out.print("#" + idNbr);
    System.out.println(" and I'm in" + program);
}
```

☞ Open the *Directory* class file. Remove the extra output statements after the calls to the *saySomething()* method. Recompile your file. Your output should now display all the pertinent information regardless of whether the entity is of type *Person*, *Politician* or *Student*.

Inheritance Modifiers

There are two modifiers that can be used to control whether and how other classes inherit from a parent class. They are...

- **abstract**
- **final**

Abstract Modifier

The **abstract** modifier lets you force a class or method to be implemented in a child class. If you make a class abstract it can't be instantiated. This means that it is only a template ... child classes must be created before any of the attributes and methods can be accessed.

Abstract Classes

You would choose to make a class abstract if you want to group together a number of related but different classes under the same umbrella. The parent class itself has no concrete use. The child classes are the useful entities. Java uses this technique to define its *Number* class. The child classes, *Integer*, *Float*, *Long*, etc. are very useful when programming applications, but the parent class *Number* is too generic to be useful.

To specify that a class is abstract you include the keyword **abstract** in the class declaration...

```
public abstract class className
{
    //attributes

    //methods
}
```

If you try to execute a statement like...

```
className entity = new className;
```

you will get a message something to the effect ...

```
"class className is an abstract class. It can't be instantiated."
```

Once you define a child class you can create all the objects of that class that you'd like...

```
public class childName extends className
{
    //attributes

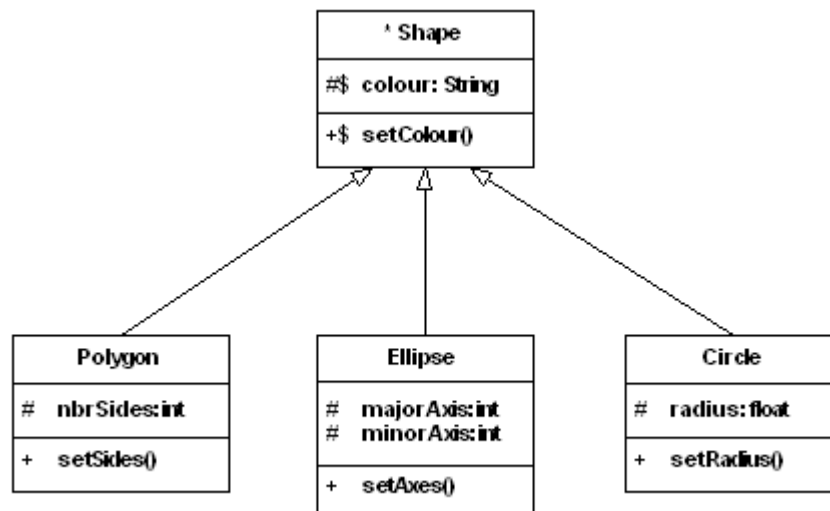
    //methods
}
```

Now if you try to execute a statement like...

```
childName entity = new childName;
```

you will have no problems.

Say for example you are asked to code the following classes...




Looking at the *Shape* class you can see a new symbol ... the asterisk (*). This indicates that the class is abstract. Remember that means there can be no objects of class *Shape*. There are three classes that inherit all the attributes and methods from *Shape* ... *Polygon*, *Ellipse* and *Circle*. Each of these also adds attributes and methods specific to its purposes.

The class definition for the *Shape* class as it is defined in the class diagram is ...

```
public abstract class Shape
{
    protected static String colour;

    public static String getColour()
    {
        return colour;
    }
}
```

 Create a **Shape.java** file that contains the above code.

☞ Using the above class diagram, create class definition files for the *Polygon*, *Ellipse* and *Circle* classes. Make sure that each of the classes inherits from the *Shape* class and that the individual files are stored in the same folder as the **Shape.java** file. Each class should have both a default constructor and an alternate constructor that accepts values for each of the attributes in the class. Include some output statements in each method indicating where you are. Add accessor methods for each attribute.

☞ Create a driver class called *showShapes*. It should contain the basic *main* method that creates an instance of *showShapes*. It must also contain the constructor for the *showShapes* class. The constructor should include code to create at least one instance of type *Polygon*, *Ellipse* and *Circle*. Use the accessor methods you wrote in the *Polygon*, *Ellipse* and *Circle* classes to display at least one attribute value for each object created.

☞ Compile and run your program.

Polymorphism and Abstract Methods

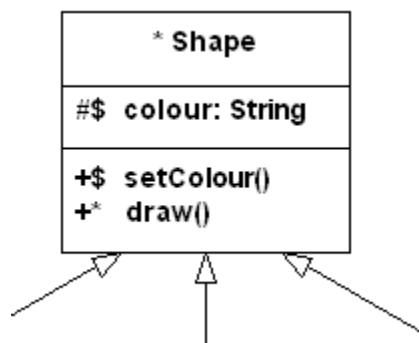
Methods can also be declared **abstract**. You would choose to make a method abstract if you have some function that is common to all child classes but has to be handled differently by each different type of object. Say for example you are go use the *Shape* class as part of graphics package. You have to be able to draw each shape on the screen but the process is quite different for each type of class ... a circle is based on its centre point and its radius, an ellipse on its centre point and its axes, and a polygon on its top left corner and its dimensions.

The process of defining a method as abstract is quite simple...

```
public abstract returnType methodName();
```

Notice there is no code in the method. Because you are going to define the process differently for each child class it is left empty. By making it an abstract method and including it in the parent class you are specifying that each child class must override the method by defining its own unique process.

Look at methods section of this modified class diagram ...



You can see that there is an abstract method called *draw()*. The asterisk (*) indicates that the method is to be declared **abstract**. Its declaration would look like...

```
public abstract void draw();
```

✎ Modify the *Shape* class to include the abstract method *draw()*. Don't forget the semi-colon at the end of the declaration.

✎ Modify each of the *Polygon*, *Ellipse* and *Circle* classes to include a *draw()* method. For now simply put an output statement in each that prints something like 'Drawing a circle.'

✎ Modify the *showShape* constructor to include a call to *draw()* for each of the objects created.

✎ Compile and run your program.

✎ Go back and comment out the *draw()* method from the *Polygon* class. Recompile your application. Make a note of the error message that you get. You'll see this message any time you forget to define an abstract method in an inherited class.

✎ Remove the comments from the draw method. Recompile and make sure your application runs properly.

Final Modifier

The **final** modifier is the opposite of the abstract modifier. It prevents a class or a method from being implemented in a child class. It also ensures that the method or class can only be implemented the way you design it ... nothing can be added and nothing can be overridden.

Final Classes

If you make a class **final** it can't be extended ... the only thing you can do with a final class is to create objects. Because of this all the methods in a final class are also considered final ... there is no way that anyone can override the original method's code.

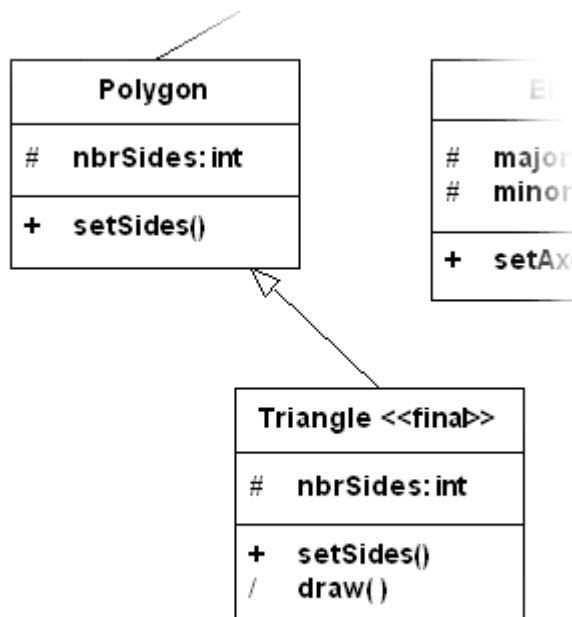
To specify that a class is **final** you include the keyword final in the class declaration...

```
public final class className
{
    //attributes

    //methods
}
```

Remember to define all the required attributes and methods in the body of the class because there is no way to extend it through inheritance.

There doesn't appear to be a symbol for indicating that a class is final. The convention seems to be to include the modifier `<<final>>` after the class name...



If you look closely you'll see one other new symbol in this class diagram. The `draw()` method included in the *Triangle* class has a slash (/) in front of it. This indicates that it is an inherited or **derived** method.

☞ Add a *Triangle* class that inherits from the *Polygon* class. Make it a **final** class and make sure that the individual files are stored in the same folder as the **Shape.java** file. The class should have both a default constructor and an alternate constructor that accepts values for each of the attributes in the class. Include some output statements in each method indicating where you are.

☞ Modify the *showShape* constructor to create at least one instance of type *Triangle*. Add a call to `draw()` for this *Triangle* object.

☞ Compile and run your program.

☞ Add an *Equilateral* class that inherits from the *Triangle* class. Make sure that the individual files are stored in the same folder as the **Shape.java** file. The class should have both a default constructor and an alternate constructor that accepts values for each of the attributes in the class. Include some output statements in each method indicating where you are.

☞ Modify the *showShape* constructor to create at least one instance of type *Equilateral*. Add a call to `draw()` for this *Equilateral* object.

☞ Recompile your program. Make a note of the error message that you get. You'll see this message any time you try to inherit from a final class.

☞ Remove the instructions to create an *Equilateral* object from the *showShape* constructor. Recompile and make sure your application runs properly.

Final Methods

If you make a method **final** it can't be overridden ... the only thing you can do with a final method is use it exactly as it was written. This helps prevent someone from altering the way you handle a particular task within your system.

To specify that a method is **final** you include the keyword final in the method declaration...

```
public final methodName( )  
{  
    //method code  
}
```

☞ Modify the *Polygon* class so that the *draw()* method is **final**.

☞ If you haven't already, add a *draw()* method to the *Triangle* class that outputs 'Drawing a Triangle' to the screen.

☞ Recompile your program. Make a note of the error message that you get. You'll see this message any time you try to override a final method.

☞ Remove the **final** modifier from the *draw()* method in the *Polygon* class. Recompile and make sure your application runs properly.

Multiple Inheritance

There are times that you want a class to inherit characteristics and behaviours from more than one parent class. This is called **multiple inheritance**. Unfortunately, Java does not support multiple inheritance. What it does provide is a special type of abstract class called an **interface**.

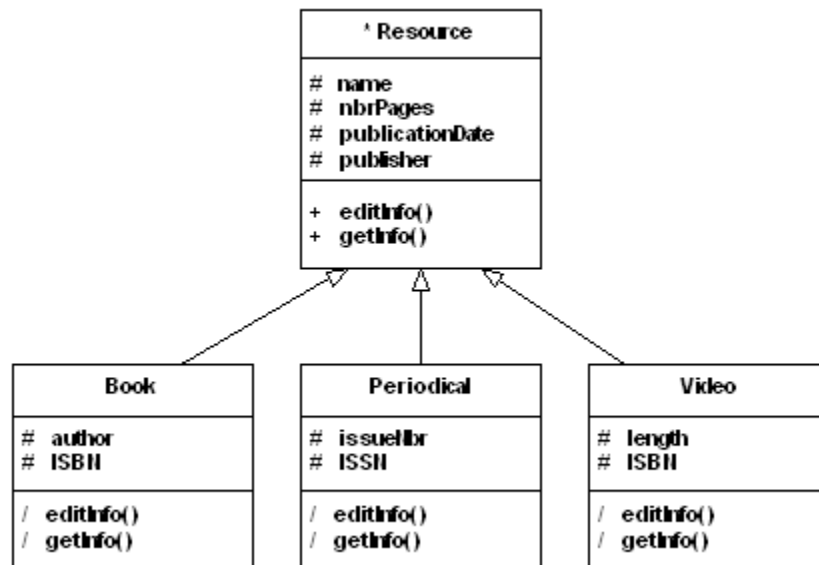
Interfaces

Sun's definition of an interface includes the following description ...

"In English, an interface is a device or a system that unrelated entities use to interact. According to this definition, a remote control is an interface between you and a television set, the English language is an interface between two people, and the protocol of behavior enforced in the military is the interface between people of different ranks. Within the Java programming language, an interface is a device that unrelated objects use to interact with each other. An interface is probably most analogous to a protocol (an agreed on behavior)."

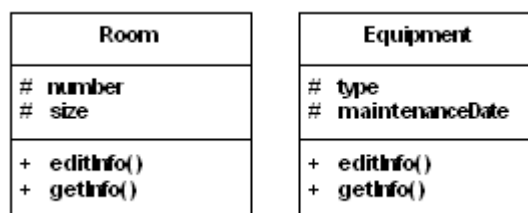
Source: <http://java.sun.com/docs/books/tutorial/java/concepts/interface.html>

Think about a library system. There are all kinds of entities in the system ... books, journals, newspapers, video tapes, audio tapes, CDs, DVDs, etc. These have some common characteristics and behaviours and some that are unique. You can see this in the following class diagram...



where the common characteristics and behaviours are defined in the abstract *Resource* class and the unique attributes defined in the *Book*, *Periodical* and *Video* classes.

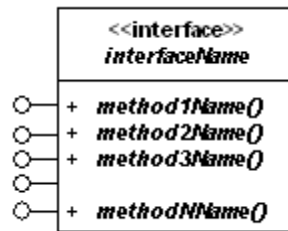
The library system also includes things like viewing rooms, and audio-visual equipment. These would each have their own class diagrams...



You'll notice that these two new classes also include *editInfo()* and *getInfo()* methods. These are not inherited from the *Resources* class and are therefore not necessarily the same. This is another example of **polymorphism**, where different classes exhibit similar behaviours.

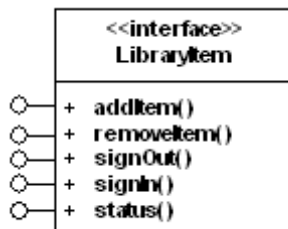
Because all of these resources exist together in a library they share a tracking system. This tracking system is quite different from the resources themselves and doesn't really care whether it is managing a resource, a room or a piece of equipment at any given time. This is where an **interface** comes in. The tracking system defines a set of methods that each class in the system implements. There is no data associated with the interface ... just method definitions.

In a class diagram an interface looks like ...



The keyword <<interface>> differentiates this rectangle from a class definition rectangle. The interface symbol (the circle/line) marks each of the methods that exist in the interface definition.

For the library system, you may choose to define an interface called *LibraryItem* to keep track of the comings and goings of all types of library resources. The class diagram component for this interface looks like...




The Java code for defining an interface is...

```
public abstract interface interfaceName
{
    public abstract returnType method1Name();
}
```

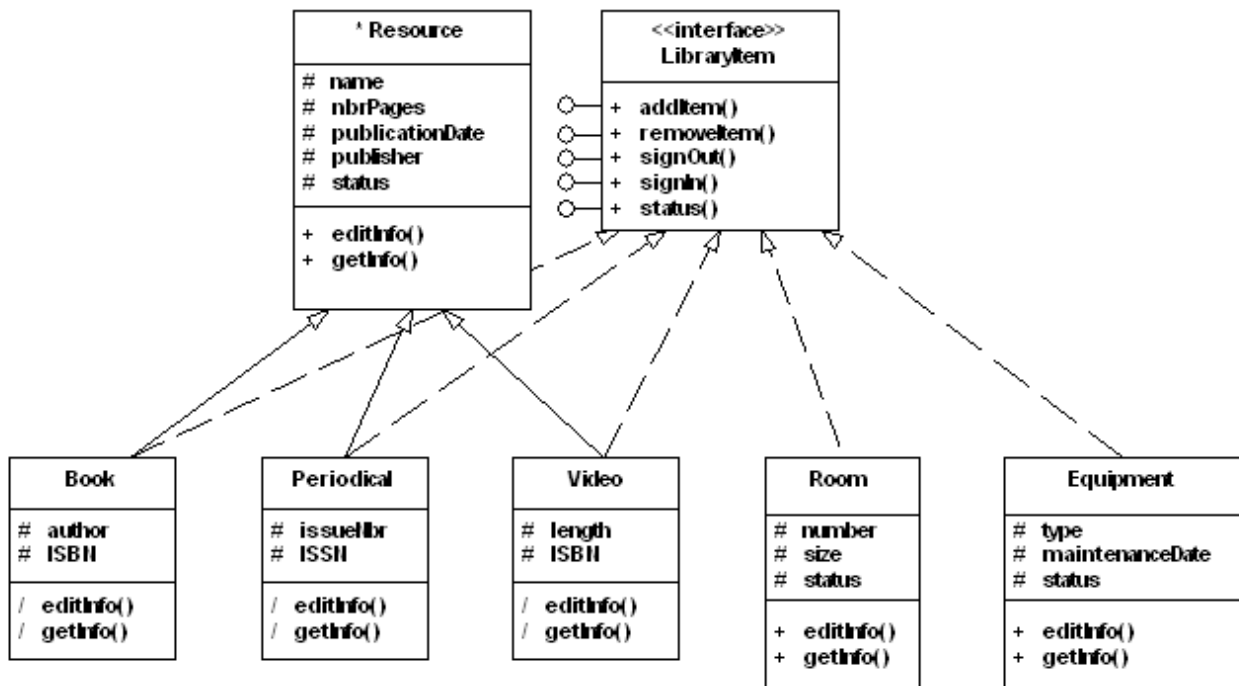
The interface definition needs both the **abstract** and the **interface** keywords. The **abstract** keyword is required because there can be no instances of an interface. Each of the methods in the interface is also defined as abstract. They must be defined in any class that implements the interface. Notice there is no code defined in the method. Each method declaration must be terminated with a semi-colon.

For the LibraryItem interface the complete interface definition would be...

```
public abstract interface LibraryItem
{
    public abstract boolean addItem();
    public abstract boolean removeItem();
    public abstract boolean signOut();
    public abstract boolean signIn();
    public abstract int status();
}
```

 Create a **LibraryItem.java** file that contains the above interface definition.

When you add an interface to your class diagram you must show which classes implement it. To do this you use an **arrow** similar to the one you use to indicate inheritance but with a **dotted line**. The class diagram for the library system would look something like...



You can see that the *Book*, *Periodical* and *Video* classes all inherit from the abstract *Resource* class. All five classes *Book*, *Periodical*, *Video*, *Room* and *Equipment* implement the *LibraryItem* interface. Notice as well that an additional attribute called *status* has been added to the *Resource*, *Room* and *Equipment* classes. This attribute will be used with the interface methods.

When a class implements an interface it must provide code for each of the methods specified in the interface. This allows any number of classes that may exist in different hierarchies, to be treated as common elements of a system. Each of them includes the attributes and methods defined or inherited as well as the methods defined in the interface. To implement an interface the class definition looks like...

```

public class className implements interfaceName
{
    //attributes go here

    //own methods go here

    //interface (implemented) methods go here
}
    
```

The keyword **implements** indicates that the class will include all the methods defined in the specified interface.

The *Room* class definition would look like...

```
public class Room implements LibraryItem
{
    static final int BOOKED = 1;
    static final int AVAILABLE = 0;

    protected int number;
    protected int size;
    protected int status;

    //own methods
    public void getInfo()
    {
        System.out.print("Room number " + number);
        System.out.println(" is " + size + " square feet");
    }

    public void setInfo(int nbr, int size)
    {
        number = nbr;
        this.size = size;
    }

    //implemented methods
    public boolean addItem()
    {
        System.out.println("Adding room " + number + " to system");
        return true;
    }


    public boolean removeItem()
    {
        System.out.println("Removing room " + number);
        return true;
    }

    public boolean signOut()
    {
        System.out.println("Room " + number + " booked");
        status = BOOKED;
        return true;
    }

    public boolean signIn()
    {
        System.out.println("Room " + number + " available");
        status = AVAILABLE;
        return true;
    }

    public int status()
    {
        return status;
    }
}
```

The class definition includes all the attributes and methods defined in its own class diagram component. It also includes all the methods defined in the *LibraryItem* interface.

 Create a **Room.java** file that contains the above class definition.

The *Book*, *Periodical* and *Video* classes all implement the *LibraryItem* interface as well as extending the *Resource* class. This is done in the class definition by first identifying the parent class and then specifying the interface(s). It is possible for a class to inherit from one parent class AND implement from any number of interfaces.

```
public class className extends parentName
    implements interface1Name, interface2Name, interfaceNName
{
    //attributes go here

    //own methods go here

    //inherited methods to be overridden go here

    //interface (implemented) methods go here
}
```

Keeping this in mind, the class definition for the *Book* class would look like...

```
public class Book extends Resource implements LibraryItem
{
    protected String author;
    protected String ISBN;

    //constructors
    Book()
    {
        super("none", 0, "none", "none");
        System.out.println("Creating a book");
        author = "none";
        ISBN = "none";
    }

    Book(String name, int pages, String date, String pub,
        String author, String ISBN)
    {
        super(name, pages, date, pub);
        System.out.println("Creating a book");
        this.author = author;
        this.ISBN = ISBN;
    }

    //inherited methods
    public void getInfo()
    {
        super.getInfo();
        System.out.println("Written by " + author);
        System.out.println("ISBN: " + ISBN);
    }
}
```

```

        public void setInfo(String author, String id)
        {
            this.author = author;
            ISBN = id;
        }

        //implemented methods
        public boolean addItem()
        {
            System.out.println("Adding book " + name + " to system");
            return true;
        }


        public boolean editItem()
        {
            System.out.println("Editing book " + name);
            return true;
        }


        public boolean signOut()
        {
            System.out.println("Book " + name + " borrowed");
            status = BOOKED;
            return true;
        }


        public boolean signIn()
        {
            System.out.println("Book " + name + " available");
            status = AVAILABLE;
            return true;
        }

        public int status()
        {
            return status;
        }
    }

```

 Create a **Book.java** file that contains the above class definition.

 Based on the class definition diagram on page 133, create a **Resource.java** file that contains the abstract class definition. Include a default constructor that sets all String attributes to “none” and all numeric attributes to zero. Include an alternate constructor that accepts values for the name, number of pages, date, and publisher attributes. Set the status to AVAILABLE. Make sure you put code in the *getInfo()* method to display the current values of the name, number of pages, date, publisher and status attributes. The *editInfo()* methods can just say something like “Editing *name*” where *name* is the attribute value.

 Create a **Library.java** file that contains the driver class. In its constructor create an object of type *Book* using the alternate constructor. Once the object is created, call the *getInfo()* method to display its information. It must also create and display the information for an object of type *Room*. Compile and test your application.

Exercise – Library System

Using the class definition diagram on page 51, complete the library system as defined. You must demonstrate that objects of each of the five classes *Book*, *Periodical*, *Video*, *Room* and *Equipment* are created and hold both common and unique information.

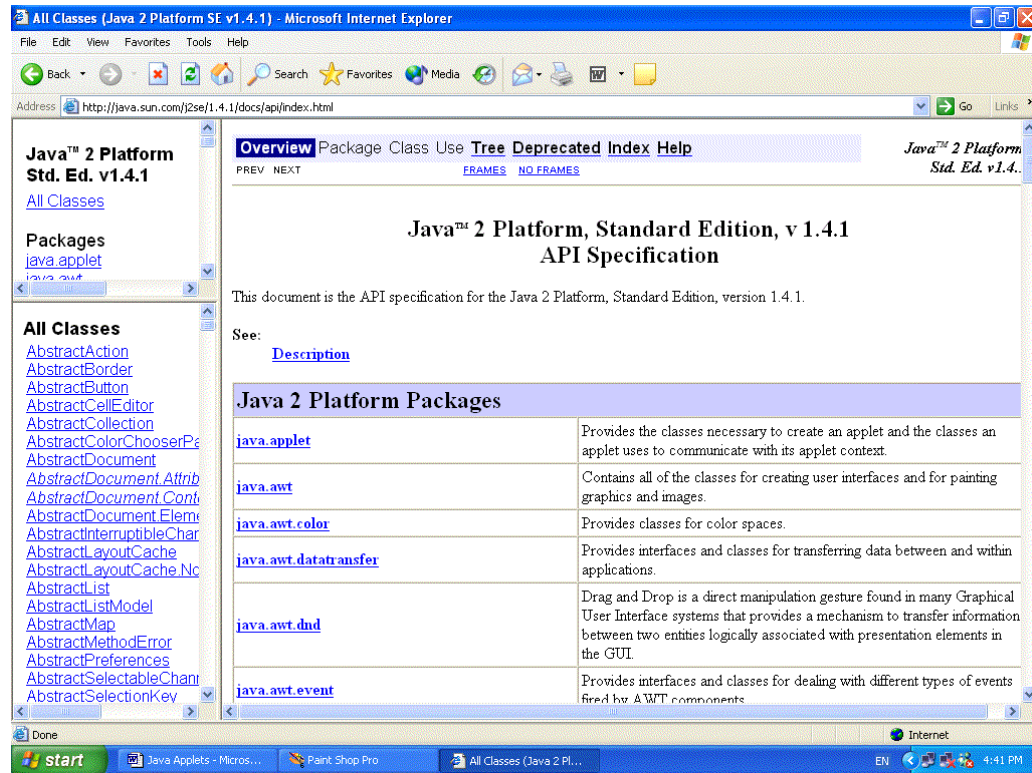
Java Packages

The Java class library is made up of a set of predefined components. These components are grouped into what are called **packages**. The thirteen main packages found in the current version of the Java Software Development Kit include these more commonly used packages...

Package	Groups together...	Such as...
java.lang	General language classes	Boolean String System Thread
java.util	General classes	Date Hashtable Stack
java.awt	GUI toolkit classes	CheckBox Button Label TextArea
java.io	Input/output classes	File InputStream OutputStream
java.net	Network classes	InetAddress Socket URL
java.text	Text classes	NumberFormat Resource

The full list of Java packages is available in the **API Specification** of the on-line documentation.

☞ Open up your web browser and locate the documentation for the Java packages. Your screen should look something like...



The full set of packages is listed in the frame in the top left corner. Scroll down through the list. Notice that many of these packages, such as java.awt, have packages that exist within the main package. These more specialized packages consist of the class definitions that provide specific functionality for Java applications and applets. Click on any of the list items to go directly to the documents for the selected package.

The large frame on the right has a brief description of each of the Java packages. This can be useful when trying to figure out which package contains the class you need.

The frame in the bottom left corner normally displays all the individual classes contained in all of the Java packages. This is the quickest way to the specific documentation for a particular class.

☞ Add this web page to your browser favourites ... you will use this set of references over and over again while writing your Java programs.

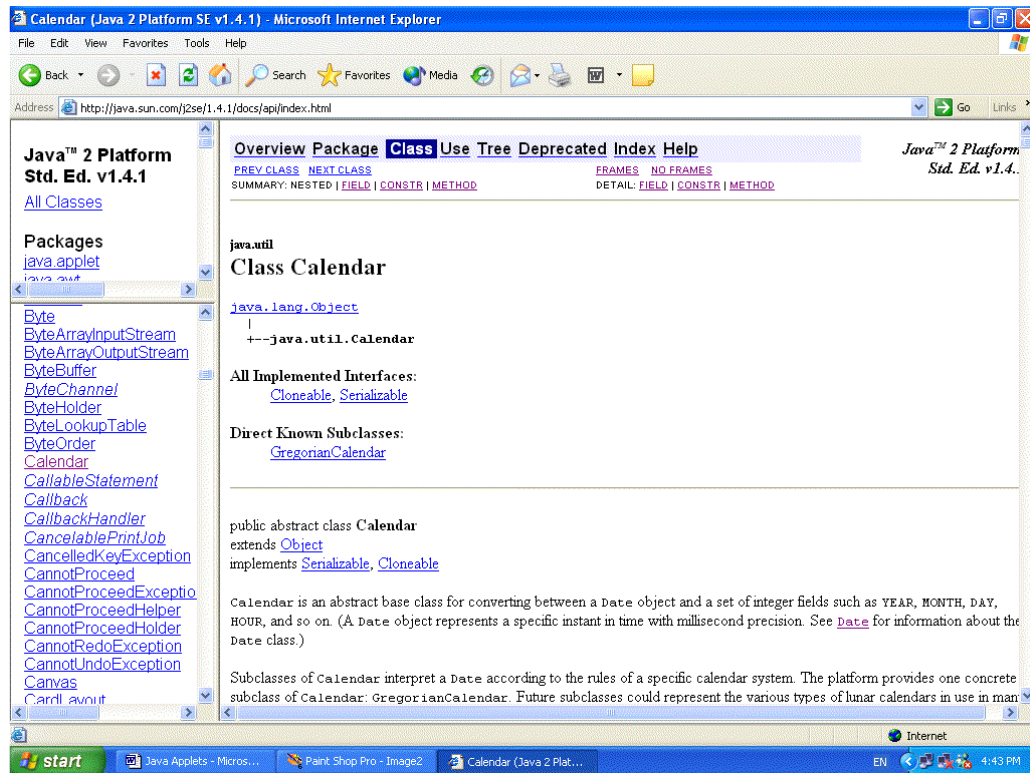
Class Documentation

Each class defined in the set of Java packages is documented in a similar fashion. Each contains ...

- A diagram showing where the selected class fits into the Java hierarchy of classes

- A basic description of the class
- A list of the attributes defined within the class
- A summary of the available constructors for the class
- A list of the available methods for the class

☞ Click on the Calendar class in the bottom left frame. You should see the documentation page for the Calendar class ...



Looking at the frame on the right side, you can see that the Calendar class belongs to the java.util package (which itself is part of the java.lang.Object package). Scrolling down you can see a description of the class. This may also include any special notes that you should be aware of when using this class in your Java programs. Always do a quick read of this description to make sure there is nothing out of the ordinary about the class or its use. This description will also include links to any related classes.

Field Summary

As you scroll past the class description you'll see a **Field Summary**. This includes all the constants and the attributes associated with the class. Constants are defined in capital letters and have a static data type. Attributes are defined in lower case and have a specified level of accessibility (public, protected or private).

Methods Summary

The screenshot shows a web browser window with the address bar displaying `http://java.sun.com/j2se/1.4.1/docs/api/index.html`. The left sidebar contains a navigation menu for the Java™ 2 Platform Std. Ed. v1.4.1, with 'All Classes' and 'Packages' sections. The 'Packages' section lists various Java packages, including `java.applet`, `java.awt`, `Byte`, `ByteArrayInputStream`, `ByteArrayOutputStream`, `ByteBuffer`, `ByteChannel`, `ByteHolder`, `ByteLookupTable`, `ByteOrder`, `Calendar`, `CallableStatement`, `Callback`, `CallbackHandler`, `CancellablePrintJob`, `CancelledKeyException`, `CannotProceed`, `CannotProceedException`, `CannotProceedHelper`, `CannotProceedHolder`, `CannotRedoException`, `CannotUndoException`, `Canvas`, and `CardLayout`. The main content area displays the `Calendar` class documentation, listing various methods and their descriptions. The methods include `equals(Object obj)`, `get(int field)`, `getActualMaximum(int field)`, `getActualMinimum(int field)`, `getAvailableLocales()`, `getFirstDayOfWeek()`, `getGreatestMinimum(int field)`, `getInstance()`, `getInstance(Locale aLocale)`, `getInstance(TimeZone zone)`, `getInstance(TimeZone zone, Locale aLocale)`, `getLeastMaximum(int field)`, and `getMaximum(int field)`. The browser's status bar at the bottom shows the taskbar with icons for 'start', 'Java Applets - Micro...', 'Paint Shop Pro - Image5', 'Calendar (Java 2 Plat...', 'EN', and a clock showing '4:44 PM'.

Calendar (Java 2 Platform SE v1.4.1) - Microsoft Internet Explorer

File Edit View Favorites Tools Help

Back Forward Stop Reload Home Search Favorites Media Mail Print TV Links

Address http://java.sun.com/j2se/1.4.1/docs/api/index.html Go Links

Java™ 2 Platform Std. Ed. v1.4.1

[All Classes](#)

Packages

[java.applet](#)

[java.awt](#)

[Byte](#)

[ByteArrayInputStream](#)

[ByteArrayOutputStream](#)

[ByteBuffer](#)

[ByteChannel](#)

[ByteHolder](#)

[ByteLookupTable](#)

[ByteOrder](#)

[Calendar](#)

[CallableStatement](#)

[Callback](#)

[CallbackHandler](#)

[CancellablePrintJob](#)

[CancelledKeyException](#)

[CannotProceed](#)

[CannotProceedException](#)

[CannotProceedHelper](#)

[CannotProceedHolder](#)

[CannotRedoException](#)

[CannotUndoException](#)

[Canvas](#)

[CardLayout](#)

`boolean` [equals](#)([Object](#) obj)
Compares this calendar to the specified object.

`int` [get](#)(`int` field)
Gets the value for a given time field.

`int` [getActualMaximum](#)(`int` field)
Return the maximum value that this field could have, given the current date.

`int` [getActualMinimum](#)(`int` field)
Return the minimum value that this field could have, given the current date.

`static Locale[]` [getAvailableLocales](#)()
Gets the list of locales for which Calendars are installed.

`int` [getFirstDayOfWeek](#)()
Gets what the first day of the week is; e.g., Sunday in US, Monday in France.

`abstract int` [getGreatestMinimum](#)(`int` field)
Gets the highest minimum value for the given field if varies.

`static Calendar` [getInstance](#)()
Gets a calendar using the default time zone and locale.

`static Calendar` [getInstance](#)([Locale](#) aLocale)
Gets a calendar using the default time zone and specified locale.

`static Calendar` [getInstance](#)([TimeZone](#) zone)
Gets a calendar using the specified time zone and default locale.

`static Calendar` [getInstance](#)([TimeZone](#) zone, [Locale](#) aLocale)
Gets a calendar with the specified time zone and locale.

`abstract int` [getLeastMaximum](#)(`int` field)
Gets the lowest maximum value for the given field if varies.

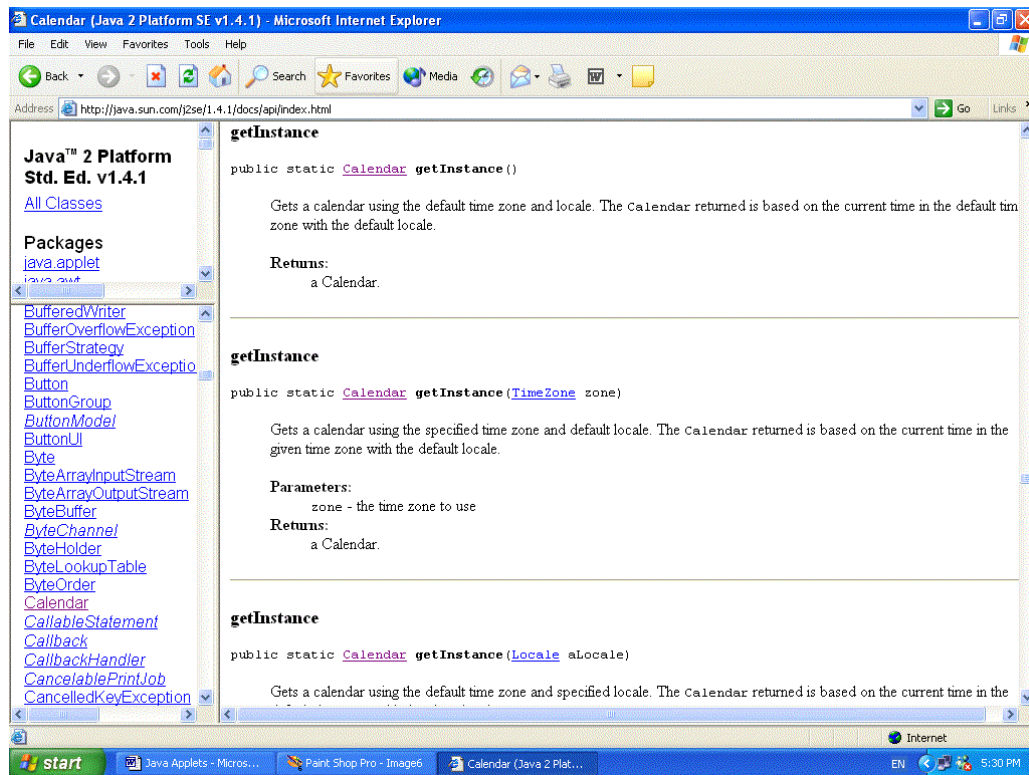
`abstract int` [getMaximum](#)(`int` field)

Internet

start Java Applets - Micro... Paint Shop Pro - Image5 Calendar (Java 2 Plat... EN 4:44 PM

Details

Finally you'll find a **Details** Section that includes a detailed description of each field, constructor and method associated with the class. You can scroll through the list or jump directly to any of these entries by clicking on the desired element in the Field, Constructor or Methods Summary sections. The Method Detail section for the Calendar class includes a prototype for each method, a brief description of its function and a description of each input parameter and the returned value...



Importing a Package

To use a built-in Java class that is defined in a Java package you must **import** the package and/or the specific class. You do this using an **import** statement. This is similar to the pre-processor directive *#include* that you are familiar with in C. The Java statement looks like...

```
import javaPackageName.*;
```

OR

```
import javaPackageName.specificClass;
```

The asterisk (*) in the first import statement is a wildcard. This means that all the classes included in the specified package are available for use in the program. For example, the statement...

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

specifies to the compiler that all the classes in the *java.awt* package are to be imported for use in this application. The second import statement indicates that only one class from the specified Java class is to be imported for use in the application. For example, the statement ...

```
import java.awt.Graphics;
```

tells the compiler that the *Graphics* class, which is part of the *java.awt* package is to be imported for use in the application.

All import statements are placed at the very beginning of *.java* file. They are always declared outside the actual class definition.

☞ Create a file called *Today.java*. Enter the code for the *main()* method that creates one instance of the **Today** class. Add a constructor for the **Today** class. For now leave it empty. Save your file in its own folder.

☞ Add a String array called **days[]** to the constructor. As part of the declaration, populate the array with seven strings ... “*Sunday*”, “*Monday*”, through to “*Saturday*”.

☞ Add a String array called **months[]** to the constructor. As part of the declaration, populate the array with twelve strings ... “*January*”, “*February*”, through to “*December*”.

☞ Import the *java.util.Calendar* class. Remember that all import statements are placed at the very beginning of the file.

☞ Add the following instruction to the constructor.

```
Calendar now = Calendar.getInstance();
```

This method of creating an instance of an object is a little different than what you’ve seen so far. Look at the **Methods Summary** documentation for the *Calendar* class. There are four different *getInstance()* methods defined for the class, each of which returns an object of type *Calendar*. The first method has an empty parameter list. The other three require input values identifying time zones and/or locales. Reading through the four descriptions, you can see that the method with no input values returns a calendar based on the default (or local) time zone. This is all you need for this application.

Again referring to the *Calendar* class documentation, notice that the selected *getInstance()* method returns a static variable of type *Calendar*. Because of this, the instruction uses the class name with the method ... *Calendar.getInstance()*. The returned value is assigned to the variable ‘*now*’ which is also of type *Calendar*.

☞ Add the following instruction to the end of your constructor...

```
System.out.print("\nCalendar says today is " + now.toString());
```

Scroll down through the *Calendar* class documentation until you find the *toString()* method. The description indicates that this method is designed for debugging purposes only. Eventually you will replace it with more appropriate methods, but for now you’ll use it to make sure the program is working properly. Notice that it is used with the *Calendar* object ‘*now*’ that you just created to print out the current calendar information.

☞ Save your program. It should look something like...


```


import java.util.Calendar;


public class Today
{
    Today()
    {
        String days[] = {"Sunday", "Monday", "Tuesday", "Wednesday",
                        "Thursday", "Friday", "Saturday"};
        String months[] = {"Jan", "Feb", "Mar", "Apr", "May", "Jun",
                          "Jul", "Aug", "Sep", "Oct", "Nov", "Dec"};

        Calendar now = Calendar.getInstance();
        System.out.print("\nCalendar says today is " + now.toString());
    }

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


```


 Make any required corrections to your code. Compile and run your application. Make a note here of how the output is displayed using the *toString()* method.

 Add the following output line at the end of the constructor...

```
System.out.println("\nThe year is " + now.get(now.YEAR));
```

The *get()* method is used with an object of class *Calendar*. *YEAR* is one of the fields defined for the class and is used with the ‘*now*’ object.

 Compile and run your program.

 Add statements to print the current month, day of the month and day of the week. Use the class fields with the *get()* method to access these values from the ‘*now*’ object you created. Make sure the month and day of the week print as text (i.e. “February”) and not numeric values. Compile and run your program.

Exercise – Number Formats

Write a Java application that uses the built-in **NumberFormat** class to display 1234.56789 as a currency value (\$1234.57), a rounded off integer value (1235) and as a number with 2 decimal values (1234.57). Hint: create separate classes for each numeric type. Set the number format in the constructor (look at the *getInstance()* methods) and format the value during the output statement.

Java Applets

The Sun Java documentation defines an applet as a...

"...small program that is intended not to be run on its own, but rather to be embedded inside another application."

More often than not Java applets run within the context of a web page. The HTML code creates and displays a specified page that, as part of its functionality, invokes an applet. There are however a few limitations on what a Java applet can do. These restrictions were initially put in place to ensure web users that their systems were secure ... that running an applet would not alter their system. This section, taken from

<http://java.sun.com/docs/books/tutorial/applet/practical/security.html>

identifies the current restrictions placed on Java applets...

- Applets cannot load libraries or define native methods.
- An applet cannot ordinarily read or write files on the host that is executing it.
- An applet cannot make network connections except to the host that it came from.
- An applet cannot start any program on the host that is executing it.
- An applet cannot read certain system properties.
- Windows that an applet brings up look different than windows that an application brings up.

There are work-arounds to some of these restrictions ... namely linking with server-side applications to accomplish what the applet cannot. Also, as time passes the browsers seem to be lifting some of these security-related limitations. Check the web page for a full description of these limitations and the suggested work-arounds.

Writing a Java Applet

The structure of a Java applet is different from a Java application. Unlike the application which requires a driver class and a *main()* method, a Java applet uses a set of five methods to initialize, display and process its tasks...

<i>init()</i>	executes once when applet begins
<i>start()</i>	executes when browser visits page containing applet
<i>paint()</i>	executes when applet appears, is resized or moved
<i>stop()</i>	executes when browser leaves page containing applet
<i>destroy()</i>	executes when applet ends or when browser is closed

✎ Create a new file called **basicMethods.java**. Save it in its own folder.

✎ Add the following import statements to your file...

```
import java.applet.Applet;  
import java.awt.Graphics;  
import java.awt.Color;
```

The first import statement is required for all applets. This allows your applet to inherit all the built-in functionality of the *java.applet* package and more specifically the *Applet* class.

The remaining two import statements provide access to some of the built-in user interface elements. The *Graphics* class allows you to draw on the screen (as a device context) and the *Color* class allows you to work with colours other than the default colour. Both of the classes are part of the *java.awt* package. **AWT** stands for **A**bstract **W**indowing **T**oolkit which is Java's name for its collection of classes required for user interfaces.

✎ Now add a public class called **basicMethods** that extends the *Applet* class. Leave the class empty for now. Note: If you called your file something other than *basicMethods.java*, you must name your class the same as your file name.

```
public class basicMethods extends Applet  
{  
  
}
```

✎ Recall from the exercises on inheritance that, because of the '**extends Applet**' instruction, you now have access to all the fields and methods defined in the *Applet* class. Use the on-line documentation to review what fields and methods are available.

✎ Now it's time to add the five basic methods required for an applet. These methods are specified as part of the main applet class, in this case *basicMethods*. The first method is *init()*...


```
public void init()  
{  
    System.out.println("In method init().");  
}
```

This is where any initialization code should be placed. Once you start placing elements like buttons, text and labels on the screen this is the method where they get defined and added. In this case, all the method does is display that it has been executed. Remember this method gets executed only once when the applet begins.

✎ Now add the *start()* method...

```
public void start()  
{  
    System.out.println("In method start().");  
}
```


This method should contain the code for any initialization that is required everytime the browser visits the page. In this case the method simply displays that has been executed.

 Add the third method *paint()* ...

```
public void paint(Graphics g)
{
    System.out.println("In method paint().");
    g.setColor(Color.blue);
    g.drawString("some text", 50, 100);
}
```

Notice that this method requires a *Graphics* object to be passed to it. This is your connection to the device context for the screen. Anytime you want to display something in the applet window it gets “painted” to the screen in the *paint()* method. In this case it displays that it has been executed to the DOS window and draws some blue text on the screen 50 pixels to the right and 100 pixels down from the top left corner of the screen.

 Now add the last two methods *stop()* and *destroy()*...

```
public void stop()
{
    System.out.println("In method stop().");
}

public void destroy()
{
    System.out.println("In method destroy().");
}
```

These are the methods where you do any clean up that may be required. The *stop()* method gets executed everytime the browser leaves the page that contains the applet and the *destroy()* method gets executed everytime the applet ends or the browser is closed.

The complete file for your applet should now look like ...

```
import java.applet.Applet;
import java.awt.Graphics;
import java.awt.Color;

public class basicMethods extends Applet
{
    public void init()
    {
        System.out.println("In method init().");
    }

    public void start()
    {
        System.out.println("In method start().");
    }
}
```


```

public void paint(Graphics g)
{
    System.out.println("In method paint().");
    g.setColor(Color.blue);
    g.drawString("some text", 50, 100);
}

public void stop()
{
    System.out.println("In method stop().");
}


public void destroy()
{
    System.out.println("In method destroy().");
}
}

```

 Make any required changes and compile your applet. The compile process is the same for both applets and applications.

Invoking Java Applets

The process of running a Java applet is different that running a Java application. First you must create an HTML document that invokes the applet.

 Create a new file called **basicMethods.html** that contains the following HTML code. Save it in the same folder as your `basicMethods.java` file.

```

<html>
<applet code="basicMethods.class" height=400 width=400></applet>
</html>

```

The code attribute in the `<applet>` tag references a source file that is not part of the actual HTML page. In this case it tells the browser to look for the compiled *basicMethods.class* file in the same folder as the source HTML document. If the *.class* file is located in a different folder you must include the **codebase** attribute that specified the URL of the required folder. If, for example, the *.class* file is in a sub-folder called ‘applets’ the HTML instruction must be changed to...

```

<applet code="basicMethods.class" codebase="applets" height=400
width=400></applet>

```

You can, if required, specify a full URL path for the codebase attribute...

```

<applet code="basicMethods.class"
codebase="http://www.conestogac.on.ca/~nnelson/oop/classes" height=400
width=400></applet>

```

The width and height attributes specify the size (in pixels) of the screen in which your applet will run.

Running Java Applets

Now that you have a compiled Java applet and an HTML file that invokes it, you are ready to run the applet. There are two ways you can run your applet...


- using AppletViewer
- in any browser

Using AppletViewer

The `appletviewer` command allows you to run applets outside of the context of a browser. The process is executed from the DOS prompt...

```
appletviewer fileName.html
```

Make sure that the file you specify is the HTML file and not the applet itself. You must include the extension for the file.


 Run your applet using AppletViewer...

```
appletviewer basicMethods.html
```

All you should see is the blue text drawn on the screen. The *System.output* instructions always output to the DOS screen so you won't see them until you return to the DOS prompt. Make a note of the order that the applet methods are executed.

Using a Browser

You can also run your applet inside your browser although this method is often slower than using AppletViewer.

 Start your browser. Select the **File/Open** options from the browser menu. Browse to find your HTML document. Click **OK** and your applet should start up in the browser. You should see the same blue text drawn on the screen. Once again the *System.output* instructions will show up on the DOS screen.

Graphics Class

In the *paint()* method you used in the last example, you encountered the *Graphics* class. Part of the Abstract Windowing Toolkit (*java.awt*) package, it is an abstract class that provides the fields and methods you need to draw various components on various devices, including offscreen. Anytime you want to draw text, geometric shapes or display images in your applet you'll be using the *Graphics* class.

Drawing Lines

The *drawLines()* method is used to draw straight lines. It gets executed in the *paint()* method and its format is...

```
g.drawLine(x_start, y_start, x_end, y_end);
```

where (*x_{start}*, *y_{start}*) defines the line's starting point and (*x_{end}*, *y_{end}*) defines the line's ending point. Both points are defined in pixels and are relative to the top left corner of the screen. For example...

```
g.drawLine(10, 20, 100, 125);
```

draws a straight line from a point 10 pixels from the left edge and 20 pixels from the top, to a point 100 pixels from the left edge and 125 pixels from the top.

Drawing Rectangles

You can draw three different types of rectangles using the methods in the Graphics class...

- outlined rectangles
- filled rectangles
- 3 dimensional rectangles

The *drawRect()* method is used to draw an outlined rectangle. It gets executed in the *paint()* method and its format is...

```
g.drawRect(x_start, y_start, width, height);
```

where (*x_{start}*, *y_{start}*) defines the point that forms the rectangle's top left corner (in pixels and relative to the top left corner of the screen). The **width** defines how many pixels wide the rectangle is and **height** defines how many pixels high the rectangle is. For example...

```
g.drawRect(100, 200, 150, 225);
```

draws a hollow rectangle 150 pixels wide and 225 pixels high with its top left corner positioned 100 pixels from the left edge and 200 pixels from the top.

The *fillRect()* method is used to draw a filled rectangle. It gets executed in the *paint()* method and its format is...

```
g.fillRect(x_start, y_start, width, height);
```

where (*x_{start}*, *y_{start}*) defines the point that forms the rectangle's top left corner, **width** defines how many pixels wide the rectangle is and **height** defines how many pixels high the rectangle is. For example...

```
g.fillRect(10, 25, 15, 22);
```

draws a solid rectangle 15 pixels wide and 22 pixels high with its top left corner positioned 10 pixels from the left edge and 25 pixels from the top. Unless otherwise specified, the rectangle is filled with the default colour.

The *draw3DRect()* method is used to draw an outlined rectangle with beveled edges. It gets executed in the *paint()* method and its format is...

```
g.draw3DRect(x_start, y_start, width, height, raised);
```

where (**x_{start}**, **y_{start}**) defines the point that forms the rectangle's top left corner (in pixels and relative to the top left corner of the screen). The **width** defines how many pixels wide the rectangle is and **height** defines how many pixels high the rectangle is. **Raised** is a boolean value which if set to true, displays a raised or beveled edge on the rectangle. If set to true, the rectangle looks like a normal hollow rectangle. For example...

```
g.draw3DRect(50, 20, 100, 150, true);
```

draws a hollow rectangle 100 pixels wide and 150 pixels high with its top left corner positioned 50 pixels from the left edge and 20 pixels from the top. The edge will appear raised.

Similarly, the *fill3DRect()* method is used to draw a solid rectangle with beveled edges. It gets executed in the *paint()* method and its format is...

```
g.fill3DRect(x_start, y_start, width, height, raised);
```

Its parameters are the same as for the *draw3DRect()* method. . For example...

```
g.fill3DRect(125, 200, 75, 90, false);
```

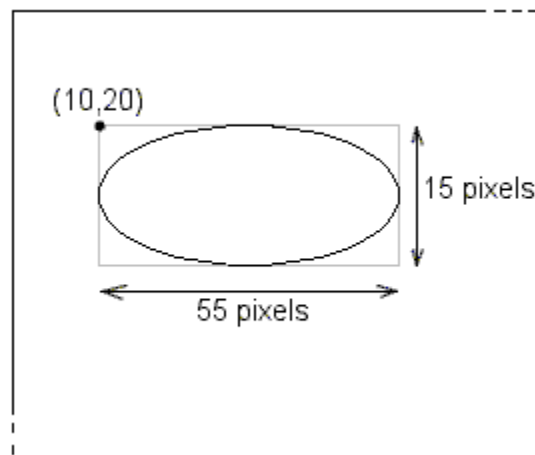
draws a solid rectangle 75 pixels wide and 90 pixels high with its top left corner positioned 125 pixels from the left edge and 200 pixels from the top. The edge will not appear raised.

Drawing Circles and Ovals

You can draw two different types of ovals or circles using the methods in the *Graphics* class...

- outlined ovals (or circles)
- filled ovals (or circles)

An oval is defined in terms of the rectangle that encloses it ...



In this case the oval is 55 pixels wide and 15 pixels high ... the same as the defining rectangle. The top left corner of the rectangle is positioned at point (10, 20).

The *drawOval()* method is used to draw an outlined oval or circle. It gets executed in the *paint()* method and its format is...

```
g.drawOval(x_start, y_start, width, height);
```

where (**x_start**, **y_start**) defines the point that forms the top left corner of the defining rectangle. **width** defines the rectangle width and **height** defines the rectangle height. Make the width and height equal for a circle. To draw the oval shown in the above diagram the instruction is...

```
g.drawOval(10, 20, 55, 15);
```

The *fillOval()* method is used to draw an solid oval or circle. It gets executed in the *paint()* method and its format is...

```
g.fillOval(x_start, y_start, width, height);
```

where (**x_start**, **y_start**) defines the point that forms the top left corner of the defining rectangle. The **width** the rectangle width and **height** defines the rectangle height. For example...

```
g.fillOval(110, 25, 120, 45);
```

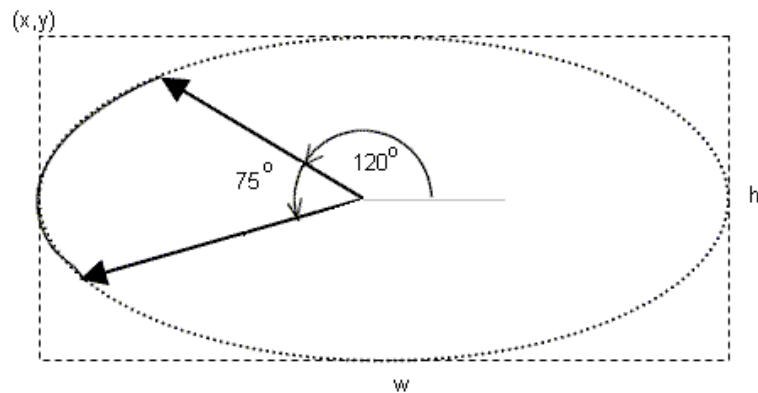
draws a solid oval in the default colour. The top left corner of the rectangle that contains the oval is positioned at (110, 25). The oval is 120 pixels wide and 45 pixels high.

Drawing Arcs

You can draw two different types of arcs using the methods in the *Graphics* class...

- outlined arcs
- filled arcs

An arc is defined in terms of an oval that fits inside a specified rectangle ...



The rectangle is defined by the point (x,y) , the width w pixels and the height h pixels. The oval sits inside this rectangle. The arc is the part of this oval that starts at an angle 120 degrees from the x axis and sweeps 75 degrees counter clockwise.

The *drawArc()* method is used to draw an arc. It gets executed in the *paint()* method and its format is...

```
g.drawArc(x_start, y_start, width, height, start, sweep);
```

where **(x_start, y_start)** defines the point that forms the top left corner of the defining rectangle. **width** defines the rectangle width and **height** defines the rectangle height. **start** defines the starting point along the enclosed oval. It is actually the number of degrees from the positive x axis. **sweep** is the number of degrees swept out by the arc from the starting point. If the sweep is counter clockwise the defined angle must be positive. If the sweep is clockwise the defined angle must be negative. To draw the arc shown in the above diagram the instruction could be either ...

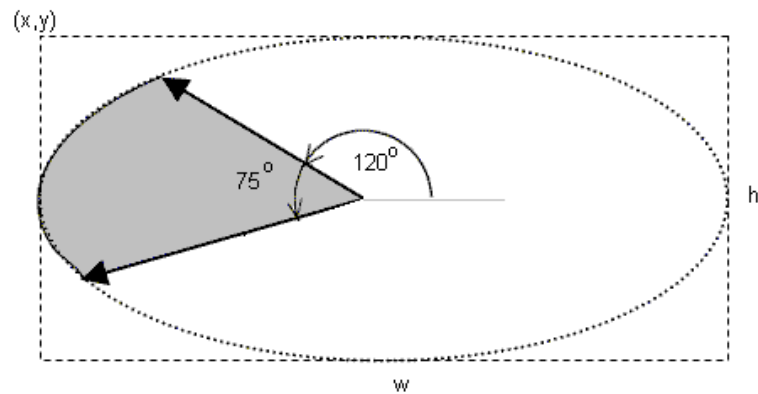
```
g.drawArc(10, 25, 50, 30, 120, 75);
```

where the arc starts 120° from the positive x axis and sweeps 75° counter clockwise, or...

```
g.drawArc(10, 25, 50, 30, 195, -75);
```

where the arc starts 195° from the positive x axis and sweeps 75° clockwise.

You can also draw a filled arc ...



where the solid area is from the center point of the oval, along the defined arc and back to the center point. The instruction to draw this solid arc could be either...

```
g.fillArc(10, 25, 50, 30, 120, 75);
```

where the arc starts 120° from the positive x axis and sweeps 75° counter clockwise, or...

```
g.fillArc(10, 25, 50, 30, 195, -75);
```

where the arc starts 195° from the positive x axis and sweeps 75° clockwise.

Defining Colours

Unless otherwise specified, all the shapes you draw on the screen are painted in the default colour... usually black. You can change the colour for any shape by using the *setColor()* method...

```
g.setColor(colour);
```

where *colour* is the desired colour. There are two ways you can define your colour. The first is to use one of the fields defined by the *Color* class. For example, to colour your shape red you would include this instruction before drawing the shape...

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

There are thirteen colours available ...

Color.black	Color.cyan	Color.gray	Color.lightGray
Color.orange	Color.red	Color.yellow	Color.blue
Color.darkGray	Color.green	Color.magenta	Color.pink
Color.white			

You can also define your own colour based on its red, green and blue components. To do this you have to create a new instance of the *Color* class...


```
Color c = new Color(redValue, greenValue, blueValue);
```

where *redValue*, *greenValue* and *blueValue* are integer values somewhere between 0 and 255. The closer the value is to 255 the more intense the color component is. You can then specify this new colour *c* as your painting colour...

```
g.setColor(c);
```


You can also define your colour right in the *setColor()* method parameter list...

```
g.setColor(new Color(206, 239, 189));
```

This sets the colour to a light green.

To use the *Color* fields or to create a new colour of your own you must import the **color** class...

```
import java.awt.color;
```

 Modify the *paint()* method of your applet so that it draws a red line that runs from the top left corner to the bottom right corner, a large blue outlined circle, a small purple solid square and a solid orange arc that sweeps 45° clockwise from the negative y axis. Compile and test your applet.

Fonts

Text is treated much the same way as the geometric shapes you just drew. It is painted onto the screen in the *paint()* method. Its characteristics such as type and style are defined using the **font** class included in the Abstract Windowing Toolkit (*java.awt*).

Defining Fonts

Unless otherwise specified all the text you draw on the screen is painted in the default font... usually black, 10 point Arial. You can change the font for any text by using the *setFont()* method...

```
g.setFont(font);
```

where *font* is an object of class **Font**.

You can define a desired font by creating a new instance of the **Font** class...

```
Font f = new Font("fontFace", fontStyle, fontSize);
```

where *fontFace* specifies the desired font, *fontStyle* defines whether the text is plain, bold and/or italic and *fontSize* defines the point size for the text. You can then specify this new font *f* as the font for your text using the *setFont()* method ...

```
g.setFont(f);
```

There are five fonts available in Java, each of which corresponds to a font you are familiar with from other PC applications...

<u>Java</u>	<u>PC Platform</u>
Courier	Courier New
Dialog	MS Sans Serif
Helvetica	Arial
TimesRoman	Times New Roman
Symbol	WingDings

There are also four font style fields defined in the **font** class...

<u>Style</u>	<u>Font Field</u>
Plain	Font.PLAIN
Bold	Font.BOLD
Italic	Font.ITALIC
Bold Italic	Font.BOLD + Font.ITALIC

Say for example you want your text displayed in 14 point MS Sans Serif italics. To specify the font you would include the following instructions...

```
Font f = new Font("Dialog", Font.ITALIC, 14);
g.setFont(f);
```

This can also be accomplished in one instruction...

```
g.setFont(new Font("Dialog", Font.ITALIC, 14));
```

Displaying Text

Once you have specified your font, you can display the text using the *drawString()* method...

```
g.drawString(string, x_start, y_start );
```

where *string* is the text to be display and (*x_{start}*, *y_{start}*) defines the starting point on the screen. For example, to display the text "This is the string" on the screen at position (42, 25) the instruction is...

```
g.drawString("This is the string", 42, 25);
```

Font Related Methods

There are a few other methods you can use when working with fonts. You can use the `getFont()` method to determine the current font...

```
Font currentFont = g.getFont();
```

This method returns an object of class **Font**. Once you know the current font you can use the `getName()` method to display the actual font name at position (x,y) on the screen...

```
g.drawString(currentFont.getName(), x, y);
```

You can also use information about the font and about the display area to position your text relative to your applet window. This is particularly useful if you always want your text centered, even if the user resizes the applet window. To do this you must first import two classes... **FontMetrics** and **Dimension** both a part of the *java.awt* class.

```
import java.awt.FontMetrics;  
import java.awt.Dimension;
```

The `getFontMetrics()` method returns an object of type `FontMetrics`. This object contains information about the font characters ... the height, spacing, leading, etc. The instruction is...

```
FontMetrics fm = getFontMetrics(font);
```

where *font* is an object of type **Font** either created by you using the *new Font()* technique or returned from the `getFont()` method. The object *fm* stores the information about the font that you'll use to position your text. First you'll need to know how wide the string you want to display will be. To do this you use the `stringWidth()` method...

```
int strWidth = fm.stringWidth(str);
```

where *str* is the string you want displayed. The returned value *strWidth* is an integer value. Next you need to know the height of the string. The `getHeight()` method returns this integer value...

```
int strHeight = fm.getHeight();
```

Now that you know the size of your text you need to calculate the size of your applet window. This is done using the `getSize()` method...


```
Dimension s = getSize();
```

This method returns an object of type **Dimension** that contains information about the applet window including its width and height. You can now use these dimensions to calculate a starting x and y position for your centered text...

```
int windowWidth = s.width;
int windowHeight = s.height;
int xpos = (windowWidth - strWidth) / 2;
int ypos = (windowHeight - strHeight) / 2;
```

The x position is calculated by subtracting the width of the string from the window width. The difference is divided by two. Say, for example, the window width is 100 pixels and the string width is 32 pixels. The difference between the two widths is 68 pixels. Dividing this by two gives a starting x position at 34 pixels. Since the string is 32 pixels wide it will end at pixel 66 leaving 34 pixels to its right. The string is centered between the left and right window edges.

Similarly the y position is calculated by subtracting the height of the string from the height of the applet window. This difference is divided by two. If the window is 140 pixels high, the center of the window is at 70 pixels. If the string is 20 pixels high, you want 10 of the pixels above the 70 pixel line and 10 of the pixels below the 70 pixel line. That means the y position should be at 60 pixels ... or using the calculation $(140 - 20) / 2$ which works out to the same 60 pixels.

 Modify the *paint()* method of your applet so that it displays your name and the font name in green 16 point italics. This text must be centred in the applet window. Compile and test your program. Resize your applet window to make sure the text stays centred.

Images

Java can display *.gif*, *.png* and *.jpeg* formatted images. In order to work with images you have to first import the **Image** class which is part of the *java.awt* package...

```
import java.awt.Image;
```

In order to make the image available to the applet you have to locate the image. You can do this in one of two ways. This first is to create an object of class **URL** that identifies where the image is located ...

```
URL url = new URL("fullURLpath");
```

The second method is to place the image in the same folder as the *.html* file that initiates the applet. You can then use the *getDocumentBase()* method to specify that the image is in the same folder as the base document ... the *.html* file. This method is part of the **Applet** class and returns an object of class **URL**...

```
URL url = getDocumentBase( );
```

Note: You will have to import the *java.net.** package to create the URL object.

Now that you know where the image is, you use the *getImage()* method to load the image...

```
Image img = getImage(url, "imageName");
```

This method returns an object of class **Image**.

Finally, you use the *drawImage()* method (also part of the **Applet** class) to place the selected image on the screen in a specified location...


```
g.drawImage(img, x, y, this);
```

where *x* and *y* define the top left corner of the image and **this**, called an **ImageObserver**, identifies which object is to be notified when the image is loaded ... in this case the method it is in ...*paint()*.

As with any methods, these image locating, loading and drawing methods can be combined. Say for example you want to display an image called “pic.gif” that is located in the same folder as the .html file that calls your applet, in the top left corner of the screen. You can use the following set of instructions...

```
Image img = getImage(getDocumentBase(), "pic.gif");  
g.drawImage(img, 0, 0, this);
```

Check the **Applet** class documentation for even more ways of displaying images.

 Modify the *paint()* method of your applet so that it displays an image somewhere in the applet window. Compile and retest your program.

Exercise – Centered Image

Write a Java applet that displays a small image in the centre of the applet window (even if the window is resized). The image must be framed by a coloured geometric shape (oval, circle, rectangle, or square). The size of the window, the image and the framing geometric shape must be displayed at the top of the applet. Your output should look something like this...



Java Components

Most Java applets and applications make use of standard on-screen components such as...

- labels
- buttons
- text fields
- check boxes
- radio buttons

This section takes a look at what components are available, how to add them to your program and finally how to handle the events triggered by these components.

Java Applications in a Window

Components such as those listed above are considered part of a graphical user interface (GUI). Up to this point however, your Java applications have all run in a DOS shell ... all the output is text that is written directly to the DOS screen. If your application requires a GUI you can configure your application to run in a window or a frame.


The first thing you have to do is import the **Frame** class. Because it is directly related to the graphical interface it is part of the *java.awt* package.

```
import java.awt.Frame;
```

If your application is to run in a frame it must extend this *Frame* class...

```
public class AppWindow extends Frame
{
}
```

where *AppWindow* is the name of both your class and your file (AppWindow.java).

 Create a new file called **AppWindow.java**. Enter the import instruction and set up the main class. Save your file in a new folder.

As in an application there must be a *main()* method. Its format is the same as all the previous applications you've written...

```
public static void main(String args[])
{
    AppWindow app = new AppWindow("Application in a Window");
}
```

where the code that it contains creates an instance of the class *AppWindow*. In this case however, it is important to define an object of type *AppWindow* so that you have access to the frame. In this statement the object is called '*app*'.

☞ Add the *main()* method to your file.

As with any class, the *AppWindow* class needs a constructor. Notice that when the ‘*app*’ object is created in the *main()* method, a string is passed to the constructor. This means that the *AppWindow* class needs a constructor that accepts a string in its parameter list...

```
public AppWindow(String frameTitle)
{
    super(frameTitle);
}
```

The only thing that this constructor does is pass the string it receives to its parent class ... in this case the *Frame* class. The *Frame* class displays this string in its title bar.

☞ Add the constructor to your file.

The ‘*app*’ object is created but not yet displayed. A size must be specified for the window and then it must be given the ‘show’ instruction. This is done in the *main()* method right after creating the ‘*app*’ object ...

```
app.setSize(width,height);
app.show();
```

where *width* is the desired window width in pixels and *height* is the desired window height in pixels.

☞ Add the *setSize()* and *show()* methods to your *main()* method. Don’t forget to substitute values in for the window width and height.

The final step in running an application in a window is to handle the application shut-down. This is done using a listener class called *winWatch* that implements the **WindowListener interface**. This separate class handles all windows events including activation, deactivation, opening and closing.

```
class winWatch implements WindowListener
{
    public void windowClosing(WindowEvent wEvt)
    {
        System.exit(0);
    }
    public void windowClosed(WindowEvent wEvt){}
    public void windowOpened(WindowEvent wEvt){}
    public void windowIconified(WindowEvent wEvt){}
    public void windowDeiconified(WindowEvent wEvt){}
    public void windowActivated(WindowEvent wEvt){}
    public void windowDeactivated(WindowEvent wEvt){}
}
```

Notice that only one of the methods in the class includes any code. Because you are only interested in what happens when the window is closing, you only need to provide code for that method ... specifying the *System.exit(0)* instruction in the *windowClosing()* method.

All of the other methods in the class are empty. **WindowListener** is one of the events handling interfaces provided in the *java.awt.event* package ... you can tell that it is an interface because it is implemented and not extended. Recall that interfaces are abstract, as are all the methods defined in it. Because of this, every method must be defined within any class that implements the interface. In this case they simply have an empty code block.

☞ Add the *winWatch* class to the end of your *AppWindow* class. It can be in the same file as long as it is outside the brackets enclosing the *AppWindow* class. **NOTE:** If you define the *winWatch* class in its own file you must make sure that the class is defined as public and that the file exists in the same folder as the *AppWindow* class.

The application must now be informed that there is a listener for all its window events. An object of class *winWatch* must be created. This is accomplished the same way any other object is created ... with an object declaration and the actual instantiation instruction.

```
winWatch wwatch;  
  
wwatch = new winWatch();
```

Now the application must be told to 'listen' for window events. This is done by 'adding' the window listener to the application...

```
initiator.addWindowListener(responder);
```

where *initiator* is the class/window/application that triggers the event and *responder* is the object that will respond to the event. In this particular application the initiator is the application called '*app*' and the responder is the *wwatch* object...

```
app.addWindowListener(wwatch);
```

Finally, since the window listener is part of the *java.awt.event* package it must be imported...

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

☞ Add these three instructions to the *main()* method of your *AppWindow* class. Your program is now complete and should look like...

```
import java.awt.Frame;  
import java.awt.event.*;  
  
public class AppWindow extends Frame  
{  
    public AppWindow(String frameTitle)  
    {  
        super(frameTitle);  
    }  
  
    public static void main(String args[])  
    {  
        winWatch wwatch;  
  
        AppWindow app = new AppWindow("Application in a Window");  
        app.setSize(400,300);  
    }  
}
```

```

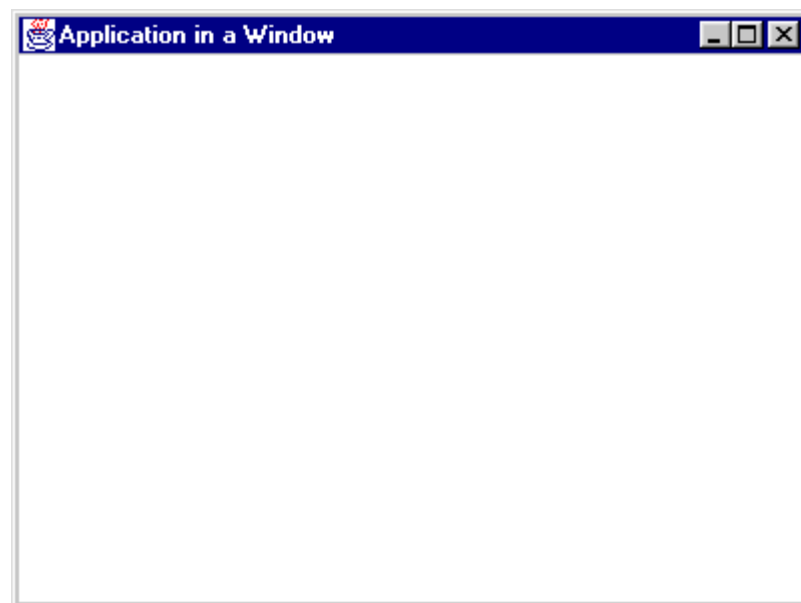
        app.show();

        wwatch = new winWatch();
        app.addWindowListener(wwatch);
    }
}

class winWatch implements WindowListener
{
    public void windowClosing(WindowEvent wEvt)
    {
        System.exit(0);
    }
    public void windowClosed(WindowEvent wEvt){}
    public void windowOpened(WindowEvent wEvt){}
    public void windowIconified(WindowEvent wEvt){}
    public void windowDeiconified(WindowEvent wEvt){}
    public void windowActivated(WindowEvent wEvt){}
    public void windowDeactivated(WindowEvent wEvt){}
}

```

☞ Compile and test your program. Your output should look something like this...



Labels

Labels are the simplest of the components that you can add to your application or applet. Because there is no interactivity associated with a label (the user simply reads what is displayed) there is very little required in terms of adding them to a program. The *Label* component is included in the *java.awt* package. Anytime you are using labels in your applet or application you must import the *Label* component by including the following ...

```
import java.awt.Label;
```

Constructors

As with any object, a label must be instantiated before it can be used in a program. There are three constructors available for a *Label* component...

```
Label()  
  
Label(String text)  
  
Label(String text, int alignment)
```

The default constructor creates a *Label* object with an empty text string. The second constructor creates a *Label* object and assigns the specified text. The third constructor creates the object, assigns the specified text to the label and defines where it is to be positioned within the frame. Constants are defined corresponding to the three possible positions ... **LEFT**, **CENTER** and **RIGHT**. These constant field values are used with the class name *Label*. Say for example you want to create a label called '*lbl*' that says "*Address*" and you want it aligned with the left edge of the window. The instruction would be...

```
Label lbl = new Label("Address", Label.LEFT);
```

Methods

The four most commonly used methods available for the *Label* class are ...

Method		Purpose
int	getAlignment()	Returns the label's current alignment
String	getText()	Returns the label's current text
void	setAlignment(int)	Sets the label's alignment (LEFT, CENTER, RIGHT)
void	setText(String)	Sets the label's text to the specified string

These methods are used with a *Label* object. For example, to change the specified text for the label '*lbl*' that was created in the last section, the instruction would be...

```
lbl.setText("This is the new label text");
```

Adding the Label

Once the label is created you must add it to the applet or application. This is normally done in the *init()* method of an applet or in the class constructor of an application that runs in a frame. The instruction is the same for both...

```
add(objectName);
```

To add the *'lbl'* created above, the instruction is...

```
add(lbl);
```

☞ Open your **AppWindow.java** file and add the following line to the *AppWindow* constructor (after the `super(frameTitle);` instruction)...

```
add (new Label("App in a Window", Label.CENTER));
```

This instruction creates an unnamed label object, centers the specified text in the application frame and adds the label to the window.

☞ Re-compile and test your program. Your output should now look something like this...



Common Code for Java Applets and Applications

It is possible to write code that will run as either an applet or as an application in a frame. To do this the file must contain three main classes...

- one that extends the **Applet** class
- a second that extends the **Frame** class
- a third for the window listener

Class Definition for the Applet Class

From the chapter on Java applets you know that the basic format of a Java applet is...

```
import java.applet.Applet;

public class master extends Applet
{
    public void init()
    {
    }

    public void start()
    {
    }

    public void stop()
    {
    }

    public void destroy()
    {
    }
}
```

 Create a new file called **master.java** that contains this applet code.

The only change you need to make to this master class is to add a *main()* method. This method gets executed when the class is run as an application. The method is the same as the one used in the ‘application in a frame’ file **AppWindow.java** ...

```
public static void main(String args[])
{
    winWatch wwatch;

    masterFrame app = new masterFrame ("master");
    app.setSize(300,300);
    app.show();

    wwatch = new winWatch();
    app.addWindowListener(wwatch);
}
```

 Add this *main()* method to the **master.java** file just before the *init()* method.

Class Definition for the Application Class

From the first section in this chapter you know that the basic code for a Java application in a frame looks like...

```
import java.awt.Frame;
import java.awt.event.*;

public class masterFrame extends Frame
{
    public masterFrame (String frameTitle)
    {
        super(frameTitle);
    }

    public static void main(String args[])
    {
        winWatch wwatch;

        master app = new master("Application in a Window");
        app.setSize(400,300);
        app.show();

        wwatch = new winWatch();
        app.addWindowListener(wwatch);
    }
}
```

The *main()* method is already a part of the class definition for the Applet class, so it will be removed from here. All that needs to be added to the class constructor is the code that gets executed when the program runs as an applet...

```
master applet = new master();
applet.init();
add ("Center", applet);
```

The first line creates a new applet object. This applet initialization code is executed and then the applet is centered in the application's frame. The complete class definition for the application class looks like...

```
import java.awt.Frame;
import java.awt.event.*;

public class masterFrame extends Frame
{
    public masterFrame (String frameTitle)
    {
        super(frameTitle);
        master applet = new master();
        applet.init();
        add ("Center", applet);
    }
}
```

☞ Add this code to the **master.java** file. Place the import instructions at the top and the complete *masterFrame* class definition at the end. Since you are adding to a file that already contains a class definition, remember to leave the **public** access modifier off the *masterFrame* class definition line...

```
class masterFrame extends Frame
{
    public masterFrame (String frameTitle)
    {
        . . .
    }
}
```

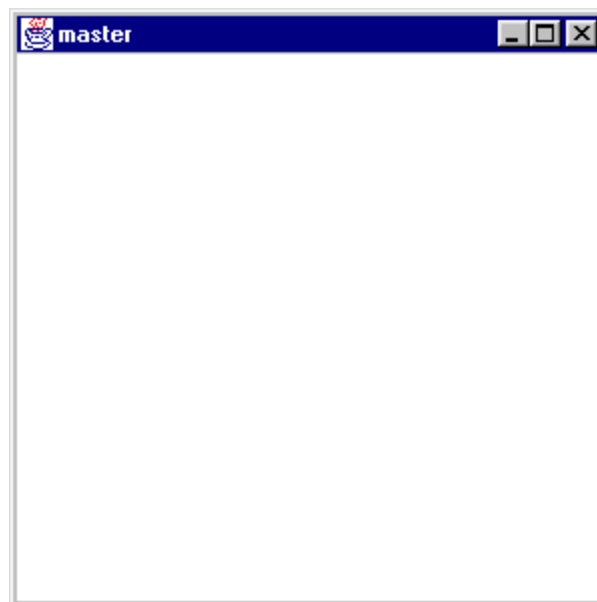
Note: You need to leave the **public** access modifier on the constructor.

Class Definition for the WindowListener Class

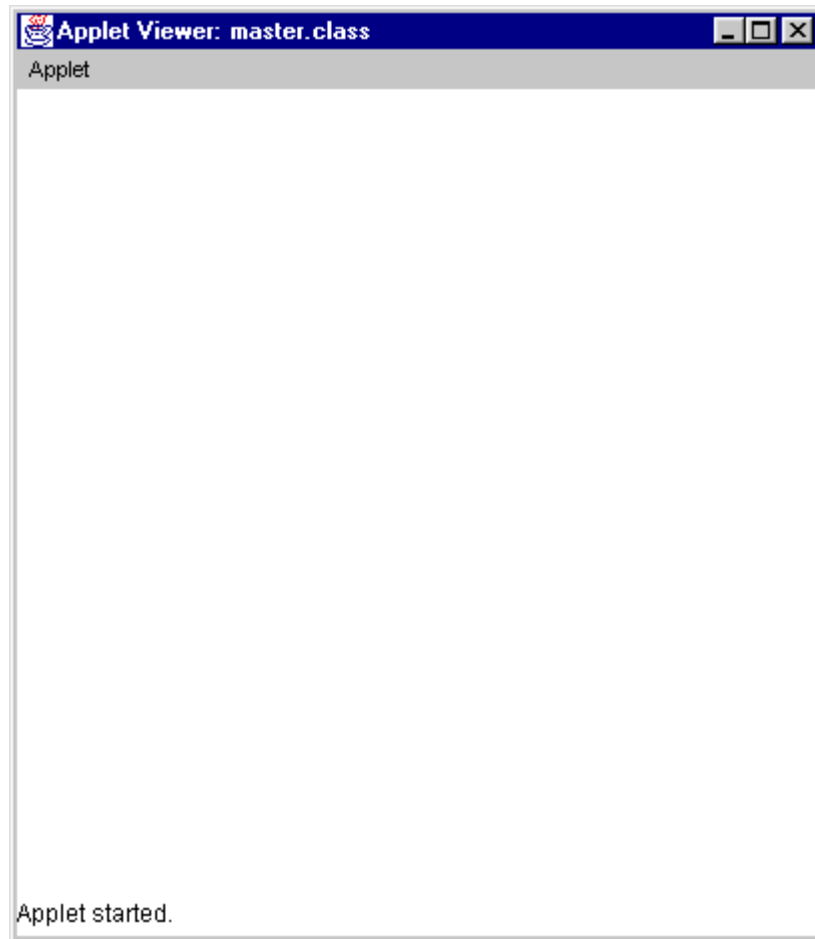
From the first section in this chapter you know that the basic code for a window listener looks like...

```
class winWatch implements WindowListener
{
    public void windowClosing(WindowEvent wEvt)
    {
        System.exit(0);
    }
    public void windowClosed(WindowEvent wEvt){}
    public void windowOpened(WindowEvent wEvt){}
    public void windowIconified(WindowEvent wEvt){}
    public void windowDeiconified(WindowEvent wEvt){}
    public void windowActivated(WindowEvent wEvt){}
    public void windowDeactivated(WindowEvent wEvt){}
}
```

☞ Add this code to the end of **master.java** file. Compile your code and run the file as an application. Your output should look like...



☞ Create an HTML file to run your applet. Run your applet using AppletViewer. Your output should look like...



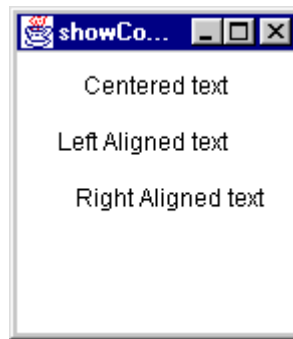
IMPORTANT NOTE:

Use this file, **master.java**, as a master file for all the Java programs you write from now on. Use a search and replace tool to replace '*master*' with the desired file name. For example, if your file is to be called **showComponents.java**, replace all occurrences of the word *master* with the word *showComponents*. Then save the file with the new file name **showComponents.java**.

☞ Create a file called **showComponents.java** from your **master.java** file. In the `init()` method of the applet class add the required code to add three labels ...

- one centered that says "Centered text"
- one left aligned that says "Left aligned text"
- one right aligned that says "Right aligned text"

☞ Compile and run your applet/application. Make sure that it runs in both environments. Your output should look like...



Notice that the output screen is fairly small here. If your output screen is too wide, your text will appear on one line. Until we take a look at screen layouts, you will have to play with your screen sizes to get output that looks the way you want it.

Buttons

Buttons are also fairly simple components that you can add to your application or applet. The `Button` component is included in the `java.awt` package. Anytime you are using buttons in your applet or application you must import the `Button` component by including the following ...

```
import java.awt.Button;
```

If you are adding more than one type of component to your applet you can use one import statement that includes all the components...

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

The asterisk (*) acts as a wild card.

Constructors

As with any object, a button must be instantiated before it can be used in a program. There are two constructors available for a *Button* component...

```
Button()
```

```
Button(String text)
```

The default constructor creates a *Button* object with no button text or label. The second constructor creates a *Button* object and assigns the specified text as the button text. To create a new **‘Continue’** button the instruction would be...

```
Button button1 = new Button("Continue");
```

Methods

The two most commonly used methods available for the `Button` class are ...

Method		Purpose
String	<code>getLabel()</code>	Returns the button's current label
void	<code>setLabel(String)</code>	Sets the button's text to the specified string

These methods are used with a *Button* object. For example, to change the specified text for the button *'button1'* that was created in the last section, the instruction would be...

```
button1.setLabel("Pause");
```

Adding the Button

Once the button is created you must add it to the applet/application. This is normally done in the *init()* method of an applet or in the class constructor of an application that runs in a frame. The instruction is the same for both...

```
add(objectName);
```

To add the *'button1'* created above, the instruction is...

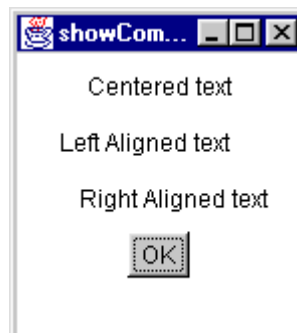
```
add(button1);
```

🔗 Open your **showComponents.java** file and add the following line to the *init()* method...

```
Button btnOK = new Button("OK");  
add(btnOK);
```

These instructions create an OK button and add it to the applet/application.

🔗 Re-compile and test your program. Your output should now look something like this...



Remember you may have to play with the applet size in order to have your display look like this.

Using the Button

Buttons aren't added to applications for their looks ... they are there to trigger some event. You as the programmer must specify which components in your application are to be monitored and what to do when an action or event has taken place. To do this you must do three things...


- implement the appropriate listener
- register interest in the event
- provide the event handling code

Java looks after all the other activities associated with event processing.

Implement the Listener

All components have a specific listener that is provided to process its events. These listeners are all part of the *awt.event* class. The listener for a *Button* is the **ActionListener**. In order to process *Button* events an applet must implement the **ActionListener** interface. The applet class definition must be altered to indicate this...

```
public class showComponents extends Applet implements ActionListener
```

 Modify the applet class definition to implement the ActionListener.

Register Interest in the Event


The class that contains the monitored component must include an instruction that attaches the listener to the component object. This is called **registering interest** in an event. To register interest in the *Button* activities you must include an instruction like...

```
buttonName.addActionListener(actualListener);
```

where *buttonName* is the name given to the *Button* object and *actualListener* is the class/window/application in which the button exists. To register interest in the 'btnOK' button, the following instruction is added to the *init()* method of the applet class definition...

```
btnOK.addActionListener(this);
```

where *this* indicates that the applet itself is monitoring for any button event.

 Add the instruction to register interest in the button event to the *init()* method.

Event Handling Code

Now you have to provide the code that is executed when a *Button* event occurs. Like the **WindowListener**, the **ActionListener** is an interface that contains abstract methods. In this case there is only one method ... *actionPerformed()*. Since you implemented the **ActionListener** you must add this *actionPerformed()* method to your applet class definition.

The event handling methods normally get added after the standard applet methods. Checking the Java documentation you'll see that the parameter list for the *actionPerformed()* method contains one object of class *ActionEvent*. The *actionPerformed()* method must therefore look like...

```
public void actionPerformed(ActionEvent e)
{
    //code goes here
}
```

Because the **ActionListener** is responsible for handling the events associated with a number of different components there has to be a way to identify the source of the event. This is where the *ActionEvent* object comes in. It has a number of methods associated with it, one of which is the *getSource()* method...

```
Object source = e.getSource();
```

The method returns an object of class *Object*. This can be compared to the component objects that may have triggered the event. For your purposes you'll want to compare the returned value to the *btnOK* object.

```
if (source == btnOK)
    //event handling code goes here
```

If the event handling code modifies any of the on-screen components you have to issue a *repaint()* instruction. With this instruction added, the *actionPerformed()* method now looks like...

```
public void actionPerformed(ActionEvent e)
{
    Object source = e.getSource();
    if (source == btnOK)
        //event handling code goes here

    repaint();
}
```

☞ Add the *actionPerformed()* method to the applet class definition. Add the instruction (to the if block) that will change the text of the centred label to "Pressed" when the button is pressed. Your if block will look something like this...

```
if (source == btnOK)
    lblCenter.setText("Pressed");
```

☞ Re-compile your program. You should get two error messages indicating that your label and your button objects are undefined variables or classes. This is because of the scope rules ... the objects are only available within the methods they are declared. This means that you have to move the object declarations out of the *init()* method and into the class itself...

```
public class showComponents extends Applet implements ActionListener
{
    Label lblCenter;
    Button btnOK;
```

followed by the applet and *main()* methods. This makes the variables available to all the methods defined in the applet class.

 Re-compile your program. After you've pressed the button your output should look like...



TextFields

TextFields are single line text boxes. The TextField component is included in the *java.awt* package. Anytime you are using text fields in your applet or application you must import the *TextField* component by including the following ...

```
import java.awt.TextField;
```

Again if you are using more than one type of component in your applet you can use one import statement that includes all the components...

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

Constructors

As with any object a text field must be instantiated before it can be used in a program. There are four constructors available for a *TextField* component...

```
TextField()

TextField(String text)

TextField(int nbrColumns)

TextField(String text, int nbrColumns)
```

The default constructor creates a *TextField* object with no associated text. The second constructor creates a *TextField* object and assigns the specified text to it. The third constructor creates a *TextField*

object that is a fixed number of characters wide. No text is associated with the object. The final constructor creates a *TextField* object that is a fixed number characters wide and has the specified text assigned to it. To create a new *TextField* that is 20 columns wide and has no default text associated with it the instruction would be...

```
TextField tf1 = new TextField(20);
```

Methods

The most commonly used methods available for the *TextField* class are ...

Method		Purpose
int	getColumns()	Returns the number of columns in the text field
char	getEchoChar()	Returns the character used to mask data entry
String	getText()	Returns the text field's current text
boolean	isEditable()	Returns a true if the text field is defined as editable
String	select(int start, int end)	Selects text from character start to character end
String	selectAll()	Selects all the text from the text field
void	setColumns(int nbr)	Specifies the width of the text field in characters
void	setEchoChar(char c)	Specifies the character for masking data entry
void	setEditable(boolean t)	Specifies whether the text field is editable or not
void	setText(String)	Sets the text field's text to the specified string

These methods are used with a *TextField* object. For example, to select the 3rd, 4th and 5th characters in the text field '*tf1*', the instruction would be...

```
tf1.select(2, 4);
```

Remember that indices in Java start at zero so you need to identify characters 2 through 4.

Adding the TextField

Once the *TextField* object is created you must add it to the applet/application. This is normally done in the *init()* method of an applet or in the class constructor of an application that runs in a frame. The instruction is the same for both...

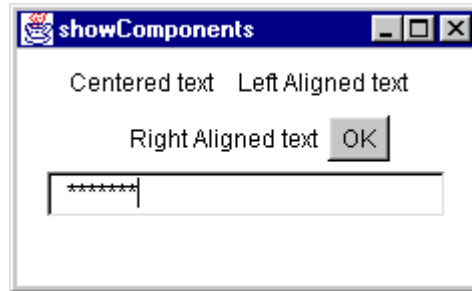
```
add(objectName);
```

To add the '*tf1*' created above, the instruction is...

```
add(tf1);
```

☞ Open your **showComponents.java** file and add the instructions required to add a `TextField` that is 25 characters wide and is password protected.

☞ Re-compile and test your program. If you type something into your text field your output should look something like this...



Using the `TextField`

`TextFields` are also components that are processed by the **ActionListener**. Therefore you must do the same three things to handle text field events as you did for buttons...

- implement the appropriate listener
- register interest in the event
- provide the event handling code

☞ Add the instruction to register interest in the text field event to the *init()* method.

Event Handling Code

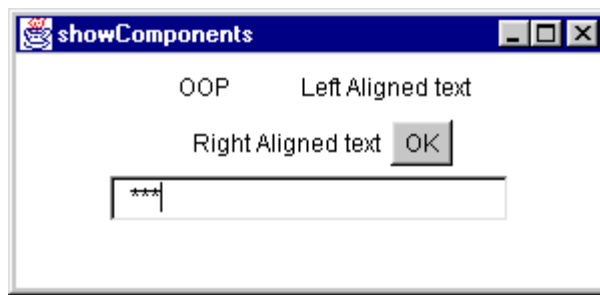
The code that is executed when the Enter key is pressed after changing a *TextField* must be put in the *actionPerformed()* method. In this case there are two possible sources for the `ActionEvent` ... the button or the text field. You already have code to determine the source of the event ...

```
Object source = e.getSource();
```

and you've tested for the button event...

```
if (source == btnOK)
    //event handling code goes here
```

☞ Add a test for the text field event in the 'else' portion of your if block. If the event is from the text field, change the text of the centered label to whatever was typed into the text field. Re-compile your program. If you've typed the word OOP into the text box (and pressed Enter) your output should look like...



Exercise – Java Components

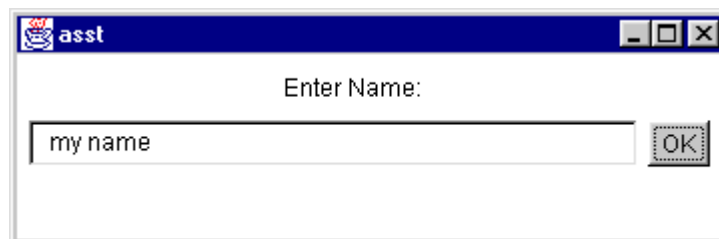
Create an applet/application that contains a labeled text field designed to hold a name and an OK button.

Focus must start at the text field and the OK button must be disabled. When text is entered into the text field (and confirmed by pressing the Enter key) the OK button is enabled and focus switches to the OK button. When the OK button is pressed the text field must be cleared and the button disabled again.

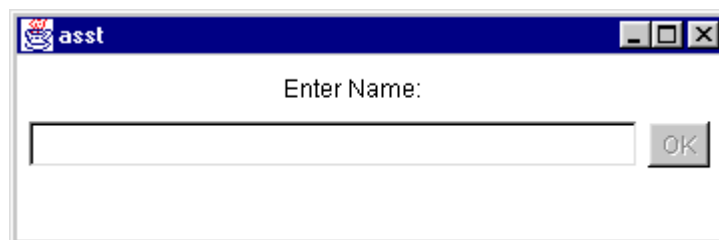
When you start up your application your screen should look something like this (with the cursor flashing in the text field)...



After you've typed in the name and pressed Enter the screen should look like...



Finally, when you press the OK button the screen should look like this again...



TextAreas

TextAreas are multiple line text boxes. The *TextArea* component is included in the *java.awt* package. Anytime you are using text areas in your applet or application you must import the *TextArea* component by including the following ...

```
import java.awt.TextArea;
```

Again if you are using more than one type of component in your applet you can use one import statement that includes all the components...

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

Constructors

As with any object a text area must be instantiated before it can be used in a program. There are five constructors available for a *TextArea* component...

```
TextArea()  
  
TextArea(String text)  
  
TextArea(int nbrRows, int nbrColumns)  
  
TextArea(String text, int nbrRows, nbrColumns)  
  
TextArea(String text, int nbrRows, nbrColumns, int scrollbars)
```

The default constructor creates a *TextArea* object with no associated text. The second constructor creates a *TextArea* object and assigns the specified text to it. The third constructor creates a *TextArea* object that is a fixed number of rows and columns. No text is associated with the object. The fourth constructor creates a *TextArea* object that is a fixed number number of rows and columns and has the specified text assigned to it. The last constructor is similar to the fourth except it also specifies what type of scrollbars are to be associated with the text area. There are four constants available to specify the type of scrollbars...

```
SCROLLBARS_NONE  
SCROLLBARS_HORIZONTAL_ONLY  
SCROLLBARS_VERTICAL_ONLY  
SCROLLBARS_BOTH
```

Note: Some platforms include scrollbars by default, while others seem to ignore the scrollbar specification. Use this constructor with care.

To create a new *TextArea* that is 40 rows deep and 30 columns wide, has no default text associated with it, and uses vertical scrollbars, the instruction would be...

```
TextArea tal = new TextArea("", 40, 20, SCROLLBARS_VERTICAL_ONLY);
```

Note: Anytime you are defining a specific type of scrollbars but you don't initially want any text in the text area, you must use the last constructor with an empty string as the first parameter.

Methods

The most commonly used methods available for the `TextArea` class are ...

Method		Purpose
void	<code>append(String s)</code>	Adds specified text 's' to the end of the current text
int	<code>getColumns()</code>	Returns the number of columns in the text area
int	<code>getRows()</code>	Returns the number of rows in the text area
char	<code>getEchoChar()</code>	Returns the character used to mask data entry
String	<code>getText()</code>	Returns the text area's current text
void	<code>insert(String s, int posn)</code>	Inserts specified text 's' starting at position ' <i>posn</i> '
boolean	<code>isEditable()</code>	Returns a true if the text area is defined as editable
void	<code>replaceRange(String s, int start, int stop)</code>	Replaces text from starting position ' <i>start</i> ' to ending position ' <i>end</i> ' with the specified text 's'
String	<code>select(int start, int end)</code>	Selects text from character ' <i>start</i> ' to character ' <i>end</i> '
String	<code>selectAll()</code>	Selects all the text from the text area
void	<code>setColumns(int nbr)</code>	Specifies the width of the text area in characters
void	<code>setRows(int nbr)</code>	Specifies the number of rows in the text area
void	<code>setEchoChar(char c)</code>	Specifies the character for masking data entry
void	<code>setEditable(boolean t)</code>	Specifies whether the text area is editable or not
void	<code>setText(String)</code>	Sets the text area's text to the specified string

These methods are used with a *TextArea* object. For example, to insert the current date at the beginning of the text area the instructions could be...

```
Date date = new Date();
ta1.insert("Date: " + date.toString() + "\n", 0);
```

Note: These instructions required the `java.util.Date` class to be imported.

Adding the TextArea

Once the text area is created you must add it to the applet/application. This is normally done in the *init()* method of an applet or in the class constructor of an application that runs in a frame. The instruction is the same for both...

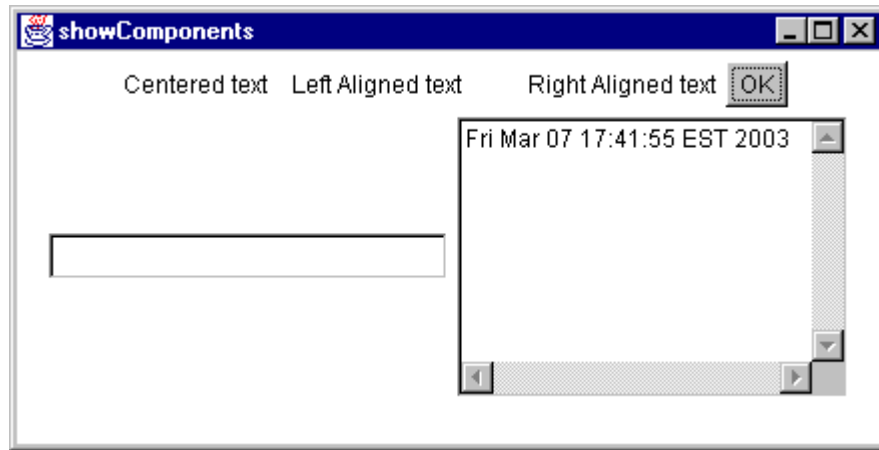
```
add(objectName);
```

To add the '*ta1*' created above, the instruction is...

```
add(ta1);
```

☞ Open your **showComponents.java** file and add the instructions required to add a `TextArea` that is 8 rows high, 25 characters wide and contains today's date.

☞ Re-compile and test your program. Your output should look something like this...



Using the `TextArea`

`TextAreas` are components that are processed by the **`TextListener`**. The process is the same for handling text area events ...

- implement the appropriate listener
- register interest in the event
- provide the event handling code

Implement the Listener

Any class can implement multiple interfaces. To specify more than one listener simply use a comma to separate the desired interfaces...

```
public class className extends Applet implements listener1, listener2
```

You can include as many as you need. Your **showComponents.java** program already implements the **`ActionListener`** interface. To also implement the **`TextListener`** interface the class declaration is...

```
public class showComponents extends Applet implements ActionListener, TextListener
```

Register Interest in the Event

Registering interest for a *TextArea* event is the similar to the method used for buttons and text fields ...

```
textAreaName.addTextListener(actualListener);
```

the only difference being `addTextListener`. The following instruction registers interest in the events caused by the changes to the text in text area *'ta1'* ...

```
ta1.addTextListener(this);
```

☞ Add the instruction to register interest in the text area event to the *init()* method.

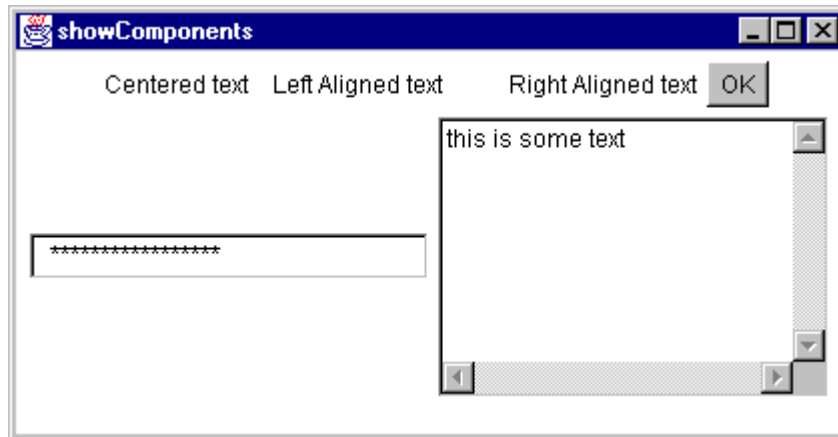
Event Handling Code

The **TextListener** interface contains one abstract method ... *textValueChanged()*. Everytime something is typed into or removed from the text area a message is sent to the **TextListener**. The parameter list for the *textValueChanged()* method contains one object of class *TextEvent*. The declaration must therefore look like...

```
public void textValueChanged(TextEvent e)
{
    //code goes here
}
```

The code that you want executed when any change occurs within a *TextArea* must be put in the *textValueChanged()* method.

☞ Add the *textValueChanged()* method to the applet class definition. Add instructions that will set the text in the text field to reflect what is typed into the text area. Re-compile and run your program. Your output should look something like...



Note: If you want the actual text to display, you'll have to remove the echo character specification for the text field.

The text field also can trigger a text event. The same *getSource()* method can be used to differentiate between the two possible sources for the event ... the text field or the text area.

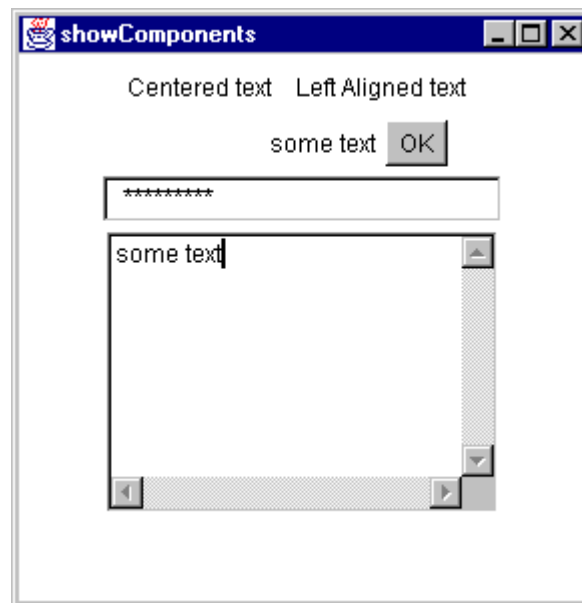
```
Object source = e.getSource();
```

This returns a value that can be compared to the text component objects...

```
if (source == tf1)
    //text field event handling code goes here
else if (source == ta1)
    //text area event handling code goes here
```

This can be extended to include as many possible sources as exist in your application.

✎ Modify the *textValueChanged()* method so that changes in the text field are echoed to the right aligned label. Changes to the text area should still be echoed to the text field. Don't forget you'll have to register interest in the *TextListener* events for the text field. Your output should now look like...



Any text you type into the text field will be echoed in the third label. Any text you type in the text area will be echoed in the text field, which in turns echoes to the label.

It is important to remember that the text field action event only triggered an event when you enter the full text and pressed enter. The text field text event triggered an event with every change to the text field. Make sure you choose the appropriate listener and event handler for the type of monitoring that is most appropriate for your application.

Absolute Layout

So far the components in your applets/applications have been positioned relative to the top left corner of the window. If they all fit across the window, they were placed that way. If not, they wrapped down to the next line. This process continued until all the specified components were added to the window. This unpredictability is not appropriate for any well designed program.

A number of layouts are provided that allow programmers the ability to place components in specific places within an application's window. These layouts are provided in a **LayoutManager** interface that

is accessible anytime you use a container object such as a *Window*, *Frame*, *Applet* or *Panel*. The absolute layout is the most basic layout. It allows you to position screen elements at fixed (x, y) positions anywhere in the application's window.

Specifying the Layout

To specify the desired layout you must include a call to the *setLayout()* method. This method is part of both the *Applet* and the *Frame* classes. It establishes communication between the **LayoutManager** and the container into which your components are placed. In the common applet/application code you are using, the *setLayout()* method is called in the *init()* method before the components are added. The instruction to specify the absolute layout is...

```
setLayout (null);
```

Positioning Components

Once the layout is specified and all of the components are added to the window, you must define the (x, y) position for each element. This is done using the *setBounds()* method...


```
objectName.setBounds(x, y, width, height);
```

where *objectName* is the component to be positioned, *x* specifies the number of pixels from the left edge of the window, *y* specifies the number of pixels from the top edge of the window, and *width* and *height* define the dimensions (in pixels) of the component.

To position a label called '*lblName*' at position (50, 25) and allow it sufficient space to be displayed properly, the instruction might be...

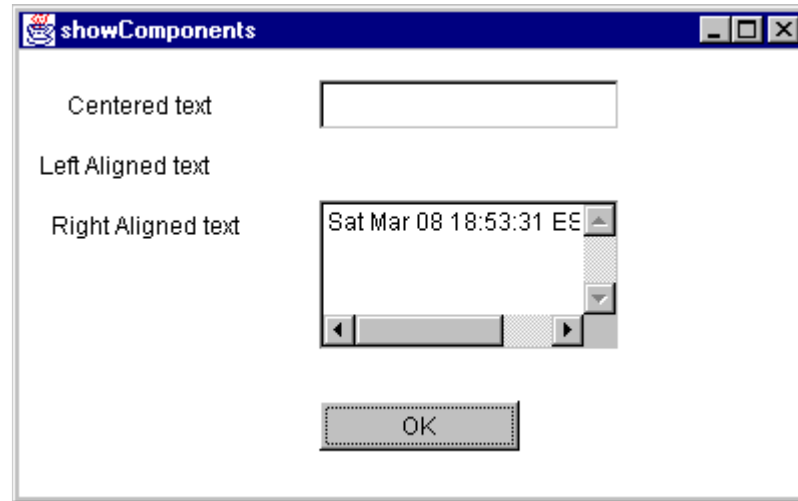
```
lblName.setBounds(50, 45, 100, 25);
```

Make sure that when you place multiple components on the screen that you leave enough space between the rows.

 Modify the **showComponents.java** file to position the existing components according to the following chart...

Component	x	y	width	height
Centered label	10	15	100	25
Left label	10	45	100	25
Right label	10	75	100	25
Button	150	175	100	25
Text field	150	15	150	25
Text area	150	75	150	75

☞ Compile and run your program. Your output should now look something like...



Notice how the labels positioned themselves within the specified layout width according to the original label position specifier ... the label specified as left aligned is positioned right at pixel 10 while the centered and right aligned labels are moved over. For any future programs make sure that all labels in a given application are aligned the same way.

Notice that the button also changed ... it stretched to fill the specified width and height dimensions. This is the way you can make all your buttons the same size.

Finally notice the change in the text area. It was resized according to the layout dimensions, overriding the dimensions specified in the text area definition. Most programmers will omit the row and column specifiers from the text area declaration and specify them as part of the layout.

Background Colour

Unless otherwise specified all your applications will have a white background with black text. You have already looked at the methods available for changing text colour. To change the background colour you must include the *setBackground()* method...

```
setBackground(new Color( redValue, greenValue, blueValue));
```

The colour values can be defined using either decimal or hexadecimal values. The following instruction sets the background colour to a light yellow...

```
setBackground(new Color(255, 255, 0xC6));
```

When choosing your background, text and graphics colours you may want stick to the standard web palette. You can generally find this 256 colour palette by doing a web search.

☞ Set the background colour to a light red in your **showComponents.java** file. Re-compile and test your program.

Checkboxes

Checkboxes allow users to select any number of options from a given set of items. The *Checkbox* component is included in the *java.awt* package. Anytime you are using checkboxes in your applet or application you must import the *Checkbox* component by including the following ...

```
import java.awt.Checkbox;
```

Again if you are using more than one type of component in your applet you can use one import statement that includes all the components...

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

Constructors

As with any object a checkbox must be instantiated before it can be used in a program. There are three constructors available for a simple *Checkbox* component...

```
Checkbox()  
  
Checkbox(String text)  
  
Checkbox(String text, boolean checked)
```

The default constructor creates a *Checkbox* object with no associated label. The second constructor creates a *Checkbox* object and displays the specified label beside the box. The third constructor creates a *Checkbox* object that displays the specified label beside the box and is either check or not checked.

To create a new *Checkbox* that is already checked and shows the label '*Option 1*' beside it, the instruction would be...

```
Checkbox cb1 = new Checkbox("Option 1", true);
```

Methods

The most commonly used methods available for the *Checkbox* class are ...

Method		Purpose
String	getLabel()	Returns the checkbox's label
boolean	getState()	Returns true/false indicating whether box is checked
void	setState(boolean state)	Sets the checkbox to checked or not checked
void	setLabel(String)	Sets the checkbox's label

These methods are used with a *Checkbox* object. For example, to ensure that the checkbox '*cb1*' is not checked, the instruction would be...

```
cb1.setState(false);
```


Adding the Checkbox

Once the Checkbox is created you must add it to the applet/application. This is normally done in the *init()* method of an applet or in the class constructor of an application that runs in a frame. The instruction is the same for both...

```
add(objectName);
```

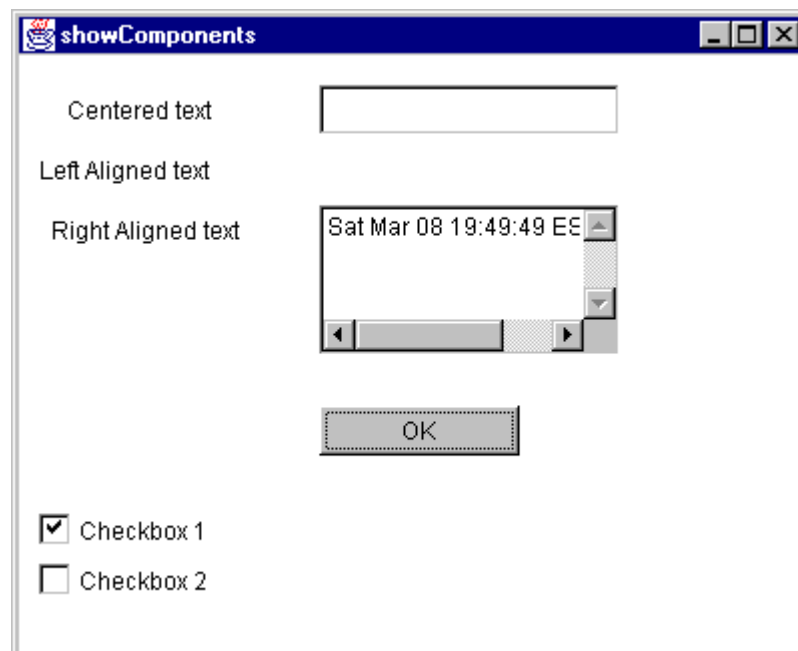
To add the 'cb1' created above, the instruction is...

```
add(cb1);
```

🔧 Open your **showComponents.java** file and add the instructions required to add two Checkboxes ... one that says 'Checkbox 1' and the other that says 'Checkbox 2'. Make the first one checked and the second one not checked. Position them according to...

Component	x	y	width	height
Checkbox 1	10	225	100	25
Checkbox 2	10	250	100	25


🔧 Re-compile and test your program. Your output should look something like this...



Using the Checkbox

Checkboxes are components that are processed by the **ItemListener**. The process is the same for handling checkbox events ...

- implement the appropriate listener
- register interest in the event
- provide the event handling code

 Modify the *showComponents* class definition to implement the **ItemListener**.

Register Interest in the Event

Registering interest for a *Checkbox* event is the similar to the method used for the other components ...

```
CheckboxName.addItemListener(actualListener);
```

the only difference being **addItemListener**. The following instruction registers interest in the events caused by the selecting checkbox 'cb1' ...

```
cb1.addItemListener(this);
```


 Add the instructions required to register interest in the checkbox events to the *init()* method.

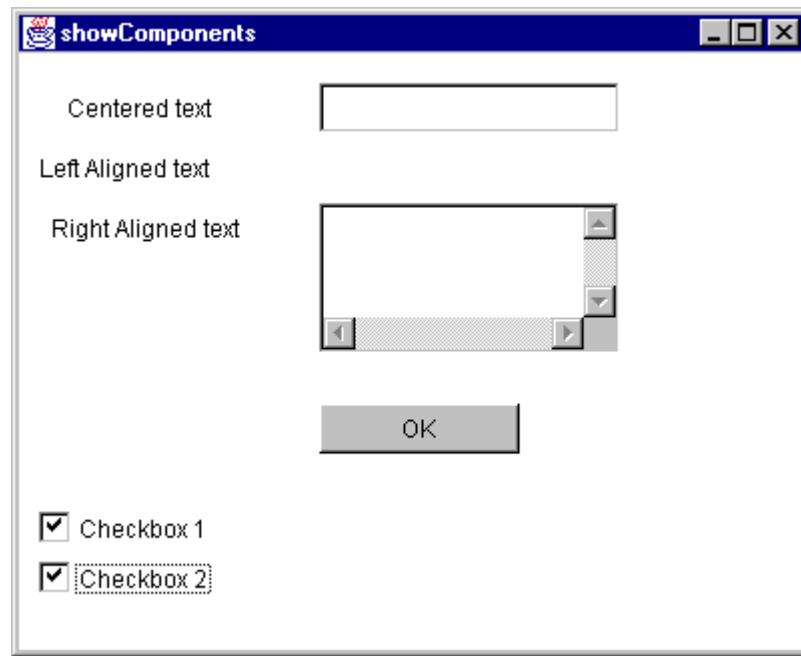
Event Handling Code

The **ItemListener** interface contains one abstract method ... *itemStateChanged()*. Everytime something a checkbox is checked or unchecked a message is sent to the **ItemListener**. The parameter list for the *itemStateChanged()* method contains one object of class *ItemEvent*. The declaration must therefore look like...

```
public void itemStateChanged(ItemEvent e)
{
    //code goes here
}
```

The code that you want executed when any change occurs to the state of a *Checkbox* must be put in the *itemStateChanged()* method.

 Add the *itemStateChanged()* method to the applet class definition. Add instructions that will clear the text in the text field if the first checkbox is checked and clear the text area if the second checkbox is checked. You'll have to use the *getSource()* method to differentiate between the two checkbox object events. Your output should now look something like...



Radio Buttons

Radio buttons are specialized checkboxes that allow users to select one option from a given set of items. The *CheckboxGroup* component is required to define radio buttons and is included in the *java.awt* package. Anytime you are using radio buttons in your applet or application you must import the *Checkbox* component by including the following ...

```
import java.awt.Checkbox;
```

Again if you are using more than one type of component in your applet you can use one import statement that includes all the components...

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

Constructors

As with any object a radio button must be instantiated before it can be used in a program. Before you create the actual radio button you must create a group for it to exist in. The constructor for the radio button group is...

```
CheckboxGroup()
```

This creates an object of type *CheckboxGroup* that needs to be specified when creating your radio buttons. There are two specialized *Checkbox* constructors available for radio button components...

```
Checkbox(String text, boolean checked, CheckboxGroup group)
```

```
Checkbox(String text, CheckboxGroup group, boolean checked)
```

Both constructors create a *Checkbox* object that belongs to a specified *CheckboxGroup*. Each also displays the specified label beside the box.

To create two new radio buttons with no default selection (none are checked to start) and the labels ‘*Radio option 1*’ and ‘*Radio option 2*’ beside them, the instructions would be...

```
CheckboxGroup cbg = new CheckboxGroup();
Checkbox rb1 = new Checkbox("Radio Option 1", false, cbg);
Checkbox rb2 = new Checkbox("Radio Option 2", false, cbg);
```

The first instruction creates the group for the radio buttons. The remaining two instructions create the actual radio buttons that are associated with the specified group.

Methods

The same methods available for the *Checkbox* class are available for radio buttons...

Method		Purpose
String	getLabel()	Returns the radio button’s label
boolean	getState()	Returns true/false indicating whether button is checked
void	setState(boolean state)	Sets the radio button to checked or not checked
void	setLabel(String)	Sets the radio button’s label

These methods are used with a radio button object. For example, to determine whether the radio button ‘*rb1*’ is checked or not, the instruction would be...

```
state = rb1.getState();
```

The variable *state* is a boolean value that can be checked for the values true or false.


Adding the Radio Button

Once a radio button is created you must add it to the applet/application. This is normally done in the *init()* method of an applet or in the class constructor of an application that runs in a frame. The instruction is the same for both...

```
add(objectName);
```

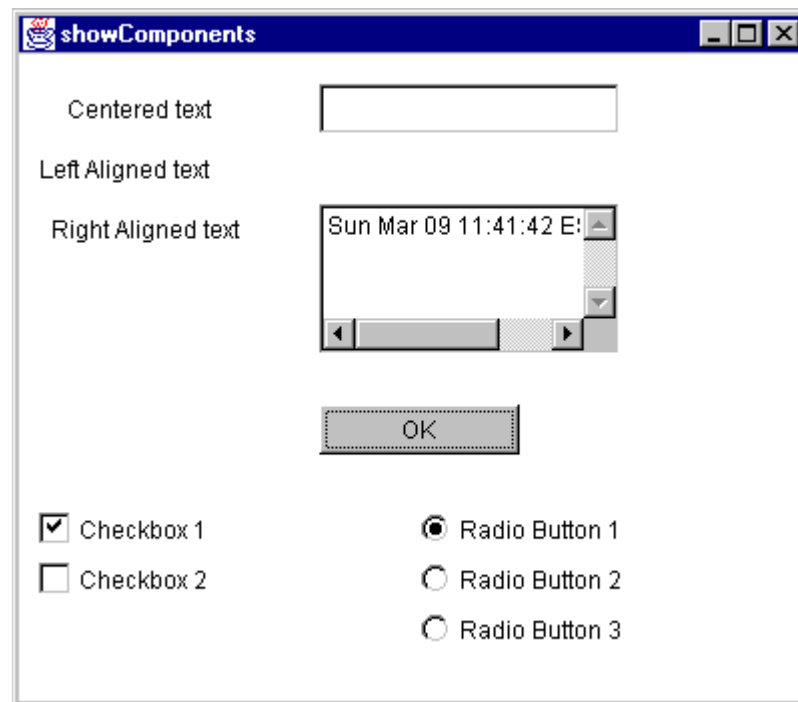
To add the ‘*rb1*’ created above, the instruction is...

```
add(rb1);
```

 Open your **showComponents.java** file. Add the instructions required to add three radio buttons that say ‘Radio Button 1’, ‘Radio Button 3’, and ‘Radio Button 3’. Make the first one the default selection. Position them according to...

Component	x	y	width	height
Radio Button 1	10	275	100	25
Radio Button 2	10	300	100	25
Radio Button 3	10	325	100	25

☞ Re-compile and test your program. Your output should look something like this...

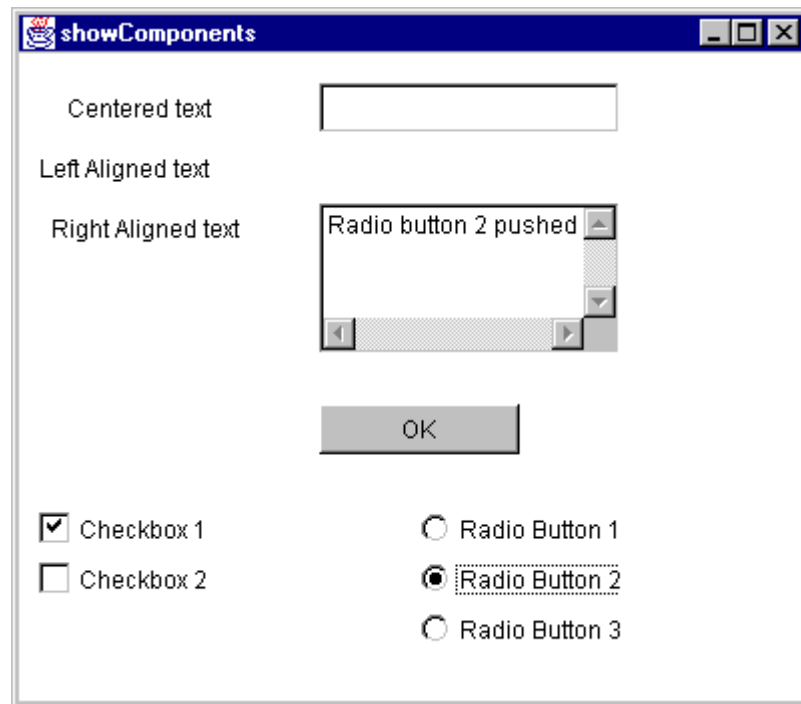


Using the Radio Button

Radio buttons events are processed by the **ItemListener** in exactly the same way as check boxes ...

- implement the appropriate listener
- register interest in the event
- provide the event handling code

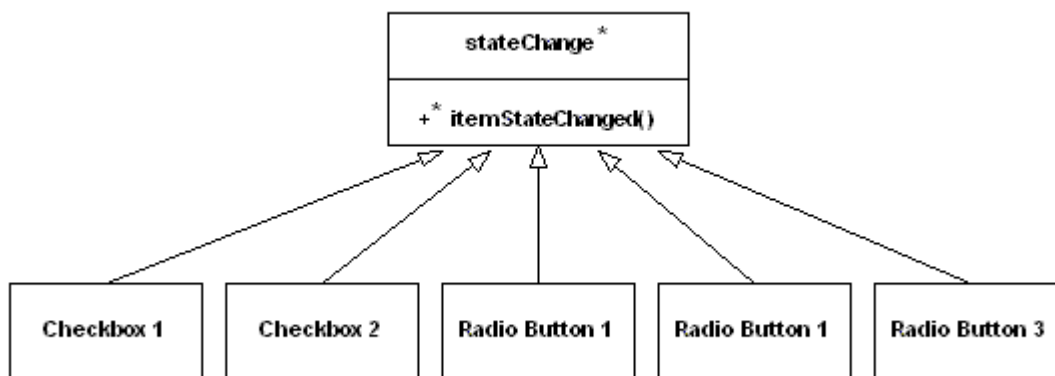
☞ Add instructions that will display which radio button is selected in the text area. You'll have to add the instructions to register interest in the three radio buttons and modify the *itemStateChanged()* method. Your output should now look something like...



Handling Events with Inner Classes


The amount code in the *itemStateChanged()* method increases with every checkbox and radio button you add to your application. Because the *getSource()* method returns an object of type *Object* and not an integer of character value you cannot use a switch structure to select between the possible sources of event. Depending on the number of components you have in your application, the event handling code can get quite convoluted.

One of the common ways of working around this problem is to handle action, text and item events using the same technique as you did for window events ... create a separate class that monitors the events for each individual component.



Defining an Abstract Class

The main class, *stateChange*, must be defined as an inner class of the applet in order to properly process events trapped by the applet. Notice that it is an abstract class that has one abstract method *itemStateChanged()*. The bottom five classes inherit this one method and define the actual event handling code.

 Comment out the *itemStateChanged()* method from the **showComponents.java** file. Add the following abstract class definition in its place...

```
abstract class stateChange implements ItemListener
{
    abstract public void itemStateChanged(ItemEvent e);
}
```

This is the standard format for defining an abstract class.

Defining the Event Handling Classes


Now each of the remaining five classes needs to be defined. Each is a child class of *stateChange* and is coded in the following way...

```
class className extends stateChange
{
    //event handling code goes here
}
```

For example, the class to handle the first radio button '*rb1*' might look something like this...

```
class rb1Change extends stateChange
{
    if (e.getStateChange() == ItemEvent.SELECTED)
        ta1.setText("radio button 1 selected");
}
```

The *getStateChanged()* method returns an integer value that can be compared to the constant value *SELECTED*. Remember that only one radio button in a group can be selected at a time, so programmers usually only test for and respond to *SELECTED* events.

 Add classes to handle each of the two checkboxes and three radio buttons. All events should be handled in the same way as they were in the original program. The class code must be placed inside the applet class definition and normally follows the abstract class definition.

Registering Interest


When you define event handling inner classes, the process of registering interest is a little bit different. No longer is the *getStateChanged()* method defined within the applet responsible for responding to events ... now it is up to the appropriate event handling class. The format is the same...

```
checkboxName.addItemListener(actualListener);
```

The difference is in the *actualListener*. Now the listener must be an instance of the appropriate event handling class. For example, to register interest in the events of the first radio button object 'rb1' the instruction is...

```
rb1.addItemListener(new rb1Change());
```

where the *actualListener* is a newly created instance of the *rb1Change* class.

 Modify the register interest instructions in the *init()* method to reflect the inner event handling classes. The new inner classes should follow the *textValueChanged()* method as shown here. The required code for the abstract class and for two of the five event handling classes is defined here for your reference...


```
public void textValueChanged(TextEvent e)
{
    Object source = e.getSource();
    if (source == tal)
        tf1.setText(tal.getText());
    else if (source == tf1)
        lblRight.setText(tf1.getText());

    repaint();
}

abstract class stateChange implements ItemListener
{
    public abstract void itemStateChanged(ItemEvent e);
}

class cb1Change extends stateChange
{
    public void itemStateChanged(ItemEvent e)
    {
        if (e.getStateChange() == e.SELECTED)
            tf1.setText("");
    }
}

class cb2Change extends stateChange
{
    public void itemStateChanged(ItemEvent e)
    {
        if (e.getStateChange() == e.SELECTED)
            tal.setText("");
    }
}
```

 Remove the *ItemListener* implementation from the *Applet* class definition. Re-compile your code. Your output should be exactly the same as it was the last time.

Note: It is recommended that for ease of maintenance you use this technique of inner classes for any listener that must respond to the events of three or more different components.

List Boxes

List boxes are specialized text areas where each line represents an option. The *List* component is included in the *java.awt* package. Anytime you are using text areas in your applet or application you must import the *List* component by including the following ...

```
import java.awt.List;
```

Again if you are using more than one type of component in your applet you can use one import statement that includes all the components...

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

Constructors

As with any object a list box must be instantiated before it can be used in a program. There are three constructors available for a *List* component...

```
List()
```

```
List(int nbrVisibleRows)
```

```
List(int nbrVisibleRows, boolean multipleMode)
```

The default constructor creates a basic *List* object. The second constructor creates a *List* object and that displays a specified number of items from which one can be selected. If the list contains more items than this a scrollbar is automatically added. The third constructor creates a *List* object that displays a certain number of items. It also, if the second parameter, *multipleMode* is set to true, allows multiple selections to be made from the list.

To create a new *List* that shows 10 items and does not allow multiple selections, the instruction would be...

```
List list1 = new List(10, false);
```

Methods

The most commonly used methods available for the *List* class are ...

Method		Purpose
void	add(String s)	Adds specified text 's' at end of the list
void	add(String s, int n)	Adds specified text 's' at position 'n' in the list
void	deselect(int n)	Unselects item number 'n'
String	getItem(int n)	Returns the text of item number 'n'

int	getItemCount()	Returns the number of items in the list
int	getSelectedIndex()	Returns the number selected item
String	getSelectedItem()	Returns the text of the selected item
boolean	isSelected(int n)	Returns true if item is selected
void	remove(int n)	Removes item number 'n' from the list
void	replaceItem(int n, String s)	Replaces item number 'n' with specified text
void	select(int n)	Selects item number 'n'
void	setMultipleSelections(boolean b)	Sets list to allow or not allow multiple selections

These methods are used with a *List* object. For example, to add the option 'red' as the second item in the list called '*list1*' the instruction would be...

```
list1.add("red ", 1);
```

As with Java arrays, the first element in a list is element number zero (0). If the data you want added to the list is not a *String* object it must either be converted or typecast to *String*.

Adding the List

Once the *List* is created you must add it to the applet/application. This is normally done in the *init()* method of an applet or in the class constructor of an application that runs in a frame. The instruction is the same for both...

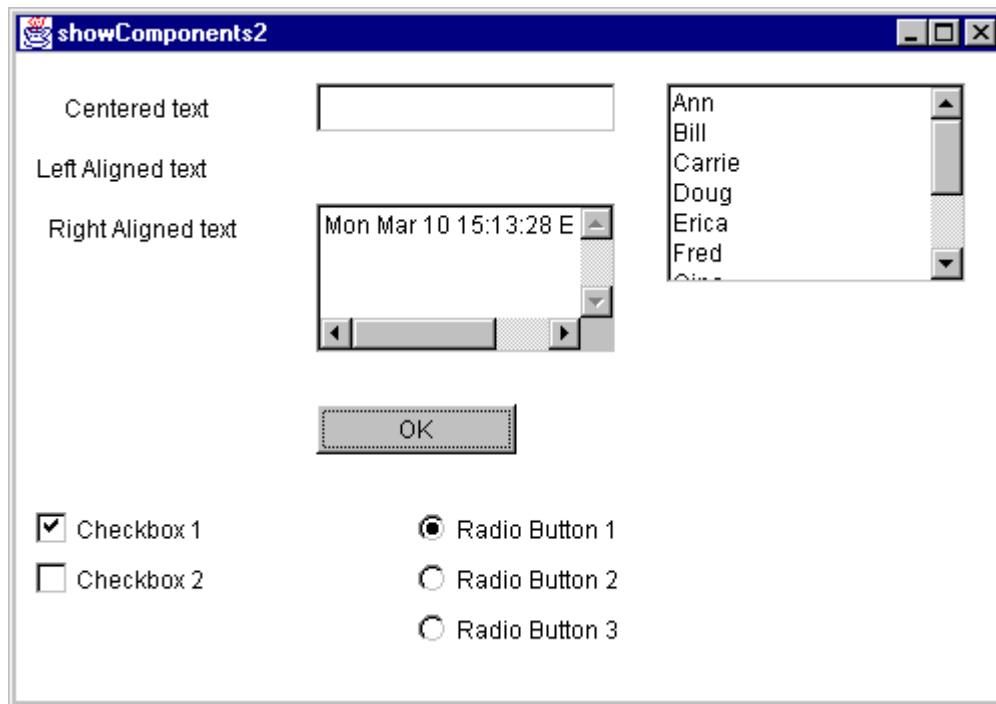
```
add(objectName);
```

To add the '*list1*' created above, the instruction is...

```
add(list1);
```

☞ Open your **showComponents.java** file and add the instructions required to add a *List* that shows five entries and allows only one selection at a time. Populate the list with at least ten names. Place the list box at position (325, 15) and make it 150 pixels wide and 100 pixels high.

☞ Re-compile and test your program. Your output should look something like this...



Using the List

Lists have two available listeners ... **ItemListener** and **ActionListener**. **ItemListener** receives messages when an item in the list is clicked and **ActionListener** receives notification when an item is double clicked. The process is the same for both events ...

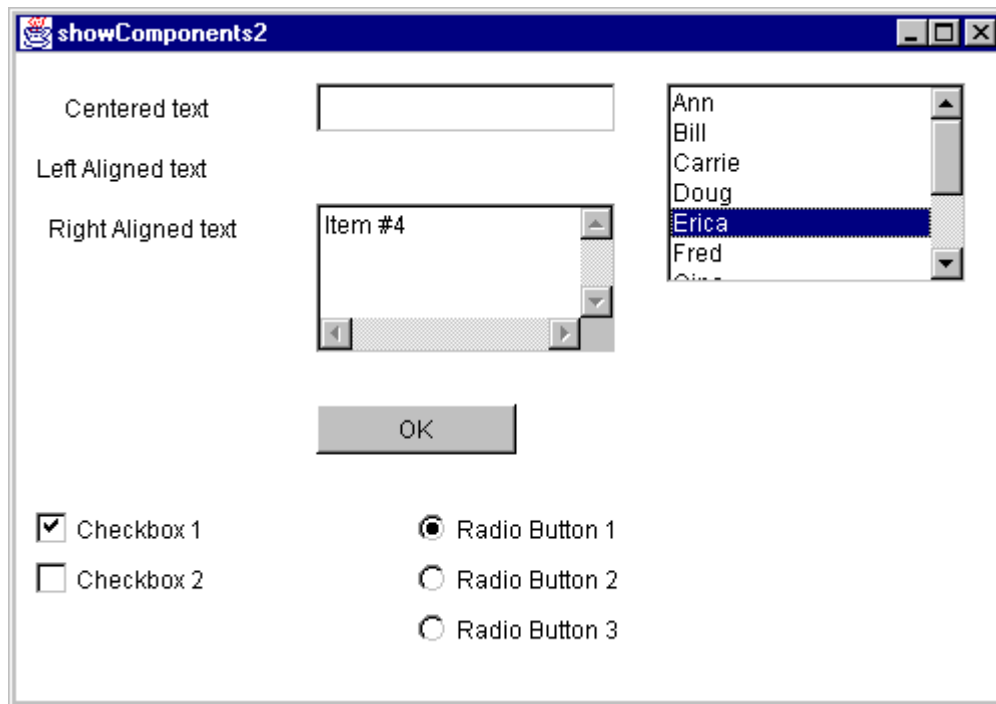
- implement the appropriate listener
- register interest in the event
- provide the event handling code

☞ Add the instructions required to register interest in both types of events to the *init()* method. Remember that you are handling action events differently from item events ... make sure the *actualListener* is defined accordingly.

Event Handling Code

Event handling for the action events and the item events is handled the same way as other component events are handled.

☞ Add the instructions to display a selected list item in the text area if it is clicked. If it is double clicked, display "Item #*n*" in the text area where *n* is the item number in the list. Re-compile and run your program. If you double click the 5th item in the list your output should now look something like...



Choice Boxes

Think of a choice box as a read-only combo box. It is a single line window with a drop-down box. You cannot enter any data in the text box area. The *Choice* component is included in the *java.awt* package. Anytime you are using text areas in your applet or application you must import the *Choice* component by including the following ...

```
import java.awt.Choice;
```

Again if you are using more than one type of component in your applet you can use one import statement that includes all the components...

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

Constructors

As with any object a list box must be instantiated before it can be used in a program. There is only one constructor available for a *Choice* component...

```
Choice()
```

This default constructor creates a basic *Choice* object. Like a list box, all items you want displayed in the choice box must be added individually.

To create a new *Choice* box, the instruction would be...

```
Choice choice1 = new Choice();
```

Methods

The most commonly used methods available for use with the *Choice* class are ...

Method		Purpose
void	add(String s)	Adds specified text 's' at the end of the choices
String	getItem(int n)	Returns the text of item number 'n'
int	getItemCount()	Returns the number of items in the choices
int	getSelectedIndex()	Returns the number selected item
String	getSelectedItem()	Returns the text of the selected item
void	insert(String s, int n)	Adds specified text 's' in as item number 'n'
void	remove(int n)	Removes item number 'n' from the choices
void	remove(String s)	Removes first occurrence of 's' from the choices
void	removeAll()	Removes all choices
void	select(int n)	Selects item number 'n'
void	select(String s)	Selects first item matching 's'

For example, to add the item “*Computer Engineering*” the choice box called ‘*choice1*’ the instruction would be...

```
choice1.add("Computer Engineering");
```


Adding the Choice Box

Once the *Choice* object is created you must add it to the applet/application. This is normally done in the *init()* method of an applet or in the class constructor of an application that runs in a frame. The instruction is the same for both...

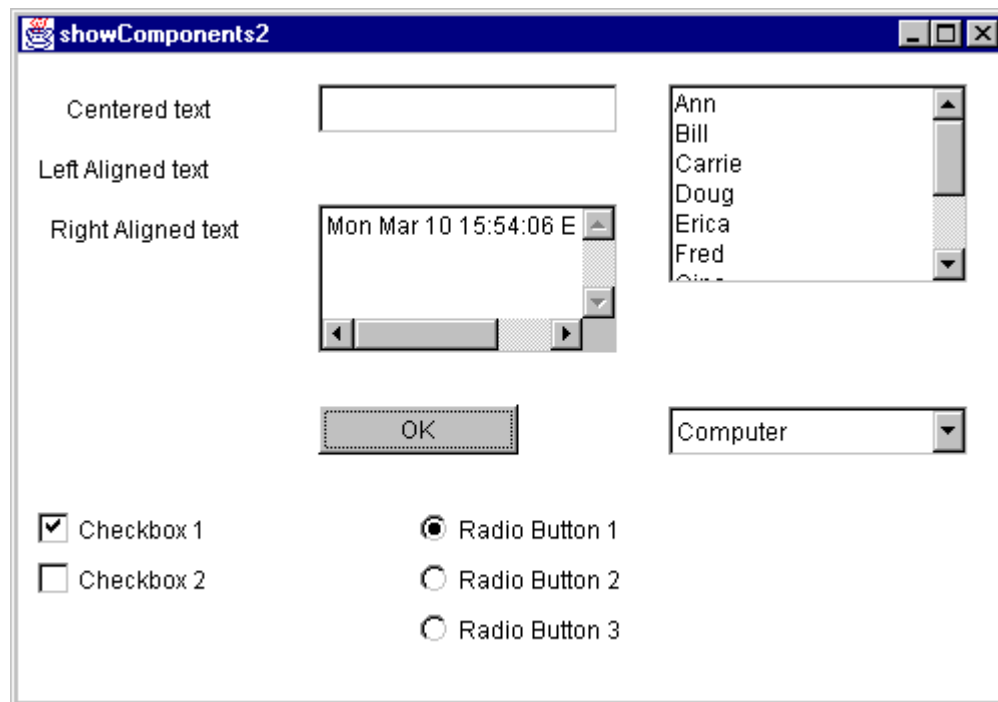
```
add(objectName);
```

To add the ‘*choice1*’ created above, the instruction is...

```
add(choice1);
```

 Open your **showComponents.java** file and add the instructions required to add a *Choice* object. Populate the list with at least five technology program names. Place the choice box at position (325, 175) and make it 150 pixels wide and 100 pixels high.

☞ Re-compile and test your program. Your output should look something like this...



Using the Choice Box

Choice boxes have only one available listener ... **ItemListener**. **ItemListener** receives messages when an item in the choice box is clicked or selected. The process is the same ...

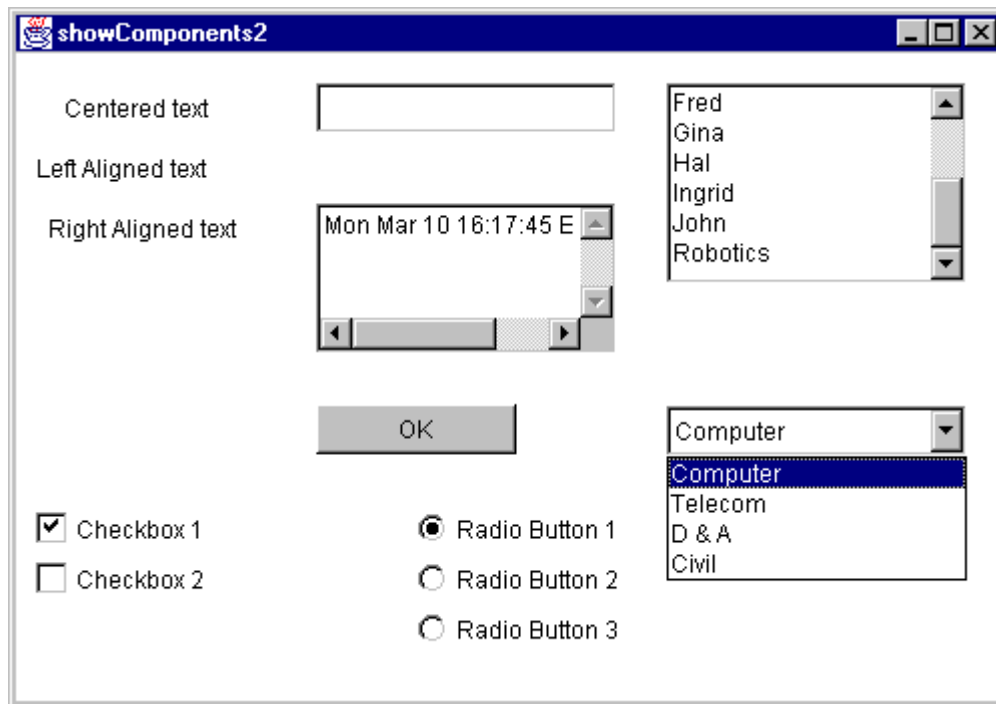
- implement the appropriate listener
- register interest in the event
- provide the event handling code

☞ Add the instructions required to register interest in item events to the *init()* method.

Event Handling Code

Event handling for the item events is handled the same way as other component events are handled.

☞ Add the instructions to remove any selected choice box item from the choice box and add it to the list box. Re-compile and run your program. Your output should now look something like...



In this case 'Robotics' was selected from the choice box. It is no longer in the combo box but was added to the list box.

Exercise – Components

Demonstrate your fully functioning **showComponents** applet/application.

Summary Sheets

The following four reference pages contain summaries of the main components, listeners and event handling methods. The also list many of the inherited attributes and methods that are available when using components in your applets and applications.

Java Components, Listener Classes and Listener Methods Summary

There are other Java components or controls that were not covered in the previous sections. The process is the same for handling all other events ...

- implement the appropriate listener
- register interest in the event
- provide the event handling code

The following table summarizes the available components, the applicable listener and the methods required for event handling.

Control	Registers Interest	Receives Event
Button List MenuItem TextField	addActionListener(Listener)	actionPerformed(ActionEvent e)
Checkbox Choice List Checkbox-MenuItem	addItemListener(Listener)	itemStateChanged(ItemEvent e)
DialogFrame	addWindowListener(Listener)	windowClosing(WindowEvent e) windowOpened(WindowEvent e) windowIconified(WindowEvent e) windowDeiconified(WindowEvent e) windowClosed(WindowEvent e) windowActivated(WindowEvent e) windowDeactivated(WindowEvent e)
Scrollbar	addAdjustmentListener(Listener)	adjustmentValueChanged(AdjustmentEvent e)
Canvas Dialog Frame Panel Window	addMouseListener(Listener)	mousePressed(MouseEvent e) mouseReleased(MouseEvent e) mouseEntered(MouseEvent e) mouseExited(MouseEvent e) mouseClicked(MouseEvent e)
Canvas Dialog Frame Panel Window	addMouseMotionListener(Listener)	mouseDragged(MouseActionEvent e) mouseMoved(MouseActionEvent e)
Component	addKeyListener(Listener)	keyPressed(KeyEvent e) keyReleased(KeyEvent e) keyTyped(KeyEvent e)
TextComponent	addTextListener(Listener)	textValueChanged(TextEvent e)

Event Class Methods

Most components inherit attributes and methods from higher order Java classes. Some of the more commonly used methods are listed in the following table. Check the on-line documentation for a full set of available methods.

Event Class	Methods	Description
EventObject	Object getSource()	source object
ComponentEvent	Component getComponent()	source component
FocusEvent	boolean isTemporary()	whether lost focus will return to this component
KeyEvent	char getKeyChar() int getKeyCode()	key affected
MouseEvent	int getX() int getY() Point getPoint()	mouse position
ActionEvent	String getActionCommand()	string identifying the command (from Button or Menu)
AdjustmentEvent	int getValue()	new value
ItemEvent	Object getItem()	the selected/deselected item
WindowEvent	Window getWindow()	the source window

General Component Methods

There are also a number of general component methods. The more commonly used methods are included in this table. For a full list, check the on-line documentation.

Syntax	Purpose
void setEnabled(boolean value) boolean isEnabled()	Makes responsive/unresponsive to user actions
void setVisible(boolean value) boolean isVisible()	Makes visible/non-visible
void setLocation(int x, int y) void setLocation(Point p) Point getLocation()	Controls position of a component
void setSize(int width, int height) void setSize(Dimension d) Dimension getSize()	Define or get size of object
void setBounds(int x, int y, int w, int h)	Reposition and resize all at once
void requestFocus()	set focus to object
Color getBackground() void setBackground(Color c)	get or set background color
Color getForeground() void setForeground(Color c)	get or set foreground color
Font getFont() void setFont(Font f)	get or set text font

Specific Component Methods

A number of the more commonly used methods associated with specific components are listed in the following table. A full list is found in the on-line documentation.

Component	Syntax	Purpose
Label	int getAlignment() setAlignment (LEFT/CENTER/RIGHT)	manipulate alignment of label text
	String getText() void setText(String text)	modify label's text
Text Components	String getText() void setText(String text)	modify text component's text
	String getSelectedText() int getSelectionStart() int getSelectionEnd()	get selected portion of text
	void select(int start, int end) void selectAll()	select text
	void setEditable(boolean t) boolean isEditable()	can text be edited or not
TextField & TextArea	void setColumns(int n) int getColumns()	modify width of text field
	void setEchoChar(char c) char getEchoChar() boolean echoCharSet()	mask data entry
TextArea	void setRows(int n) int getRows()	modify height of text area
	int getScrollbarVisibility()	check scrollbar setting
	void append(String s) void insert(String s, int pos) void replaceRange(String s, int start, int end)	manipulate text in text area

Mouse, Keyboard and Menu Events

Most applications involve mouse and/or keyboard control as well as a number of menu selections. Each of these requires its own event handling process. This section looks at the types of events that can be monitored as well as how they can be handled.


Mouse Events


Depending on your application there may be times when it is necessary to track mouse motion and monitor for mouse actions such as clicking. Unlike components there is no physical thing to add to the screen, it is just a matter of monitoring for activity.


Monitoring for Mouse Activity

There are two listeners associated with mouse activity ... **MouseMotionListener** and **MouseListener**. **MouseMotionListener** receives notification when the mouse is moved or dragged. **MouseListener** receives messages for all other mouse activities. The process is the same for both events ...

- implement the appropriate listener
- register interest in the event
- provide the event handling code

 Starting with your master file, search and replace all occurrences of the word '*master*' with '*mouse*'. Save your program as **mouse.java** in a new folder.

 Add the *paint()* method to the applet class. Add instructions to set the drawing colour to a changeable value (use a variable called something like *drawingColour*), and to draw a filled circle (at a variable position (*circleX*, *circleY*), and of variable *size*). Declare the variables for *drawingColour*, *circleX*, *circleY*, and *size* so that they are available to all the methods in the applet class. Assign a starting position of (20, 20) and a starting size of 15.

 Create two *Color* objects ... one for a foreground colour and the other for a background colour. In the *init()* method set your applet background to the background colour and the *drawingColour* variable to the foreground colour. Your code for the applet class should look something like...

```
import java.applet.Applet;
import java.awt.event.*;
import java.awt.Frame;
import java.awt.*;

public class mouse extends Applet
{
    static int screenSize= 300;
    int circleX = 20;
    int circleY = 20;
    int size = 15;
    Color bkColour = new Color(0,0,0);           //black
    Color foreColour = new Color(255,255,0);      //yellow
    Color drawColour;
```

```

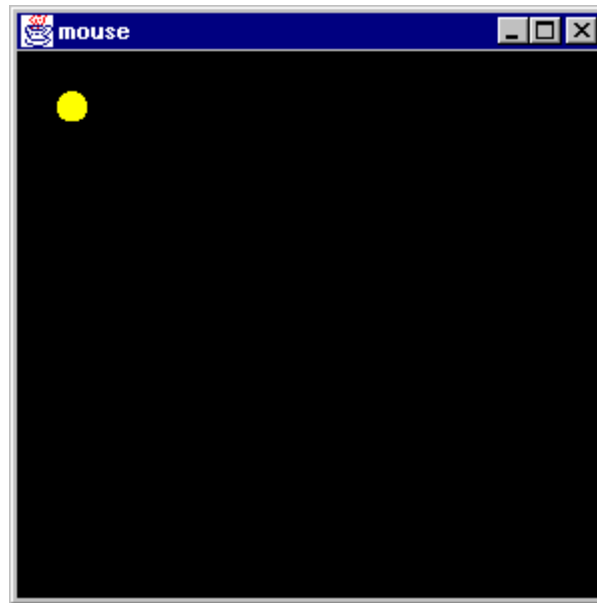
public static void main(String args[])
{
    mouseFrame app = new mouseFrame ("Mouse Application");
    app.setSize(screenSize, screenSize);
    app.show();
}

public void init()
{
    setBackground(bkColour);
    drawColour = foreColour;

public void paint(Graphics g)
{
    g.setColor(drawColour);
    g.fillOval(circleX, circleY, size, size);
}
}

```

☞ Compile and test your program. You should see a small circle displayed on the screen ...



Mouse Motion

To track mouse motion it is necessary to implement the **MouseMotionListener**. This is typically done in a class similar to the **winWatch** class used to monitor for window events. In order for the mouse motion monitoring class to have easy access to information about the mouse (with respect to its environment) it is often created as an inner class of the applet class. Its format is...

```

class mouseMotionWatch implements MouseMotionListener
{
    public void mouseMoved(MouseEvent mEvt) {}
    public void mouseDragged(MouseEvent mEvt) {}
}

```

☞ Add the mouse motion monitoring class to the applet class.

The **MouseMotionListener** listener defines two abstract classes that must be included in the class ... *mouseMoved()* and *mouseDragged()*. Depending on the application, event handling code is added where required. Each method receives a *MouseEvent* object that identifies key information about the mouse and its position. The methods most associated with mouse motion are...

Method		Purpose
int	<code>getX()</code>	Returns the current x coordinate position
int	<code>getY()</code>	Returns the current y coordinate position
void	<code>translatePoint(int x, int y)</code>	Translates the event coordinates to a new position specified by x and y

☞ Add instructions to the *mouseMoved()* method that gets the current (x,y) position of the mouse and assigns these values to the circle's x and y variables...

```
circleX = mEvt.getX();  
circleY = mEvt.getY();  
repaint();
```

where *circleX* and *circleY* are the position parameter in the `g.fillOval()` instruction in the *paint()* method. The *repaint()* instruction forces the circle to be redrawn each time a mouse moved event is triggered.

The only thing left to do is to register interest in the mouse motion events. This is done in the *init()* method. The initiator of the mouse motion event is the applet itself while the **mouseMotionWatch** class is the responder. The instruction to register interest is therefore...

```
this.addMouseMotionListener(new mouseMotionWatch());
```

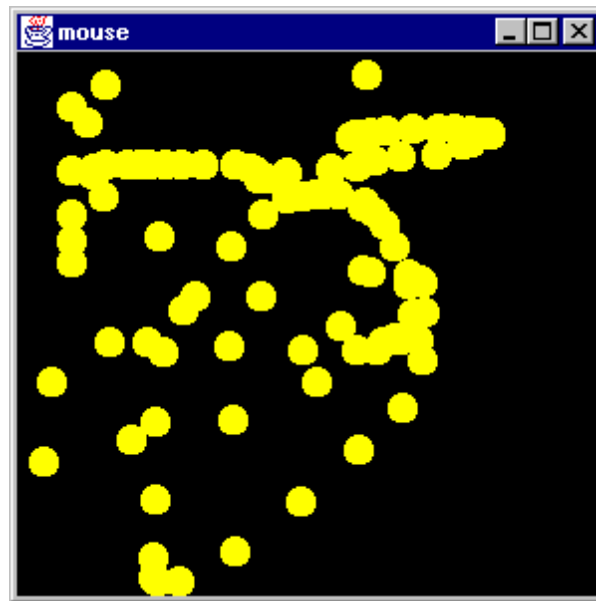
☞ Add the instruction to register interest in the mouse motion events to the *init()* method of the applet. Compile and test your program. As you move the mouse around you should see the circle follow the mouse.

Every time the applet is told to repaint the screen a build-in method called *update()* is called. This clears the screen in preparation for the repaint. If you choose not to have the screen cleared before it is updated you can override the built-in *update()* method with your own...

```
public void update(Graphics g)  
{  
    paint(g);  
}
```

This method simply calls the specified *paint()* method without clearing the screen first.

☞ Add the *update()* method to the applet class. Re-compile your program. You should now see a trail of circles that follows the mouse...

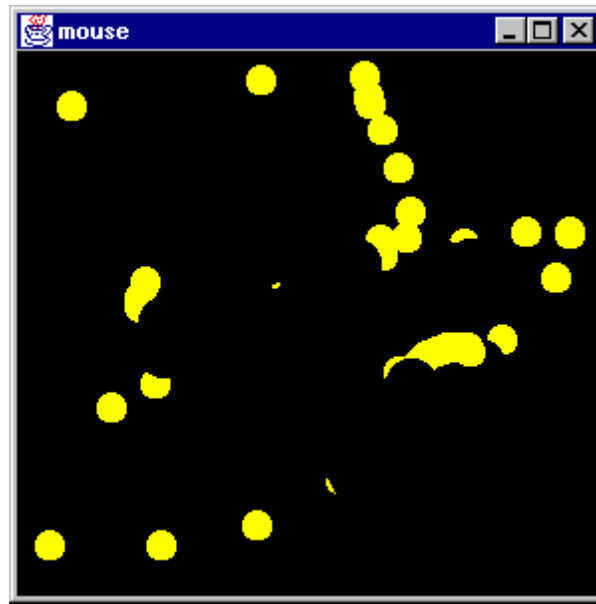


Mouse motion also includes dragging the mouse ... moving the mouse with the left button pressed. Event handling for mouse drags is processed in the *mouseDragged()* method that is part of the **mouseMotionListener**.

☞ Modify your **mouse.java** program so that the trail of circles is still drawn when the mouse is moved but if you drag the mouse it erases beneath the mouse. To accomplish this you'll have to...

- modify the *mouseMoved()* method to ensure the *drawColour* is the foreground colour. Set the size of the circle to the default value.
- add code to the *mouseDragged()* method to set the *drawColour* to the background colour (you erase by actually painting with the background colour). Set the size of the circle to a value larger than the default value ... this is typically done when you are 'erasing' something. You also need to add mouse tracking and screen updating code that is in the *mouseMoved()* method.
- Make sure that the *paint()* method sets the active colour to the *drawColour* variable

☞ Re-compile and test your program. Moving the mouse should display a trail of circles. With the left mouse button pressed, any motion should appear to erase the display...



Other Mouse Events

For all other mouse events it is necessary to implement the **MouseListener**. This is also done in a class similar to the **winWatch** class used to monitor for window events. In order for the mouse monitoring class to have easy access to information about the mouse (with respect to its environment) it is often created as an inner class of the applet class. Its format is...

```
class mouseWatch implements MouseListener
{
    public void mousePressed(MouseEvent mEvt){}
    public void mouseReleased (MouseEvent mEvt){}
    public void mouseEntered (MouseEvent mEvt){}
    public void mouseExited (MouseEvent mEvt){}
    public void mouseClicked (MouseEvent mEvt){}
}
```

☞ Add the mouse monitoring class to the applet class.

☞ Add the instruction to register interest in general mouse events to the *init()* method.

The **MouseListener** listener defines five abstract classes that must be included in the class...

mousePressed ()	Tracks mouse down action on any mouse button
mouseReleased()	Tracks mouse up actions on any mouse button
mouseClicked()	Tracks mouse down followed by mouse up actions on any mouse button
mouseEntered()	Tracks when the mouse enters the listening component
mouseExited()	Tracks when the mouse leaves the listening component

Depending on the application, event handling code is added where required. Each method receives a *MouseEvent* object that identifies key information about the mouse and its position. The methods most associated with general mouse events are...

Method		Purpose
int	getButton()	Returns which mouse button was pushed (NOBUTTON, BUTTON1, BUTTON2, BUTTON3)
int	getX()	Returns the current x coordinate position
int	getY()	Returns the current y coordinate position

Now let's modify the program so that when the background is cleared every time the mouse moves outside the application's frame.

☞ Add a boolean variable called *outOfWindow* to the applet class. Its default value should be false.

☞ Add code to the *mouseExited()* method to set the *outOfWindow* flag to true. Issue the *repaint()* instruction to force the *paint()* method to be executed.

☞ Add code to the *mouseEntered()* method to set the *outOfWindow* flag to false. Issue the *repaint()* instruction to force the *paint()* method to be executed.

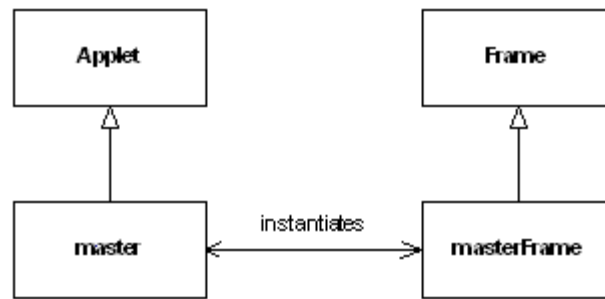
☞ Add code to the *paint()* method that test the *outOfWindow* flag. If it is false the mouse is in the frame and you should draw the circle as normal. If the flag indicates the mouse is out of the frame, clear the frame by painting a filled rectangle (the size of the applet) in the background colour. Just like erasing portions of the screen by painting with the background colour, this is a very common way of 'clearing' the screen.

☞ Re-compile and test your program. Every time you move the mouse off the applet/application window all the circle trails should disappear. Bringing the mouse back over the window should start the regular drawing process again.

Double Buffering

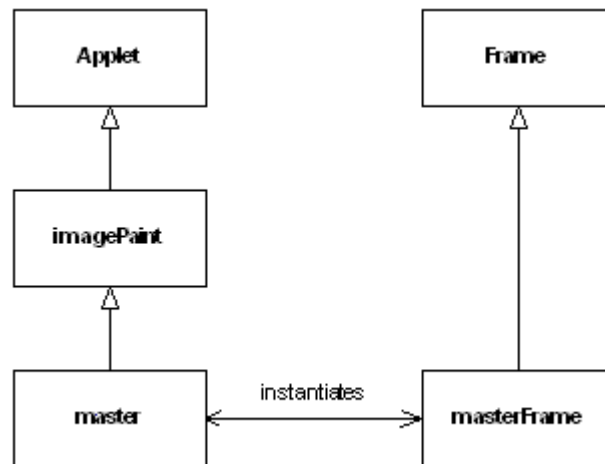
If you watch screen carefully when running the mouse applet/application you'll see some flickering as the mouse moves. If you are trying to move an image around the screen this flickering becomes very noticeable and can be quite disturbing to some users. To prevent this you can use a technique called **double buffering** ... writing the screen to memory before actually displaying it.

Think about what happens when you run a program created using the common applet/application format. The class structure has the **master** class inheriting from the **Applet** class. The **masterFrame** class inherits from the **Frame** class...



If the program is run as an applet, execution starts with the *init()* method. If it runs as an application, the *main()* method in the **master** class creates and shows the frame which in turn calls the *init()* method.

In order to use double buffering and prevent flicker, the class structure has to be modified slightly...



A new **imagePaint** class is added to separate the image processing from the basic functionality of the program. The **master** class inherits from the **imagePaint** class, which in turn inherits from the **Applet** class.

☞ Starting with your master file, search and replace all occurrences of the word '*master*' with '*imageDrag*'. Save your program as **imageDrag.java** in a new folder.

Any time you are working with images (.gif, .jpeg, or .png) you must import the **java.awt.Image** and the **java.awt.*** packages. In addition, because you will be double buffering you need to import the **java.awt.Dimension** package to work with screen elements.

☞ Import the **java.awt.***, **java.awt.Image** and **java.awt.Dimension** packages...

```

import java.awt.Dimension;
import java.awt.Image;
import java.awt.*;
  
```

☞ Modify the **imageDrag** class declaration to show that **imageDrag** inherits from the **imagePaint** class and not the **Applet** class...

```
public class imageDrag extends imagePaint
{
```

☞ Add the **imagePaint** class declaration. Make sure it is outside the **imageDrag**, **imageDragFrame** and **winWatch** classes and that it inherits from the **Applet** class...

```
class imagePaint extends Applet
{
}
```

☞ Move the applet methods (*init()*, *start()*, *paint()*, *stop()* and *destroy()*) from the **imageDrag** class to the **imagePaint** class. Leave the *main()* method in the **imageDrag** class.

☞ Move all but the

```
imageDragFrame app = new imageDragFrame ("imageDrag");
```

instruction from the *main()* method in the **imageDrag** class to the constructor in the **imageDragFrame** class. That leaves the *main()* method looking like...

```
public static void main(String args[])
{
    imageDragFrame app = new imageDragFrame ("imageDrag");
}
```

and the **imageDragFrame** constructor looking something like...

```
class imageDragFrame extends Frame
{
    public imageDragFrame (String frameTitle)
    {
        super(frameTitle);
        imageDrag applet = new imageDrag();
        applet.init();
        add ("Center", applet);

        winWatch wwatch;

        app.setSize(300,300);
        app.show();

        wwatch = new winWatch();
        app.addWindowListener(wwatch);
    }
}
```

Because of this move you must make a few modifications...

☞ Move the *wwatch* declaration statement out of the **imageDragFrame** constructor so that it is a class attribute...

```
class imageDragFrame extends Frame
{
    winWatch wwatch;

    public imageDragFrame (String frameTitle)
    {
```

☞ Because you are now setting the screen size and issuing the show command from the frame you no longer need (or have access to) the frame object '*app*'. Remove the '*app*' object reference from the *setSize()* and *show()* methods...

```
        setSize(300,300);
        show();
```

☞ Move the call to the *init()* method to the end of the constructor. It must be the last instruction otherwise you'll generate a run-time error when you try to write to the screen. The complete **imageDrag** and **imageDragFrame** classes should now look like...

```
public class imageDrag extends imagePaint
{
    public static void main(String args[])
    {
        imageDragFrame app = new imageDragFrame ("imageDrag");
    }
}

class imageDragFrame extends Frame
{
    winWatch wwatch;

    public imageDragFrame (String frameTitle)
    {
        super(frameTitle);
        imageDrag applet = new imageDrag();
        add ("Center", applet);

        setSize(300,300);
        show();

        wwatch = new winWatch();
        app.addWindowListener(wwatch);

        applet.init();
    }
}
```

Class Attributes

Now it's time to populate the **imagePaint** class. First you need two instance attributes that will be used to buffer the contents of the window in memory...

```
Image offScreenImg;  
Graphics offScreenG;
```

The first attribute, *offScreenImg*, holds the buffered image while the second, *offScreenG*, is the device context.

 Add these two attributes to the **imagePaint** class.


You also need attributes to position the image within the frame, to set the background colour and to work with the screen dimensions...

```
int xpos = 100;  
int ypos = 100;  
  
Color bkColor = Color.white;  
  
Dimension s;  
int winWidth;  
int winHeight;
```

 Add these six attributes to the **imagePaint** class.

Finally you need attributes to store and identify the actual image...

```
Image img;  
String imageName = "name.gif";
```

 Find yourself a small .gif, .jpeg or .png image. Store it in the same folder as the imageDrag.java file. Add the two image attributes to the **imagePaint** class. Make sure you substitute the name of your image in for 'name.gif'.

init() Method

There are a few things that need to be initialized as the program begins...

- set the background colour
- register interest in mouse motion events
- determine the actual size of the window
- create an off screen image and a graphics context for it
- get the desired image

☞ Set the background colour and register interest in mouse motion events by adding the following instructions to the *init()* method...

```
setBackground(bkColor);  
this.addMouseMotionListener(new mouseMotionWatch());
```

☞ Determine the window size by adding the following instructions to the *init()* method...

```
s = getSize();  
winWidth = s.width;  
winHeight = s.height;
```

☞ Create an off screen image and a graphics context for it by adding the following instructions to the *init()* method ...

```
offScreenImg = createImage(winWidth, winHeight);  
offScreenG = offScreenImg.getGraphics();
```

☞ Finally get the actual image so you are ready to paint it. Add the following instructions to the *init()* method...

```
Toolkit t = Toolkit.getDefaultToolkit();  
img = t.getImage(imageName);
```

The first instruction is required because your program is designed to run as either an applet or an application. It determines what windowing toolkit is currently being used. Once it knows applicable toolkit it can load in the desired image.

Note: If you are just writing an applet you can use the *getImage()* and *getDocumentBase()* methods first examined in the Java Applets chapter...

```
img = getImage(getDocumentBase(), imageName);
```

***paint()* Method**

Now that the off screen buffer is created the desired display can be added to it by...

- setting the drawing colour to the background colour
- clear the background by drawing a filled rectangle the size of the window
- drawing the image at the specified position

☞ Prepare the off screen display by adding the following instructions to the *paint()* method...

```
offScreenG.setColor(bkColor);  
offScreenG.fillRect(0,0,winWidth, winHeight);  
offScreenG.drawImage(img,xpos,ypos,this);
```

Recall that the last parameter in the *drawImage()* method parameter list identifies what object will be notified when the image is loaded. In this case 'this' specifies that the *paint()* method is to be notified.

Finally it is time to copy the off screen buffer from memory to the screen. The instruction...

```
this.getGraphics().drawImage(offScreenImg, 0, 0, this);
```

creates a graphics context for either the applet or application screen and then draws the off screen image onto the screen starting at the top left corner. Once again the *paint()* method is to be notified that the image is loaded.

☞ Add the instruction to copy the off screen buffer to the screen.

***update()* method**

☞ Add the *update()* method to override the default method that clears the screen with each repaint...

```
public void update(Graphics g)
{
    paint(g);
}
```

Mouse Motion Events

The last stage in dragging or moving an image around the screen is to provide the mouse motion event handling code. You have already registered interest in the mouse motion events so all that is left is to add the inner class responsible for handling the events...

```
class mouseMotionWatch implements MouseMotionListener
{
    public void mouseMoved(MouseEvent mEvt){}

    public void mouseDragged(MouseEvent mEvt)
    {
        xpos = mEvt.getX()-10;
        ypos = mEvt.getY()-15;

        repaint();
    }
}
```

When the image is dragged across the screen the mouse's current (x, y) position is used to center the image at that location. Say for example the image is 20 pixels wide and 30 pixels high, and the mouse is currently at position (150, 200). You want the image centered at this position so you want half the image to the left of this point and half the image to the right of this point. The desired x coordinate for the image is therefore $150 - 20/2$ or 140. Similarly the desired y coordinate is $200 - 30/2$ or 185. The *repaint()* instruction will cause the image to be redrawn at these new coordinates.

☞ Add the mouse motion event handling code. Make sure the **mouseMotionWatch** class is defined as an inner class of the **imagePaint** class and that you change the calculated x and y positions to reflect the size of your image.

The complete code should now look something like...

```
import java.applet.Applet;
import java.awt.Frame;
import java.awt.event.*;
import java.awt.Dimension;
import java.awt.Image;
import java.awt.*;

public class imageDrag extends imagePaint
{
    public static void main(String args[])
    {
        imageDragFrame app = new imageDragFrame ("imageDrag");
    }
}

class imageDragFrame extends Frame
{
    winWatch wwatch;

    public imageDragFrame (String frameTitle)
    {
        super(frameTitle);
        imageDrag applet = new imageDrag();
        add ("Center", applet);

        setSize(300,300);
        show();

        wwatch = new winWatch();
        app.addWindowListener(wwatch);

        applet.init();
    }
}

class winWatch implements WindowListener
{
    public void windowClosing(WindowEvent wEvt)
    {
        System.exit(0);
    }

    public void windowClosed(WindowEvent wEvt){}
    public void windowOpened(WindowEvent wEvt){}
    public void windowIconified(WindowEvent wEvt){}
    public void windowDeiconified(WindowEvent wEvt){}
    public void windowActivated(WindowEvent wEvt){}
    public void windowDeactivated(WindowEvent wEvt){}
}
}
```



```

class imagePaint extends Applet
{
    Image offScreenImg;
    Graphics offScreenG;

    int xpos = 100;
    int ypos = 100;

    Color bkColor = Color.white;

    Dimension s;
    int winWidth;
    int winHeight;

    Image img;
    String imageName = "off.gif";

    public void init()
    {
        setBackground(bkColor);

        this.addMouseMotionListener(new mouseMotionWatch());

        s = getSize();
        winWidth = s.width;
        winHeight = s.height;

        offScreenImg = createImage(winWidth, winHeight);
        offScreenG = offScreenImg.getGraphics();

        Toolkit t = Toolkit.getDefaultToolkit;
        img = t.getImage(imageName);
    }

    public void update(Graphics g)
    {
        paint(g);
    }

    public void paint(Graphics g)
    {
        offScreenG.setColor(bkColor);

        offScreenG.fillRect(0,0,winWidth, winHeight);

        offScreenG.drawImage(img,xpos,ypos,this);

        this.getGraphics().drawImage(offScreenImg, 0, 0, this);
    }
}


```

```

class mouseMotionWatch implements MouseMotionListener
{
    public void mouseMoved(MouseEvent mEvt){}
    public void mouseDragged(MouseEvent mEvt)
    {
        xpos = mEvt.getX()-10;
        ypos = mEvt.getY()-10;

        repaint();
    }
}

```

 Compile and test your program. You should be able to click on the image and drag it to any location within the window.

Keyboard Events

Depending on your application there may be times when it is necessary to monitor for keyboard events. Like the mouse events, it is just a matter of monitoring for activity.

Monitoring for Keyboard Activity

There is just one listener associated with keyboard activity ... **KeyListener**. This listener receives notification every time any key (or combination of keys) is pressed. The process is the same as it has been for all component and mouse events ...

- implement the appropriate listener
- register interest in the event
- provide the event handling code


 Starting with your master file, search and replace all occurrences of the word '*master*' with '*moveLetter*'. Save your program as **moveLetter.java** in a new folder.

To track keyboard events it is necessary to implement the **KeyListener**. This is typically done in a class similar to the **winWatch** class used to monitor for window events. As with the mouse monitoring classes, the keyboard monitoring class is usually created as an inner class of the applet class. Its format is...

```

class keyWatch implements KeyListener
{
    public void keyPressed(KeyEvent kEvt){}
    public void keyReleased(KeyEvent kEvt){}
    public void keyTyped(KeyEvent kEvt){}
}

```

 Add the **keyWatch** class to your file. Make sure it is defined as an inner class of the **Applet** class.

The **KeyListener** listener defines three abstract classes that must be included in the class ... *keyPressed()*, *keyReleased()* and *keyTyped()*. Depending on the application, event handling code is added where required. Each method receives a *KeyEvent* object that identifies information about keyboard action. The methods most associated with keyboard events are...

Method		Purpose
char	getKeyChar()	Returns the character associated with the pressed key
int	getKeyCode()	Returns the integer key code of the pressed key
String	getKeyText()	Returns a string describing the key code such as "HOME", "F1" or 'a'
void	setKeyChar(char <i>character</i>)	Sets the key to the specified character

Java uses virtual key codes to represent each key on the keyboard. Each key code starts with **VK_** and is followed by the actual key specifier. The virtual key codes for the more commonly used keys are...

VK_A ... VK_Z	Upper case letters
VK_a ... VK_z	Lower case letters
VK_0 ... VK_9	Number keys
VK_SHIFT, VK_ALT, VK_CONTROL	Shift, alt and control keys
VK_F1 ... VK_F12	Function keys
VK_DOWN, VK_UP, VK_LEFT, VK_RIGHT	Arrow keys

Please refer to the documentation for the full set of virtual key codes.

A number of events are triggered when a key is pressed. Say for example the key **N** is pressed. The key listener class is notified of the following sequence of events...

```

KEY_PRESSED    VK_SHIFT
KEY_PRESSED    VK_N
KEY_TYPED      N
KEY_RELEASED   VK_N

```

As the programmer you can monitor and respond to any or all of these events.

The **moveLetter** program you are about to create will move a specified letter left and right on the screen using the arrow keys. If the control key is held down it will move faster. First you need to define an attribute for the font and the letter that will be displayed...

```

private Font f = new Font("Serif", Font.BOLD, 36);
private char letter = 'A';

```

This font will give you a fairly large letter that in this case is initialized to an upper case A ... you can initialize yours to whatever you'd like.

You also need constants and a variable that will allow you to change between fast and slow speeds...

```
public static int SLOW = 2;
public static int FAST = 10;

private int deltaX = SLOW;
```

These two constant values will actually define the number of pixels that the letter will move each time a left or right arrow key is pressed. The variable is initialized to the slower speed which is the default speed.


Finally you need variables to position the letter in the window. In this case the letter will start at position (100, 100) in the window...

```
private int x = 100;
private int y = 100;
```

 Add these attributes to the applet class of the **moveLetter** program.

Not much has to happen during initialization. You need to...


- set the font
- register interest in the keyboard events
- set focus to the applet

 Add the following instructions to the *init()* method...

```
setFont(f);
this.addKeyListener(new keyWatch());
requestFocus();
```

The *paint()* method is also very simple. The *drawString()* method is used to draw the letter onto the screen at the specified coordinates. Since the letter is a character and the *drawString()* method requires a *String* object, the character must be converted to a *String* first. This is done using the *String* method *valueOf()*...

```
g.drawString(String.valueOf(letter), x, y);
```

 Add the *drawString()* instruction to the *paint()* method.

Now for the event handling. You have to watch for ...

- a letter typed --- it will be displayed and moved around
- the control key pressed --- it will increase the speed of movement
- the control key released --- it will return the speed of movement to normal
- left arrow key pressed --- move letter to left
- right arrow key pressed --- move letter to right

There are three ‘key pressed’ events that need to be handled. These are all processed in the *keyPressed()* method of the **keyWatch** class. The *KeyEvent* object specifies the key code for the key pressed. The *getKeyCode()* method is used to determine which key it was...

```
int code = e.getKeyCode();
```

The returned value can then be compared to the **VK_CONTROL**, **VK_LEFT** and **VK_RIGHT** constants defined by the **KeyEvent** class...

```
if (code == KeyEvent.VK_CONTROL)
    deltaX = FAST;
else if (code == KeyEvent.VK_RIGHT)
{
    . . .
}
```

☞ Add code to handle the control, left and right keys to the *keyPressed()* method. If the left arrow key takes the letter off the left edge of the window have it pop in the right side. Similarly if the right arrow key takes the letter off the right edge of the window have it pop in the left side.

The only key you need to monitor for release is the control key. This code goes in the *keyReleased()* method.

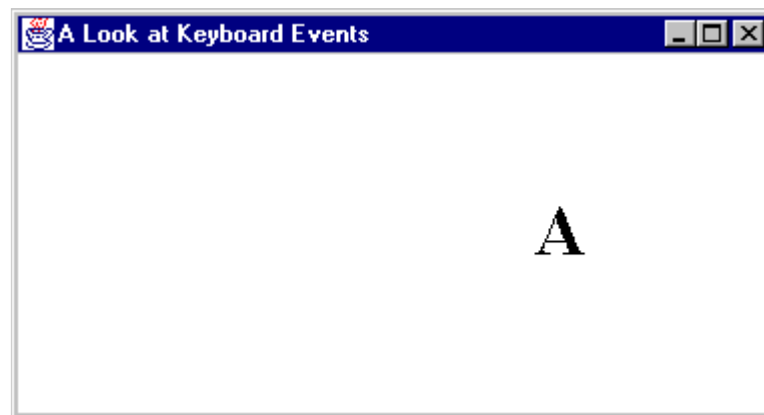
☞ Add code to the *keyReleased()* method that changes the speed back to SLOW when the control key is released. Make sure you check for the control key ... you don’t want the speed changing with the release of any other key.

Finally you need to check for a typed character. This code goes in the *keyTyped()* method. The *KeyEvent* object specifies which character was typed and the *getKeyChar()* method is used to determine which character it was...

```
letter = e.getKeyChar();
```

☞ Add code to the *keyTyped()* method that assigns the typed character to the *letter* attribute. Make sure you include the *repaint()* instruction so the letter gets changed on your display.

☞ Compile and test your code. Your output should look something like ...



Menus

Menus are made up of three distinct parts... the *menu bar* which holds the main *menu* labels which in turn holds specific *menu items*. Each of these components is included in the *java.awt* package. Anytime you are using menus in your applet or application you must import the *Menu* component by including the following ...

```
import java.awt.Menu;
```

Normally if you are using more than one type of component in your applet you can use one import statement that includes all the components...

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

but occasionally you'll get a platform where you have to import the Menu package individually.

Constructors

As with any object each part of a menu must be instantiated before it can be used in a program. There is just one constructor available for a *MenuBar* component...

```
MenuBar()
```

This default constructor creates a *MenuBar* object that can be bound to a **Frame**.

The *Menu* component, a pull down menu item that is part of a menu bar, has three constructors...

```
Menu()
```

```
Menu(String label)
```

```
Menu(String label, boolean tearOff)
```

The default constructor creates a new menu with no label. The second constructor creates a menu with a specified label, while the third constructor creates a menu with a specified label that can or cannot be torn off meaning it can be pulled off and dragged away from the parent menu bar or menu.

The *MenuItem* component, an option within a *Menu*, has three constructors...

```
MenuItem()
```

```
MenuItem(String label)
```

```
MenuItem(String label, MenuShortcut shortcut)
```

The default constructor creates a new menu item with no label. The second constructor creates a menu item with a specified label, while the third constructor creates a menu with a specified label and an associated keyboard shortcut.

Building a Menu

The first step in building a menu is to define each of the components ... the menu bar itself, each of the main menu options and each of the sub menu items. First you create the menu bar ...

```
MenuBar menuBarName = new MenuBar();
```

Next you define a main menu object for each option that will be included in the menu bar...

```
Menu menuName = new Menu("label");
```

Finally you define a menu item object for each item that will be included in each main menu option...

```
MenuItem menuItemName = new MenuItem("label");
```


Because the menu is attached to the frame, all of these declarations must be made in the **Frame** class of your program.

Say for example you want to set up a menu bar that includes 'File' and 'Edit' options. The 'File' menu must have 'Open', 'Save' and 'Exit' options, while the 'Edit' menu needs 'Undo' and 'Find' options. The component definitions would be...

```
MenuBar mBar = new MenuBar();

Menu fileMenu = new Menu("File");
Menu editMenu = new Menu("Edit");

MenuItem fileOpen = new MenuItem("Open");
MenuItem fileSave = new MenuItem("Save");
MenuItem fileExit = new MenuItem("Exit");
MenuItem editUndo = new MenuItem("Undo");
MenuItem editFind = new MenuItem("Find");
```

 Add code to your **moveLetter.java** file to define a menu that allows you change the speed and the desired letter. The Speed menu should have a Fast and a Slow option. Put a few alternate letters in the Letter menu. Remember to put these definitions in the **Frame** class.

Once the individual components are defined you can construct the actual menu. First the menu bar gets bound to the frame ...

```
setMenuBar(menuBarName);
```

The main menu options get added to the menu bar...


```
menuBarName.add(menuName);
```

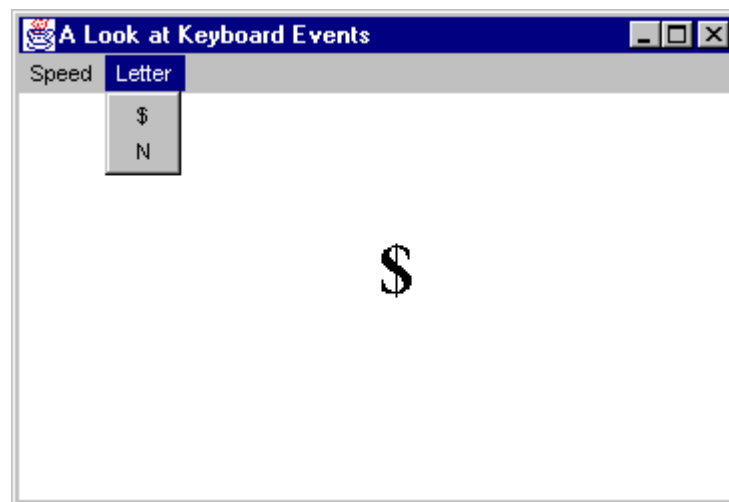
and the individual menu items get added to the main menu options ...

```
menuName.add(menuItemName);
```

For the File/Edit menu defined above, the menu would be constructed with the following set of instructions...

```
setMenuBar (mBar) ;  
  
mBar.add(fileMenu) ;  
mBar.add(editMenu) ;  
  
fileMenu.add(fileOpen) ;  
fileMenu.add(fileSave) ;  
fileMenu.add(fileExit) ;  
  
editMenu.add(editUndo) ;  
editMenu.add(editFind) ;
```

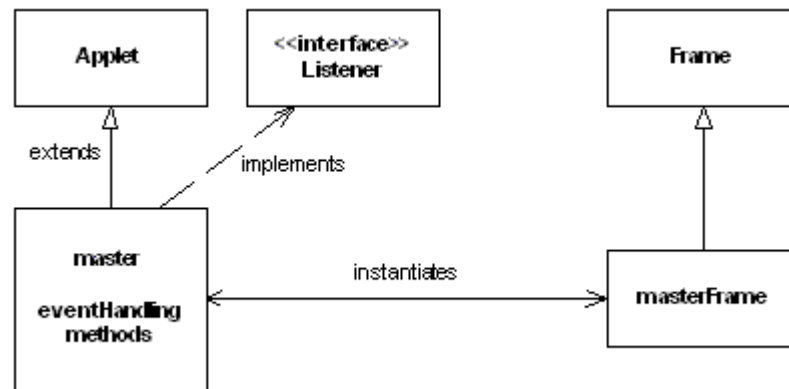
 Add code to your **moveLetter.java** file to create your menu. Your screen should look something like...



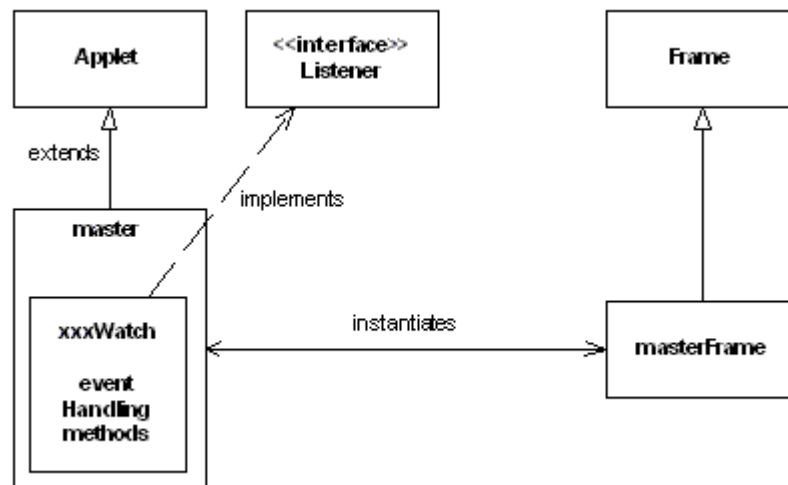
Menu Events

Up to this point events have been handled in one of two ways...

1. implementing the appropriate listener in the **Applet** class and including the required listener methods as part of the applet



2. creating an inner listening class that implements the appropriate listener and includes the required listener methods



Menu events have to be handled a little bit differently. The menu itself is bound to the frame but the menu events are processed by the applet. Neither of the two event handling methods examined so far allow for this. This is where adapters come in.

Adapters

Adapters are objects that act as a liaison between two other objects whose interfaces are incompatible. This liaison allows the two objects to communicate with each other. It also allows the event handling code to be separated from the application's code.

Simple Adapters

You have already been using a simple adapter ... the **winWatch** class acts a liaison between the window events in the GUI and the functional part of the application...

```
class winWatch implements WindowListener
{
    public void windowClosing(WindowEvent wEvt)
    {
        System.exit(0);
    }
    public void windowClosed(WindowEvent wEvt){}
    public void windowOpened(WindowEvent wEvt){}
    public void windowIconified(WindowEvent wEvt){}
    public void windowDeiconified(WindowEvent wEvt){}
    public void windowActivated(WindowEvent wEvt){}
    public void windowDeactivated(WindowEvent wEvt){}
}
```

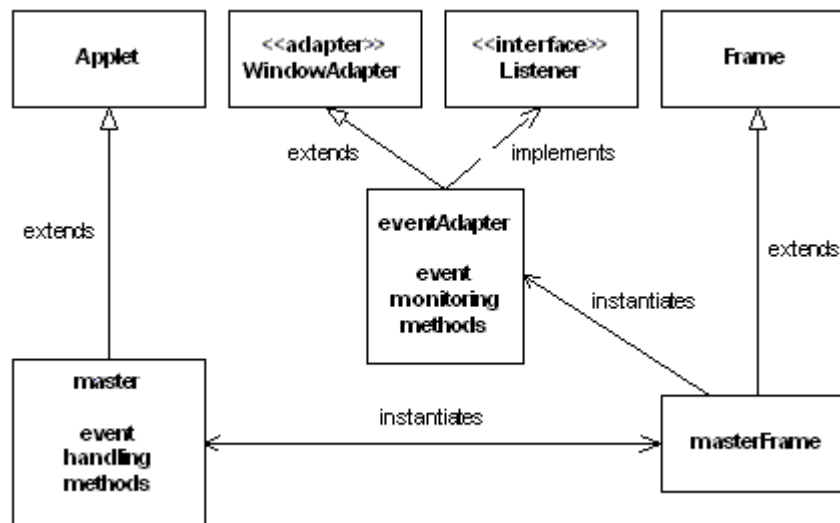
Right now though, it only 'listens' to events that happen in the frame. This is because the application creates an instance of the **winWatch** class and registers interest in window events specifying *wWatch* as the object that will respond to any window events...

```
wWatch = new winWatch();
this.addWindowListener(wWatch);
```

It also has to define every method included in the **WindowListener** even though only one of these methods is actually used.

Event Adapter Class

One of the more common ways to address these problems with the **winWatch** class as well as to provide a way for handling events that must be linked to both the frame and the applet, is to reorganize the class structure...



First you can see that the event handling methods have been differentiated from the event monitoring methods. The event monitoring methods have been extracted from the applet class and placed in a separate **eventAdapter** class. This allows event monitoring to come from either the applet (components, the mouse and keys) or the application's frame (menus). Regardless of whether the event is triggered from the applet or the frame, the event handling should be common. That is why they are included only once, in the applet class.

This **eventAdapter** class inherits from an abstract adapter class called **WindowAdapter** that is set up to receive window events. It implements the **WindowListener** so if you create an event handling class that inherits from the **WindowAdapter**, you only have to include methods for the window events you choose to monitor.

The **eventAdapter** class must also implement any other listeners that are required to handle component, mouse or keyboard events.

Menu Events

Menu events require implementation of the **ActionListener**. Recall that it has one abstract method ... *actionPerformed()*. Using the *getSource()* method you can determine which menu item triggered the event...

```
public void actionPerformed(ActionEvent d)
{
    if (e.getSource() == frameObject.menuItemName)
        . . .
```

For example to see if the *fileOpen* menu item was selected in the application frame called *frame* the instruction is...

```
if (e.getSource() == frame.fileOpen)
    . . .
```


EventAdapter Class

Your **moveLetter.java** program already has methods in place to handle keyboard events. You have also added menu items that trigger action events. All of this event monitoring, plus the window event monitoring will be done by the **eventAdapter** class. First let's create the basic **eventAdapter** class...

```
class EventAdapter extends WindowAdapter
{
    moveLetter applet;
    moveLetterFrame frame;

    EventAdapter(moveLetter applet, moveLetterFrame frame)
    {
        this.applet = applet;
        this.frame = frame;
    }
}
```

The class has two attributes... one to provide a link to the applet and the other to provide a link to the application frame. Its constructor is quite simple. It receives two objects... one an instance of the applet and the other an instance of the application's frame. These two parameters are assigned to the appropriate attributes.

 Comment out or remove the **winWatch** class from your **moveLetter.java** file. Add the **EventAdapter** class to the end of the file. Make sure it is outside the applet and the application class definitions.

Monitoring for Window Events


You can see from the class diagram and the class definition that the **EventAdapter** class inherits from the **WindowAdapter** class. This adapter provides default code for each of the window events defined in the **WindowListener** ...you can override any of these methods if you have specific tasks you want executed. Your applications specify tasks for the *WindowClosing* event. Adding this to the **EventAdapter** class gives you...

```
class EventAdapter extends WindowAdapter
{
    moveLetter applet;
    moveLetterFrame frame;

    EventAdapter(moveLetter applet, moveLetterFrame frame)
    {
        this.applet = applet;
        this.frame = frame;
    }

    public void windowClosing(WindowEvent wEvt)
    {
        applet.quit();
    }
}
```

You can see that the actual event handling is left to the applet ... here it simply passes control to a method in the applet called *quit()*. You can add this method later.

 Add the *windowClosing()* method to your **EventAdapter** class.

Monitoring for Menu Events

All menu items for which interest has been registered need to be monitored in the **EventAdapter** class. Since notification is sent to the **ActionListener** you must add two things to your **EventAdapter** class...

- implement the **ActionListener**
- add the event monitoring code in the *actionPerformed()* method

The **EventAdapter** class now looks like...

```
class EventAdapter extends WindowAdapter implements ActionListener
{
    moveLetter applet;
    moveLetterFrame frame;

    EventAdapter(moveLetter applet, moveLetterFrame frame)
    {
        this.applet = applet;
        this.frame = frame;
    }

    public void windowClosing(WindowEvent wEvt)
    {
        applet.quit();
    }

    public void actionPerformed(ActionEvent e)
    {
        if (e.getSource() == frame.speedSlow)
            applet.changeSpeed(moveLetter.SLOW);

        else if (e.getSource() == frame.speedFast)
            applet.changeSpeed(moveLetter.FAST);

        else if (e.getSource() == frame.letter$)
            applet.changeLetter('$');

        else if (e.getSource() == frame.letterN)
            applet.changeLetter('N');
    }
}
```

The source of the action event is tested against each of the menu items (two to change the speed and two to change the letter). Notice that once again, the actual event handling is left to the applet ... in this case to two methods *changeSpeed()* and *changeLetter()*. The event monitoring code simply calls the appropriate event handling method in the applet.

 Add the *actionPerformed()* method to your **EventAdapter** class. Don't forget to implement the **ActionListener**.

Monitoring for Keyboard Events

All keyboard events for which interest has been registered also need to be monitored in the **EventAdapter** class. Since notification is sent to the **KeyListener** you must add two things to your **EventAdapter** class...

- implement the **KeyListener**
- add the event monitoring code in the *keyPressed()*, *keyReleased()* and *keyTyped()* methods

The **EventAdapter** class is now complete and looks like...

```
class EventAdapter extends WindowAdapter implements ActionListener, KeyListener
{
    moveLetter applet;
    moveLetterFrame frame;

    EventAdapter(moveLetter applet, moveLetterFrame frame)
    {
        this.applet = applet;
        this.frame = frame;
    }

    public void windowClosing(WindowEvent wEvt)
    {
        applet.quit();
    }

    public void actionPerformed(ActionEvent e)
    {
        if (e.getSource() == frame.speedSlow)
            applet.changeSpeed(moveLetter.SLOW);
        else if (e.getSource() == frame.speedFast)
            applet.changeSpeed(moveLetter.FAST);
        else if (e.getSource() == frame.letter$)
            applet.changeLetter('$');
        else if (e.getSource() == frame.letterN)
            applet.changeLetter('N');
    }

    public void keyPressed(KeyEvent e)
    {
        int code = e.getKeyCode();

        if (code == KeyEvent.VK_CONTROL)
            applet.changeSpeed(moveLetter.FAST);

        else if (code == KeyEvent.VK_RIGHT)
            applet.moveLetter(moveLetter.RIGHT);

        else if (code == KeyEvent.VK_LEFT)

            applet.moveLetter(moveLetter.LEFT);
    }

    public void keyReleased(KeyEvent e)
    {
        if (e.getKeyCode() == KeyEvent.VK_CONTROL)
            applet.changeSpeed(moveLetter.SLOW);
    }

    public void keyTyped(KeyEvent e)
    {
        applet.changeLetter(e.getKeyChar());
    }
}
```

These methods are slightly different than what you had before ...the testing is the same but again the actual event handling is left to the applet ... the same two methods *changeSpeed()* and *changeLetter()*. The event monitoring code is different but the event handling methods provided in the applet are the same.

☞ Add the keyboard event monitoring methods to the **EventAdapter** class. Don't forget to implement the **KeyListener**.

Registering Interest

Interest must still be registered for the keyboard, menu and window events. The keyboard events are registered in the applet while the menu and window events are registered in the application's frame. In all cases the event listener will be an instance of the **EventAdapter** class. This object is created in the application's frame...

```
EventAdapter adapter = new EventAdapter(applet, this);
```

Because it passes a reference to the applet and the frame to the adapter, this instruction must be placed after the frame creates the applet object. It must also be placed before you try to register interest in either the menu or the window events.

To register interest in the window events you must use the *addWindowListener()* method...

```
this.addWindowListener(adapter);
```

The keyword '*this*' indicates that the frame is the originator of the window events. The *adapter* object will receive notification of these events.

Each individual menu item must register interest in its events. The *addActionListener()* method is used...

```
menuItemName.addActionListener(adapter);
```

where *menuItemName* is the actual menu item name.

☞ Add the instructions to register interest in each menu item and the window events to the application frame's constructor. The **moveLetterFrame** class is now complete and should look something like this...

```
class moveLetterFrame extends Frame
{
    MenuBar mBar = new MenuBar();
    Menu speedMenu = new Menu("Speed");
    Menu letterMenu = new Menu("Letter");
    MenuItem speedFast = new MenuItem("Fast");
    MenuItem speedSlow = new MenuItem("Slow");
    MenuItem letter$ = new MenuItem("$");
    MenuItem letterN = new MenuItem("N");
}
```

```

public moveLetterFrame (String frameTitle)
{
    super(frameTitle);

    moveLetter applet = new moveLetter();
    EventAdapter adapter = new EventAdapter(applet, this);
    applet.init(adapter);
    add ("Center", applet);

    setMenuBar (mBar);
    mBar.add(speedMenu);
    mBar.add(letterMenu);
    speedMenu.add(speedFast);
    speedMenu.add(speedSlow);
    letterMenu.add(letter$);
    letterMenu.add(letterN);

    speedFast.addActionListener(adapter);
    speedSlow.addActionListener(adapter);

    letter$.addActionListener(adapter);
    letterN.addActionListener(adapter);

    this.addWindowListener(adapter);
}
}

```

Keyboard events are registered by the applet. As before, this is done in the *init()* method. To register the keyboard events you must use the *addKeyListener()* method...

```
this.addKeyListener(adapter);
```

The keyword '*this*' indicates that the applet is the originator of the keyboard events. The *adapter* object will receive notification of these events. Because the application created the adapter object, the applet won't know it exists unless the application tells it. This is done when the application calls the applet's *init()* method...

```
applet.init(adapter);
```

The applet's *init()* method must be modified accordingly ... an *EventAdapter* object must be added to the parameter list...

```

public void init(EventAdapter adapter)
{
    setFont(f);
    this.addKeyListener(adapter);
    requestFocus();
}

```

Now when the method registers interest in the keyboard events it is able to specify the external adapter class.

☞ Modify the call to the *init()* method in the application's constructor. Modify the *init()* method declaration in the applet class.

Code for Handling Events

All the event handling code is part of the applet class. Right now the keyboard event handling code is in the **keyWatch** class, an inner class of the applet class. This **keyWatch** class is no longer needed, but some its code is. As you can see from the *EventAdapter* class you must provide the following event handling methods in the applet class...

```
void quit( )

void changeSpeed(int newSpeed)

void changeLetter(char newLetter)

void moveLetter(int direction)
```

☞ Add the *quit()* method to your applet class. It will contain the `System.exit(0);` instruction that was originally in the *windowClosing()* method in the **winWatch** class.

☞ Add the *changeSpeed()* method to the applet class. It will contain code from the control key portion of the *keyPressed()* and *keyReleased()* methods in the **keyWatch** class. Modify it to accept an integer value from the calling method.

☞ Add the *changeLetter()* method to the applet class. It will contain code from the *keyTyped()* method in the **keyWatch** class. Modify it to accept a character integer value from the calling method.

☞ Add the *moveLetter()* method to the applet class. It will contain code from the arrow key portions of the *keyPressed()* method in the **keyWatch** class. Modify it to accept an integer value from the calling method.

☞ If you haven't already removed the **keyWatch** class, do it now. The applet class is now complete and should look something like...

```
public class moveLetter extends Applet
{
    //put variable declarations here
    public static int SLOW = 2;
    public static int FAST = 10;
    public static int SIZE = 500;
    public static int LEFT = 1;
    public static int RIGHT = 2;

    private int x = 100, y = 100;
    private char letter = 'A';
    private Font f = new Font("Serif", Font.BOLD, 36);
    private int deltaX = SLOW;
```

```

public static void main(String args[])
{
    moveLetterFrame frame = new moveLetterFrame ("Keyboard Events");
    frame.setSize(SIZE, SIZE);
    frame.show();
}

public void init(EventAdapter adapter)
{
    setFont(f);
    this.addKeyListener(adapter);
    requestFocus();
}

public void paint (Graphics g)
{
    g.drawString(String.valueOf(letter), x, y);
}


public void quit()
{
    System.exit(0);    //exit on System exit box clicked
}

public void changeSpeed(int newSpeed)
{
    deltaX = newSpeed;
}

public void changeLetter(char newLetter)
{
    letter = newLetter;
    repaint();
}

public void moveLetter(int direction)
{
    if (direction == RIGHT)
    {
        if (x + deltaX > SIZE)
            x = 0;
        else
            x += deltaX;
    }
    else if (direction == LEFT)
    {
        if (x - deltaX < 0)
            x = SIZE-20;
        else
            x -= deltaX;
    }
    repaint();
}
}

```

 Compile and test your program. You should be able to change the speed using either the control key or the menu. You should also be able to change the letter using either the keyboard or the menu.

Exercise – Modified Moving Letter

Modify the **moveLetter.java** program so that in addition to what it does now it also ...

- uses the up and down arrow keys to move the letter up and down. If the letter moves off the screen it should pop in from the other side
- has menus to set the direction to up, down, left or right
- uses the space bar to move in the specified direction (make sure you have a default direction specified so that the space bar works even if you haven't

Containers and Layouts

The absolute layout was introduced a few chapters back when it was necessary to place components at specific locations on the screen. This is just one of the layout classes provided in the **LayoutManager** interface.

Containers

Recall that any container object such as a **Window**, **Frame**, **Applet** or **Panel** has a *setLayout()* method that establishes communication between a layout manager object and the container object. The layout manager is notified each time a component is added to the container. It keeps a list of the components as well as any constraints on where they should appear within the container.

When the container is establishing the position of its components it relies on the **LayoutManager** to calculate the appropriate position for each. The **LayoutManager** does this by querying each object for its preferred size and its minimum size. These returned values are then used to determine final screen positions.

Layout Manager Classes

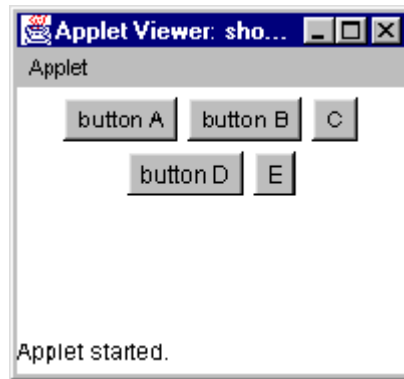
There are five additional layout classes provided in the **LayoutManager** interface...

Flow	Controls flow left to right. Moves to new line when required.
Border	Controls are added to the north, south, east, west and center of the container. Any space not defined goes to the center.
Grid	Grid n rows by m columns made up of equal size rectangles. Controls are added left to right starting with the top row.
Card	Container appears as a stack of cards with only one card visible at a time.
GridBag	Complex grid where each item can take up several rows or columns.

The more commonly used layouts are the **Border** and the **Grid** classes.

Flow Layout

If you choose to use the flow layout, the screen is very similar to what you saw with no layout specified at all. Say for example you have five buttons components to place on the screen. The order the buttons displayed on the screen is the same order that they are added to the container...



Unless otherwise specified, the component takes up only the space required. Notice that the 'C' and 'E' buttons are only big enough to display the specified label.

Constructors

There are three constructors for the **FlowLayout** class...

```
FlowLayout()
```

```
FlowLayout(int alignment)
```

```
FlowLayout(int alignment, int horizontalGap, int verticalGap)
```

The default constructor creates a new **FlowLayout** object where all components will be centered across the container. As many elements as possible will be placed across the screen. The second constructor allows an alignment other than center. The options are...

CENTER	Each row of components is aligned to the center of the container
LEFT	Each row of components is aligned with the left edge of the container
RIGHT	Each row of components is aligned with the right edge of the container
LEADING	Each row of components is aligned with the leading edge of the container
TRAILING	Each row of components is aligned with the trailing edge of the container

The third constructor allows you to define the alignment and specify a vertical and horizontal gap other than the default 5-unit gap.

☞ Starting with your **master.java** file, replace all occurrences of the word ‘*master*’ with the word ‘*showFlow*’. Save your file in a new folder as **showFlow.java**.

☞ Add the following instruction to the *init()* method...

```
setLayout(new FlowLayout());
```

This creates a new **FlowLayout** object and tells the container that the components will flow left to right across the display area, wrapping to the next ‘line’ when necessary.

☞ Create the five button objects shown in the sample screen.

Adding Components

☞ Add the five buttons to the container using the *add()* method provided in the **Container** class. Remember these instructions are part of the *init()* method. For example..

```
add(buttonA);
```

adds the first button to the container.

Displaying the Layout

☞ Add the following instruction after all the components are added to the container...

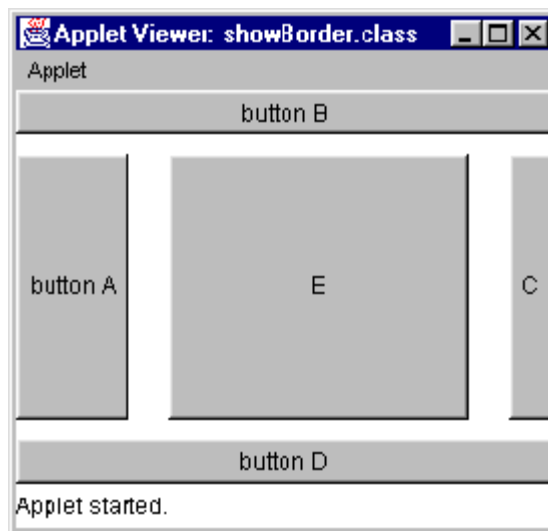
```
setVisible(true);
```

This makes the container visible.

☞ Compile and test your program. Depending on the size of your window, you should see something like the sample diagram.

Border Layout

The border layout divides the screen into five working areas ... one along the left or west edge, another along the right or east edge, a third along the top or north edge, the fourth along the bottom or south edge, and the last in the center. If one of the five buttons is added to each working area of the screen, the display might look like...



The east and west areas are sized to the width of the components they contain. The north and south areas are sized to the height of the components they contain. All remaining space is left to the center.


Constructors


There are two constructors for the **BorderLayout** class...

```
BorderLayout()
```

```
BorderLayout (int horizontalGap, int verticalGap)
```

The default constructor creates a new **BorderLayout** object with no gaps between the five working areas. The second constructor allows both the horizontal gap between the west, center, and east areas, and the vertical gap between the north, center and south areas to be specified.

 Starting with your **master.java** file, replace all occurrences of the word '*master*' with the word '*showBorder*'. Save your file in a new folder as **showBorder.java**.

 Add the following instruction to the *init()* method...

```
setLayout(new BorderLayout(20, 10));
```

This creates a new **BorderLayout** object and tells the container that the components will be placed in the north, south, east, west and center working areas. The gap between the vertical elements is 10 units and the gap between the horizontal elements is 20 units.


Adding Components

Now when adding components to the container it is necessary to specify which of the working areas the component will be placed. The *add()* method requires both the component object name plus a specifier for the working area. These specifiers are defined as constants in the **BorderLayout** class...

NORTH	Place in top working area
SOUTH	Place in bottom working area
EAST	Place in right working area
WEST	Place in left working area
CENTER	Place in center working area

The format of the *add()* method is...


```
add(componentName, workingAreaSpecifier);
```

 Add the five buttons to the container using the *add()* method provided in the **Container** class. Remember these instructions are part of the *init()* method. For example...

```
add(buttonA, BorderLayout.EAST);
```


adds the first button to the right working area of the container.

Displaying the Layout

 Add the following instruction after all the components are added to the container...

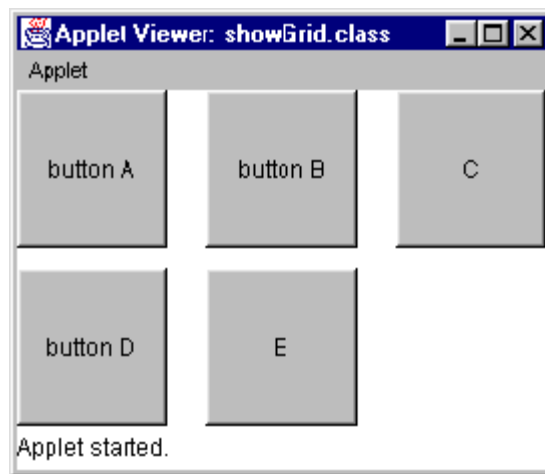
```
setVisible(true);
```

This makes the container visible.

 Compile and test your program. You should clearly see the divisions between the five working areas as in the sample diagram.

Grid Layout

The grid layout divides the screen into a fixed number of rows and columns. Each cell holds one component and they are all the same size. To display the five buttons a grid 2x3 or 3x2 is required. This sample screen shows a 2 row by 3 column grid...



If no horizontal or vertical gap is defined the rows and columns touch each other.

Constructors

There are three constructors for the **GridLayout** class...

```
GridLayout()
```

```
GridLayout(int rows, int columns)
```

```
GridLayout(int rows, int columns, int horizontalGap, int verticalGap)
```

The default constructor creates a new **GridLayout** object with one column and one row. The second constructor specifies the number of rows and columns. The third constructor specifies a fixed size grid as well as the horizontal spacing between the columns and the vertical spacing between the rows.

✎ Starting with your **master.java** file, replace all occurrences of the word ‘*master*’ with the word ‘*showGrid*’. Save your file in a new folder as **showGrid.java**.

✎ Add the following instruction to the *init()* method...

```
setLayout(new GridLayout(2, 3, 20, 10));
```

This creates a new **GridLayout** object and tells the container that the components will be placed in a 2 row by 3 column grid. The gap between the vertical elements is 10 units and the gap between the horizontal elements is 20 units.

Adding Components

Components are added to the container in order starting with the top left corner of the grid. Like adding components to the **FlowLayout**, the `add()` method only requires the component object name to be specified ...

```
add(componentName);
```

☞ Add the five buttons to the container using the `add()` method provided in the **Container** class. Remember these instructions are part of the `init()` method. For example...

```
add(buttonA);
```

adds the first button to the top left corner of the container's grid.

Displaying the Layout

☞ Add the following instruction after all the components are added to the container...

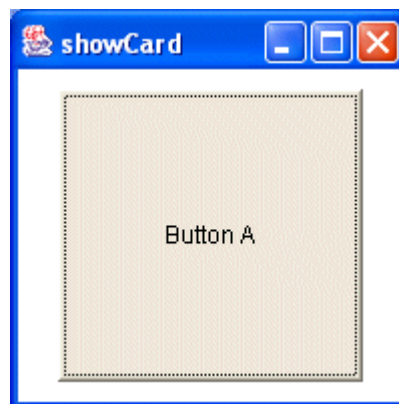
```
setVisible(true);
```

This makes the container visible.

☞ Compile and test your program. You should see five of the six cells in the grid.

Card Layout

The card layout treats each component as one card in a stack of cards, showing only one card at a time. The top card may look something like...



Constructors

There are two constructors for the **CardLayout** class...

```
CardLayout()  
  
GridLayout(int horizontalGap, int verticalGap)
```

The default constructor creates a new **CardLayout** object with no gap between the component and the container. The second constructor specifies horizontal spacing between the container and the left and right edges and the vertical spacing between the container and the top and bottom edges.

✎ Starting with your **master.java** file, replace all occurrences of the word ‘*master*’ with the word ‘*showCards*’. Save your file in a new folder as **showCards.java**.

✎ Add the following instruction to the *init()* method...

```
setLayout(new CardLayout(20, 10));
```

This creates a new **CardLayout** object and tells the container that the components will be placed in a series of cards. The vertical gap is 10 units and the horizontal gap is 20 units.

Adding Components

Because only one card is displayed at a time a different *add()* method is used to add components to the container...

```
add(String identifier, componentName);
```

where *identifier* is a string that gives each card a unique name. This allows for random access of the cards in the stack.

✎ Add the five buttons to the container using the *add()* method provided in the **Container** class. Remember these instructions are part of the *init()* method. For example...

```
add("A", buttonA);
```

adds the first button to the top card in the stack.

Displaying the Layout

✎ Add the following instruction after all the components are added to the container...

```
setVisible(true);
```

This makes the container visible.

✎ Compile and test your program. You should see just one card that contains the first component.

Working with the Cards

The following program shows you how you can use **CardLayout** methods to move between two cards. Please check the on-line documentation for the full set of **CardLayout** methods.

```
import java.applet.Applet;
import java.awt.event.*;
import java.awt.Frame;
import java.awt.*;

public class showCard extends Applet
{
    private static int SIZE=200;

    public static int FORWARD = 1;
    public static int BACK = 0;

    //add two buttons each of which will be on a separate card
    Button buttonA = new Button("button A");
    Button buttonB = new Button("button B");

    //create a CardLayout object to allow movement between cards
    CardLayout cards = new CardLayout(20, 10);

    public static void main(String args[])
    {
        showCardFrame frame = new showCardFrame ("showCard");
        frame.setSize(SIZE,SIZE);
        frame.show();
    }

    public void init(EventAdapter adapter)
    {
        setLayout(cards);
        add("A", buttonA);
        add("B",buttonB);

        buttonA.addActionListener(adapter);

        buttonB.addActionListener(adapter);

        setVisible(true);
    }

    public void quit()
    {
        System.exit(0);
    }

    public void changeCard(int direction)
    {
        if (direction == FORWARD)
            cards.next(this);
        else
            cards.previous(this);
    }
}
```

```
//=====
class showCardFrame extends Frame
{

    public showCardFrame (String frameTitle)
    {
        super(frameTitle);

        showCard applet = new showCard();
        EventAdapter adapter = new EventAdapter(applet, this);
        applet.init(adapter);
        add ("Center", applet);

        this.addWindowListener(adapter);
    }
}

//=====
class EventAdapter extends WindowAdapter implements ActionListener
{
    showCard applet;
    showCardFrame frame;

    EventAdapter(showCard applet, showCardFrame frame)
    {
        this.applet = applet;
        this.frame = frame;
    }


    public void windowClosing(WindowEvent wEvt)
    {
        applet.quit();
    }

    public void actionPerformed(ActionEvent e)
    {
        Object source = e.getSource();

        if (source == applet.buttonA)
            applet.changeCard(showCard.FORWARD);
        else if (source == applet.buttonB)
            applet.changeCard(showCard.BACK);
    }
}
}
```

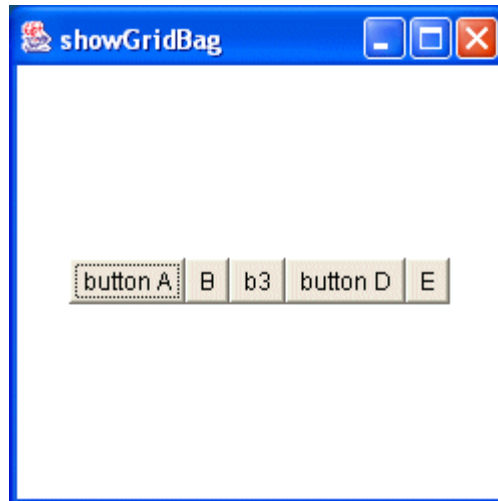
Notice first of all that there had to be a **CardLayout** object in order to move between the cards. This is created in the applet class along with the button components to be added to the cards. The two buttons are added in the *init()* method and interest is registered in their events.

All events are handled with the **EventAdapter**. The *actionPerformed()* method to monitor for button pushes is included here. When an action event occurs, the *actionPerformed()* method checks for the source of the event. Depending on which button triggered the event, the *changeCard()* method (defined in the applet) is called. This method uses the **CardLayout** methods *next()* and *previous()* to move between the cards.

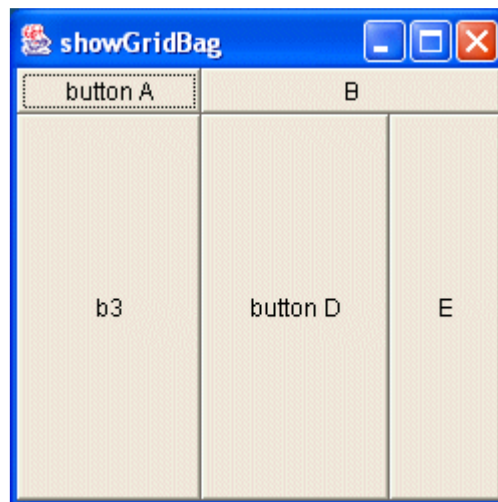
 Modify your program so that it can move between the five cards.

GridBag Layout

The **GridBagLayout** is a flexible layout manager that aligns components vertically and horizontally, without requiring that the components be of the same size. Used in its simplest fashion, the five buttons added to the container look something like this...



The **GridBagLayout** also allows you to modify constraints or characteristics for each component added to the container. These constraints affect how elements are positioned, sized, and spaced. The same five buttons may be displayed as...



Constructors

There is just one constructor for the **GridBagLayout** class...

```
GridBagLayout()
```

This default constructor simply creates a new **GridBagLayout** object.

☞ Starting with your **master.java** file, replace all occurrences of the word ‘*master*’ with the word ‘*showGridBag*’. Save your file in a new folder as **showGridBag.java**.

☞ Create a new **GridBagLayout** object. This declaration should be in the attribute declaration area of the applet...

```
GridBagLayout gridbag = new GridBagLayout();
```

☞ Add the following instruction to the *init()* method...

```
setLayout(gridbag);
```

Adding Components

The challenge when working with the **GridBagLayout** is to define the appropriate attributes in the **GridBagConstraints** object associated with the desired component. This attribute definition has to be done before the component is added to the container. The attributes in the **GridBagConstraints** class are...


gridx	cell containing the leading edge of the component's display area
gridy	cell at the top of the component's display area
gridwidth	number of cells in a row for the component's display area
gridheight	number of cells in a column for the component's display area
weightx	specifies how to distribute extra horizontal space
weighty	specifies how to distribute extra vertical space
anchor	used when the component is smaller than its display area. It determines where, within the display area, to place the component
fill	used when the component's display area is larger than the component's requested size. It determines whether to resize the component, and if so, how
insets	specifies the external padding of the component, the minimum amount of space between the component and the edges of its display area
ipadx	specifies the internal padding of the component, how much space to add to the minimum width of the component
ipady	specifies the internal padding, that is, how much space to add to the minimum height of the component

Before you can modify these attributes there has to be an object of type **GridBagConstraints**. This is created using the following instruction...

```
GridBagConstraints gridBagObject = new GridBagConstraints();
```

where *gridBagObject* is the name of the object created. For example if the object is called 'c', the instruction is...

```
GridBagConstraints c = new GridBagConstraints();
```

 Create a **GridBagConstraints** object for your application. Place the definition of the object in the normal attribute declaration area of the applet class.

Setting Constraints

Once this object exists, specific attributes can be modified to affect the output. This is done using a simple assignment statement...

```
constraintsObject.constraintsAttribute = desiredValue;
```

where *constraintsObject* is the name of the **gridBagConstraints** object, *constraintsAttribute* is the attribute to be changed and *desiredValue* is either a defined value or a **gridBagConstraints** class constant. For example, to resize components to fill the available horizontal and vertical space the fill attribute of the 'c' object is modified...

```
c.fill = GridBagConstraints.BOTH;
```

To make components double height, the instruction is...

```
c.gridheight = 2;
```

One uses a **gridBagConstraints** class constant while the other assigns a fixed value. Please check the on-line documentation for the complete list of **gridBagConstraints** class constants.

Associating Attributes with Components

Once all the desired attributes are modified, they are assigned to a specific component using the *setConstraints()* method that is part of the **GridBagLayout** Class...

```
gridbagObject.setConstraints(component, constraintsObject);
```

where *gridBagObject* is the name of the **gridBag** object, *component* is the component to which the attributes are to be assigned and *constraintsObject* is the name of the **gridBagConstraints** object. For example to assign the attributes to a **Button** component called *buttonA*, the instruction is...

```
gridBag.setConstraints(buttonA, c);
```


As a point of reference, the code to generate the second sample screen is...

```
//resize the component both horizontally and vertically
c.fill = GridBagConstraints.BOTH;
//prevent row spacing from being at edges
c.weightx = 1.0;
gridbag.setConstraints(buttonA, c);
add(buttonA);

c.gridwidth = GridBagConstraints.REMAINDER; //end row
gridbag.setConstraints(buttonB, c);
add(buttonB);

//number of cells in a row
c.gridwidth = 1; //reset to the default
c.gridheight = 1;
//prevent all vertical spacing at top and bottom
c.weighty = 1.0;
gridbag.setConstraints(buttonC, c);
add(buttonC);

//number of cells in a column
c.gridheight = 2;
gridbag.setConstraints(buttonD, c);
add(buttonD);


c.gridwidth = GridBagConstraints.REMAINDER; //end row
gridbag.setConstraints(buttonE, c);
add(buttonE);
```

Adding Components

Once the attributes are assigned to the individual components they are added to the container...

```
add(componentName);
```

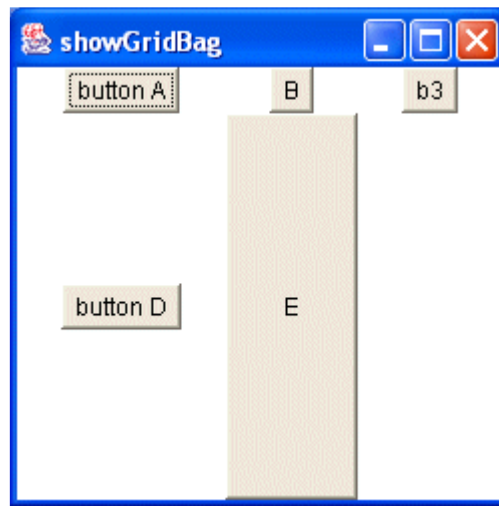
Displaying the Layout

 Add the following instruction after all the components are added to the container...

```
setVisible(true);
```

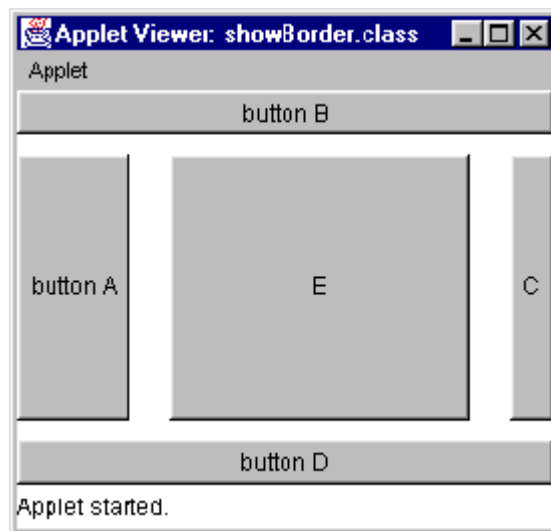
This makes the container visible.

☞ Define the constraints for, and add the five buttons to the container so that your output resembles the screen shown here. You will probably have to play with the layout constraints a bit to figure this out. Remember these instructions are part of the *init()* method.

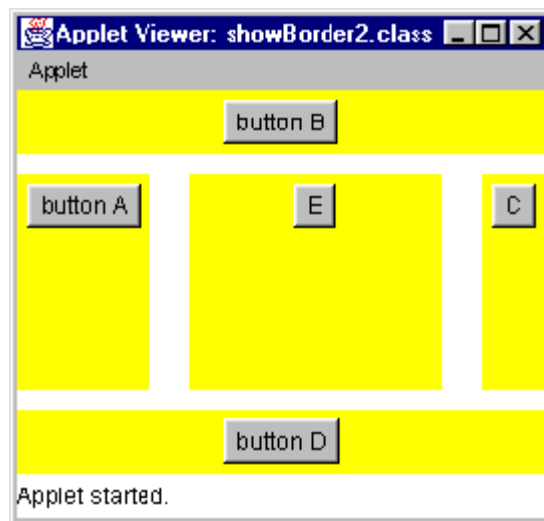


Panels

Panels are another type of container that can be used with layouts to better control what components look like and where they are placed. Recall the **showBorders.java** program you wrote where the output looked like...



You had very little control over the size and shape of the button components as they were added to the north, south, east, west, and center of the container. With panels you can control this to display 'normal' buttons in each of the quadrants...



In this case there are five **Panel** class objects created ... one for each of the north, south, east, west, and center areas available in the layout. The desired button is added to the appropriate panel and each panel is added to the appropriate area within the layout.

Containers can exist on their own or within other containers. Because panels are containers they can exist within specified areas of certain layouts. They can also contain layouts of their own.

Constructors

There are two constructors for the **Panel** class...

```
Panel ()
```

```
Panel (LayoutManager layout)
```

The default constructor creates a new **Panel** object that uses the default layout. The second constructor allows specification of the desired layout for components within the panel.

✎ Starting with your **master.java** file, replace all occurrences of the word '*master*' with the word '*showPanels*'. Save your file in a new folder as **showPanels.java**.

✎ Create five **Panel** objects that use the default layout. These declarations should be in the attribute declaration section of the applet class.

✎ Create the five button objects shown in the sample screen. These declarations should also be in the attribute declaration section of the applet class.

✎ Add the following instruction to the *init()* method...

```
setLayout (new BorderLayout ());
```

This creates a new **BorderLayout** object with the north, south, east, west and center areas.


Adding Components

Components are added to **Panel** objects before the panels are added to the container. This process is a bit different than adding the components directly to the container...

```
panelObject.add(componentName);
```

Notice that both the panel object '*panelObject*' and the component '*componentName*' are specified. For example, to add **Button** object '*buttonA*' to **Panel** object '*p1*' the instruction is...

```
p1.add(buttonA);
```

 Add one of the five buttons to each of the five panels. Remember these instructions are part of the *init()* method.

Adding Panels

Once all the required components are added to a **Panel** it can be added to the container. This process is exactly the same as it was to add a component to the container...


```
add(panelObject);
```

In this case, because it is a **BorderLayout**, the *add()* instruction is slightly different...


```
add(panelObject, workingAreaSpecifier);
```

For example, to add **Panel** object '*p1*' to the west section of the screen, the instruction is...

```
add(p1, BorderLayout.WEST);
```


 Add each of the five panels to one of the areas of the screen. Remember these instructions are part of the *init()* method.

Displaying the Layout

 Add the following instruction after all the components are added to the container...

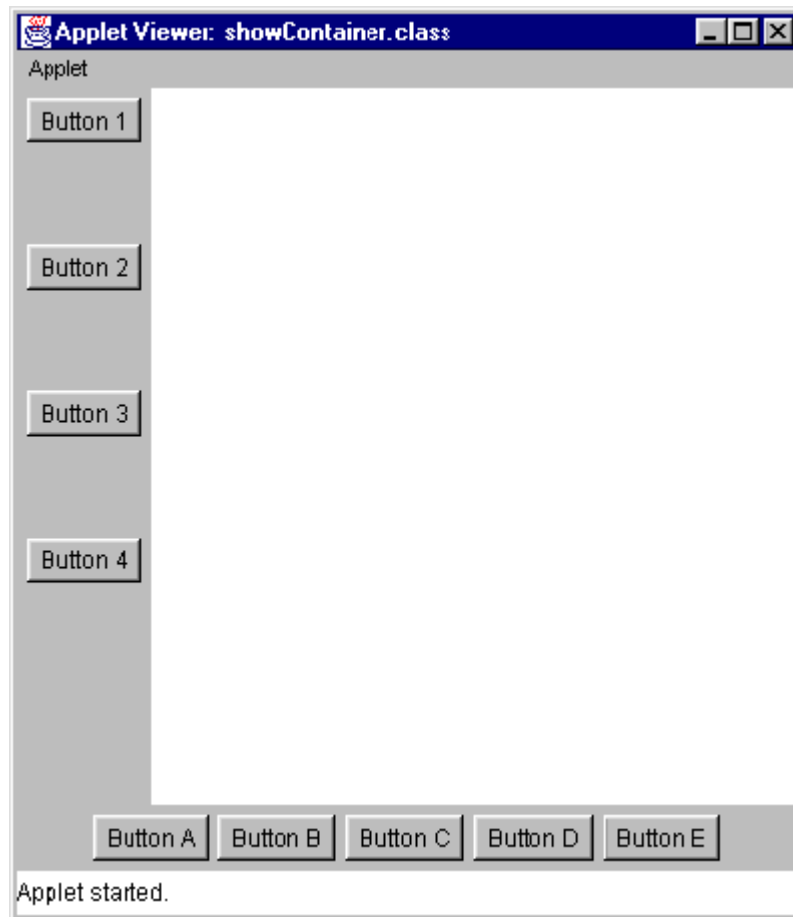
```
setVisible(true);
```

This makes the container visible.

 Compile and test your program. Depending on the size of your window, you should see something like the sample diagram.

Class Exercise – Containers, Panels and Layout Managers

Write a program that generates a screen that looks something like...



When any button is pressed it displays its label somewhere in the white space.

Hint: Remember that panels can be placed inside layout areas and layout managers can be defined for panels. This means that different layouts can be used within panels that are assigned to the master layout.

Handling Data

In the C programming language arrays, strings, structures and unions are considered to be *aggregate* data types ... those which contain organized collections of data in a definite order. Strings and arrays must by definition hold the same type of data, while structures and unions can hold varying types.

As you know, Java organizes its data (along with the appropriate methods) into classes. A number of system defined classes have been examined as well as the process of planning and programming your own classes. This section will focus on three system defined classes specifically created for ‘traditional’ data management.

Arrays

Like in C, an array is an indexed list of elements, all of which have to be of the same type. In Java arrays are considered somewhere between a primitive data type and a class. They function very much like a class in that there is a set of methods associated with arrays that allows you to examine and manipulate the data. They however cannot be subclassed. This means that you cannot add your own methods to work with the data.

Declaring Arrays

Arrays are defined as part of the **Object** class. As with any primitive data type or class object there must be both a data declaration and an instantiation. The data declaration specifies what type of data will be stored in the array, and can be done in one of two ways...

```
dataType arrayName[];  
or  
dataType[] arrayName;
```

Neither allocates any space for the array ... they simply indicate that an array called *arrayName* will be created to hold data of type *dataType*. For example, to specify an array of integers called ‘*numbers*’ the instruction is...

```
int numbers[];
```

Similarly to create an array of **Person** objects called ‘*people*’ the instruction is...

```
Person people[];
```

Once the array is declared it must be instantiated. This process is just like creating a class object...

```
arrayName = new dataType[size];
```

This creates the actual array, reserving enough space in memory for *size* elements of type *dataType*. The variable *size* can be either a constant or variable value. For example to create an array to hold ten *people* the instruction is...

```
people = new Person[10];
```

It is very common for the array declaration and instantiation to be done at the same time...

```
dataType arrayName[] = new dataType[size];
```

For example to declare and instantiate an array to hold 50 integers the instruction would be...

```
int numbers[] = new int[50];
```

Notice the presence of the square brackets in this combined declaration and instantiation. Don't forget to include these.

Array Initialization

By default, array elements are initialized to a default value ... usually zero or empty. They can also be initialized to specific values at instantiation time. To initialize an array whose elements are one of the primitive data types (integer, character, boolean, etc.), the instantiation instruction may look something like...

```
int numbers[] = {2, 3, 6, 12, 16, 24};
```

with curly braces enclosing the set of elements. Commas are used to separate each element. This particular example creates an array of integers with enough room for six elements. For arrays of objects the instantiation instruction may look something like...

```
Label tags[] = {new Label("Name"), new Label("Age"), new Label("Gender")};
```

This creates an array of three label components. The individual elements are accessed the standard way...

```
add(tags[0]);
```

This instruction adds the first label to the container.

Using Arrays

Once the array object is created, manipulation of its elements is almost identical to C arrays. By using an integer index, you can assign, modify, move through and access individual elements.

```
numbers[index] = value * 25 + 3 * numbers[index-1];
```

One of the key differences between C and Java is that Java will not allow you to access an element beyond its range. Any attempt to do this will result in a run-time error called an **exception**. These can be trapped and will be discussed in an upcoming chapter.

Array objects have an associated length that identifies how many elements exist in the array. This length is accessed using ...

```
arrayName.length
```

This is often used in loops to ensure that only valid index values are used...

```
for (int i =0; i < numbers.length; i++)
```

☞ Starting with your **master.java** file, replace all occurrences of the word ‘*master*’ with the word ‘*showArrays*’. Save your file in a new folder as **showArrays.java**.

☞ Create an array of integers that contains all the odd numbers from 1 to 50 in order.

☞ Create an array of five **Label** components that contain the text “Value 1”, “Value 2” ... “Value 5”. Add the labels to the container.

☞ Add a **List** component that displays three items at a time and allows only one selection. Add the List to the container.

Passing Array Elements to Methods

The process of passing an individual array element to a method is the same in both Java and C. The calling instruction includes the desired element in the parameter list. For example to pass the second element of an integer array called ‘*numbers*’ to a method called ‘*changeElement()*’ the instruction is...

```
changeElement(numbers[1]);
```

The method declaration must specify that an integer is expected and in this case specifies that an integer must be returned...

```
public int changeElement(int element)
{
    . . .
    return element;
}
```

Passing Arrays to Methods

The process of passing an entire array to a method is also the same in Java and C. The calling instruction includes the array name in the parameter list. For example to pass an entire array of Labels called ‘*tags*’ to a method called ‘*changeAllElements()*’ the instruction is...

```
changeAllElements(tags);
```


The method declaration must specify that an integer is expected...

```
public void changeAllElements(Label lbl[])
{
    . . .
}
```

Inside the *changeAllElements()* method the array can be used exactly the same as it would be in the original or calling method. Any changes made to the array are reflected back to the calling method.

Multidimensional Arrays

As in C, Java arrays can have more than one dimension. Declaration, instantiation and initialization of multidimensional arrays follow the same rules as for one dimensional arrays. Declarations must include a set of square brackets for each dimension. For a two dimensional array of integers, the declaration may look like...

```
int values[][];
```

The instantiation instruction must include a size for all but the last dimension. For the two dimensional array just declared the instantiation instruction could be...

```
values[][] = new int[4][2];
```

This creates a four by two array. Each element is initialized to zero.

The following instruction could be used to initialize elements to non-zero values...

```
values[][] = {{1, 2, 3}, {5, 6, 7}, {8, 9, 10}, {11, 12, 13}};
```

This creates a four by three array with each element initialized to the specified value.

Strings

Strings are one of the more commonly used classes in Java. Anytime you display or work with text you are using the **String** class. The class itself includes methods for examining individual characters of the sequence, for comparing strings, for searching strings, for extracting substrings, and for creating a copy of a string with all characters translated to uppercase or to lowercase.

Although strings are represented internally as an array of characters, they are in fact different than an array. The most interesting of these differences is that **String** objects cannot be changed ... once it has been created its content and length are fixed.

Constructors

The **String** class is part of the *java.lang* package. There is no need to import this package ... all applications and applets have direct access to it. There are eleven constructors for the **String** class. The most commonly used constructors include...

```
String()

String(char[] value)

String(String original)
```

The first constructor creates an empty **String** object. The second constructor is a conversion constructor that creates a new **String** object containing the contents of the character array specified in the argument list. The third constructor is a copy constructor that creates a new **String** object containing the literal string specified in the argument list.

You can also create a **String** object using an assignment statement...

```
String stringName = "string literal";
```

This method is every bit as valid as using a constructor in the traditional way.

String Operators

Since **String** objects are constant, Java provides special support for the binary string concatenation operator (+). This operator makes it very easy to piece text and numbers together to form usable strings. You started using this operator with your earliest Java applications when you pieced together output strings using '+';

Concatenation is actually implemented through the related **StringBuffer** class and its *append()* method. The instruction...

```
x = "a" + 4 + "c"
```

is actually compiled to the equivalent of:

```
x = new StringBuffer().append("a").append(4).append("c").toString()
```

This creates an empty string buffer, appends the string representation of each operand to the string buffer, and then converts the contents of the string buffer to a string.

String Length

The number of characters in a **String** object '*str*' can be found using the *length()* method...

```
int size = str.length();
```

The returned integer '*size*' defines the number of characters in the string.

Converting Data to Strings

There are often times that a programmer has to convert other data types to strings. The most common reasons are for display and text file purposes. You have already seen two examples of converting other objects and data to strings... the conversion constructor that converted a character array to a string and the concatenation example that converted an integer to a string. This concatenation method is one of the easiest ways to convert any primitive data type or any other object to a string. The compiler invokes whatever method is necessary to do the conversion.

Another method used to convert a primitive data value to a **String** object is the *valueOf()* method that is part of the **String** class. This method is **overloaded** meaning that there are actually nine different *valueOf()* methods defined within the **String** class. Each method has a different argument list and the compile determines which method to call based on the type of data passed to the method.

There is one *valueOf()* method defined for each primitive data type...

```
String valueOf(boolean b)
String valueOf(char c)
String valueOf(int i)
String valueOf(long l)
String valueOf(double d)
String valueOf(float f)
```

There are two *valueOf()* methods for converting character arrays to strings...

```
String valueOf(char ca[])
String valueOf(char ca[], int offset, int count)
```

The first converts the entire character array to a **String** object while the second converts a sub-string of the specified character array to a **String** object.

The remaining *valueOf()* method is used to convert objects to strings...


```
String valueOf(Object obj)
```

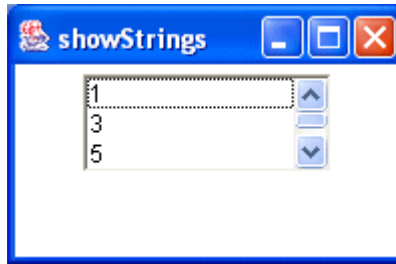
For example to convert an integer attribute called 'nbr' to a string and assign it to a **String** object called 'str' the instruction is...

```
str = String.valueOf(nbr);
```

Notice that the *valueOf()* method is used with the class name **String**.

☞ Create an array of **String** objects large enough to hold the string equivalent of your integer array. Convert each integer to its equivalent string and store each in the array. Populate the **List** object with these **String** elements.

 Compile and run your program. You should see something like...



The *toString()* method is often used to convert objects to **String** objects. There are almost 300 versions of the *toString()* method ... luckily the compiler is able to select the proper method given the type of data in the parameter list. The format for the *toString()* method is...

```
objectName.toString();
```

When you are defining your own classes for an application it is a good idea to override the *toString()* method in the class. This allows you to specify exactly what class information you want displayed when the object is converted to a string.

Converting Strings to Data

Wrapper classes are defined for each of the primitive data types that allow conversion from strings to the data type. The wrapper classes...

Boolean
Character
Integer
Long
Float
Double

each contain a *valueOf()* method that returns the specified string as the appropriate wrapper data type. Each of these wrappers also contains a conversion method...

```
booleanValue( )  
charValue( )  
intValue( )  
longValue( )  
floatValue( )  
doubleValue( )
```

that can then be used to convert the returned value to the primitive data type. Say for example you have a String object called 'str' that contains "253". The following instructions will convert this string to an integer value...

```
Integer x = Integer.valueOf(str);  
int value = x.intValue();
```

Once these two instructions are complete, the attribute *'value'* contains the integer value 253. Since the dot operator groups from left to right these two instructions could be re-written as...

```
int value = Integer.valueOf(str).intValue();
```

If you want to convert a **String** object *'str'* to a character array called *'newStr'* you can use the *toCharArray()* method...

```
char[] newStr = str.toCharArray();
```

This new character array will be exactly the same length as the original **String** object.

Working with Characters

Although **String** objects cannot be changed once they are created it is possible to access the character at a certain position within a string. The *charAt()* method is used to do this. Its prototype is...

```
char charAt(int i);
```

where *i* is the position of the desired character within the string. For example, given a string called *str* that contains the text “The sky is blue”, the instruction...

```
char ch = str.charAt(5);
```

stores the character *'k'* in the attribute *ch*.

There are also methods that allow you to search for the first or last occurrence of a specific character within a string. The *indexOf()* method finds the first occurrence while the *lastIndexOf()* method finds the last occurrence. For example, given the same string *str* that contains the text “The sky is blue”, the instruction...

```
int first = str.indexOf('e');
```

finds the first *'e'* in the string which is in position 2. This value is assigned to the attribute *'first'*. Similarly, the instruction...

```
int last = str.lastIndexOf('e');
```

finds the last *'e'* in the string which is in position 14. This value is assigned to the attribute *'last'*.

Comparing Strings

There are a number of methods in the **String** class that allow strings or sub-strings to be compared...

Return Type	Method	Operation
int	<i>strName</i> .compareTo(String <i>s</i>)	Compares <i>strName</i> to <i>s</i> . Returns difference between ASCII values of first characters that are different. Returns zero if strings are the same.
boolean	<i>strName</i> .endsWith(String <i>s</i>)	Returns true if <i>strName</i> ends with <i>s</i> . Returns false otherwise.
boolean	<i>strName</i> .equals(Object <i>obj</i>)	Returns true if the two strings <i>strName</i> and <i>obj</i> (which must be a string) are identical. Returns false otherwise.
boolean	<i>strName</i> .equalsIgnoreCase(String <i>str</i>)	Returns true if the two strings <i>strName</i> and <i>str</i> are identical without consideration of upper and lower case letters. Returns false otherwise.
Boolean	<i>strName</i> .startsWith(String <i>s</i>)	Returns true if <i>strName</i> starts with <i>s</i> . Returns false otherwise.

Modifying Strings

There are also a number of methods in the **String** class that allow strings to be modified...

Return Type	Method	Operation
String	<i>str</i> .concat(String <i>s</i>)	Returns a new string that adds string <i>s</i> to the end of string <i>str</i> .
String	<i>str</i> .replace(char <i>original</i> , char <i>new</i>)	Returns a new string that replaces every instance of the <i>original</i> character with the <i>new</i> character.
String	<i>str</i> .substring(int <i>start</i>)	Returns a new string that is a sub-string of the original string <i>str</i> . The sub-string includes all the characters from position <i>start</i> to the end of the string.

String	<code>str.substring(int start, int end)</code>	Returns a new string that is a sub-string of the original string <i>str</i> . The sub-string includes all the characters from position <i>start</i> up to and including position <i>end</i> .
String	<code>str.toLowerCase()</code>	Returns a new string in which all characters in the original string <i>str</i> are converted to lower case.
String	<code>str.toUpperCase()</code>	Returns a new string in which all characters in the original string <i>str</i> are converted to upper case.
String	<code>str.trim()</code>	Returns a new string in which all leading and trailing white space (blanks, tabs, returns, etc.) in the original string <i>str</i> are removed..

Vectors

Vectors are an additional class that Java provides for handling groups of data. Think of a vector as an array of unknown, variable and unlimited size. They can hold any type of object although primitive data types must be converted to the appropriate wrapper class before they can be added to a vector.

Constructors

The **Vector** class is part of the *java.util* package. Any time you want to use vectors in your application you must import this package...

```
import java.util.*
```

There are four constructors for the Vector class...

```
Vector( );

Vector(Collection c );

Vector(int initialCapacity);

Vector(int initialCapacity, int capacityIncrement );
```

The default constructor creates a vector with an initial capacity of ten elements. The second constructor creates a vector large enough to hold the elements contained in the specified collection. Each element in the collection is copied into the vector in its original order. The third constructor allows you to define the starting number of elements. The final constructor allows the starting number of elements to be defined as well as how many more elements will be added each time the vector reaches its current capacity.

For example, to create a **Vector** that will hold a vector of strings, the instruction is...

```
Vector strings = new Vector( );
```

creates a vector called *strings* that can actually hold any type of object. Similarly, the instruction...

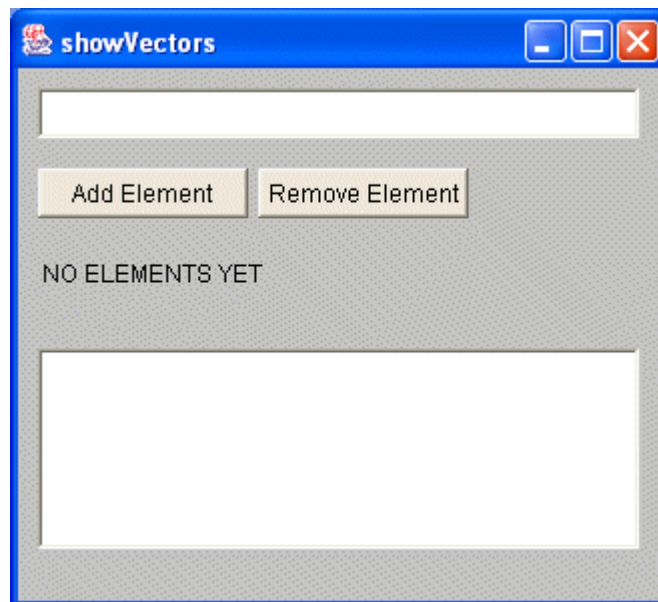
```
Vector members = new Vector(25);
```

creates a vector that starts off with 25 elements and is capable of holding any type of object.

✎ Starting with your **master.java** file, replace all occurrences of the word ‘*master*’ with the word ‘*showVectors*’. Save your file in a new folder as **showVectors.java**.

✎ Create a vector object.

✎ Add a text field, two buttons, one label, and a list box so that your display looks something like...



✎ Add the appropriate listeners for the two buttons.

✎ Add event handling code so that when the Add Element button is clicked, the text in the text field is added to the list box, and the text field is cleared.

Vector Methods

The most commonly used methods available for the **Vector** class are...

Return Type	Method	Operation
void	<code>add (Object <i>obj</i>)</code>	Add the specified element <i>obj</i> at the end of the vector
void	<code>addElement(Object <i>obj</i>)</code>	Add the specified element <i>obj</i> at the end of the vector
boolean	<code>equals(Object <i>obj</i>)</code>	Compares the specified object <i>obj</i> with the vector object
Object	<code>get(int <i>index</i>)</code>	Returns the object stored at position <i>index</i> in the vector
void	<code>insertElementAt(Object <i>obj</i>, int <i>index</i>)</code>	Add the specified element <i>obj</i> at the position <i>index</i> in the vector
boolean	<code>isEmpty()</code>	Returns true if the vector has no elements
void	<code>removeElement(Object <i>obj</i>)</code>	Remove the first occurrence of element <i>obj</i> from the vector
void	<code>removeElementAt(int <i>index</i>)</code>	Remove element from position <i>index</i> in the vector
int	<code>size()</code>	Returns the number of elements in the vector


For example, to add an element called ‘*data*’ to the end of a vector called ‘*vect*’ the instruction is...

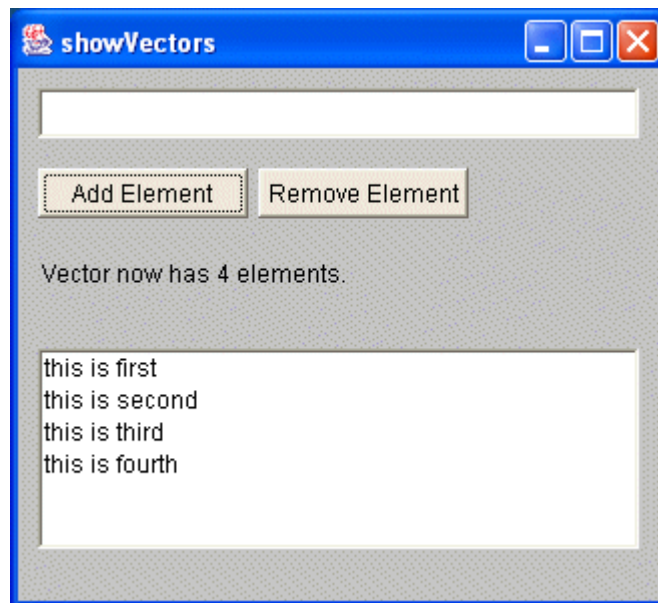
```
vect.addElement(data);
```

The instruction to determine the size of the vector called ‘*vect*’ is...

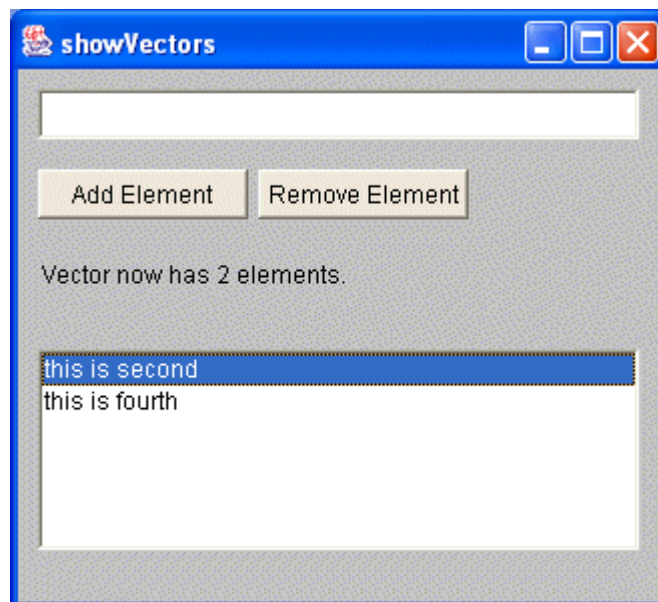
```
int vectSize = vect.size();
```

 Add code to the event handling code for the Add Element button that adds the specified text to the vector as well as to the list.

 After the string is added to the vector, calculate the vector size and change the label displaying “NO ELEMENTS YET” to “Vector now has *n* elements.” where *n* is the current number of elements. Compile and test your program. After a few elements are added to the vector, the display should look something like...



☞ Add code to the event handling code for the Remove Element button. When the button is clicked the item selected in the list is deleted from both the vector and the list. Make sure the label displaying the number of elements in the vector is updated. Compile and test your program. After a few elements are removed, the display should look something like...



Make sure you check the on-line documentation for the full set of **Vector** methods.

Class Exercise – Strings, Arrays and Vectors

Demonstrate your fully functioning **showVectors** applet/application.

Exceptions

Exceptions are run-time errors that may occur at any time during program execution. Some exceptions are critical and cause the application to terminate prematurely. Others are non-critical and simply identify that an error has occurred during execution. In Java the process of catching and notifying the system that a run-time error has occurred is called *throwing an exception*.

Java provides a powerful error-handling class that allows programs to monitor for and trap typical errors. In doing this, the programmer can prevent critical exceptions from crashing the program.

Exception Class

The **Exception** class has over fifty different child classes that handle different types of errors such as instantiation errors, runtime errors, data formatting errors, and illegal access errors. Each of these child classes has its own set of child classes to look after more specialized errors. The **RuntimeException** class, for example, contains child classes to look after arithmetic errors, indices that are out of bounds, illegal arguments, and missing resources. This section looks at some of the more commonly used exception classes.

Runtime Exceptions

The **RuntimeException** class exists in the *java.lang* package. There are approximately thirty child classes, some of which are discussed here. Please check the on-line documentation for the complete set.

The **ArithmeticException** is thrown when an arithmetic error, such as divide by zero occurs. For example if the variable denominator happens to equal zero, the following arithmetic expression throws an arithmetic exception...

```
result = numerator/denominator;
```

The **ArrayStoreException** is thrown when an attempt is made to store the wrong type of object into an array of objects. For example, the following code generates an array store exception...

```
String str[] = new String[10];  
str[0] = new Integer(0);
```

The **IllegalArgumentException** is thrown when a method is passed an illegal or inappropriate argument. This class is extended to a number of child classes including **NumberFormatException** that is thrown when an application tries to convert a string to one of the numeric types, but the string doesn't have the appropriate format.

The **IndexOutOfBoundsException** is thrown when an index of some sort (such as to an array, to a string, or to a vector) is out of range. This class has child classes that deal specifically with array index boundaries and with string index boundaries.

The **NullPointerException** is thrown when an application attempts to use `null` in a case where an object is required. This is a very common error and usually means that the required object has not yet been instantiated or if it has, that the system has no way to access it.

IO Exceptions

The **IOException** class exists in the *java.io* package. There are approximately twenty child classes, some of which are discussed here. Please check the on-line documentation for the complete set.

The **EOFException** is thrown when an end of file or end of stream has been reached unexpectedly during input.

The **FileNotFoundException** is thrown when an attempt open a specified file fails. It is thrown by the **FileInputStream**, **FileOutputStream** and the **RandomAccessFile** classes when the specified file is inaccessible or does not exist.

The **MalformedURLException** is thrown when a specified URL is incorrectly formatted. It may be that no legal protocol could be found in the specified URL string or the string itself could not be parsed.

Trapping Errors

If you as the programmer don't do anything to prevent an exception, the error message filters up the call stack until it is eventually caught by one of the Java classes. At that point, it identifies the type of error and the classes it has passed through to get to where it was finally caught. The error type and the call stack are written to the system default output ... normally the screen.

To prevent this from happening, you want to trap these errors. This is accomplished using a *try/catch/finally* structure...

```
try
{
    //code where exception might occur goes here
}
catch (ExceptionClass ex1)
{
    //code to be executed if this type of error occurs
}
catch (AnotherExceptionClass ex2)
{
    //code to be executed if this type of error occurs
}
finally
{
    //code that is executed regardless of whether there is an exception
    //this block is optional
}
```

The *try* block is designed to include the code where an exception may occur. There can be as many *catch* blocks as required ... one for each type of exception that may be caused during execution of the *try* block. Each *catch* block is passed an exception class object than can be used within the block. The

finally block is an optional block. If it is included its code gets executed whether an exception occurred or not.

For example, it is possible for an error to occur when division is performed. If the denominator happens to be zero, the answer is undefined. The following block of code could be used to catch this type of arithmetic error...

```
try
{
    result = numerator/denominator;
    label.setText("Result is " + result);
}
catch (ArithmeticException e)
{
    label.setText("Result is undefined");
}
```

Throwing Exceptions

It is possible for you as a programmer to generate your own exceptions instead of waiting for Java to generate them. This is done using a *throw* statement...

```
try
{
    if (condition you want to trap)
        throw new ExceptionClass( );
}
catch (ExceptionClass ex1)
{
    //code to be executed error you trapped occurs
}
```

This technique is useful when you want to separate the code used to deal with rare but possible errors from code used to deal with expected errors.

Using Exceptions

Try to use the **Exception** class only when results are out of your control as a programmer, not to trap common, expected errors. Like any well designed code, you should test for and prevent all expected errors. It is also more efficient ... a series of tests will actually run faster than throwing an exception. Keep in mind that the more exceptions you throw, the more complex your code is. Try to only throw exceptions that have a reasonable chance of happening.

Class Exercise – Array Boundary Exceptions

Write a simple application that creates and populates an array of 5 integers. Given a user-entered index, display the corresponding element. Use the *try/catch* structure to prevent access beyond the five elements.

Input and Output

In the applications you've written to date, all the input has come from either the command line or from a component such as a text field. Similarly, output has been directed to the system or to a component such as a label or a list box. This section takes a look at two more aspects of I/O... file handling and keyboard input.

File Handling

File handling in Java is done using *stream* classes that are part of the *java.io* package. A stream is simply an ordered sequence of objects. Methods are available to write or add objects to the stream, and to read or take objects from the stream.

File Menus and Dialog Boxes

Before data is actually read from or written to a file, there has to be a way to specify the desired file. This is normally done using a standard File Dialog box. Java provides an easy way to do this using the **FileDialog** class. This class has three constructors...

```
FileDialog(Frame parent)

FileDialog(Frame parent, String title)

FileDialog(Frame parent, String title, int mode)
```

The first constructor creates a file dialog box object and associates it with the application's frame. The second constructor creates the dialog box, associates it with the frame and specifies the box's title. The third constructor, in addition to what the second does, also specifies whether the dialog box will be used to load or save a file. This third constructor is the one most often used. For example to create a load file dialog box with the title "Open a File", the instruction is...

```
FileDialog fdlg = new FileDialog(frame, "Open a File", FileDialog.LOAD);
```

where *frame* is the application frame's name, and `FileDialog.LOAD` is constant defined in the **FileDialog** class.

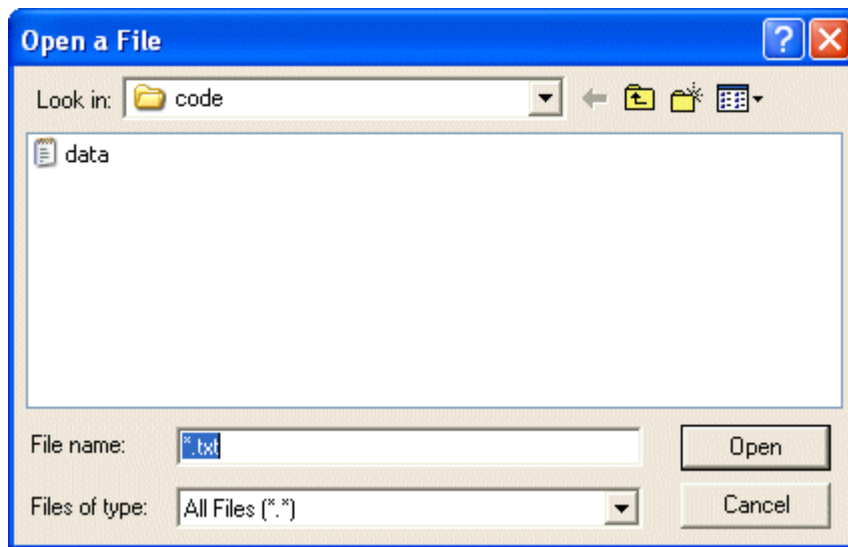
Before the **FileDialog** object is displayed you can identify either a specific file or a file filter that will display only applicable files. This is done using the *setFile()* method...

```
setFile(String fileSpecifier)
```

For example, to specify that all text files should be displayed in the dialog box called 'fdlg', the instruction is...

```
fdlg.setFile("*.txt");
```

This will display only subset of all the files in the folder ... those that have the 'txt' extension. The dialog box will look something like...



If this *setFile()* instruction is left out of the application the file dialog box displays all files in the folder.


The *show()* method is used to display the dialog box ...


```
show( )
```

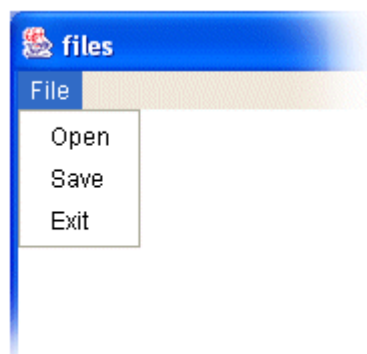
For example, to show the '*fdlg*' dialog box the instruction is...

```
fdlg.show( ) ;
```

You'll now see a dialog box similar to the one shown above. Depending on your use of the *setfile()* method you will see all or some of the file names displayed.

 Starting with your **master.java** file, replace all occurrences of the word '*master*' with the word '*files*'. Save your file in a new folder as **files.java**.

 Add a menu system to the frame class that looks like...



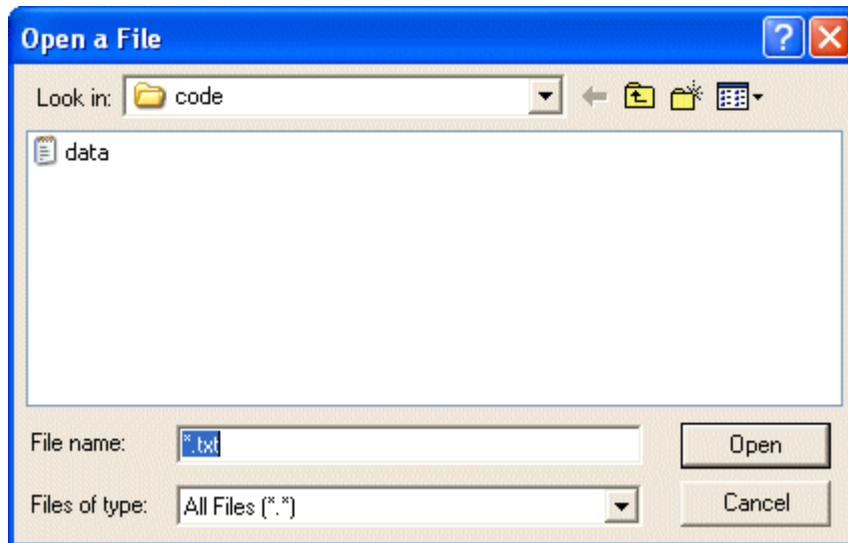
☞ Add event handling code for each of the three menu items. The *Open* option should call a *FileOpen()* method in the applet class. The *Save* option should call a *FileSave()* method in the applet class. Add these two file methods to the applet but leave them empty for now. The *Exit* option should call the *quit()* method that should already exist in the applet class.

☞ Compile and test your program. You should be able to select any of the three menu items without crashing your program. The *Exit* option should close the application.

☞ Add the following code to the *FileOpen()* method in the applet class...

```
FileDialog fdlg = new FileDialog(frame, "Open a File", FileDialog.LOAD);  
fdlg.setFile("*.txt");  
fdlg.show();
```

☞ Compile and test your program. Now when you click the File Open option you should see a dialog box like...



It won't do anything yet, so just hit cancel once it is displayed.

Once the user makes a file selection, control returns from the dialog box to the application. Two **FileDialog** methods *getDirectory()* and *getFile()* are used to extract the selected file name and folder.

☞ Add the following code to the end of the *FileOpen()* method in the applet class...


```
loadData(fdlg.getDirectory() + fdlg.getFile());  
fdlg.dispose();  
  
repaint();
```

This passes a string containing the full path and file name to a local method called *loadData()* that will actually open and read the file contents. It also frees up the system resources allocated to the dialog box before repainting the screen.

 Add the *loadData()* method to the applet class...

```
private void loadData(String fileName)
{
    System.out.println("The selected file is " + filename);
}
```

Make sure you specify a **String** argument that will accept the filename.

 Compile and test your program. Select a file using the dialog box. You should see its name and full path displayed on the console screen.

Reading from Files

As mentioned earlier, data is considered a stream of objects. All the classes associated with reading data are part of the *java.io* package. The **InputStream** class and its many child classes are responsible for all the activity required for reading data files. Reading text files is slightly different than reading data files. Each will be discussed separately.

Reading Text

There are two main classes required when reading text files ... **FileReader** and **BufferedReader**. These two classes work together with the **FileReader** class reading a stream of characters, and the **BufferedReader** class buffering the text for easy and efficient reading. First an object of class **FileReader** is created...

```
FileReader f = new FileReader(fileName);
```

where '*f*' is the **FileReader** object that will access the characters from the file '*filename*'. Now the **BufferedReader** object is created to store the characters read from the file...

```
BufferedReader in = new BufferedReader(f);
```

where '*in*' is the object that will be used to actually read the data from the character stream '*f*'. Because these instructions may cause a file related error, they should be monitored for an **IOException**.

 Add these instructions to the *loadData()* method in the applet class...

```
try
{
    FileReader f = new FileReader(fileName);
    BufferedReader in = new BufferedReader(fstrm);
}
catch (IOException e)
{
    System.err.println(e);
}
```

If something happens while opening the file the exception information will be displayed.

The **BufferedReader** class has a *readLine()* method that returns one line of text as a **String**. The instruction...

```
s = in.readLine();
```

reads one line of text 's' from the character buffer 'in'.

☞ Add the following instructions to the *try* block in the *loadData()* method.

```
s = in.readLine();
while (s != null)
{
    System.out.println(s);
    s = in.readLine();
}
```

This reads the file one line at a time and displays each line on the screen. Testing the string against null makes sure you don't read past the end of the file. Make sure you declare a **String** variable for 's'.

☞ It is also a good idea to monitor for an end-of-file exception. Add the following catch block to the existing try/catch structure in the *loadData()* method ...

```
catch (EOFException e)
{
    //no action required ... just prevent an error from occurring
}
```

As in any environment, once you have finished working with a file, it should be closed. This is done using the *close()* method...

```
in.close();
```

☞ Add the *close()* method to the end of the try block. The *loadData()* method should now look like...

```
private void loadData(String fileName)
{
    String s;

    System.out.println("The selected file is " + fileName);


    try
    {
        FileReader f = new FileReader(fileName);
        BufferedReader in = new BufferedReader(fstrm);
        s = in.readLine();
        while (s != null)
        {
            System.out.println(s);
            s = in.readLine();
        }
    }
```


```

        in.close();
    }

    catch (EOFException e)
    {
        //no action required ... just prevent an error from occurring
    }
    catch (IOException e)
    {
        System.err.println(e);
    }
}

```


 Create a text file that has five to ten short strings in it.


 Compile and test your program. Select your text file using the dialog box. You should see its name and full path displayed on the console screen. You should also see each line of the file displayed.

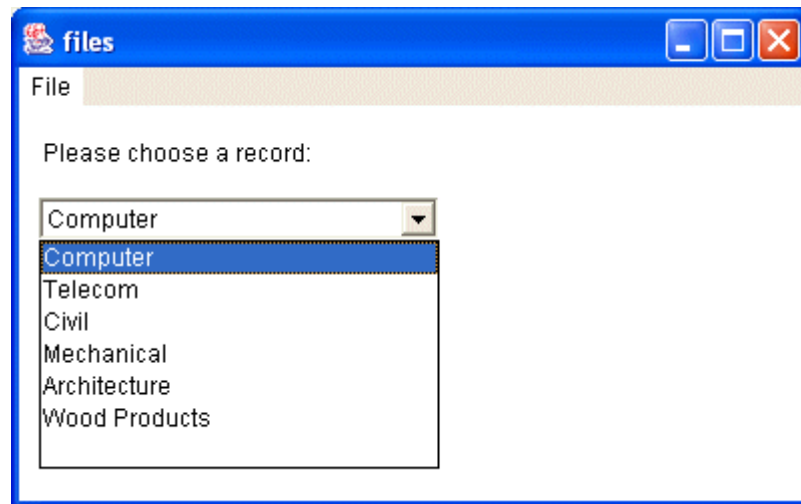
 Add a label and a choice box to your *init()* method so that your display looks something like ...



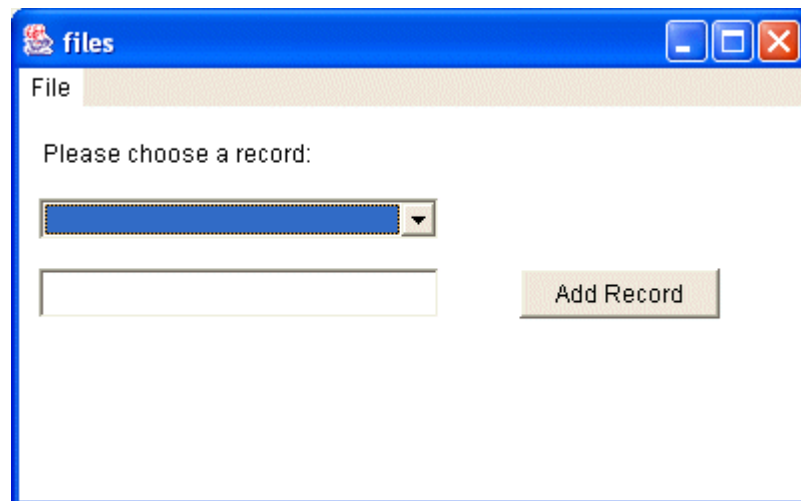
Using the Data

 Add a **Vector** object to your applet. Modify your *loadData()* method so that it stores each string read from the file as one element in the Vector. After the file is closed, populate the choice box with the elements of the vector. Remove the output statements that display the file name and contents to the console.

 Compile and test your program. After you've opened your text file the strings from the file should be in the choice box. With the choices revealed, your display should look something like...



☞ Add a text field and a button to your window. It should now look something like...



☞ Add a listener for the 'Add Record' button. Add the event handling code that takes the contents of the text field and adds it to the vector. After the item is added to the vector, clear the text field. Make sure that you can add records both before and after you've opened the file.

Writing to Files

Like reading from files, all the classes associated with writing data are part of the *java.io* package. The **OutputStream** class and its many child classes are responsible for all the activity required for writing data files. Writing text files is slightly different than writing data files. Each will be discussed separately.

Dialog Boxes

The **FileDialog** constructor you used for the Open Dialog box can also be used for a Save Dialog box...

```
FileDialog(Frame parent, String title, int mode)
```

The only changes you need to make are in the title and the *mode*...

```
FileDialog fdlg = new FileDialog(frame, "Save a File", FileDialog.SAVE);
```

where *frame* is the application frame's name, and `FileDialog.SAVE` is constant defined in the **FileDialog** class.

Once the dialog box is instantiated you can use the same sequence of methods to filter and show it. Once the dialog box is closed you use the same methods to extract the relevant file path and name information.

☞ Add code to the *fileSave()* method to display a Save dialog box. Include instructions to call a *saveData()* method passing in the file path and name.

☞ Add the *saveData()* method that expects a **String** specifying the desired file. For now add an instruction to display the specified file name to the console.

☞ Compile and test your program. Make sure that the Save menu item opens the Save Dialog box. It doesn't do anything yet so just press cancel to close the box.

Writing Text

There are two main classes required when writing text files ... **FileWriter** and **BufferedWriter**. These two classes work together with the **FileWriter** class writing a stream of characters, and the **BufferedWriter** class writing text to a character stream, buffering it for easy and efficient writing. First an object of class **FileWriter** is created...

```
FileWriter f = new FileWriter(fileName);
```

where '*f*' is the **FileWriter** object that will send the characters to the file '*filename*'. Now the **BufferedWriter** object is created to transfer the characters to the file...

```
BufferedWriter out = new BufferedWriter(f);
```

where '*out*' is the object that will be used to actually write the data to the character stream '*f*'. Because these instructions may cause a file related error, they should be monitored for an **IOException**. You also need to include a file close instruction.

 Add these instructions to the *saveData()* method in the applet class...

```
try
{
    BufferedWriter out = new BufferedWriter(new FileWriter(fileName));


    out.close();
}
catch(IOException e)
{
    System.err.println(e);
}
```

The **BufferedWriter** class includes a number of methods used to write text to a file. The two most commonly methods are *write()* and *newLine()*...

```
write(String s, int offset, int length);

newLine();
```

The *write()* method specifies the string to be written, the starting position within the string and how many characters are to be written. No line feed is added at the end of the string. If the string does not contain a line feed then the *newLine()* method is used to add this to the file.

 Modify the *saveData()* method by adding the following instructions to the middle of the try block...

```
for (int i = 0; i < data.size(); i++)
{
    str = String.valueOf(data.get(i));
    out.write(str, 0, str.length());
    out.newLine();
}
```

This code extracts each string from the vector (called *data*), converts it to a string and writes it to the file. Make sure you create a string variable called *str* and change the name of the vector to whatever your vector is called. The complete *saveData()* method should look something like this...

```
public void saveData(String fileName)
{
    String str;

    try
    {
        BufferedWriter out = new BufferedWriter(new FileWriter(fileName));
        for (int i = 0; i < data.size(); i++)
        {
            str = String.valueOf(data.get(i));
            out.write(str, 0, str.length());
            out.newLine();
        }
        out.close();
    }
}
```

```

        catch(IOException e)
        {
            System.err.println(e);
        }
    }
}

```

☞ Compile and test your program. Make sure you can open a file, add to it and save it. Check your new file contents by opening it in a text editor or by loading it back into your program.

Reading and Writing Data Files

Working with data files is similar to working with text files. The differences are in the classes used to stream the data, and in the read and write methods.

Reading Data Files

There are two main classes required when reading data files ... **FileInputStream** and **DataInputStream**. These two classes work together with the **FileInputStream** class reading a stream of bytes, and the **DataInputStream** class letting the application read primitive Java data types from the byte stream. First an object of class **FileInputStream** is created...

```
FileInputStream f = new FileInputStream(fileName);
```

where *f* is the **FileInputStream** object that will access the bytes from the file '*filename*'. Now the **DataInputStream** object is created to read the data from the byte stream...

```
DataInputStream in = new DataInputStream(f);
```

The **DataInputStream** class has a number of methods that allow reading of primitive data types...

```

int readInt();
char readChar();
float readFloat();
double readDouble();
long readLong();
boolean readBoolean();

```

Each of these methods is used with the **DataInputStream** object. For example, to read a Boolean value from a file the instruction is...

```
boolean boolValue = in.readBoolean();
```

Recall that if a primitive value is to be stored in a **Vector** object, it must be converted to a wrapper. This is done by creating a new wrapper object whose value is the primitive data type. For example to convert the boolean value *boolValue* for storage in a **Vector** called *data*, the instruction is...

```
data.addElement(new Boolean(boolValue));
```

Writing Data Files

There are two main classes required when writing data files ... **FileOutputStream** and **DataOutputStream**. These two classes work together with the **FileOutputStream** class writing a stream of bytes, and the **DataOutputStream** class letting the application write primitive Java data types to the byte stream. First an object of class **FileOutputStream** is created...

```
FileOutputStream f = new FileOutputStream(fileName);
```

where '*f*' is the **FileOutputStream** object that will write the bytes to the file '*filename*'. Now the **DataOutputStream** object is created to write the data to the byte stream...

```
DataOutputStream out = new DataOutputStream(f);
```

The **DataOutputStream** class has a number of methods that allow writing of primitive data types...

```
int writeInt();  
char writeChar();  
float writeFloat();  
double writeDouble();  
long writeLong();  
boolean writeBoolean();
```

Each of these methods is used with the **DataOutputStream** object. For example, to write an integer value to a file the instruction is...

```
out.writeInt(12);
```

Since each primitive value stored in a **Vector** is stored using a wrapper class, it must be converted back to the primitive data type before writing it to a file. This is done using wrapper methods...

```
intValue();  
charValue();  
floatValue();  
doubleValue();  
longValue();  
booleanValue();
```

For example to convert and write the first **Integer** element from a **Vector** called *data*, the instruction is...

```
out.writeInt(data.get(0).intValue());
```

The first part of the argument accesses the first **Vector** element. The second part of the argument converts the value from **Integer** to int. This is the value written to the file.

Things to Keep in Mind

One of the problems with data files of this type is that what is read must be exactly what was written ... if three integers were written, followed by twelve long values and then two more integers, the values must be read back in exactly the same order. Keep this in mind as you create any data files.

Another thing to remember about files is that they can only be used with Java applications. The security features built into the Java language prevent file access when running applets.

Class Exercise – Text and Data Files

Demonstrate the fully functional text file application.

Copy this program and modify it to create and read a data file that contains at least 2 integer values, 2 double values and one boolean value. Display these values in the Choice box. There is no need to include the Add Record functionality in this application.

Threads

A thread is the basic unit of program execution in Java. It exists and runs within an applet or an application. Java provides built-in support for threads, so a process can have several threads running concurrently, each performing a different job. This section looks at threads and how they can be used in applets and applications.

Creating Threads

The **Thread** class is part of the *java.lang* package. It implements the **Runnable** interface which provides a common protocol for objects that want to execute code while they are active. There are eight constructors for the **Thread** class. The most commonly used constructor is...

```
Thread(Runnable target)
```

where *target* is the object that will manage the thread. For example, to create a thread in an applet that will be managed by the applet, the instruction is...

```
Thread t = new Thread(this);
```

The target, being the applet itself, is defined using '*this*'.

Managing Threads

Once a thread is instantiated it must be started. This is done using the *start()* method defined in the **Thread** class. This automatically calls a *run()* method that must be included in the target ... in this case, the applet. For example, the instruction to start the thread '*t*' is...

```
t.start();
```

This calls the *run()* method...

```
public void run()
{
    //thread execution code goes here
}
```

When the *run()* method is complete, the **Thread** class will clean up and terminate the thread.

Terminating Threads

Once a thread is running there must be a way to stop it. This is often done using the *stop()* method defined in the **Thread** class but this method has been deprecated because it can cause system instability. The recommended way for terminating a thread now is to use some type of terminating flag in the *run()* method...

```

public void run()
{
    while (!terminated)
    {
        //thread execution code goes here
    }
}

```

The ‘*terminated*’ flag should be defined in the target class. That allows any component that has access to the target class to terminate the thread by calling a *stopThread()* method...

```

public void stopThread()
{
    terminated = true;
}

```

This process will be discussed in more detail in the next sections.

Using Threads in Java

There are two basic approaches to using multiple threads in an applet or an application....

- make the applet or its event handling adapter **Runnable**
- create a **Runnable** class that contains a **Thread** object

Runnable Applet

If the applet needs only one alternate thread, then the simplest way to implement that thread is by making either the applet or its event handling class **Runnable**...

```

class appletName extends Applet implements Runnable

```


or

```

class EventAdapterName extends WindowAdapter implements Runnable

```

*Recall that the **Runnable** interface provides a common protocol for objects that wish to execute code while they are active ... in this case active means after a thread has been started and before it is stopped.*

 Starting with your **master.java** file, replace all occurrences of the word ‘*master*’ with the word ‘*simpleThread*’. Save your file in a new folder as **simpleThread.java**.

☞ Add a label and two buttons so that your display looks like...



☞ Make sure you import the *java.util* package by including...

```
import java.util.*;
```

☞ Add the following variables to the applet class. Make sure that these, along with the label and button declarations, are in the applet declaration section and not inside one of the applet methods.

```
Thread t;  
boolean terminated = false;  
boolean started = false;
```


The '*terminated*' flag is used to indicate whether the **Thread** '*t*' has been terminated. The '*started*' flag is used to indicate whether the **Thread** '*t*' has been started.

☞ Add the appropriate listeners for the two buttons. Implement the listener in the event handling adapter class and add the corresponding event handling method. Include code to test for the source of the event.

☞ Add the following instructions to the event handling block for the Start Thread button...

```
if (!applet.started)  
{  
    applet.started = true;  
    applet.t = new Thread(this);  
    applet.t.start();  
}
```


As long as the thread has not already been started, the first instruction sets the applet attribute '*started*' to true indicating the thread is about to be started. The second instruction creates a new thread '*t*' and uses '*this*' to identify that the event handling adapter class is to manage the thread. Finally the thread is started. This calls the *run()* method which you're about to add to the event handling adapter class.

 Add the following *run()* method to the adapter class...

```
public void run()
{
    applet.terminated = false;

    while (!applet.terminated)
    {
        applet.date.setText((new Date()).toString());
        try
        {
            Thread.sleep(1000);
        }
        catch (Exception e)
        {
        }
    }
}
```

The first instruction ensures that the ‘*terminated*’ flag is set to false. The while loop ensures that the thread continues to execute until the thread is properly terminated. The actual thread activity sets the **Label** text to the current date and time and then waits for 1000 milliseconds (a second) before updating. The sleep method throws an exception that should be caught.

 Add the following instructions to the event handling block for the Stop Thread button...


```
applet.stopThread();
applet.started = false;
```

The first instruction calls an applet method called *stopThread()*. This method will be responsible for all the activities related to stopping the thread. The second instruction set the applet flag ‘*started*’ to false indicating that the thread is no longer running.

 Add the following *stopThread()* method to the applet class...

```
public void stopThread()
{
    terminated = true;
}
```

This method simply sets the ‘*terminated*’ flag to true. Remember that the thread continues to run until this variable is true. This is all you need to do to stop the thread.

 It is also important that the thread be stopped when the applet/application shuts down. Add the following instruction to the beginning of applet’s *quit()* method...

```
stopThread();
```

☞ Compile and test your program. After you've pressed the Start Thread button, the output should look something like...



Pressing the Stop Thread button should leave the current time displayed ... it will no longer be updated each second. Pressing the Start Thread button again will create a new thread and re-start the clock.

For your information, the complete **simpleThread.java** code is included here...

```
import java.applet.Applet;
import java.awt.event.*;
import java.awt.Frame;
import java.awt.*;
import java.util.*;

//=====
public class simpleThread extends Applet
{
    static int VSIZE = 150;
    static int HSIZE = 300;

    Label date;
    Button startDate, stopDate;

    Thread t;
    boolean terminated = false;
    boolean started = false;

    public static void main(String args[])
    {
        simpleThreadFrame frame = new simpleThreadFrame ("simpleThread");
        frame.setSize(HSIZE,VSIZE);
        frame.show();
    }

    public void init(EventAdapter adapter)
    {
        setLayout(null);

        date = new Label("Press the Start Thread button to see the date");
        add(date);
        date.setBounds(10,10,250,25);

        startDate = new Button("Start Thread");
```

```

        add(startDate);
        startDate.setBounds(10, 50, 100, 25);
        startDate.addActionListener(adapter);

        stopDate = new Button("Stop Thread");
        add(stopDate);
        stopDate.setBounds(120, 50, 100, 25);
        stopDate.addActionListener(adapter);
    }

    public void stopThread()
    {
        terminated = true;
    }

    public void quit()
    {
        stopThread();
        System.exit(0);    //exit on System exit box clicked
    }
}

//=====
class simpleThreadFrame extends Frame
{
    public simpleThreadFrame (String frameTitle)
    {
        super(frameTitle);

        simpleThread applet = new simpleThread();
        EventAdapter adapter = new EventAdapter(applet, this);
        applet.init(adapter);
        add ("Center", applet);

        this.addWindowListener(adapter);
    }
}

//=====
class EventAdapter extends WindowAdapter implements Runnable, ActionListener
{
    simpleThread applet;
    simpleThreadFrame frame;

    EventAdapter(simpleThread applet, simpleThreadFrame frame)
    {
        this.applet = applet;
        this.frame = frame;
    }

    public void windowClosing(WindowEvent wEvt)
    {
        applet.quit();
    }
}

```

```

public void run()
{
    applet.terminated = false;

    while (!applet.terminated)
    {
        applet.date.setText((new Date()).toString());
        try
        {
            Thread.sleep(1000);
        }
        catch (Exception e) {}
    }
}

public void actionPerformed(ActionEvent e)
{
    if (e.getSource() == applet.stopDate)
    {
        applet.stopThread();
        applet.started = false;
    }


    else if (!applet.started)
    {
        applet.started = true;
        applet.t = new Thread(this);
        applet.t.start();
    }
}
}

```

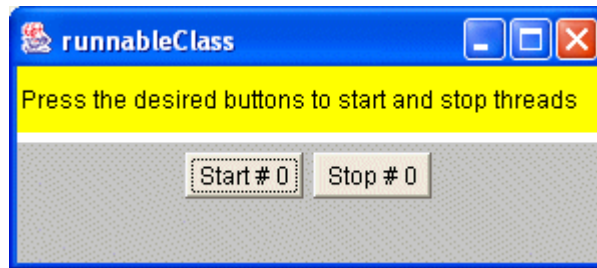
Runnable Class containing a Thread

If the threaded process centers on a single object, it is more efficient to extend that object's class and give it its own thread. The process involves similar activities to those included when creating a **Runnable** applet, but these activities are organized a little bit differently. The stages involved in the process are...

- create a **Runnable** subclass of the desired object
- create a thread in the constructor of this **Runnable** subclass with itself as the target
- overload the thread *start()* and *stop()* methods in the **Runnable** subclass so that the applet/application can access the thread
- provide a *run()* method that contains the thread code
- allow the applet/application to create as many instances of the **Runnable** subclass as it needs

 Starting with your **master.java** file, replace all occurrences of the word '*master*' with the word '*runnableClass*'. Save your file in a new folder as **runnableClass.java**.

☞ Add a label and two buttons so that your display looks like...



For help in laying out your components, this screen uses the north and center sections of a BorderLayout. The north portion contains the label with the instructions, and the center portion contains a panel that contains a start button and a stop button.

☞ Make sure you import the *java.util* package by including...

```
import java.util.*;
```

Creating a Runnable Subclass

In this application the **Runnable** subclass will display the current date and time as a **Label** component. The class is named **Clock** and extends **Label**. It also implements the **Runnable** interface so that it can run a thread...

```
class Clock extends Label implements Runnable
```

The class itself must contain a constructor and the *run()* method (because of the **Runnable** interface.) It will also contain a *start()* and *stop()* method to provide thread control...

```
class Clock extends Label implements Runnable
{
    public Clock()
    {
    }

    public void start()
    {
    }

    public void run()
    {
    }

    public void stop()
    {
    }
}
```

☞ Add the **Clock** class with its constructor, *start()*, *run()* and *stop()* methods. Make sure that the **Clock** class is not an inner class of the applet class, application class or the adapter class. It exists as a separate class.

Creating the Thread

The **Clock** class constructor is responsible for creating the thread ... when the **Clock** is created, so is the thread. A **Thread** variable must be both declared and instantiated.

✎ Add the **Thread** variable declaration to the **Clock** class attribute declaration area...

```
private Thread t;
```

✎ Add the thread object instantiation to the **Clock** constructor...

```
t = new Thread(this);
```

Notice that the target for this thread is the **Clock** object itself. Keeping in mind that the **Clock** class extended the **Label** class, text can be written to the **Clock** object...

```
this.setText("Clock created");
```

✎ Add this *setText()* instruction to the **Clock** constructor.

Overloading the *start()* and *stop()* Methods

In order for components on the main screen to control the thread, *start()* and *stop()* methods must be included in the **Clock** class. The *start()* method will initialize the '*terminated*' flag used to stop the thread. It will also issue the thread *start()* instruction.

✎ Add the following instructions to the **Clock** class *start()* method...

```
terminated = false;  
t.start();
```

Make sure that you have declared a boolean variable called *terminated* in the attribute declaration section of the **Clock** class.

The *stop()* method is responsible for setting the '*terminated*' flag to true, indicating that the thread is to stop.

✎ Add the following instructions to the **Clock** class *stop()* method...

```
terminated = true;
```

Defining the *run()* Method

The *run()* method contains the actual code to be executed by the thread. It is called automatically when the *t.start()* instruction is executed.

✎ Add the following instructions to the **Clock** class *run()* method...

```
while (!terminated)
{
    this.setText((new Date()).toString());
    try
    {
        Thread.sleep(1000);
    }
    catch (Exception e) { }
}
this.setText("Clock Terminated");
```

While the thread is active (the terminated flag is true) the current date and time is displayed as a **Label**. It is updated every 1000 milliseconds. The thread is stopped when the terminated flag changes to false. It falls out of the while loop and displays “Clock terminated” in place of the date and time.

Creating Clock Class Instances

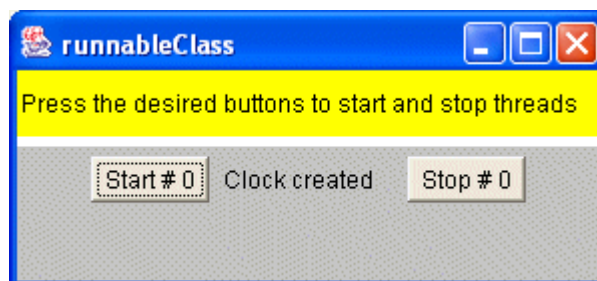
The start and stop buttons are responsible for starting and stopping the **Clock** thread, but before the thread is activated, an instance of the **Clock** class must be created.

✎ Add the following instruction to the *init()* method of the applet.

```
myClock = new Clock();
```

Make sure you have declared a **Clock** variable called ‘*myClock*’ in the attribute declaration area of the applet class.

✎ Add this **Clock** object between the start and stop buttons on the main screen. Compile and test your program. The display should now look something like...



Starting and Stopping the Thread

The start and stop buttons must be activated so that they can control the thread.

☞ Register interest for each button with the appropriate listener. Add code to the adapter to differentiate between events coming from the start and stop buttons.

The start button should call the clock's *start()* method...

```
applet.myClock.start();
```

Remember that the **Clock** object is defined in the applet class and the event handling code is in the adapter class, so you need to specify the variable as '*applet.myClock*'.

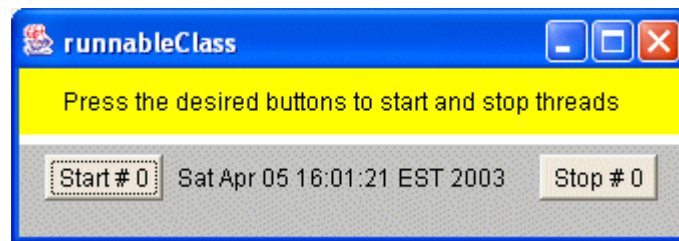
The stop button should call the clock's *stop()* method...

```
applet.myClock.stop();
```

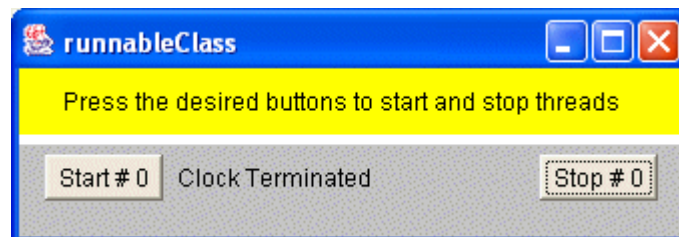
Again because the **Clock** object is defined in the applet class and the event handling code is in the adapter class, you need to specify the variable as '*applet.myClock*'.

☞ Add the start and stop button event handling code to appropriate block in the event handling adapter.

☞ Compile and test your program. Pressing the start button you should see something like...



Pressing the stop button you should see...



The complete code is include here for comparison purposes...

```

import java.applet.Applet;
import java.awt.event.*;
import java.awt.Frame;
import java.awt.*;
import java.util.*;

//=====
public class runnableClass extends Applet
{
    static int HSIZE = 300;
    static int VSIZE = 130;

    Label info;

    Panel p1, p2;
    Button starts;
    Button stops;
    Clock myClock;

    public static void main(String args[])
    {
        runnableClassFrame frame = new runnableClassFrame ("runnableClass");
        frame.setSize(HSIZE,VSIZE);
        frame.show();
    }

    public void init(EventAdapter adapter)
    {
        info = new Label("Press the desired buttons to start and stop threads");

        myClock = new Clock();
        starts = new Button("Start #0");
        stops = new Button("Stop #0");

        setLayout(new BorderLayout(5,5));

        p1 = new Panel();
        p1.setBackground(Color.yellow);
        p1.add(info);
        add(p1, BorderLayout.NORTH);

        p2 = new Panel();
        p2.setBackground(Color.lightGray);
        p2.add(starts);
        p2.add(myClock);
        p2.add(stops);
        add(p2, BorderLayout.CENTER);

        setVisible(true);

        starts.addActionListener(adapter);
        stops.addActionListener(adapter);
    }
}

```

```

    public void quit()
    {
        myClock.stop();
        System.exit(0);    //exit on System exit box clicked
    }
}

//=====
class runnableClassFrame extends Frame
{
    public runnableClassFrame (String frameTitle)
    {
        super(frameTitle);

        runnableClass applet = new runnableClass();
        EventAdapter adapter = new EventAdapter(applet, this);
        applet.init(adapter);
        add ("Center", applet);

        this.addWindowListener(adapter);
    }
}

//=====
class EventAdapter extends WindowAdapter implements ActionListener
{
    runnableClass applet;
    runnableClassFrame frame;

    EventAdapter(runnableClass applet, runnableClassFrame frame)
    {
        this.applet = applet;
        this.frame = frame;
    }

    public void windowClosing(WindowEvent wEvt)
    {
        applet.quit();
    }

    public void actionPerformed(ActionEvent e)
    {
        Object source = e.getSource();

        if (source == applet.starts)
            applet.myClock.start();
        else
            applet.myClock.stop();
    }
}

```

```
//=====
class Clock extends Label implements Runnable
{
    private Thread t;
    boolean terminated = false;

    public Clock()
    {
        t = new Thread(this);
        this.setText("Clock created");
    }

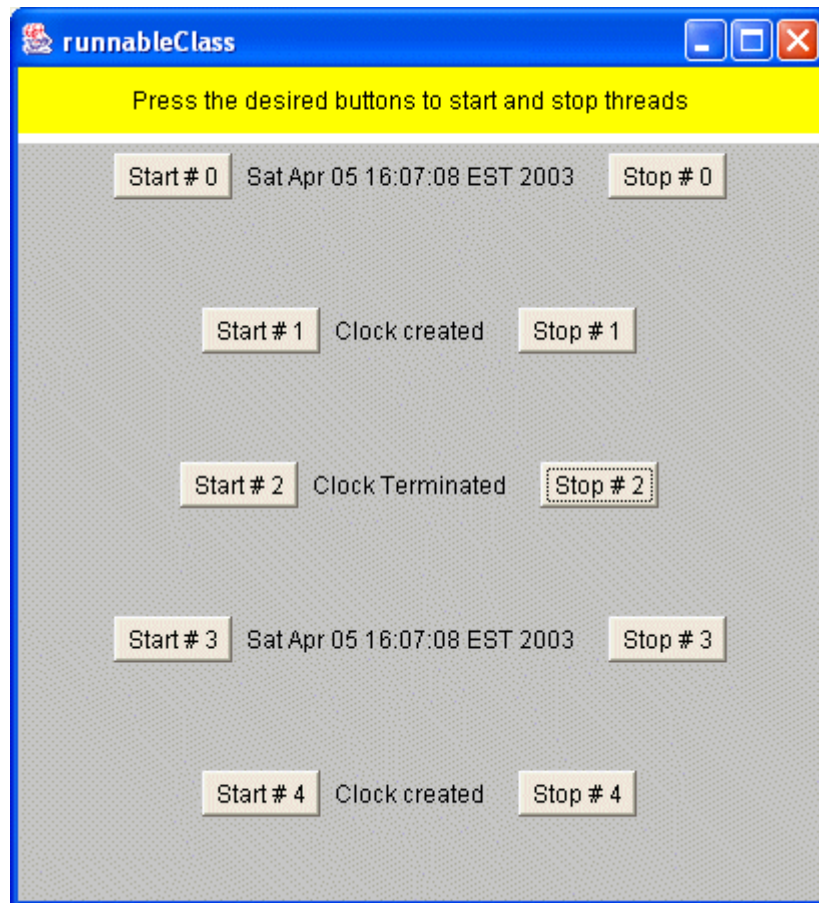
    public void start()
    {
        terminated = false;
        t.start();
    }

    public void run()
    {
        while (!terminated)
        {
            this.setText((new Date()).toString());
            try
            {
                Thread.sleep(1000);
            }
            catch (Exception e) { }
        }
        this.setText("Clock Terminated");
    }


    public void stop()
    {
        terminated = true;
    }
}

```

This output of this program is very similar to what you saw in the **simpleThread.java** program you wrote earlier. The problem with that first method is that you can only have one thread running. With this method of defining a **Runnable** subclass, you can easily run multiple threads simply by creating new **Clock** objects. For example, the following output screen shows five thread running, some created, some active and some that have been terminated...



To accomplish this five different **Clock** objects were created, each with their own start and stop buttons. Each of them creates its own thread which operates quite independently of the other four.

 Add a second **Clock** thread to your program. You should be able to start and stop both clocks independently.

Class Exercise – Multi-Threading

- a) Demonstrate a fully functional two Clock threads applet/application.
- b) Create a new applet/application that runs two threads ... one to display the current date and time, and a second to cycle the colour of the background through a fixed number of colours. The speed that the colour changes is dependent on the position of the scrollbar ... if it is up, the colours change very quickly (every 10th of a second), if it is down, the colours change very slowly (every second). The display should look something like...



Hints:

1. Use the **Clock** class created in the previous exercise to display the current date and time.
2. Use a **Scrollbar** object to control the speed of the colour change. Use its minimum and maximum values to define the range of milliseconds you need for the colour change.
3. Make a second **Runnable** class for the changing colours. Make the class extend the **Panel** class and then change its background colour in the thread code. Read the scrollbar's value (*getValue()* method) and use this to set the time that the thread will sleep.
4. Use the **BorderLayout** to position the three elements on the screen.

Classes in C++

Classes in C++ serve the same function as they do in Java ... to group together the characteristics and behaviours of a particular object. While the function is the same, the format and syntax are slightly different. This section looks at defining classes for use in C++ applications.

Defining Classes

In C++, all classes are defined in header files. The class definition contains...

- constant declarations
- class attribute declarations
- instance attribute declarations
- prototypes for most class and instance functions
- in-line functions for short (1 to 2 lines) class and instance functions

A class definition in C++ is structured a little bit differently than in Java...

```
//required classes included here
using namespace std;

class className
{
private:
    //class attributes and constants declared here

    //instance attributes may be declared here

protected:
    //instance attributes may be declared here

public:

    //class constructor declared here
    className(){}

    //class destructor declared here
    ~className()
        { }

    //short instance functions defined here

    // instance functions prototyped here

};
```

As in any C code the first section provides access to the required libraries. Recall that namespaces are C++'s way of creating scope for global identifiers. In most cases the required libraries are in the standard namespace 'std'.

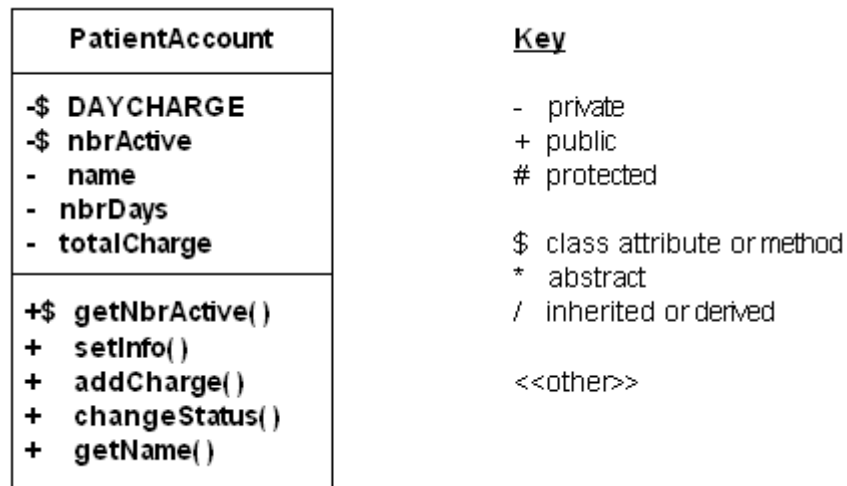
The class definition is next. The keyword `class` is followed by the desired class name. Curly brackets enclose the entire class. Pay close attention to the semi-colon at the end of the class definition. This must be there for proper compiling.

Inside the class block there are three sections... one for private attributes and functions, another for protected attributes and functions, and the third for public attributes and functions. Each starts with the appropriate keyword followed by a colon.

Within the public, protected and private blocks attributes are defined as they would in any C program... data type followed by the variable name. Each line is terminated with a semi-colon.

Unless a function is only one or two lines long, the class definition simply holds its prototype. This prototyping is done exactly the same way as functions are prototyped in a regular C program. If the function is very short it is defined in full. This is called defining the function in-line. The code is actually substituted in at compile time making a larger executable. The advantage is in the increased performance that results from the in-line code.

Say for example the following class diagram is to be coded...



The class includes a constant value *DAYCHARGE*, a class variable '*nbrActive*', three instance variables '*name*', '*nbrDays*' and '*totalCharge*', one class function *getNbrActive()*, and four instance functions *setInfo()*, *addCharge()*, *changeStatus()* and *getName()*.

First the basic class structure is implemented...

```
class PatientAccount
{
private:
protected:
public:
};
```

Next the required library access is specified within the standard namespace.

```
include <string>
using namespace std;

class PatientAccount
{
private:
protected:
public:
};
```

Defining Constants

Constants are defined in the private section. As in Java, they are preceded by the keyword **static** since they are considered class attributes. They are also preceded by the keyword **const** indicating that the values will not change. They cannot however, be initialized in the class definition. This must be done outside the class.

Adding the constant attributes to the class definition gives...

```
include <string>
using namespace std;

class PatientAccount
{
private:
    static const float DAYCHARGE;

protected:
public:
};
```

Defining Class Variables

Class variables are also defined in the private section. As in Java, they too are preceded by the keyword **static**. Adding the class attributes to the class definition gives...

```
include <string>
using namespace std;

class PatientAccount
{
private:
    static const float DAYCHARGE;
    static int nbrActive;

protected:
public:
};
```

Defining Instance Variables

Instance variables may be defined in the private, protected or public section depending on the type of access you want to allow. Keeping in mind the basic principles of object oriented programming, all attributes should be hidden unless it is absolutely necessary to make them globally accessible. That means that most if not all instance attributes should be either private or protected.

Adding the instance attributes to the class definition gives...

```
include <string>
using namespace std;

class PatientAccount
{
private:
    static const float DAYCHARGE;
    static int nbrActive;

    string name;
    int nbrDays;
    float totalCharge;

protected:
public:
};
```

Declaring Class Functions

Class functions are defined in the public section of the class declaration. Like class variables they are preceded by the keyword **static**. Adding the class function definition gives...

```
include <string>
using namespace std;

class PatientAccount
{
private:
    static const float DAYCHARGE;
    static int nbrActive;

    string name;
    int nbrDays;
    float totalCharge;

protected:
public:
    static float getnbrActive()
        { return nbrActive; }
};
```

Because this function just returns the value of a class attribute it is defined as an in-line function.

Declaring Instance Functions

Instance functions are may be defined in the public, private or protected section of the class declaration. Short functions are defined in-line, while longer functions are prototyped. Adding the instance functions to the class definition gives...

```
include <string>
using namespace std;

class PatientAccount
{
private:
    static const float DAYCHARGE;
    static int nbrActive;

    string name;
    int nbrDays;
    float totalCharge;

public:
    static float getnbrActive()
        { return nbrActive; }

    void addCharge(const float cost)
        { totalCharge += cost; }

    string getName( ) const
        { return name; }

    void changeStatus(const int change);

    void setInfo(string patient, int days);

};
```

Class Constructor

The class constructor serves the same purpose in C++ as it does in Java. Any initialization that needs to be done when a new object of this class is instantiated should be done here. If the constructor contains more than an instruction or two, it gets prototyped in the class definition and defined elsewhere. If it is small, it can be defined in-line.

In this particular case there is no specific initialization required. The constructor must still be included though. Adding the constructor to the public section of class definition gives...

```
public:

    PatientAccount() { }

    static float getnbrActive()
        { return nbrActive; }

    . . .
```

Class Destructor

Each class in a C++ application needs a class destructor as well as a class constructor. This destructor function is called when the object goes out of scope or when it is released. Normally the class destructor contains any “clean-up” code for the class. Like any other function, if it is small it can be defined in-line, otherwise it is declared here and defined elsewhere.

In this particular case there is no specific clean-up required. The destructor must still be included though. Adding the destructor to the public section of class definition gives...

```
public:
    PatientAccount() { }

    ~PatientAccount() { }

    static float getnbrActive()
    { return nbrActive; }
    . . .
```

Notice that the only difference between the constructor and the destructor is the tilde (~) in front of the name. The class definition is now complete and is shown here in its entirety. The protected section was removed since it contained no attributes or functions.

```
include <string>
using namespace std;

class PatientAccount
{
private:
    static const float DAYCHARGE;
    static int nbrActive;

    string name;
    int nbrDays;
    float totalCharge;

public:
    PatientAccount() { }

    ~PatientAccount() { }

    static float getnbrActive()
    { return nbrActive; }

    void addCharge(const float cost)
    { totalCharge += cost; }

    string getName( ) const
    { return name; }

    void changeStatus(const int change);

    void setInfo(string patient, int days);
};
```

Defining Class and Instance Functions and Initializing Constants

All functions prototyped in the class definition must be defined in one of two ways...

- in an accompanying *.cpp* file
- or
- in the same header file but outside the class definition

This is also the place where any constants declared in the header file are initialized.

Accompanying .cpp File

The more common way of defining class and instance functions is in an accompanying file. The file itself must be given the same name as the class definition file. For example, if your class definition is called **PatientAccount.h**, then the accompanying file must be called **PatientAccount.cpp**.

The format for this **.cpp** file is...

```
include "headerFile.h"

//constants initialized here

//functions defined here
```

The basic format for the **PatientAccount.cpp** file is...

```
include "PatientAccount.h"

//constants initialized here

//functions defined here
```

Defining Constants

Initializing constants is pretty straightforward ... the data type, class and attribute names, and the assigned value are all specified. The format for constant initialization is...

```
const dataType className::constantName = assignedValue;
```

The scope resolution operator (::) is used to connect constant name with the class to which it belongs.

In the **PatientAccount.h** file there was only one constant value declared. The file with this constant initialization is now...

```
include "PatientAccount.h"

const float PatientAccount::DAYCHARGE = 150.00;
```


Defining Functions

Defining functions is also pretty straightforward ... it is almost identical to defining a normal C function. The only difference is that you must include the class name in the function declaration statement. The format for defining functions is...

```
returnDataType className::functionName (parameter list)
{
    //function code goes here
}
```

With the functions that were declared in **PatientAccount.h** defined, the **PatientAccount.cpp** file is now...

```
include "PatientAccount.h"

const float PatientAccount::DAYCHARGE = 150.00;

void PatientAccount::changeStatus(int change)
{
    if (change == 0)    //not leaving
    {
        nbrDays++;
        totalCharge += DAYCHARGE;
        cout << "\n" << name << "'s stay is now at " << nbrDays << " days";
    }
    else                //leaving
    {
        nbrActive--;
        cout << "\n" << name << "stayed " << nbrDays << " days";
        cout << " and accumulated a total bill of $" << totalCharge;
        cout << "\nThere are now " << nbrActive << " active accounts";
    }
}

void PatientAccount::setInfo(string patient, int days)
{
    nbrActive++;

    name = patient;
    nbrDays = days;
    totalCharge = days * DAYCHARGE;

    cout << "\n" << name << " admitted for " << days << " days";
}
```

In the Header File

If you choose to define the constants and functions in the header file, they must be defined outside the class definition. The definition code would follow the class definition in the **PatientAccount.h** file...

```
class PatientAccount
{
    //class contents as declared previously
};

const float PatientAccount::DAYCHARGE = 150.00;

void PatientAccount::changeStatus(int change)
{
    if (change == 0)    //not leaving
    {
        nbrDays++;
        totalCharge += DAYCHARGE;
        cout << "\n" << name << "'s stay is now at " << nbrDays << " days";
    }
    else                //leaving
    {
        nbrActive--;
        cout << "\n" << name << "stayed " << nbrDays << " days";
        cout << " and accumulated a total bill of $" << totalCharge;
        cout << "\nThere are now " << nbrActive << " active accounts";
    }
}

void PatientAccount::setInfo(string patient, int days)
{
    nbrActive++;

    name = patient;
    nbrDays = days;
    totalCharge = days * DAYCHARGE;

    cout << "\n" << name << " admitted for " << days << " days";
}
```

Using C++ Classes

Once the class definition header file and the accompanying *.cpp* file are created, the class can then be used by any C++ application. The first step in using a class is to specify the link to the class header file. This is done using a standard include statement...

```
#include "className.h"
```

For example, to be able to use the **PatientAccount** class you must include the **PatientAccount.h** file in your application...

```
#include "PatientAccount.h"
```

Instantiating Objects

This driver class is where all required objects are instantiated. An object can be created as a static object...

```
className objectName;
```

as a dynamic object..

```
className *objectName = new className();
```

as a static array of objects...

```
className objectName[SIZE];
```

or as a dynamic array of objects...

```
className *objectName = new PatientAccount[SIZE];
```

Accessing Instance Functions

Once an object is instantiated its instance functions are accessible using the dot operator...

```
objectName.functionName(parameter list);
```

Accessing Class Functions

Class functions are accessible using the class name and the scope resolution operator...

```
className::functionName();
```

Freeing Dynamic Objects

If you choose to use dynamic objects, make sure that you free them up once they are no longer needed. The instruction to free up one dynamic object is...

```
delete(objectName);
```

The instruction to free up a dynamic array of objects is...

```
delete[] objectName;
```

To see these instructions in an application, take a look at the following **hospital.cpp** driver file. Line numbers have been added for reference purposes only.

```

1.  //hospital.cpp

2.  #include <iostream>
3.  #include <string>
4.  using namespace std;

5.  #include "PatientAccount.h"

6.  enum {BLANK, ADMIT, EXTEND, RELEASE, COUNT, DONE};

7.  const int MAXPATIENTS = 10;

8.  int PatientAccount::nbrActive = 0;

9.  int getMenuChoice();

10. void admitPatient(PatientAccount patients[], int nbr);
11. void extendStay(PatientAccount patients[]);
12. void releasePatient(PatientAccount patients[]);
13. void showNbrPatients();
14. int findPatient(PatientAccount patients[], string patient);

15. //=====
16. void main()
17. {
18.     bool done = false;
19.     int patientNbr = 0;

20.     PatientAccount * patients = new PatientAccount[MAXPATIENTS];

21.     do
22.     {
23.         switch(getMenuChoice())
24.         {
25.             case ADMIT:
26.                 admitPatient(patients, patientNbr++);
27.                 break;
28.             case EXTEND:
29.                 extendStay(patients);
30.                 break;
31.             case RELEASE:
32.                 releasePatient(patients);
33.                 break;
34.             case COUNT:
35.                 showNbrPatients();
36.                 break;
37.             case DONE:
38.                 delete[] patients;
39.                 char ch;
40.                 cin >> ch;
41.                 done = true;
42.             }
43.         }
44.         while (!done);
45.     }

```

```

46. //=====
47. int getMenuChoice()
48. {
49.     int choice;
50.     bool okay= false;

51.     do
52.     {
53.         cout << "\nPlease choose an item from the menu:";
54.         cout << "\n\t1. Admit Patient";
55.         cout << "\n\t2. Extend Stay for Patient";
56.         cout << "\n\t3. Release Patient";
57.         cout << "\n\t4. Prescribe Meds";
58.         cout << "\n\t5. Order Surgery";
59.         cout << "\n\t6. Show Patient Count";
60.         cout << "\n\t7. Done";
61.         cout << "\n\nYour choice: ";

62.         cin >> choice;

63.         if (choice < ADMIT || choice > DONE)
64.             cerr << "Invalid choice ... try again\n\n";
65.         else
66.             okay = true;
67.     }
68.     while (!okay);

69.     return choice;
70. }

71. //=====
72. void admitPatient(PatientAccount patients[], int nbr)
73. {
74.     string name;
75.     int stay;

76.     cout << "Name of patient: ";
77.     cin >> name;
78.     cin.ignore();
79.     cout << "Length of stay: ";
80.     cin >> stay;

81.     patients[nbr].setInfo(name.lower(), stay);
82. }

83. //=====
84. void extendStay(PatientAccount patients[])
85. {
86.     string name;
87.     int which;

88.     cout << "Name of patient: ";
89.     cin >> name;
90.     cin.ignore();
91.     which = findPatient(patients, name);
92.     if (which < MAXPATIENTS)
93.         patients[which].changeStatus(EXTEND);
94. }

```

```

95. //=====
96. void releasePatient(PatientAccount patients[])
97. {
98.     string name;
99.     int which;

100.     cout << "Name of patient: ";
101.     cin >> name;
102.     cin.ignore();
103.     which = findPatient(patients, name);
104.     if (which < MAXPATIENTS)
105.         patients[which].changeStatus(RELEASE);
106. }

107. //=====
108. void showNbrPatients()
109. {
110.     cout << "\nThere are " << PatientAccount::getNbrActive() << " patients\n";
111. }

112. //=====
113. int findPatient(PatientAccount patients[], string patient)
114. {
115.     bool found = false;
116.     int patientNbr = 0;

117.     while (!found && patientNbr < MAXPATIENTS)
118.     {
119.         if (patients[patientNbr].getName() == patient.toLowerCase())
120.             found = true;
121.         else
122.             patientNbr++;
123.     }

124.     if (!found)
125.         cerr << "\nRequested patient not found\n\n";

126.     return patientNbr;
127. }

```

The first class related line is the link to the class header file. This is done in line 5. There is no need to link specifically to the accompanying *.cpp* file ... this is done automatically.

Line 6 contains an anonymous enumeration statement. Each of the defined constants is assigned a sequential integer value starting with zero. This means that any time BLANK is used in the code, the value 0 is substituted. Anytime RELEASE is used, the value 3 is substituted. This is an easy way to define menu selection items.

Line 8 is used to initialize the class variable *nbrActive*. The class name and scope resolution operator are used to access this variable. Remember that because a class variable is declared as a static variable, its scope is the lifetime of the program ...it comes into existence before any instances of the class are created.

In line 20 a dynamic array is created to hold 10 patient objects. This instruction will call the **PatientAccount** constructor 10 times, one for each of the objects in the array.

Each of the functions *admitPatient()*, *extendStay()*, *releasePatient()* and *findPatient()* expects the array of **PatientAccount** objects as an argument. This array of objects is passed in exactly the same way as any standard array is passed in C.

The next reference to an object of the **PatientAccount** class is in line 81 in the *admitPatient()* function. This call is to *setInfo()*, an instance function of the **PatientAccount** class. The user has just entered the patient's name and how many days the patient will be hospitalized. This information is passed to that particular patient's record for storage. A similar call is made in line 105.

Line 110 calls the class function *getNbrActive()*. This call determines how many of the 10 available accounts are active. Notice that the class name and the scope resolution operator are used for this class function. Like static variables, static functions are accessible before any instance of the class is created. This can be very useful during initialization.

Finally, line 119 calls the accessor function *getName()*. The loop that this call is in is designed to cycle through each of the available patient records searching for the specified patient's name. If found, the *findPatient()* function returns the record number. Otherwise it generates an error message and return record number 0.

***this Pointer**

Every time that you call an object's instance function in the C++ environment a pointer to the actual object is passed to the function along with the specified arguments. For example, in the previous program the instruction on line 128 ...

```
patients[nbr].setInfo(name.tolower(), stay);
```

calls the *setInfo()* instance function. Two attributes in the **PatientAccount** object *patients[nbr]* are assigned the passed in values for the patient's name and length of stay. When this instruction is compiled it becomes...

```
setInfo(&patients[nbr], name.tolower(), stay);
```

where the first argument is the address of the object itself. This allows the attributes in the object to be modified. Also at compile time, the function's prototype becomes...

```
void setInfo(PatientAccount *this, string patient, int days);
```

where the first argument, called the ***this pointer**, provides the required access to the objects attributes.

Note: The ***this** pointer is passed in automatically. There is no need to pass the address of the object explicitly.

The ***this** pointer can also be used to return an object from a function. The instruction...

```
return (*this);
```

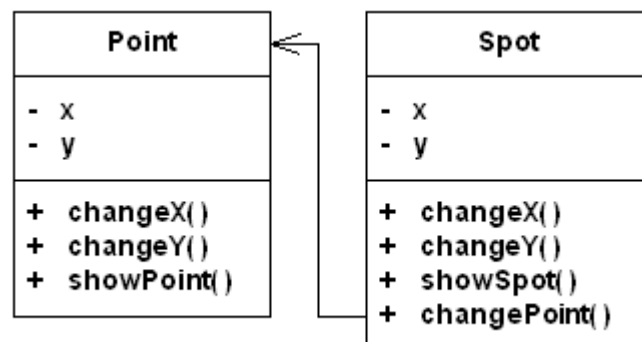
passes the entire object back to the calling function.

Friends of Classes

One of the four basic goals of object oriented programming is to hide as much information as possible from the “outside world”. For this reason, most if not all, instance attributes are declared as private and accessor methods or functions are written to provide access. Occasionally however, there are times that you need to give direct data access to certain functions in the application that aren’t in the class. This is where friend functions come in.

A friend of a class is a function that is not a member of the class, but has access to the private members of the class. Friend functions are treated as if they are true members of the class. They can exist as stand-alone functions or as functions defined within another class. The friendship must however, be specifically stated by the class allowing this special access privilege.

Say for example the following two classes are defined for an application...



The **Point** class is straightforward. It has two private attributes and three public functions. The **Spot** class also has two private attributes and three public functions. It also has a public class that requires access to attributes within the **Point** class. To allow this access the **Point** class must declare the *changePoint()* function as a friend function.

Unfortunately, friend functions are not passed the **this* pointer when they are called. That means that you as the programmer have to provide the address of the original class (in this case the **Point** class) as one of the arguments in the parameter list.

Forgetting about the friend function for the moment, the basic class definition file for the **Point** class is...

```
class Point
{
private:
    int x, y;

public:
    Point(){x = 0; y = 0;};
    Point(int a, int b) {x = a; y = b;};

    void changeX(int value) {x = value;}
    void changeY(int value) {y = value;}

    void showValues() {cout << "\n(" << x << ", " << y << ")";}
};
```

The basic definition for the **Spot** class is...

```
class Spot
{
private:
    int x, y;

public:
    Spot()(){x = 0; y = 0;};
    Spot(int a, int b) {x = a; y = b;};

    void changeX(int value) {x = value;}
    void changeY(int value) {y = value;}

    void showValues() {cout << "\n(" << x << ", " << y << ")";}
};
```

Both have a default constructor as well as a second constructor that accepts two integer values ... one for x and the other for y. The *changeX()* and *changeY()* functions accept an integer value and assign it to the appropriate attribute. The *showValues()* function displays the x and y values as a coordinate in the form (x, y).

The **Point** class must identify that the friend function *changePoint()* is allowed access to its private attributes. It does this by using the keyword **friend** ...

```
friend void otherClass::friendFunctionName(arguments);
```

One of the arguments must be the address of an object of this class. For example, to define the *changePoint()* function as a friend, the **Point** class public section must include the following...

```
friend void Spot::changePoint(int, int, Point &);
```

The class where the function is actually defined is the **Spot** class. The last argument in a call to this function must be the address of a **Point** object. Adding this prototype to the **Point** class gives...

```

class Point
{
private:
    int x, Y;

public:
    Point(){x = 0; y = 0;};
    Point(int a, int b) {x = a; y = b;};

    void changeX(int value) {x = value;}
    void changeY(int value) {Y = value;}

    void showValues() {cout << "\n(" << x << "," << y << ")";}

    friend void Spot::changePoint(int, int, Point &);
};

```

The corresponding code in the class that actually defines the friend function looks like a normal prototype...

```
void friendFunctionName(arguments);
```

The argument list must be the address of an object the class that the friend function will access. The prototype for the *changePoint()* function is therefore...

```
void changePoint(int, int, Point &);
```

The last argument is the address of a **Point** object. With this friend function added, the Spot class definition is...

```

class Spot
{
private:
    int x, y;

public:
    Spot() {x = 0; y = 0;};
    Spot(int a, int b) {x = a; y = b;};

    void changeX(int value) {x = value;}
    void changeY(int value) {y = value;}

    void showValues() {cout << "\n(" << x << "," << y << ")";}

    void changePoint(int a, int b, Point &p) {p.x = a; p.y = b};
};

```

Looking at the friend function specifier in the **Point** class definition, you can see that it will need to know of the existence of the **Spot** class. This is done by including a reference to the **Spot** class definition...

```
#include "Spot.h"
```

Similarly, because of the friend function prototype, the **Spot** class will need to know of the existence of the **Point** class. This is done using a forward declaration that ensures the compiler knows that the **Point** class is defined some where else...

```
class Point;
```

This line is included before the **Spot** class definition begins. Because each of these classes has a reference to the other, you need to add a preprocessor directive to each class header file that ensures that each class definition is only included once. The complete **Point** class definition is enclosed by...

```
#ifndef Point_h
#define Point_h

.
.
.

#endif
```

The first instruction checks to see if the variable *Point_h* is defined. If not, it defines the variable and continues with the class definition. If the variable is already defined, the entire definition is skipped over. The complete **Point** class definition is now...

```
#ifndef Point_h
#define Point_h

#include "Spot.h"          //for friend declaration

class Point
{
private:
    int x, y;

public:
    Point(){x = 0; y = 0;};
    Point(int a, int b) {x = a; y = b;};

    void changeX(int value) {x = value;}
    void changeY(int value) {Y = value;}

    void showValues() {cout << "\n(" << x << ", " << y << ")";}

    friend void Spot::changePoint(int, int, Point &);
};

#endif
```

Similarly, the **Spot** class needs to check to see if it has already been included. With the preprocessor directive added, the complete **Spot** class definition is...

```
#ifndef Spot_h
#define Spot_h

class Point;           //forward declaration

class Spot
{
private:
    int x, y;

public:
    Spot() {x = 0; y = 0;};
    Spot(int a, int b) {x = a; y = b;};

    void changeX(int value) {x = value;};
    void changeY(int value) {y = value;};

    void showValues() {cout << "\n(" << x << ", " << y << ")";}

    void changePoint(int a, int b, Point &p) {p.x = a; p.y = b;};
};

#endif
```

The driver class can now create objects of both the **Point** and **Spot** classes. The instructions...

```
Point p1(4, 3);
Spot s1(10, 20);
```

create two objects. The **Point** object *p1* has the coordinates (4,3). The **Spot** object *s1* has the coordinates (10, 20). Either set of coordinates can be modified using the instance functions *changeX()* and *changeY()*...

```
p1.changeX(12);
s1.changeY(30);
```

The **Point** object *p1* now has the coordinates (12,3) while the **Spot** object *s1* has the coordinates (10, 30).

Any **Spot** object can also modify a **Point** object using the friend function...

```
s1.changePoint(9, 13, p1);
```

This instruction changes the coordinates of **Point** object *p1* to (9, 13).

Because friend functions really defeat the object oriented principle of data encapsulation, they must be used with care. The only time you really need to use friend functions is for operator overloading. This is discussed in an upcoming section.

Memberwise Assignment

Just like with primitive data types the assignment operator (=) may be used to assign one object to another or to initialize one object with another object's data. By default each member of one object is copied to its counterpart in the other object.

For example if two **Point** objects are created using the following instructions...

```
Point p1(4, 3);  
Point p2();
```

the first object *p1* has coordinates of (4, 3). The second point *p2* gets the default coordinates of (0, 0). The coordinates of one can be assigned to the other using the assignment operator...

```
p2 = p1;
```

After this instruction is executed, both *p1* and *p2* have the coordinates (4, 3). Each member of object *p1* was assigned to the corresponding element in object *p2*.

This method of object copying works well as long as the object only contains primitive data types. If the object contains non-primitive attributes such as strings or pointers, this method may cause problems. To prevent this from happening you must create a specialized function called a **copy constructor**.

Copy Constructors

A copy constructor is a class-defined function that overrides the compiler's default copy. This function must ensure that each variable in the original object is copied to the corresponding variable in the new object. Before you can copy objects that contain non-primitive attributes, a copy constructor must be coded. The format of a copy constructor is quite simple...

```
className(const className &obj);
```

Say for example, that the following **Book** class is required in an application...

```
include <string>  
using namespace std;  
  
class Book  
{  
private:  
    string title;  
    int nbrChapters;  
    string *chapters;  
  
public:  
    Book();  
    Book(string, int, const string[]);  
  
    ~Book() { }
```

```

    void setChapter(const string &, int);

    void setTitle(const string t){title = t;}

    void printData( );
};

Book::Book()
{
    title = "";
    nbrChapters = 0;
}

Book::Book(string t, int nbr, const string chptrs[])
{
    title = t;
    nbrChapters = nbr;
    chapters = new string[nbr];
    for (int i = 0; i < nbr; i++)
        chapters[i] = chptrs[i];
}

void Book::setChapter(const string &chptr, int nbr)
{
    chapters[nbr] = chptr;
}

void Book::printData()
{
    cout << title << " contains the following chapters\n";

    for (int i = 0; i < nbr; i++)
        cout << "\t" << chapters[i] << "\n";
}

```

A simple driver function that uses this class to create one **Book** object is...

```

#include <iostream>
#include <string>
using namespace std;

#include "Book.h"

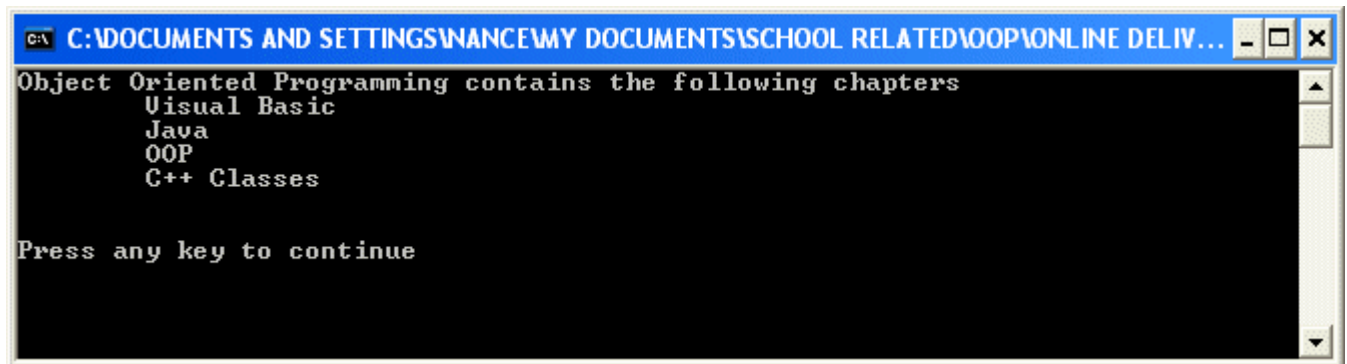
void main()
{
    string chapters[] = {"Visual Basic", "Java ", "OOP", "C++ Classes"};

    Book book1("Object Oriented Programming", 4, chapters);
    book1.printData();

    char ch;
    cout << "\n\nPress any key to continue";
    cin >> ch;
}

```

The output of this application is...



```
C:\DOCUMENTS AND SETTINGS\NANCE\MY DOCUMENTS\SCHOOL RELATED\OOP\ONLINE DELIV...
Object Oriented Programming contains the following chapters
    Visual Basic
    Java
    OOP
    C++ Classes

Press any key to continue
```

Now code is added to the driver to create a second book. The highlighted section shows how memberwise assignment is used to “copy” the contents of the first book to the second.

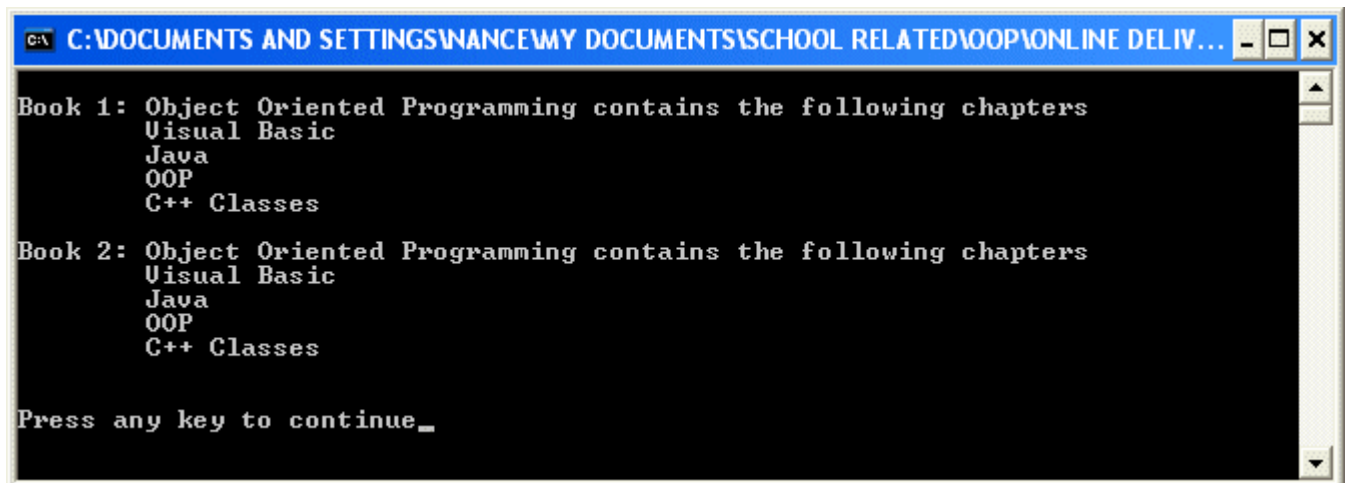
```
void main()
{
    string chapters[] = {"Visual Basic", "Java ", "OOP", "C++ Classes"};

    cout << "\nBook 1: ";
    Book book1("Object Oriented Programming", 4, chapters);
    book1.printData();

    cout << "\nBook 2: ";
    Book book2 = book1;
    book2.printData();

    char ch;
    cout << "\n\nPress any key to continue";
    cin >> ch;
}
```

The output of this code is...



```
C:\DOCUMENTS AND SETTINGS\NANCE\MY DOCUMENTS\SCHOOL RELATED\OOP\ONLINE DELIV...
Book 1: Object Oriented Programming contains the following chapters
    Visual Basic
    Java
    OOP
    C++ Classes

Book 2: Object Oriented Programming contains the following chapters
    Visual Basic
    Java
    OOP
    C++ Classes

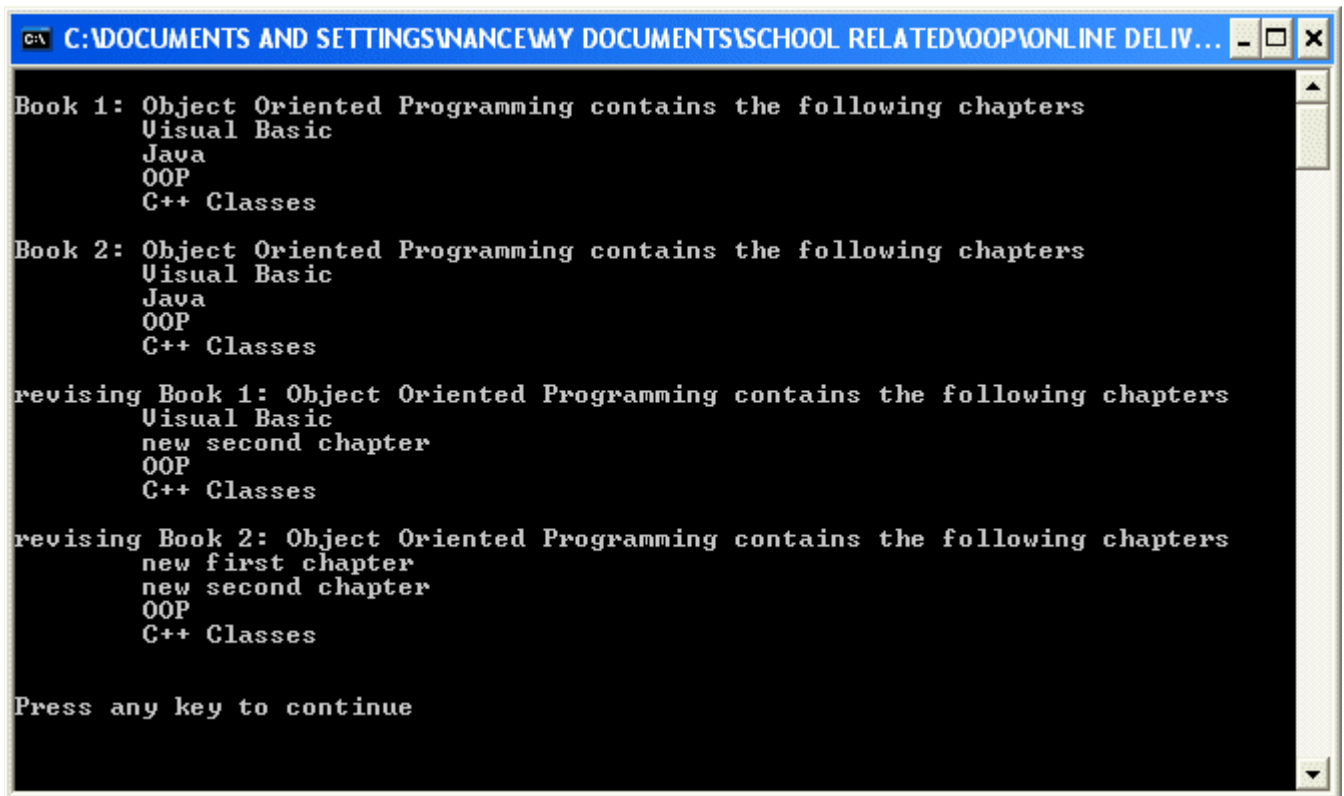
Press any key to continue_
```

It appears that the two books now have the same information. To make sure, two of the chapter titles are changed using the *setChapter()* function. The following code is added to the end of the *main()* function...

```
cout << "\nrevising Book 1: ";
book1.setChapter("new second chapter", 1);
book1.printData();

cout << "\nrevising Book 2: ";
book2.setChapter("new first chapter", 0);
book2.printData();
```

Looking closely at the output after the chapter revisions are made...



```
C:\DOCUMENTS AND SETTINGS\NANCE\MY DOCUMENTS\SCHOOL RELATED\OOP\ONLINE DELIV...
Book 1: Object Oriented Programming contains the following chapters
        Visual Basic
        Java
        OOP
        C++ Classes
Book 2: Object Oriented Programming contains the following chapters
        Visual Basic
        Java
        OOP
        C++ Classes
revising Book 1: Object Oriented Programming contains the following chapters
        Visual Basic
        new second chapter
        OOP
        C++ Classes
revising Book 2: Object Oriented Programming contains the following chapters
        new first chapter
        new second chapter
        OOP
        C++ Classes
Press any key to continue
```

The change to the first book looks fine. The change to the second book should flag a problem... the change to the first book exists here too. In actuality, there is only one list of chapters. When the memberwise assignment operator is used, it copies the primitive data types... in this case it copied the pointer. That means that both books are pointing to the same list of chapter headings. To get around this a copy constructor is required.

A copy constructor must be added to the public section of the **Book** class definition. Its form is...

```
Book(const Book&);
```

Because the object contains an array of strings, the copy constructor code must create an array the same size and then copy each string from the original array to the copy. This function, called *copyList*(), is declared in the private section the **Book** class definition...

```
void copyList(const Book &);
```

The actual code for the copy constructor and the *copyList*() function are defined after the **Book** class definition...

```
Book::Book(const Book &original)
{
    chapters = 0;
    copyList(original);
}

void Book::copyList(const Book &original)
{
    delete[] chapters;

    if (original.chapters != 0)
    {
        nbrChapters = original.nbrChapters;
        chapters = new string[nbrChapters];

        for (int j = 0; j < nbrChapters; j++)
            chapters[j] = original.chapters[j];
    }
}
```

The complete **Book.h** file with the changes highlighted, now looks like...

```
#include <iostream>
#include <string>
using namespace std;

class Book
{
private:
    string title;
    int nbrChapters;
    string *chapters;

    void copyList(const Book &);

public:
    Book();
    Book(string, int, const string[]);

    ~Book() { }

    Book(const Book&);
}
```

```

    void setChapter(const string &, int);

    void setTitle(const string t){title = t;}

    void printData( );
};

Book::Book()
{
    title = "";
    nbrChapters = 0;
}

Book::Book(string t, int nbr, const string chptrs[])
{
    title = t;
    nbrChapters = nbr;
    chapters = new string[nbr];
    for (int i = 0; i < nbr; i++)
        chapters[i] = chptrs[i];
}

void Book::setChapter(const string &chptr, int nbr)
{
    chapters[nbr] = chptr;
}

void Book::printData()
{
    cout << title << " contains the following chapters\n";

    for (int i = 0; i < nbrChapters; i++)
        cout << "\t" << chapters[i] << "\n";
}

Book::Book(const Book &original)
{
    chapters = 0;
    copyList(original);
}

void Book::copyList(const Book &original)
{
    delete[] chapters;

    if (original.chapters != 0)
    {
        nbrChapters = original.nbrChapters;
        chapters = new string[nbrChapters];

        for (int j = 0; j < nbrChapters; j++)
            chapters[j] = original.chapters[j];
    }
}

```

The first thing that the *copyList()* function does is ensure that no existing chapter list exists. It does this by deleting the array. As long as there are chapter titles in the object being copied, the function creates a new array and then copies the strings one by one.

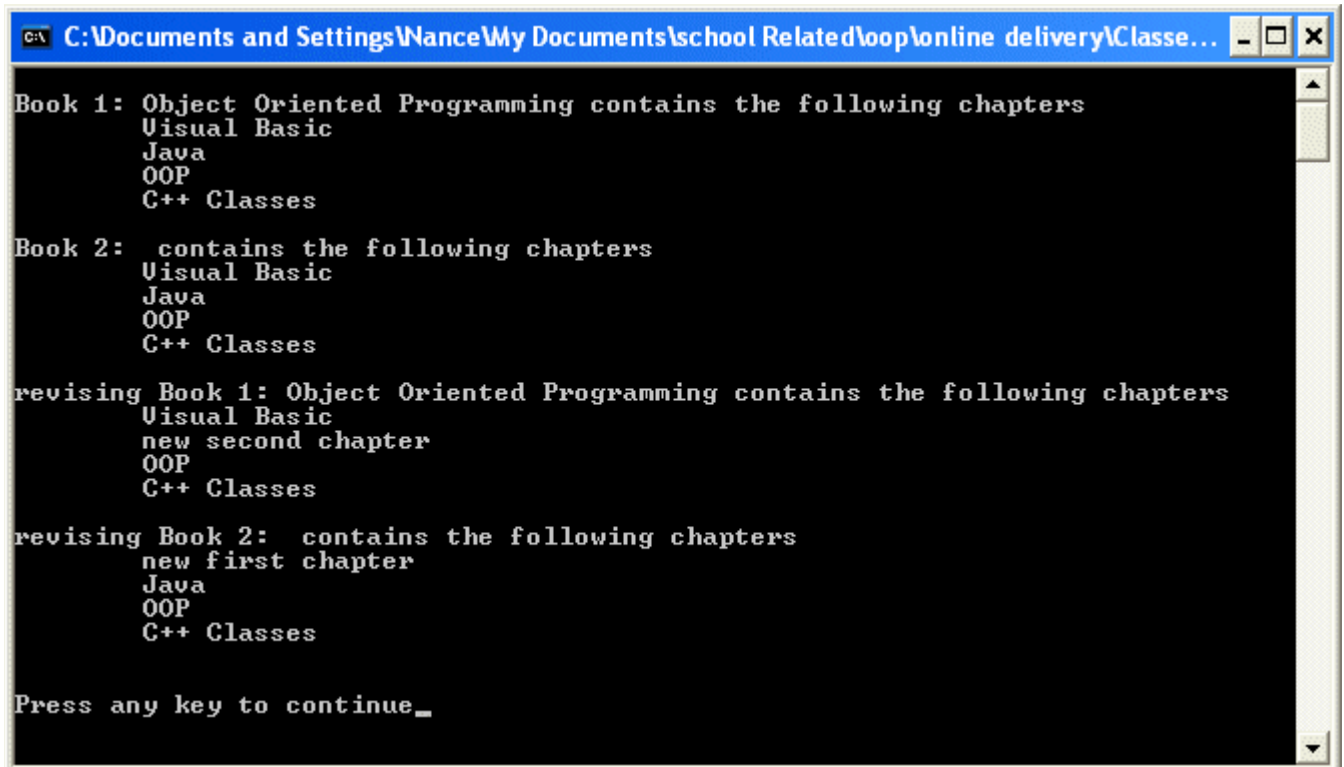
The driver application also needs to be modified to use the copy constructor instead of the memberwise assignment operator. The following line ...

```
Book book2(book1);
```

replaces the line...

```
Book book2 = book1;
```

The output of the program now shows that the entire object is duplicated and that each of the two books is independent of the other...



```
C:\Documents and Settings\Wance\My Documents\school Related\oop\online delivery\Classe...
Book 1: Object Oriented Programming contains the following chapters
    Visual Basic
    Java
    OOP
    C++ Classes
Book 2:  contains the following chapters
    Visual Basic
    Java
    OOP
    C++ Classes
revising Book 1: Object Oriented Programming contains the following chapters
    Visual Basic
    new second chapter
    OOP
    C++ Classes
revising Book 2:  contains the following chapters
    new first chapter
    Java
    OOP
    C++ Classes
Press any key to continue_
```

Operator Overloading

C++ allows you to redefine how the standard operators work when used with class objects. This is particularly useful when your objects contain non-primitive data types, but you want to do typical types of operations such as addition, input and output. This process of redefining standard operators is called **operator overloading**. Put simply, the operator is programmed to function differently depending on the data it receives.

The process of overloading operators is straightforward...you create, as part of your class definition, an **operator function** that will get executed anytime you use the specified operator with an object of that class. This operator function must return the same data type that the normal operator would ... normally the same type of data that is being operated on. For example the assignment operator returns the same type of data that is being assigned. If you overload this operator within one of your classes, your operator function must return an object of this class.

Defining Operator Functions

Like any other member function, an operator function must be both prototyped and defined. The format for the prototype is...

```
className operator operatorSymbol (const className &);
```

where *className* is the name of the class, and *operatorSymbol* is the operator you are overloading. The **const** keyword indicates that the input value will not change in the function. For example, to overload the assignment operator (=) within the **Book** class, the prototype is...

```
Book operator = (const Book &);
```

The operator function itself has the structure...

```
className operator operatorSymbol (const className &originalObject)
{
    //code to be executed when operatorSymbol is used
    //    with an object of this class

    return *this;
}
```

Notice that the parameter passed to the operator function is passed by reference. Remember that when something is passed by reference, you don't need the dereferencing operator (*) to use it in the function, but any changes you make are reflected back to the calling function.

The final line of code in the operator function is very important. Recall that the ***this pointer** is passed explicitly to a function and points to the object itself. In this case, the *this pointer is used to return the contents of the original object.

The operator function for the assignment operator in the **Book** class might look something like...

```
Book Book::Book operator = (const Book &original)
{
    chapters = 0;
    copyList(original);

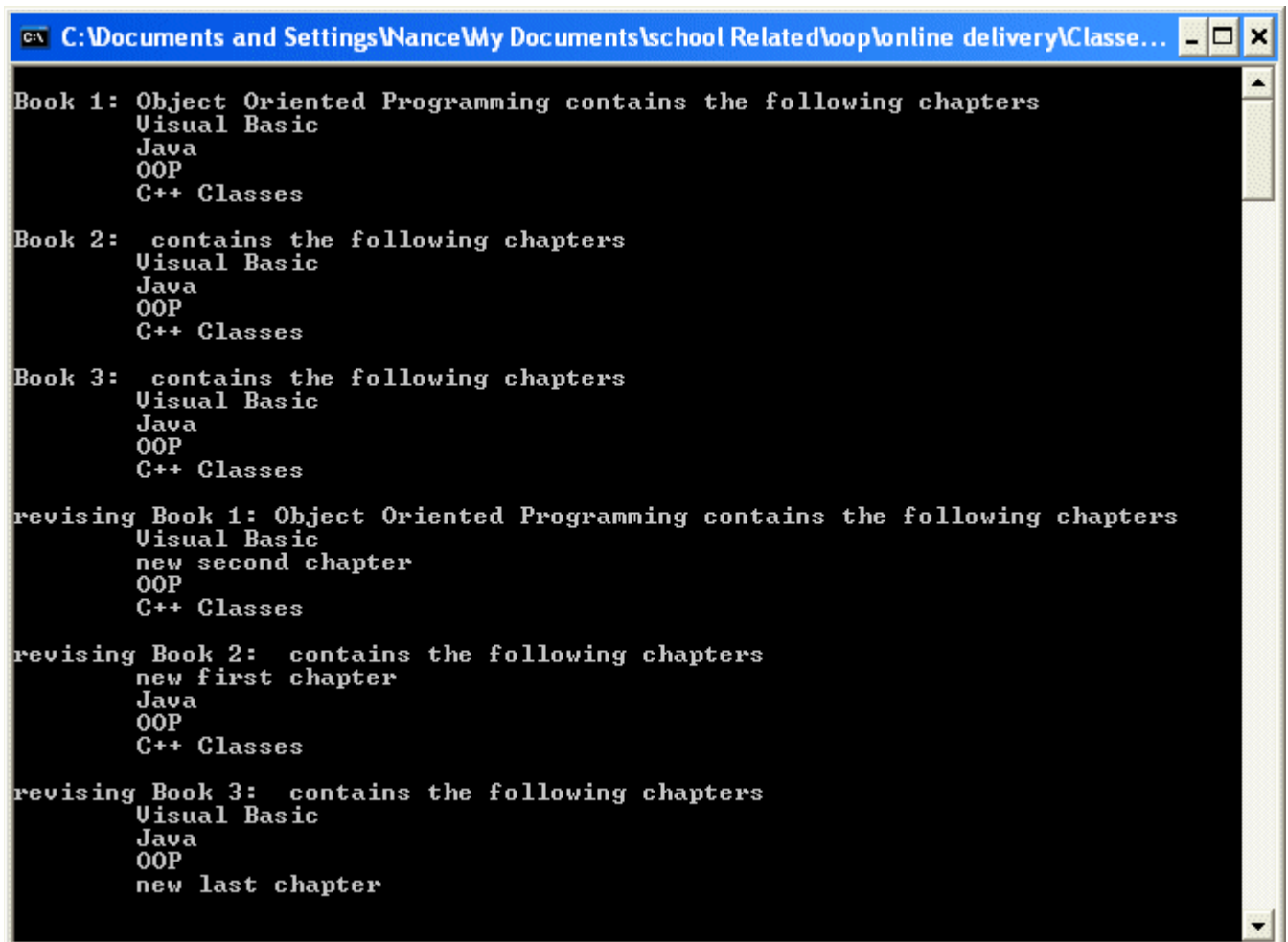
    return *this;
}
```

The first **Book** refers to the return data type. The rest of the function declaration line has a similar structure to all other member functions. You may recognize the code in the function (other than the last line) as the code included in the **Book** class copy constructor. Since the assignment operator needs to perform the same tasks as the copy constructor (ensuring that each element in the original object is copied to the corresponding element in the receiving object) the bulk of the code must be the same.

The assignment operator can now be used as it would with a primitive data type...

```
Book book3 = book1;
book3.printData();
```

Adding the operator function to the **Book** class definition and the above code to the driver class gives...



```
C:\Documents and Settings\Wance\My Documents\school Related\oop\online delivery\Classe...
Book 1: Object Oriented Programming contains the following chapters
Visual Basic
Java
OOP
C++ Classes

Book 2:  contains the following chapters
Visual Basic
Java
OOP
C++ Classes

Book 3:  contains the following chapters
Visual Basic
Java
OOP
C++ Classes

revising Book 1: Object Oriented Programming contains the following chapters
Visual Basic
new second chapter
OOP
C++ Classes

revising Book 2:  contains the following chapters
new first chapter
Java
OOP
C++ Classes

revising Book 3:  contains the following chapters
Visual Basic
Java
OOP
new last chapter
```

Overloading Operators for Input and Output

It is very common to overload the stream insertion (<<) and the stream extraction (>>) operators to make input and output of objects simpler. Because they involve the input and output streams they must however be treated a little bit differently than the other operators ... instead of being declared as member functions they are declared as friend functions. The prototypes, declared in the public section, look like...

```
friend ostream & operator << (ostream &strm, className &);  
friend istream & operator >> (istream &strm, className &);
```

In order to function as closely to the original stream operators, each operator function must return the address of the current stream. As parameters, these functions are passed a reference to the current stream and a reference to the object. For example, adding these overloaded operator prototypes to the **Book** class gives...

```
friend ostream & operator << (ostream &strm, Book &);  
friend istream & operator >> (istream &strm, Book &);
```

The friend functions can be defined in the same file as the member functions. Each has a similar format...

```
ostream & operator << (ostream &strm, className &object)  
{  
    //output code goes here  
    return strm;  
}  
  
istream & operator >> (istream &strm, className &object)  
{  
    //input code goes here  
    return strm;  
}
```

Notice in both cases that the operator function must return the stream. This allows the operator to be used exactly the same way a normal << or >> is used. The function to output all **Book** information might look like...

```
ostream & operator<< (ostream &strm, Book &obj)  
{  
    strm << obj.title << " contains the following chapters\n";  
  
    for (int i = 0; i < obj.nbrChapters; i++)  
        strm << "\t" << obj.chapters[i] << "\n";  
  
    return strm;  
}
```

Each attribute within the **Book** class is accessed using the *obj* object. References to *cout* are replaced with the stream variable *strm*. Other than that, this function is pretty much the same as *printData()*.

The function to allow input of all **Book** information might look like...

```
istream & operator >> (istream &strm, Book &obj)
{
    delete[] obj.chapters;

    cout << "\n\nHow many chapters?";
    strm >> obj.nbrChapters;
    obj.chapters = new string[obj.nbrChapters];

    for (int i = 0; i < obj.nbrChapters; i++)
    {
        cout << "\tEnter chapter #" << i+1 << ":";
        strm >> obj.chapters[i];
    }

    return strm;
}
```

This is very similar to the constructor code other than all references to *cin* that are replaced with *strm*. Notice also that the **Book** attributes are access using the *obj* object.

Some final words about operator overloading...

5. Only use operator overloading when you want to program an operator to work specifically with a class. Make sure that the code you write performs the same type of function that the original operator does. Because you are the one writing the code for the operator function, you can completely change the functionality of an operator. This is not recommended!
6. Make sure that the number of operands for your operator function is identical to the original operator. For example, since the assignment operator is a binary operator which requires one operand on either side of the operator, any assignment operator function you write must also be binary.
7. The **?:** **.** **.*** **::** and **sizeof** operators cannot be overloaded.
8. The most commonly overloaded operators are the assignment(=), math(+, -, *, /), relational(<, >, <=, >=, !=), stream insertion(<<) and stream extraction(>>) operators.

Object Oriented C++

Recall that the full power of object oriented programming is summarized by four concepts...

- **ABSTRACTION** – seeing things from a broad, more general perspective
- **ENCAPSULATION** – hiding data and controlling access to it
- **INHERITANCE** – defining objects based on other more generic objects
- **POLYMORPHISM** – defining the same attribute or behaviour in different classes

This section looks at applying these basic concepts in C++.

this Pointer

The **this pointer** is often used in object oriented C++. You'll recall from the last section that it is automatically passed to all instance functions when they are called and points to the instance of the class making the function call.

The **this pointer** serves a number of purposes. First it provides access to member variables...

```
void someFunction(int x)
{
    this->x = x;
}
```

In this case, as in Java, it allows the instance variable x to be assigned the value passed in from the parameter list. Notice that it uses pointer notation to access the instance variable.

The **this pointer** also allows objects to be compared...

```
boolean someFunction(someClass &obj)
{
    boolean same = false;

    if (this == &obj)
        same = true;

    return same;
}
```

This simple function determines if the address of the calling object and the one passed in through the parameter list are the same. If they are the same true is returned, otherwise false is returned.

Look, for example, at the following class definition...

```
#include <iostream>
using namespace std;

class Sample
{
private:
    long id;

public:
    Sample();
    Sample(const long &id);
    bool compareSamples(const Sample &item) const;
};

Sample::Sample()
{
    id = 0;
    cout << "\n\nCreating object with id of " << id;
}

Sample::Sample(const long &id)
{
    this->id = id;
    cout << "\n\nCreating object with id of " << id;
}

bool Sample::compareSamples(const Sample &item) const
{
    bool state = false;
    if (this == &item)
        state = true;
    return state;
}
```

The default constructor creates a **Sample** object with an *id* of 0. The alternate constructor creates a **Sample** object with the specified *id*. The *compareSamples()* function determines if the current object is the same object as the one passed to it.

Now take a look at a driver function...

```
#include <iostream>
using namespace std;

#include "sample.h"

void main()
{
    Sample s1(12345);
    bool compare = s1.compareSamples(s1);
    cout << "\n\nSamples s1 and s1 are the same? ... " << boolalpha << compare;
```

```

Sample s2(23456);
compare = s1.compareSamples(s2);
cout << "\n\nSamples s1 and s2 are the same? ... " << boolalpha << compare;
}

```

The first block creates a **Sample** object with an *id* of 12345. This object, *s1*, is then compared to itself. A second **Sample** object is created with an *id* of 23456. This object, *s2*, is compared to object *s1*. The output of this program is...

Basic Inheritance

As in Java, a child or derived class inherits all of the variables and functions of the parent or base class. New data and functions can be added to the derived class to add any required specialization.

The process of inheriting from a parent class is pretty much the same as in Java. The main difference is that in C++ you must identify the **base class access specification**. This specification determines how private, protected and public parent class members may be accessed by the child class. The following table defines the access...

Base Class Access Specification	base class members defined ...		
	Public	Protected	Private
public	public	protected	inaccessible
protected	protected	protected	inaccessible
private	private	private	inaccessible

For example, if the base class access specification is set to *public* then all the parent class members declared to be public are considered public in the child class. All protected variables are treated as protected, but none of the private variables are accessible. On the other hand if the base class access specification is set to *protected* then all parent class members defined as either public or protected are

considered protected variables. None of the private variables are accessible. If the base class access specification is set to *private* then all variables declared to be public or protected are treated as private variables of the child class. None of the variables declared private in the parent class are accessible.

The format for declaring that class **Child** inherits from class **Parent** is...

```
class Child : baseClassAccessSpecification Parent
{
    //class definition
}
```

For example if class **Square** inherits from class **Quadrilateral** with a *public* base class access specification the class definition is...

```
class Square : public Quadrilateral
{
    //class definition
}
```

Constructors and Destructors

Parent class constructors are called before the constructor of a child class. Destructors are called in reverse order ... the child class destructor is called, followed by the parent class destructor. Make sure that you keep this in mind when writing the code for your constructors and destructors.

Overriding Base Class Functions

All public and protected functions defined in the parent class are inherited. There are times however when you want a function in a child class to operate differently from that defined in the parent class. This requires you to override the parent class function. To do this you must both re-prototype and re-define the function in the child class...

```
class Parent
{
public:
    void someFunction();
}

void someFunction()
{
    //parent class code goes here;
}

class Child : public Parent
{
    void someFunction();
}

void someFunction()
{
    //child class code goes here;
}
```

Even if a function is overridden in a child class, the parent class function can still be accessed. Assuming the parent class is called **Parent** and the child class is called **Child**, the highlighted instruction contained in the overridden function *someFunction()* in the **Child** class accesses the original *someFunction()* function defined in the **Parent** class.

```
void someFunction()
{
    Parent::someFunction();

    //rest of code for the Child function goes here
}
```

Notice that in order to call the original function the parent class name must precede the function name. They are separated by the scope resolution operator.

Take a look at the following **Parent** class definition found in a **Parent.h** file...

```
#include <iostream>
using namespace std;

class Parent
{
public:
    enum state {PENDING, INPROCESS, COMPLETE};

protected:
    state status;

public:
    Parent();
    Parent(state s);
    ~Parent();

    void printStatus ();
};

//constructors
Parent::Parent()
{
    status = PENDING;
    cout << " ... in Parent class default constructor";
    printStatus();
}

Parent::Parent(state s)
{
    status = s;
    cout << " ... in Parent class alternate constructor";
    printStatus();
}
```

```

//destructor
Parent::~~Parent()
{
    cout<< "... in Parent destructor";
    printStatus();
}

//other functions
void Parent::printStatus()
{
    cout <<"\n\tStatus: ";

    switch(status)
    {
    case PENDING:
        cout <<"pending";
        break;
    case INPROCESS:
        cout <<"in process";
        break;
    case COMPLETE:
        cout <<"complete";
    }
}

```

First notice that the *status* variable is declared to be a protected variable. This ensures that it can be inherited by any child classes. The **Parent** class itself has two constructors ... one that sets the *status* to **PENDING**, displays that it is in the default constructor and then calls the *printStatus()* function, and another that accepts a *status*, displays that it is in the alternate constructor and then calls the same *printStatus()* function. It also contains a destructor that identifies that it is in the destructor function and calls the *printStatus()* function.

The class also contains the *printStatus()* function which prints out the current status of the class.

The **Child** class is similar...

```

#include "Parent.h"

class Child : public Parent
{
private:
    int colour;

public:
    Child();
    Child(state s, int clr);
    ~Child();

    void printStatus();
};

```

```

//constructors
Child::Child()
{
    colour = 20;
    cout << "\n ... in Child default class constructor";
    printStatus();
}

Child::Child(state s, int clr)
{
    status = s;
    colour = clr;
    cout << "\n ... in Child alternate class constructor";
    printStatus();
}

//destructor
Child::~~Child()
{
    cout<< "\n... in Child destructor";
    printStatus();
}

//other functions
void Child::printStatus()
{
    Parent::printStatus();
    cout << " and colour is " << colour;
}

```

First it is important to notice that this inheriting class includes the **Parent** class header file. This is necessary because of the references made to it. The instance variables are declared to be private. This is fine since this class will not act as a parent class to any other class. **Parent** contains two constructors, one that assigns a default *status*, and the other that allows specification of the *status*. Its destructor works the same as the **Parent** class destructor. The *printStatus()* function is overridden in this **Child** class. It does however call the **Parent's** original function in its first instruction.

A very simple driver function may look something like...

```

#include <iostream>
using namespace std;

#include "Child.h"

void main()
{
    cout << "Creating a Parent class object called data1 (default status)";
    Parent data1;
    cout << "\nCalling printStatus for data1...";
    data1.printStatus();

    cout << "\n\nCreating a Parent class object called data2 (status on)";
    Parent data2(Parent::COMPLETE);
    cout << "\nCalling printStatus for data2...";
    data2.printStatus();
}

```

```

        cout << "\n\nCreating a Child class object called data3 ";
        cout << "(default status and colour)";
        Child data3;
        cout << "\nCalling printStatus for data3...";
        data3.printStatus();

        cout << "\n\nCreating a Child class object called data4 ";
        cout << " (in process, colour 10)";
        Child data4(Parent::INPROCESS, 10);
        cout << "\nCalling printStatus for data4...";
        data4.printStatus();

        char ch;
        cout << "\n\nPress any key to continue";
        cin >> ch;
    }
}

```

The first block of code creates a **Parent** class object using the default constructor. Its *status* is displayed as part of the constructor class and again in the direct call to the *printStatus()* function. The second block of code creates a **Parent** class object using the alternate constructor. Its *status* is displayed as part of the constructor class and again in the direct call to the *printStatus()* function.

The third block of code creates a **Child** class object using the default constructor. Its *status* is displayed as part of each constructor class and again in the direct call to the *printStatus()* function. Finally, the fourth block of code creates a **Child** class object using the alternate constructor. Its *status* is displayed as part of each constructor class and again in the direct call to the *printStatus()* function.

The output of this program looks like...

```

C:\DOCUMENTS AND SETTINGS\WANCEMY DOCUMENTS\SCHOOL RELATED\OOP\ONLINE DELIV...
Creating a Parent class object called data1 (default status)
... in Parent class default constructor
Status: pending
Calling printStatus for data1...
Status: pending

Creating a Parent class object called data2 (status on)
... in Parent class alternate constructor
Status: complete
Calling printStatus for data2...
Status: complete

Creating a Child class object called data3 (default status and colour)
... in Parent class default constructor
Status: pending
... in Child default class constructor
Status: pending and colour is 20
Calling printStatus for data3...
Status: pending and colour is 20

Creating a Child class object called data4 (in process, colour 10)
... in Parent class default constructor
Status: pending
... in Child alternate class constructor
Status: in process and colour is 10
Calling printStatus for data4...
Status: in process and colour is 10

Press any key to continue

```

After any key is pressed at the end of the program, the following is displayed on the screen...

```
Press any key to continue
... in Child destructor
    Status: in process and colour is 10
... in Parent destructor
    Status: in process
... in Child destructor
    Status: pending and colour is 20
... in Parent destructor
    Status: pending
... in Parent destructor
    Status: complete
... in Parent destructor
    Status: pending
```

Notice that the objects are “destroyed” in the reverse order that they were instantiated. You can see that each **Child** class object executes its **Child** class destructor followed by the **Parent** class destructor. This is followed by execution of the two **Parent** class destructors.

Note: To see this display you have to run your program from the command prompt and redirect the output to a file.

Standard Template Library (STL)

A template is a specification for a function or a class that can perform the same operation on almost any data type. C++ provides a library of templates called the Standard Template Library or STL. This library contains many generic templates for implementing abstract data types and algorithms.

There are two basic data structures in the Standard Template Library ... containers and iterators. These structures are used with a set of predefined algorithms.

Containers

A container is a class that stores and organizes data. It grows and shrinks automatically as data is added and removed. There are two types of container classes in the STL...sequential and associative containers.

Sequential Containers

As the name implies, sequential containers organize data sequentially, much like a standard array does. There are three sequence containers ... **vectors**, **deque**s and **lists**. **Vectors** are expandable arrays with access to the elements generally made at the end or in the middle. **Deque**s are like vectors but access to the elements is normally made at the beginning. A **list** can be accessed at any position.

There are a number of functions available for use with sequential containers...

<code>int size()</code>	Returns the number of elements in the container
<code>iterator begin()</code>	Returns an iterator pointing to the first element
<code>iterator end()</code>	Returns an iterator pointing to one past the last element
<code>iterator rbegin()</code>	Returns a reverse iterator pointing to the last element
<code>iterator rend()</code>	Returns a reverse iterator pointing to one before the first element
<code>void insert(iterator, element)</code>	Inserts <i>element</i> before location of <i>iterator</i>
<code>void clear()</code>	Eliminates all elements in the container
<code>void erase(iterator)</code>	Eliminates element at location of <i>iterator</i>
<code>void push_back(value)</code>	Inserts the specified value after the last element
<code>void push_front(value)</code>	Inserts the specified value before the first element
<code>void pop_back()</code>	Pops the first element out of the container
<code>void pop_front()</code>	Pops the last element out of the container

Two containers can also be compared using the equality (==) and non-equality (!=) operators. The equality operator returns true if the size of the two containers is the same and each element in the first container is equal to the corresponding element in the second container. The non-equality operator returns the inverse of the equality operator.

Associative Containers

Associative containers are like simple databases that use keys to allow quick access to elements. There are four associative containers... **sets**, **multisets**, **maps** and **multimaps**. A **set** stores a set of keys. No duplicate keys are allowed. A **multiset** also stores a set of keys, but does allow duplicates. A **map** relates a set of keys to corresponding data elements. One key is provided for each element. No duplicates are permitted. Finally a **multimap** maps a set of keys to data elements. In this case multiple keys are allowed per element and duplicates are permitted.

There are a few functions available for use with all the associative containers...

<code>int size()</code>	Returns the number of elements in the container
<code>void insert(iterator, element)</code>	Inserts <i>element</i> before location of <i>iterator</i>
<code>bool empty()</code>	returns true if the container is empty; false otherwise

Two containers can also be compared using the equality (==) operator. If both containers contain the same elements it returns true. Otherwise the operation returns false.

Iterators

An iterator is like a pointer. It is used specifically to access information stored in a container. There are five iterators... **forward**, **bidirectional**, **random-access**, **input** and **output**. The **forward** iterator can only move forward through a container. It uses the ++ operator to do this. The **bidirectional** iterator uses the ++ and – operators to move in either direction through a container. As the name implies, the **random-access** iterator can move forward or backward through the elements in a container. It can also jump to a specific data element anywhere in the container. The **input** and **output** iterators are used specifically with cin and cout to read from or write to devices or files.

There are a number of functions designed to work with iterators...

<code>int count(start, stop, &target)</code>	Returns the number of elements in the range equal to the specified target
<code>iterator find(start, stop, &target)</code>	Returns <i>iterator</i> located at first occurrence of target in specified range
<code>bool equal(start1, stop1, start2)</code>	Returns true if specified range in the first container has the same elements in the same order as the second container

<i>iterator</i> search(<i>start1</i> , <i>stop1</i> , <i>start2</i> , <i>stop2</i>)	Returns <i>iterator</i> located at element in first specified range that contains the second specified range. Returns <i>stop1</i> if the second specified range is not a subset of the first.
<i>bool</i> binary_search(<i>start</i> , <i>stop</i> , & <i>target</i>)	Assumes specified range is sorted into ascending order. Returns true is specified target is contained in the specified range. Returns false otherwise.

Algorithms

The algorithms contained in the Standard Template Library perform various operations on the elements of containers. They work for all types of containers. A partial list of available algorithms includes...

<i>iterator</i> copy(<i>start1</i> , <i>stop1</i> , <i>start2</i>)	Copies specified range of elements from one container to another container. The second container must be the same size as the specified range in the first container.
<i>iterator</i> max_element (<i>start</i> , <i>stop</i>)	Returns an iterator to the largest object
<i>void</i> merge(<i>start1</i> , <i>stop1</i> , <i>start2</i> , <i>stop2</i> , <i>result</i>)	Assumes each of the two specified ranges is already in ascending order. Merges and sorts the specified elements from two containers into one <i>result</i> container.
<i>iterator</i> min_range(<i>start</i> , <i>stop</i>)	Returns an iterator to the smallest object
<i>void</i> random_shuffle(<i>start</i> , <i>stop</i>)	Randomly shuffles the elements in a specified range
<i>iterator</i> remove(<i>start</i> , <i>stop</i> , & <i>target</i>)	Moves elements equal to a specified <i>target</i> from the specified range of elements to the end of the range. Size of the container is not changed.
<i>void</i> reverse(<i>start</i> , <i>stop</i>)	Reverses the order of the elements in a specified range
<i>void</i> swap(& <i>target1</i> , & <i>target2</i>)	Interchange values of specified elements
<i>void</i> sort(<i>start</i> , <i>stop</i>)	Sorts objects in a specified range

Operations for Sorted Containers

There is also a set of operations that works with any container whose elements are sorted into ascending order. These operations include...

<code>bool includes(<i>start1</i>, <i>stop1</i>, <i>start2</i>, <i>stop2</i>)</code>	Returns true if every element in the second specified range also occurs in the first specified range. Returns false otherwise.
<code>void set_union(<i>start1</i>, <i>stop1</i>, <i>start2</i>, <i>stop2</i>, <i>result</i>)</code>	Sorts all the elements of the two specified ranges into a result container.
<code>void set_intersection(<i>start1</i>, <i>stop1</i>, <i>start2</i>, <i>stop2</i>, <i>result</i>)</code>	Sorts all the common elements from the two specified ranges into a result container.
<code>void set_difference(<i>start1</i>, <i>stop1</i>, <i>start2</i>, <i>stop2</i>, <i>result</i>)</code>	Copies all the elements from the first specified range that are not in the second specified range into a result container.

Vector Class Template

A vector is a sequence container that acts like an expandable array. Although elements can be accessed at any location in the vector, insertions and deletions are most efficient if done at the end. This section looks at declaring and using vectors.

Declaring Vectors

Like an array, a vector must specify which single type of data it will contain. The format is...

```
vector<dataType> vectorName;
```

For example to declare a vector called *numbers* that will hold integer values the declaration is...

```
vector<int> numbers;
```

Your code must include the vector library for any vector access...

```
#include <vector>
```

Vector Functions

Because vectors work most efficiently when elements are added to the end, the *push_back()* and *pop_back()* functions are often used to add and remove elements. You can use the *insert()* function to insert elements into a specific position within the vector. This operation however is not as efficient.

You can also use the array format to access individual elements in the vector. For example, to print out the first element of the *numbers* vector, you can use...

```
cout << numbers[0];
```

Let's start with a simple program that creates a vector of integers called *Vect*...

```
#include <iostream>
#include <vector>          //for all vector access
using namespace std;

void displayVector(vector<int> v);

void main()
{
    int x;

    //declare vector object ... vector holds integers
    vector<int> Vect;

    //display number of elements in vector
    cout << "Vect starts with " << Vect.size() << " elements." << endl;

    //add ten elements to vector
    for (x = 0; x < 10; x++)
        Vect.push_back(x);

    //display number of elements in vector
    cout << "Vect now has " << Vect.size() << " elements." << endl;

    //display value of each element in vector using vector methods
    cout << "\tElements are: " ;
    displayVector(Vect);

    //remove elements from vector
    cout << "\nNow emptying the vector ..." << endl;
    int size = Vect.size();

    for (x = 0; x < size; x++)
    {
        Vect.pop_back();
        cout << "\tafter pop_back #" << x << ": ";
        displayVector(Vect);
    }

    //display number of elements in vector
    cout << "Vect ends with " << Vect.size() << " elements." << endl;
}

//=====
void displayVector(vector<int> v)
{
    for (int x = 0; x < v.size(); x++)
        cout << v[x] << " ";
    cout << endl;
}
```

First the vector library is included. This is necessary anytime you are using vectors in your program. Looking at the prototype for the *displayVector()* function you can see that it is passed a vector. The “data type specifier” for a vector is always *vector<dataType>*.

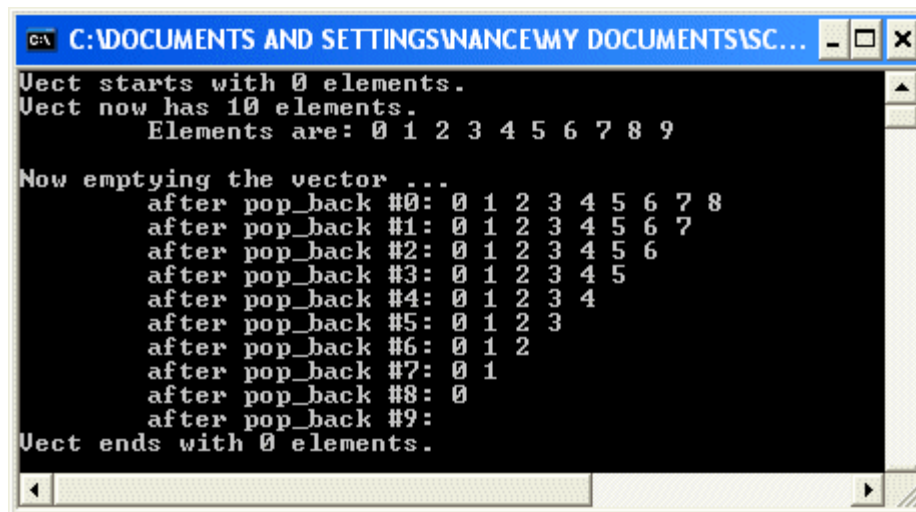
The vector itself is declared at the top of the *main()* function. The *size()* function is then called to display the current number of elements in the vector. This function call is associated with an object just like any other instance function. In this case the object is the vector *Vect*.

A for loop populates the vector with ten integer elements ... the values from zero to nine. The *push_back()* function is used to add each element to the end of the vector. The *size()* function is called again to display the current number of elements in the vector.

The *displayVector()* function is called to display the individual elements of the vector. The function is passed the vector and then array notation is used to print out the value of each element.

Finally, the vector is emptied using the *pop_back()* function. As with any dynamic data structure in C, it is important that you free up the memory once it is no longer needed.

The output of this program shows both the addition and removal of the ten elements...



```
C:\DOCUMENTS AND SETTINGS\NANCE\MY DOCUMENTS\SC...
Vect starts with 0 elements.
Vect now has 10 elements.
Elements are: 0 1 2 3 4 5 6 7 8 9

Now emptying the vector ...
after pop_back #0: 0 1 2 3 4 5 6 7 8
after pop_back #1: 0 1 2 3 4 5 6 7
after pop_back #2: 0 1 2 3 4 5 6
after pop_back #3: 0 1 2 3 4 5
after pop_back #4: 0 1 2 3 4
after pop_back #5: 0 1 2 3
after pop_back #6: 0 1 2
after pop_back #7: 0 1
after pop_back #8: 0
after pop_back #9: 
Vect ends with 0 elements.
```

Just like pointers, iterators can also be used access the elements in the vector. To use an iterator you must first declare it...

```
vector<dataType>::iterator iteratorName;
```

For the vector in the previous program, the iterator declaration might look like...

```
vector<int>::iterator Iter;
```

where *Iter* is the name of the iterator that will be used to access element in the integer vector.

Once the iterator is declared it can be used like any other pointer-type variable. For example, to display the element of the vector from last to first you could use the following for loop...

```
cout << "\n\tElements last to first are...";

for (Iter = Vect.end()-1; Iter >= Vect.begin(); Iter --)
    cout << *Iter << " ";

cout << endl;
```

The *begin()* and *end()* functions return an iterator to the vector that allows each element to be accessed in turn.

Iterators can be passed to functions the same as any other variable. For example, the *displayVector()* function can be overloaded to work with iterators. Its prototype would be...

```
void displayVector(vector<int> v, vector<int>::iterator i, int direction);
```

The first argument is the vector itself. The second argument is an iterator for the vector. The last argument specifies whether the elements are to be displayed in forward or reverse order.

Assuming an enumerated data type has been defined as...

```
enum {FORWARD, BACKWARD};
```

the second *displayVector()* function might look something like...

```
void displayVector(vector<int> v, vector<int>::iterator i, int direction)
{
    if (direction == FORWARD)
        for (i = v.begin(); i < v.end(); i++)
            cout << *i << " ";
    else
        for (i = v.end()-1; i >= v.begin(); i--)
            cout << *i << " ";

    cout << endl;
}
```

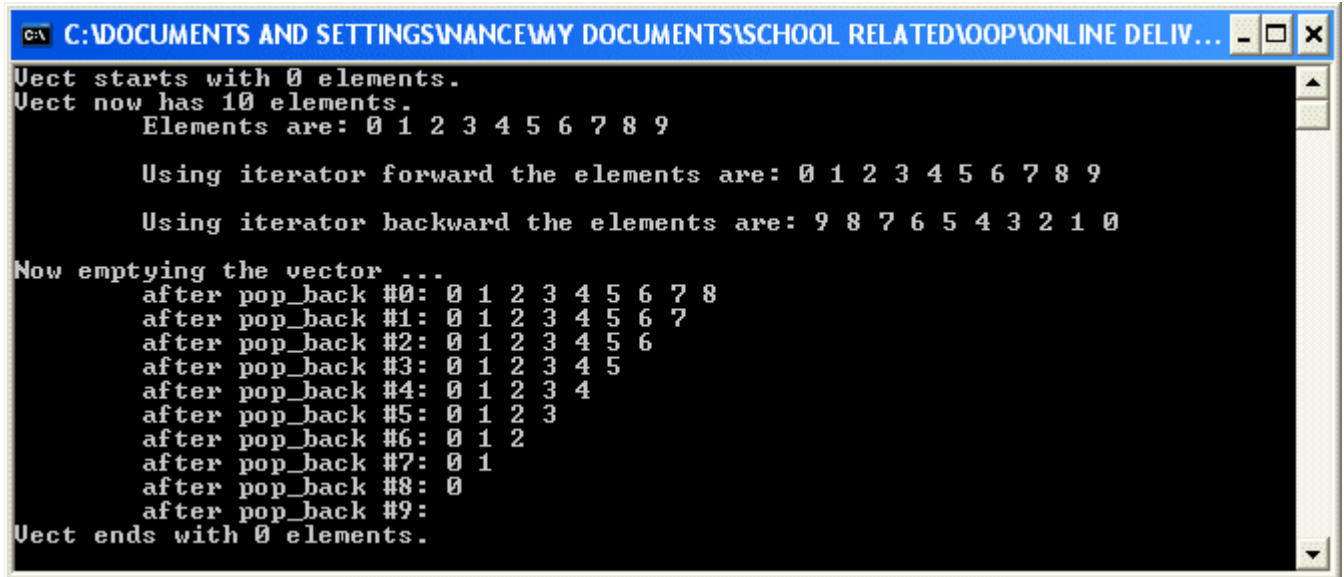
As before, the dereferencing operator (*) is used with the iterator to access the vector elements.

The code to execute this function to display in the forward and then the reverse direction is...

```
//display value of each element in vector using iterator ... forward
cout << "\n\tUsing iterator forward the elements are: " ;
displayVector(Vect, Iter, FORWARD);

//display value of each element in vector using iterator ... backward
cout << "\n\tUsing iterator backward the elements are: " ;
displayVector(Vect, Iter, BACKWARD);
```

The output of the program is now...



```
C:\DOCUMENTS AND SETTINGS\NANCE\MY DOCUMENTS\SCHOOL RELATED\VOOP\ONLINE DELIV...
Vect starts with 0 elements.
Vect now has 10 elements.
  Elements are: 0 1 2 3 4 5 6 7 8 9

  Using iterator forward the elements are: 0 1 2 3 4 5 6 7 8 9

  Using iterator backward the elements are: 9 8 7 6 5 4 3 2 1 0

Now emptying the vector ...
  after pop_back #0: 0 1 2 3 4 5 6 7 8
  after pop_back #1: 0 1 2 3 4 5 6 7
  after pop_back #2: 0 1 2 3 4 5 6
  after pop_back #3: 0 1 2 3 4 5
  after pop_back #4: 0 1 2 3 4
  after pop_back #5: 0 1 2 3
  after pop_back #6: 0 1 2
  after pop_back #7: 0 1
  after pop_back #8: 0
  after pop_back #9:
Vect ends with 0 elements.
```

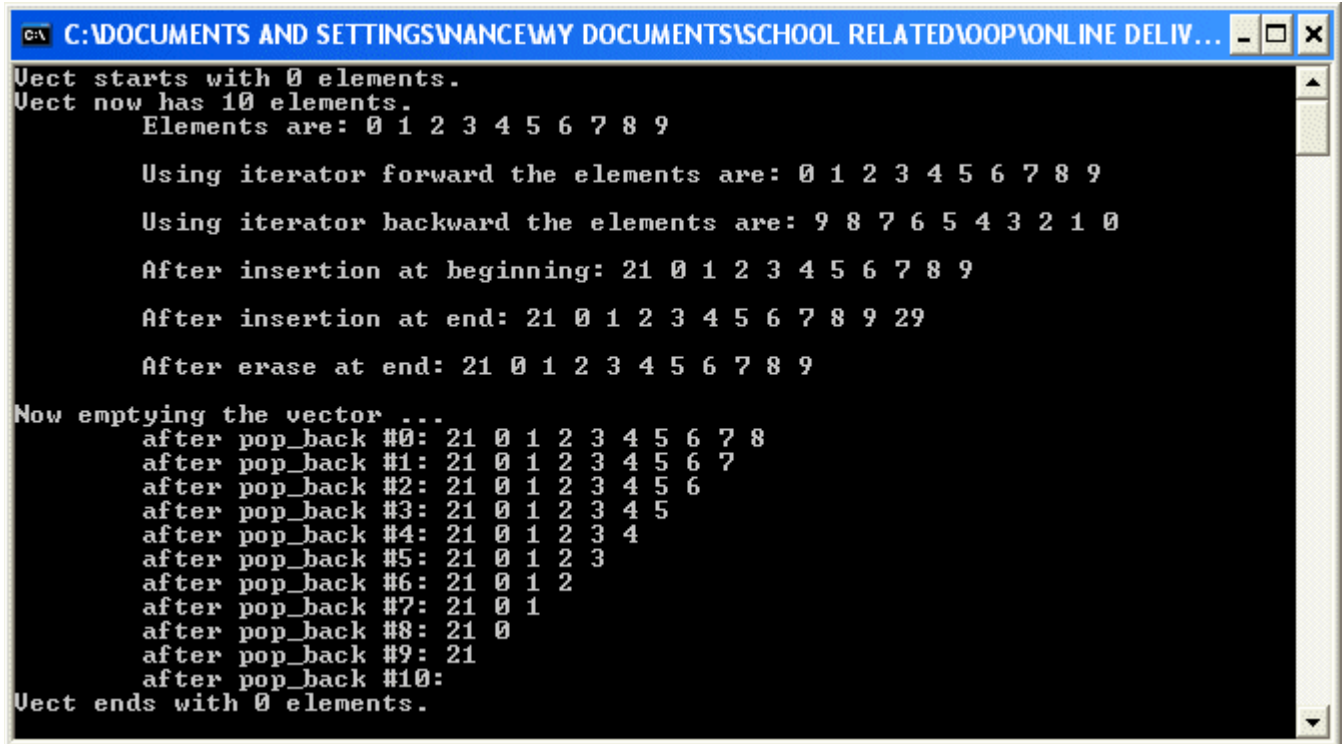
The `insert()` and `erase()` functions are used to add and remove vector elements. The following code inserts the value 21 at the beginning of the vector and the value 29 at the end. It then removes the last element...

```
//insert element at beginning
Vect.insert(Vect.begin(), 21);
cout << "\n\tAfter insertion at beginning: " ;
displayVector(Vect);

//insert element at end
Vect.insert(Vect.end(), 29);
cout << "\n\tAfter insertion at end: " ;
displayVector(Vect);

//erase element at end
Vect.erase(Vect.end());
cout << "\n\tAfter erase at end: " ;
displayVector(Vect);
```


Adding this code before popping back the elements gives an output of...



A screenshot of a Windows command prompt window. The title bar reads "C:\ \ C:\DOCUMENTS AND SETTINGS\NANCEWY\DOCUMENTS\SCHOOL RELATED\VOOP\ONLINE DELIV...". The window has a black background with white text. The text shows the output of a C++ program using vectors. It starts with "Vect starts with 0 elements." followed by "Vect now has 10 elements." and a list of elements from 0 to 9. Then it shows iterations forward and backward. Next, it shows insertion at the beginning (adding 21) and at the end (adding 29). Then it shows erasing the last element. Finally, it shows a loop of 11 "pop_back" operations, each followed by the current state of the vector, until it is empty. The window has standard Windows window controls (minimize, maximize, close) in the top right corner.

```
C:\ \ C:\DOCUMENTS AND SETTINGS\NANCEWY\DOCUMENTS\SCHOOL RELATED\VOOP\ONLINE DELIV...
Vect starts with 0 elements.
Vect now has 10 elements.
  Elements are: 0 1 2 3 4 5 6 7 8 9

  Using iterator forward the elements are: 0 1 2 3 4 5 6 7 8 9
  Using iterator backward the elements are: 9 8 7 6 5 4 3 2 1 0

  After insertion at beginning: 21 0 1 2 3 4 5 6 7 8 9
  After insertion at end: 21 0 1 2 3 4 5 6 7 8 9 29
  After erase at end: 21 0 1 2 3 4 5 6 7 8 9

Now emptying the vector ...
after pop_back #0: 21 0 1 2 3 4 5 6 7 8
after pop_back #1: 21 0 1 2 3 4 5 6 7
after pop_back #2: 21 0 1 2 3 4 5 6
after pop_back #3: 21 0 1 2 3 4 5
after pop_back #4: 21 0 1 2 3 4
after pop_back #5: 21 0 1 2 3
after pop_back #6: 21 0 1 2
after pop_back #7: 21 0 1
after pop_back #8: 21 0
after pop_back #9: 21
after pop_back #10:
Vect ends with 0 elements.
```

Vector Algorithms

The Standard Template Library algorithms can be used with vectors. As long as you include the algorithm library...

```
#include <algorithm>
```

you can use these algorithms with your containers.

The following code, for example, can be used to shuffle, search for a specific element and then resort the elements in the vector...

```
//randomly shuffle elements
random_shuffle(Vect.begin(), Vect.end());
cout << "\tShuffled elements are: " ;/
displayVector(Vect);

//search for element
if (binary_search(Vect.begin(), Vect.end(), 5))
    cout << "\nThe value 5 was found in the vector" << endl;
else
    cout << "\nThe value 5 was not found in the vector" << endl;

//sort elements
sort(Vect.begin(), Vect.end());
cout << "\tSorted elements are: " ;
displayVector(Vect);
```

```

//search for element
if (binary_search(Vect.begin(), Vect.end(), 6))
    cout << "\nThe value 6 was found in the vector" << endl;
else
    cout << "\nThe value 6 was not found in the vector" << endl;

```

The program output is...

```

C:\DOCUMENTS AND SETTINGS\WANCEMY DOCUMENTS\SCHOOL RELATED\LOOP\ONLINE DELIV...
Vect starts with 0 elements.
Vect now has 10 elements.
  Elements are: 0 1 2 3 4 5 6 7 8 9

  Using iterator forward the elements are: 0 1 2 3 4 5 6 7 8 9
  Using iterator backward the elements are: 9 8 7 6 5 4 3 2 1 0
  After insertion at beginning: 21 0 1 2 3 4 5 6 7 8 9
  After insertion at end: 21 0 1 2 3 4 5 6 7 8 9 29
  After erase at end: 21 0 1 2 3 4 5 6 7 8 9
  Shuffled elements are: 3 2 21 1 9 6 7 8 4 0 5

The value 6 was not found in the vector
  Sorted elements are: 0 1 2 3 4 5 6 7 8 9 21

The value 6 was found in the vector

Now emptying the vector ...
  after pop_back #0: 0 1 2 3 4 5 6 7 8 9
  after pop_back #1: 0 1 2 3 4 5 6 7 8
  after pop_back #2: 0 1 2 3 4 5 6 7
  after pop_back #3: 0 1 2 3 4 5 6
  after pop_back #4: 0 1 2 3 4 5
  after pop_back #5: 0 1 2 3 4
  after pop_back #6: 0 1 2 3
  after pop_back #7: 0 1 2
  after pop_back #8: 0 1
  after pop_back #9: 0
  after pop_back #10:
Vect ends with 0 elements.

```

Notice that the binary search algorithm did not find the 6 the first time even though it is in the vector. This is because the binary search assumes the elements are stored in increasing order. If you intend to use this search algorithm make sure you sort the elements first. Notice that the sort algorithm sorts the element into increasing order.

Deque Class Template

A deque is a double ended queue. Some people pronounce it “deck” and other “d-queue”. Although elements can be inserted into deques equally efficiently at either the beginning or the end, deques are usually used when elements are to be added at the beginning. Vectors are more efficient if elements are to be added at the end. This section looks at declaring and using deques.

Declaring Deques

Like a vector, a deque must specify which single type of data it will contain. The format is...

```
deque<dataType> dequeName;
```

For example to declare a deque called *numbers* that will hold integer values the declaration is...

```
deque<int> numbers;
```

Your code must include the deque library for any deque access...

```
#include <deque>
```

Deque Functions

The functions available in the deque class include most of the set defined for the containers in general. The *push_front()* and *pop_front()* functions are the preferred functions for element insertion and removal.

A deque program that is comparable to the vector program you looked at in the first section is shown here...

```
#include <iostream>
#include <deque>           //for all deque access
#include <algorithm>       //for random, sort and search
using namespace std;

enum {FORWARD, BACKWARD};

void displaydeque(deque<int> v);
void displaydeque(deque<int> v, deque<int>::iterator i, int direction);

void main()
{
    int x;

    //declare deque object ... deque holds integers
    deque<int> Deck;

    //declare iterator for deque
    deque<int>::iterator Iter;

    //display number of elements in deque
    cout << "Deck starts with " << Deck.size() << " elements." << endl;

    //add number of elements to front of deque
    for (x = 0; x < 10; x++)
        Deck.push_front(x);

    //display number of elements in deque
    cout << "Deck now has " << Deck.size() << " elements." << endl;
```

```

//display value of each element in deque using deque methods
cout << "\tElements are: " ;
displaydeque(Deck);

//display value of each element in deque using iterator ... forward
cout << "\n\tUsing iterator forward the elements are: " ;
displaydeque(Deck, Iter, FORWARD);

//display value of each element in deque using iterator ... backward
cout << "\n\tUsing iterator backward the elements are: " ;
displaydeque(Deck, Iter, BACKWARD);

//insert element at beginning
Deck.insert(Deck.begin(), 21);
cout << "\n\tAfter insertion at beginning: " ;
displaydeque(Deck);

//insert element at end
Deck.insert(Deck.end(), 29);
cout << "\n\tAfter insertion at end: " ;
displaydeque(Deck);

//erase element at end
Deck.erase(Deck.end()-1);
cout << "\n\tAfter erase at end: " ;
displaydeque(Deck);

//randomly shuffle elements
random_shuffle(Deck.begin(), Deck.end());
cout << "\tShuffled elements are: ";
displaydeque(Deck);

//search for element
if (binary_search(Deck.begin(), Deck.end(), 6))
    cout << "\nThe value 6 was found in the deque" << endl;
else
    cout << "\nThe value 6 was not found in the deque" << endl;

//sort elements
sort(Deck.begin(), Deck.end());
cout << "\tSorted elements are: " ;
displaydeque(Deck);

//search for element
if (binary_search(Deck.begin(), Deck.end(), 6))
    cout << "\nThe value 6 was found in the deque" << endl;
else
    cout << "\nThe value 6 was not found in the deque" << endl;

//remove elements from deque
cout << "\nNow emptying the deque ..." << endl;
int size = Deck.size();
for (x = 0; x < size; x++)
{
    Deck.pop_front();
    cout << "\tafter pop_front #" << x << ": ";
    displaydeque(Deck);
}

```

```

        //display number of elements in deque
        cout << "Deck ends with " << Deck.size() << " elements." << endl;
    }

    //=====

void displaydeque(deque<int> v)
{
    for (int x = 0; x < v.size(); x++)
        cout << v[x] << " ";
    cout << endl;
}

void displaydeque(deque<int> v, deque<int>::iterator i, int direction)
{
    if (direction == FORWARD)
        for (i = v.begin(); i < v.end(); i++)
            cout << *i << " ";
    else
        for (i = v.end()-1; i >= v.begin(); i--)
            cout << *i << " ";

    cout << endl;
}

```

The output of this program is...

```

C:\DOCUMENTS AND SETTINGS\WANCEMY DOCUMENTS\SCHOOL RELATED\OOP\ONLINE DELIV...
Deck starts with 0 elements.
Deck now has 10 elements.
  Elements are: 9 8 7 6 5 4 3 2 1 0

  Using iterator forward the elements are: 9 8 7 6 5 4 3 2 1 0
  Using iterator backward the elements are: 0 1 2 3 4 5 6 7 8 9
  After insertion at beginning: 21 9 8 7 6 5 4 3 2 1 0
  After insertion at end: 21 9 8 7 6 5 4 3 2 1 0 29
  After erase at end: 21 9 8 7 6 5 4 3 2 1 0
  Shuffled elements are: 6 7 21 8 0 3 2 1 5 9 4

  The value 6 was not found in the deque
  Sorted elements are: 0 1 2 3 4 5 6 7 8 9 21

  The value 6 was found in the deque

  Now emptying the deque ...
  after pop_front #0: 1 2 3 4 5 6 7 8 9 21
  after pop_front #1: 2 3 4 5 6 7 8 9 21
  after pop_front #2: 3 4 5 6 7 8 9 21
  after pop_front #3: 4 5 6 7 8 9 21
  after pop_front #4: 5 6 7 8 9 21
  after pop_front #5: 6 7 8 9 21
  after pop_front #6: 7 8 9 21
  after pop_front #7: 8 9 21
  after pop_front #8: 9 21
  after pop_front #9: 21
  after pop_front #10:
Deck ends with 0 elements.

```

List Class Template

The last of the sequential containers available in the Standard Template Library is the list. The efficiency of inserting elements into a list is the same regardless of the location. Access time however, is higher for elements in the middle of the list. This section looks at declaring and using lists.

Declaring Lists

A list must specify which single type of data it will contain. The format is...

```
list<dataType> listName;
```

For example to declare a list called *names* that will hold a number of strings, the declaration is...

```
list<string> names;
```

Your code must include the list library for any list access...

```
#include <list>
```

Look at this simple list program ...

```
#include <iostream>
#include <string>
#include <list>
using namespace std;

void dump (list<string>& l);

void main()
{
    list<string> names;

    //insert names into list
    names.insert(names.begin(), "Ann");
    names.insert(names.end(), "Tom");
    names.insert(names.begin(), "Mary");
    names.insert(names.begin(), "Dick");
    cout << endl << "Names in original order" << endl;
    dump(names);

    //reverse the list
    names.reverse();
    cout << endl << "Names in reverse order" << endl;
    dump(names);

    //sort the list ... increasing order (alphabetical)
    names.sort();
    cout << endl << "Names in sorted order" << endl;
    dump(names);

    //remove elements from list
    cout << "\nNow emptying the list ..." << endl;
    int size = names.size();
```

```

        for (int x = 0; x < size; x++)
        {
            names.pop_back();
            cout << "\nafter pop_back #" << x << endl;
            dump(names);
        }
    }

void dump (list<string>& lst)
{
    //const_iterator used if iteration doesn't change any elements
    list<string>::const_iterator iter;

    for (iter = lst.begin(); iter != lst.end(); iter++)
        cout << *iter << endl;
}

```

The main differences between this program and the previous ones are caused because the list is holding strings. Take a close look at the *dump()* function. The parameter list identifies that the list is passed by reference. Because the list may be large, this is a more efficient way of passing the information to the function. Now take a look at the iterator declaration. Instead of **iterator** it uses **const_iterator**. This keyword should be used anytime the iteration is just reading the elements. If any of the elements will be changed, stick to the **iterator** keyword. The output of this program is...

```

C:\DOCUMENTS AND SETTINGS\NANCE\MY DOCUMENTS\SCHOOL RELATED\VOOP\ONLINE DELIV...
Names in original order
Dick
Mary
Ann
Tom

Names in reverse order
Tom
Ann
Mary
Dick

Names in sorted order
Ann
Dick
Mary
Tom

Now emptying the list ...

after pop_back #0
Ann
Dick
Mary

after pop_back #1
Ann
Dick

after pop_back #2
Ann

after pop_back #3

```

Set Class Template

The first of the associative containers available in the Standard Template Library is the set. A set stores a list of unique keys that are sorted as they are inserted into the set. Once a key has been added to the set, it cannot be changed. Instead you must insert a modified key as a new element and then erase the old one. This section looks at declaring and using sets.

Declaring Sets

A set must specify which single type of data it will contain. It may also define a comparer ... an identification of how the keys are sorted as they are inserted. The basic format without the comparer is...

```
set<dataType> setName;
```

Some compilers require the comparer to be included. The format is...

```
set<dataType, comparer<dataType> > setName;
```

The two available comparers are *less* and *greater*. *Less* is the default comparer and sorts the keys in increasing order. If you want the keys sorted in decreasing order you must include the *greater* comparer.

For example to declare a set called *s1* that will hold a number of strings in alphabetical order, the declaration is either ...

```
set<string> s1;  
  
set<string, less<string> > s1;
```

To declare a set called *s2* that will hold a number of strings in reverse alphabetical order, the declaration is...

```
set<string, greater<string> > s2;
```

In order to compile this set declaration, your code must include the set library for any set access...

```
#include <set>
```

Look at this simple set program ...

```
#include <iostream>  
#include <string> // some compilers need the string library included  
#include <set>    //before the container libraries  
#include <algorithm>  
using namespace std;  
  
void dump(set<string, less<string> > &s);  
void dump(set<string, greater<string> > &s);
```



```

void main()
{
    //create and populate first set
    set<string> s1;
    s1.insert("Sam");
    s1.insert("Mary");
    s1.insert("Ann");
    s1.insert("Fred");
    cout << "Set 1: ";
    dump(s1);

    //create and populate second set
    set<string> s2;
    s2.insert("Sam");
    s2.insert("Mary");
    s2.insert("Tanya");
    s2.insert("George");
    cout << "Set 2: ";
    dump(s2);

    //search for and modify an element in the second set
    string key, newkey;
    cout << "\nWhich element do you want to modify in set 2? ";
    cin >> key;
    cout << "\nEnter replacement element: ";
    cin >> newkey;

    if (s2.find(key) == s2.end())
        cout << "\n\tElement doesn't exist in set 2\n";
    else
    {
        s2.insert(newkey);
        s2.erase(key);
    }

    cout << "\nSet 1: ";
    dump(s1);
    cout << "Set 2: ";
    dump(s2);

    //find all elements in both sets
    set<string> sResult;

    set_union(s1.begin(), s1.end(), s2.begin(), s2.end(),
              inserter(sResult, sResult.begin()));
    cout << "\nUnion of 2 sets: ";
    dump(sResult);
    sResult.erase(sResult.begin(), sResult.end());

    //find and sort common elements from both sets
    set_intersection(s1.begin(), s1.end(),
                    s2.begin(), s2.end(),
                    inserter(sResult, sResult.begin()));
    cout << "Intersection of 2 sets: ";
    dump(sResult);
    sResult.erase(sResult.begin(), sResult.end());
}

```

```

//find elements in first set that aren't in second one
set_difference(s1.begin(),s1.end(),
               s2.begin(),s2.end(),
               inserter(sResult,sResult.begin()));
cout << "Elements in set 1 not in set 2: ";
dump(sResult);
sResult.erase(sResult.begin(),sResult.end());

//clear both sets of all elements
cout << "\n\nClearing set 1. ";
s1.clear();
cout << "\tElements left: " << s1.size() ;
dump(s1);

cout << "\nClearing set 2. ";
s2.clear();
cout << "\tElements left: " << s2.size() ;
}

//=====

void dump(set<string, less<string> > &s)
{
    set<string, less<string> >::const_iterator pos;

    for (pos = s.begin(); pos != s.end(); ++pos)
        cout << *pos << ' ';

    cout << endl;
}

```