

Diploma Information Technology

Module: Programming 2B

Module Code: PRG220

STUDY GUIDE




Damelin









www.damelin.co.za

Contents

Module Information	iii
STUDY UNIT 1: ADVANCED ARRAY CONCEPTS.....	1
1.1 Introduction.....	1
1.2 Sorting Array Elements.....	1
1.3 Sorting Arrays of Objects.....	5
1.4 Using Two-Dimensional and Other Multidimensional Arrays.....	7
1.5 Using the Arrays Class	12
1.6 Creating Enumerations	17
1.7 Summary	19
STUDY UNIT 2 INTRODUCTION TO INHERITANCE	21
2.1 Introduction.....	21
2.2 Learning About the Concept of Inheritance.....	21
2.3 Extending Classes.....	25
2.4 Summary	36
STUDY UNIT 3 ADVANCED INHERITANCE CONCEPTS	37
3.1 Introduction.....	37
3.2 Creating and Using Abstract Classes	37
3.3 Programmers of an abstract class can include two method types.....	38
3.4 Using Dynamic Method Binding.....	40
3.5 Using a Superclass as a Method Parameter Type	41
3.6 Creating Arrays of Subclass Objects	42
3.7 Using the Object Class and Its Methods	43
3.8 Using the toString() Method.....	43
3.9 Using the equals() Method.....	44
3.10 Using Inheritance to Achieve Good Software Design.....	45
3.11 Creating Interfaces to Store Related Constants	50
3.12 Creating and Using Packages	51
Summary.....	53
STUDY UNIT 4 EXCEPTION HANDLING.....	54
4.1 Introduction.....	54
4.2 Learning About Exceptions	54
4.3 Trying Code and Catching Exceptions.....	57
4.4 Throwing and Catching Multiple Exceptions	58
4.5 Using the finally Block.....	60
4.6 Understanding the Advantages of Exception Handling	62
4.7 Specifying the Exceptions, a Method Can Throw.....	64

4.8 Tracing Exceptions Through the Call Stack	66
4.9 Creating Your Own Exceptions.....	67
4.10 Using Assertions.....	67
Summary	68
STUDY UNIT 5 FILE INPUT AND OUTPUT	69
5.1 Introduction.....	69
5.2 Understanding Computer Files	69
5.3 Java provides built-in classes that contain methods to help you with these tasks.	70
5.4 ObjectOutputStream.....	74
Summary	74
STUDY UNIT 6 INTRODUCTION TO SWING COMPONENTS	75
6.1 Introduction.....	75
6.2 Understanding Swing Components.....	75
6.3 Using the JFrame Class	76
6.4 Customizing a JFrame's Appearance	78
6.5 Extending the JFrame Class.....	83
6.6 Learning About Event-Driven Programming.....	88
Summary	96
STUDY UNIT 7 ADVANCED GUI TOPICS	97
7.1 Introduction.....	97
7.2 Understanding the Content Pane.....	97
7.3 Using Colour.....	99
7.4 Learning More About Layout Managers.....	101
7.5 Using BorderLayout.....	101
Summary	120
References.....	120

Module Information

Name of programme	Diploma Information Technology
Type of programme	Full-Time & Part Time
NQF Level	6
Name of module	Programming 2B
Credits	15
Notional hours	150
Learning Outcomes:	<p>At the end of this module learners should be able to:</p> <ul style="list-style-type: none"> • Discuss basic issues of programming. • Discuss various types of data that can be incorporated into a program. • Use a method as a reusable sequence of statements in each task. • Explain and apply classes. • Discuss and apply selection and repetition structures. • Correctly use arrays in programs. • Discuss and apply inheritance. • Correctly handle exceptions. • Critically evaluate and apply Graphical user interface and Visual Studio IDE. • Explain and apply controls. • Explain and apply event handling. • Use files and streams.
Prescribed textbooks and other sources	Java Programming Farrell, J. 2016 Cengage 9781285856919
Icons	 Additional information  Self-check activity  Reading in prescribed textbook  Bright ideas  Think point  Case Study  Study Group Discussion  Vocabulary

STUDY UNIT 1: ADVANCED ARRAY CONCEPTS

1.1 Introduction

Study unit 1 discusses advanced array concepts, including array sorting techniques. Students will also learn about multidimensional arrays, and the Arrays and ArrayList utility classes.

After studying this unit, you should be able to:

- Sort array elements using the bubble sort algorithm
- Sort array elements using the insertion sort algorithm
- Use two-dimensional and other multidimensional arrays
- Use the Arrays class
- Use the ArrayList class

1.2 Sorting Array Elements

Sorting is the process of arranging a series of objects in some logical order. When you place objects in order, beginning with the object that has the lowest value, you are sorting in ascending order; conversely, when you start with the object that has the largest value, you are sorting in descending order. The simplest possible sort involves two values that are out of order. To place the values in order, you must swap the two values. Suppose that you have two variables `valA` and `valB` and further suppose that `valA = 16` and `valB = 2`. To exchange the values of the two variables, you cannot simply use the following code:

```
valA = valB; // 2 goes to valA
```

```
valB = valA; // 2 goes to valB
```

If `valB` is 2, after you execute `valA = valB`; both variables hold the value 2. The value 16 that was held in `valA` is lost. When you execute the second assignment statement, `valB = valA`; each variable still holds the value 2. The solution that allows you to retain both values is to employ a variable to hold `valA`'s value temporarily during the swap:

```
temp = valA; // 16 goes to temp
```

```
valA = valB; // 2 goes to valA
```

```
valB = temp; // 16 goes to valB
```

Using this technique, `valA`'s value (16) is assigned to the `temp` variable. The value of `valB` (2) is then assigned to `valA`, so `valA` and `valB` are equivalent. Then, the `temp` value (16) is assigned to `valB`, so the values of the two variables finally are swapped. If you want to sort any two values, `valA` and `valB`, in ascending order so that `valA` is always the lower value, you

use the following if statement to make the decision whether to swap. If valA is more than valB, you want to swap the values. If valA is not more than valB, you do not want to swap the values.

```
if(valA > valB)
{
    temp = valA;
    valA = valB;
    valB = temp;
}
```

Sorting two values is a simple task; sorting more values (valC, valD, valE, and so on) is more complicated. The task becomes manageable when you know how to use an array. As an example, you might have a list of five numbers that you want to place in ascending order. One approach is to use a method popularly known as a bubble sort.

In a bubble sort, you continue to compare pairs of items, swapping them if they are out of order, so that the smallest items “bubble” to the top of the list, eventually creating a sorted list. The bubble sort is neither the fastest nor most efficient sorting technique, but it is one of the simplest to comprehend and provides deeper understanding of array element manipulation. To use a bubble sort, you place the original, unsorted values in an array, such as the following:

```
int[ ] someNums = {88, 33, 99, 22, 54};
```

You compare the first two numbers; if they are not in ascending order, you swap them. You compare the second and third numbers; if they are not in ascending order, you swap them. You continue down the list. Generically, for any someNums[x], if the value of someNums[x] is larger than someNums [x + 1], you want to swap the two values. With the numbers 88, 33, 99, 22, and 54, the process proceeds as follows:

- Compare 88 and 33. They are out of order. Swap them. The list becomes 33, 88, 99, 22, and 54.
- Compare the second and third numbers in the list, 88 and 99. They are in order. Do nothing.
- Compare the third and fourth numbers in the list, 99 and 22. They are out of order. Swap them. The list becomes 33, 88, 22, 99, 54.
- Compare the fourth and fifth numbers, 99 and 54. They are out of order. Swap them. The list becomes 33, 88, 22, 54, 99.

When you reach the bottom of the list, the numbers are not in ascending order, but the largest number, 99, has moved to the bottom of the list. This feature gives the bubble sort its name the “heaviest” value has sunk to the bottom of the list as the “lighter” values have bubbled to the top.

Assuming `b` and `temp` both have been declared as integer variables, the code so far is as follows:

```
for(b = 0; b < someNums. Length - 1; ++b)
    if(someNums[b] > someNums [b + 1])
    {
        temp = someNums[b];
        someNums[b] = someNums [b + 1];
        someNums [b + 1] = temp;
    }
```

Notice that the `for` statement tests every value of `b` from 0 through 3. The array `someNums` contains five integers. The subscripts in the array range in value from 0 through 4. Within the `for` loop, each `someNums[b]` is compared to `someNums [b + 1]`, so the highest legal value for `b` is 3 when array element `b` is compared to array element `b + 1`. For a sort on any size array, the value of `b` must remain less than the array’s length minus 1. The list of numbers that began as 88, 33, 99, 22, 54 is currently 33, 88, 22, 54, 99. To continue to sort the list, you must perform the entire comparison-swap procedure again.

- Compare the first two values, 33 and 88. They are in order; do nothing.
- Compare the second and third values, 88 and 22. They are out of order. Swap them so the list becomes 33, 22, 88, 54, 99.
- Compare the third and fourth values, 88 and 54. They are out of order. Swap them so the list becomes 33, 22, 54, 88, 99.
- Compare the fourth and fifth values, 88 and 99. They are in order; do nothing.

After this second pass through the list, the numbers are 33, 22, 54, 88, and 99 close to ascending order, but not quite. You can see that with one more pass through the list, the values 22 and 33 will swap, and the list is finally placed in order. To fully sort the worst-case list, one in which the original numbers are descending (as out-of-ascending order as they could possibly be), you need to go through the list four times, making comparisons and swaps.

At most, you always need to pass through the list as many times as its length minus one. The example below shows the entire procedure.

```
for(b = 0; b < someNums.length - 1; ++b)
    if(someNums[b] > someNums [b + 1])
    {
        temp = someNums[b];
        someNums[b] = someNums [b + 1];
        someNums [b + 1] = temp;
    }
```

When you use a bubble sort to sort any array into ascending order, the largest value “falls” to the bottom of the array after you have compared each pair of values in the array one time. The second time you go through the array making comparisons, there is no need to check the last pair of values. The largest value is guaranteed to already be at the bottom of the array.

You can make the sort process even more efficient by using a new variable for the inner for loop and reducing the value by one on each cycle through the array. The example shows how you can use a new variable named `comparisonsToMake` to control how many comparisons are made in the inner loop during each pass through the list of values to be sorted. In the shaded statement, the `comparisonsToMake` value is decremented by 1 on each pass through the list.

```
int comparisonsToMake = someNums.length - 1;
for(a = 0; a < someNums.length - 1; ++a)
{
    for(b = 0; b < comparisonsToMake; ++b)
    {
        if(someNums[b] > someNums[b + 1])
        {
            temp = someNums[b];
            someNums[b] = someNums[b + 1];
            someNums[b + 1] = temp;
        }
    }
    --comparisonsToMake;
}
```


1.3 Sorting Arrays of Objects

You can sort arrays of objects in much the same way that you sort arrays of primitive types. The major difference occurs when you make the comparison that determines whether you want to swap two array elements. When you construct an array of the primitive element type, you compare the two array elements to determine whether they are out of order. When array elements are objects, you usually want to sort based on an object field. Assume you have created a simple Employee class, as shown below.

The class holds four data fields and get and set methods for the fields.

```
public class Employee
{
    private int empNum;
    private String lastName, firstName;
    private double salary;
    public int getEmpNum()
    {
        return empNum;
    }
    public void setEmpNum(int emp)
    {
        empNum = emp;
    }
    public String getLastName()
    {
        return lastName;
    }
    public void setLastName(String name)
    {
        lastName = name;
    }
    public String getFirstName()
    {
```

```

        return firstName;
    }

    public void setFirstName(String name)
    {
        firstName = name;
    }

    public double getSalary()
    {
        return salary;
    }

    public void setSalary(double sal)
    {
        salary = sal;
    }
}

```

You can write a program that contains an array of five Employee objects using the following statement:

```
Employee[ ] someEmps = new Employee[5];
```

Assume that after you assign employee numbers and salaries to the Employee objects, you want to sort the Employees in salary order. You can pass the array to a bubbleSort() method that is prepared to receive Employee objects.

The example below shows the method.

```

public static void bubbleSort(Employee[] array)
{
    int a, b;
    Employee temp;
    int highSubscript = array.length - 1;
    for(a = 0; a < highSubscript; ++a)
        for(b = 0; b < highSubscript; ++b)
            if(array[b].getSalary() > array[b + 1].getSalary())
                {

```

```

        temp = array[b];
        array[b] = array[b + 1];
        array[b + 1] = temp;
    }
}

```

The bubbleSort() method is very similar to the bubbleSort() method you use for an array of any primitive type, but there are three major differences:

- The bubbleSort() method header shows that it receives an array of type Employee.
- The temp variable created for swapping is type Employee. The temp variable will hold an Employee object, not just one number or one field. It is important to note that even though only employee salaries are compared, you do not just swap employee salaries. You do not want to substitute one employee's salary for another's. Instead, you swap each Employee object's empNum and salary as a unit.
- The comparison for determining whether a swap should occur uses method calls to the getSalary() method to compare the returned salary for each Employee object in the array with the salary of the adjacent Employee object.

1.4 Using Two-Dimensional and Other Multidimensional Arrays

Java is an object-oriented and component-oriented language. When you declare an array such as `int[] someNumbers = new int[3];`, you can envision the three declared integers as a column of numbers in memory, as shown in the example. In other words, you can picture the three declared numbers stacked one on top of the next. An array that you can picture as a column of values, and whose elements you can access using a single subscript, is a one-dimensional or single-dimensional array. You can think of the size of the array as its height.

Java also supports two-dimensional arrays. Two-dimensional arrays have two or more columns of values, as shown in the example below. The two dimensions represent the height and width of the array. Another way to picture a two-dimensional array is as an array of arrays. It is easiest to picture two-dimensional arrays as having both rows and columns. You must use two subscripts when you access an element in a two-dimensional array. When mathematicians use a two-dimensional array, they often call it a matrix or a table; you might have used a two-dimensional array called a spreadsheet.

```

someNumbers [0][0]  someNumbers [0][1]  someNumbers [0][2]  someNumbers [0][3]
someNumbers [1][0]  someNumbers [1][1]  someNumbers [1][2]  someNumbers [1][3]
someNumbers [2][0]  someNumbers [2][1]  someNumbers [2][2]  someNumbers [2][3]

```

When you declare a one-dimensional array, you type a set of square brackets after the array's data type. To declare a two-dimensional array in Java, you type two sets of brackets after the array type.

For example, the array in the example can be declared as follows, creating an array named `someNumbers` that holds three rows and four columns:

```
int [ ][ ] someNumbers = new int[3][4];
```

Just as with a one-dimensional array, if you do not provide values for the elements in a two-dimensional numeric array, the values default to zero. You can assign other values to the array elements later. For example, `someNumbers [0][0] = 14;` assigns the value 14 to the element of the `someNumbers` array that is in the first column of the first row. Alternatively, you can initialize a two-dimensional array with values when it is created. For example, the following code assigns values to `someNumbers` when it is created:

```
int [ ][ ] someNumbers = { {8, 9, 10, 11}, {1, 3, 12, 15}, {5, 9, 44, 99} };
```

The `someNumbers` array contains three rows and four columns. You do not need to place each row of values for a two-dimensional array on its own line. However, doing so makes the positions of values easier to understand. You contain the entire set of values within an outer pair of curly braces. The first row of the array holds the four integers 8, 9, 10, and 11. Notice that these four integers are placed within their own inner set of curly braces to indicate that they constitute one row, or the first row, which is row 0.

Similarly, 1, 3, 12, and 15 make up the second row (row 1), which you reference with the subscript 1. Next, 5, 9, 44, and 99 are the values in the third row (row 2), which you reference with the subscript 2. The value of `someNumbers [0][0]` is 8. The value of `someNumbers [0][1]` is 9. The value of `someNumbers [2][3]` is 99. The value within the first set of brackets following the array name always refers to the row; the value within the second brackets refers to the column. As an example of how useful two-dimensional arrays can be, assume you own an apartment building with four floors—a basement, which you refer to as floor zero, and three other floors numbered one, two, and three. In addition, each of the floors has studio (with no bedroom) and one- and two-bedroom apartments. The monthly rent for each type of apartment is different the higher the floor, the higher the rent (the view is better), and the rent is higher for apartments with more bedrooms. Below are the rental amounts.

Floor	Zero Bedrooms	One Bedroom	Two Bedrooms
0	400	450	510
1	500	560	630
2	625	676	740
3	1000	1250	1600

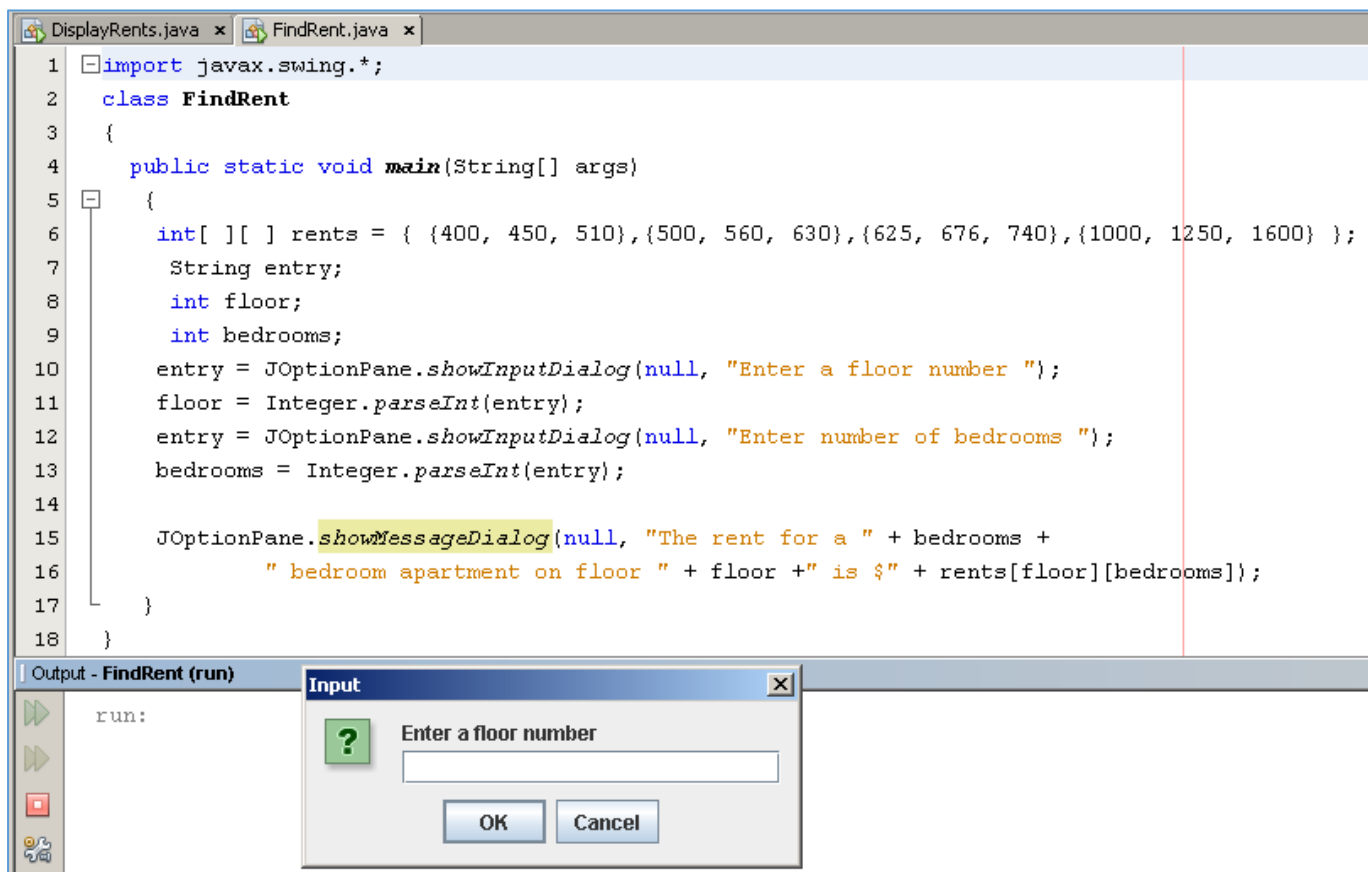
To determine a tenant's rent, you need to know two pieces of information: the floor on which the tenant rents an apartment and the number of bedrooms in the apartment. Within a Java program, you can declare an array of rents using the following code:

```
int [ ][ ] rents = { {400, 450, 510}, {500, 560, 630}, {625, 676, 740}, {1000, 1250, 1600} };
```

Assume you declare two integers to hold the floor number and bedroom count, as in the following statement:

```
int floor, bedrooms;
```

Then any tenant's rent can be referred to as `rents[floor][bedrooms]`. The example below shows an application that prompts a user for a floor number and number of bedrooms.



Passing a Two-Dimensional Array to a Method

When you pass a two-dimensional array to a method, you pass the array name just as you do with a one-dimensional array. A method that receives a two-dimensional array uses two bracket pairs following the data type in the parameter list of the method header. For example, the following method headers accept two-dimensional arrays of ints, doubles, and Employees, respectively:

```
public static void displayScores(int[][]scoresArray)
```

```
public static boolean areAllPricesHigh(double[][] prices)
```

```
public static double computePayrollForAllEmployees(Employee[][] staff)
```

In each case, notice that the brackets indicating the array in the method header is empty. There is no need to insert numbers into the brackets because each passed array name is a starting memory address. The way you manipulate subscripts within the method determines how rows and columns are accessed.

Using the length Field with a Two-Dimensional Array

Previously we have learnt that a one-dimensional array has a length field that holds the number of elements in the array. With a two-dimensional array, the length field holds the number of rows in the array. Each row, in turn, has a length field that holds the number of columns in the row. For example, suppose you declare a rents array as follows:

```
Int [ ][ ] rents = { {400, 450, 510},{500, 560, 630},{625, 676, 740},{1000, 1250, 1600} };
```

The value of `rent.length` is 4 because there are four rows in the array. The value of `rents[0].length` is 3 because there are three columns in the first row of the rents array. Similarly, the value of `rents[1].length` also is 3 because there are three columns in the second row. The example below shows an application that uses the length fields associated with the rents array to display all the rents. The floor variable varies from 0 through one less than 4 in the outer loop, and the bdrms variable varies from 0 through one less than 3 in the inner loop.

```
DisplayRents.java x
1  class DisplayRents
2  {
3      public static void main(String[] args)
4      {
5          int[][] rents = {{400, 450, 510},{500, 560, 630},{625, 676, 740},{1000, 1250, 1600}};
6
7          int floor;
8          int bdrms;
9
10         for(floor = 0; floor < rents.length; ++floor)
11             for(bdrms = 0; bdrms < rents[floor].length; ++bdrms)
12                 System.out.println("Floor " + floor + " Bedrooms " + bdrms + " Rent is R " + rents[floor][bdrms]);
13     }
14 }
```

Output - DisplayRents (run)

```
run:
Floor 0 Bedrooms 0 Rent is R 400
Floor 0 Bedrooms 1 Rent is R 450
Floor 0 Bedrooms 2 Rent is R 510
Floor 1 Bedrooms 0 Rent is R 500
Floor 1 Bedrooms 1 Rent is R 560
Floor 1 Bedrooms 2 Rent is R 630
Floor 2 Bedrooms 0 Rent is R 625
Floor 2 Bedrooms 1 Rent is R 676
Floor 2 Bedrooms 2 Rent is R 740
Floor 3 Bedrooms 0 Rent is R 1000
Floor 3 Bedrooms 1 Rent is R 1250
Floor 3 Bedrooms 2 Rent is R 1600
BUILD SUCCESSFUL (total time: 0 seconds)
```

Understanding Ragged Arrays

In a two-dimensional array, each row is an array. In Java, you can declare each row to have a different length. When a two-dimensional array has rows of different lengths, it is a ragged array because you can picture the ends of each row as uneven. You create a ragged array by defining the number of rows for a two-dimensional array, but not defining the number of columns in the rows.

For example, suppose you have four sales representatives, each of whom covers a different number of states as their sales territory. Further suppose you want an array to store total sales for each state for each sales representative. You would define the array as follows:

```
double [ ] [ ] sales = new double[4][ ];
```

This statement declares an array with four rows, but the rows are not yet created. Then, you can declare the individual rows, based on the number of states covered by each salesperson as follows:

```
sales [0] = new double [12];
```

```
sales [1] = new double [18];
```

```
sales [2] = new double [9];
```

```
sales [3] = new double [11];
```

Using Other Multidimensional Arrays

Besides one- and two-dimensional arrays, Java also supports arrays with three, four, and more dimensions. The general term for arrays with more than one dimension is multidimensional arrays. For example, if you own an apartment building with several floors and different numbers of bedrooms available in apartments on each floor, you can use a two-dimensional array to store the rental fees. If you own several apartment buildings, you might want to employ a third dimension to store the building number. An expression such as `rents [building][floor][bedrooms]` refers to a specific rent figure for a building whose building number is stored in the `building` variable and whose floor and bedroom numbers are stored in the `floor` and `bedrooms` variables. Specifically, `rents [5][1][2]` refers to a two-bedroom apartment on the first floor of building 5. When you are programming in Java, you can use four, five, or more dimensions in an array. If you can keep track of the order of the variables needed as subscripts, and if you don't exhaust your computer's memory, Java lets you create arrays of any size.

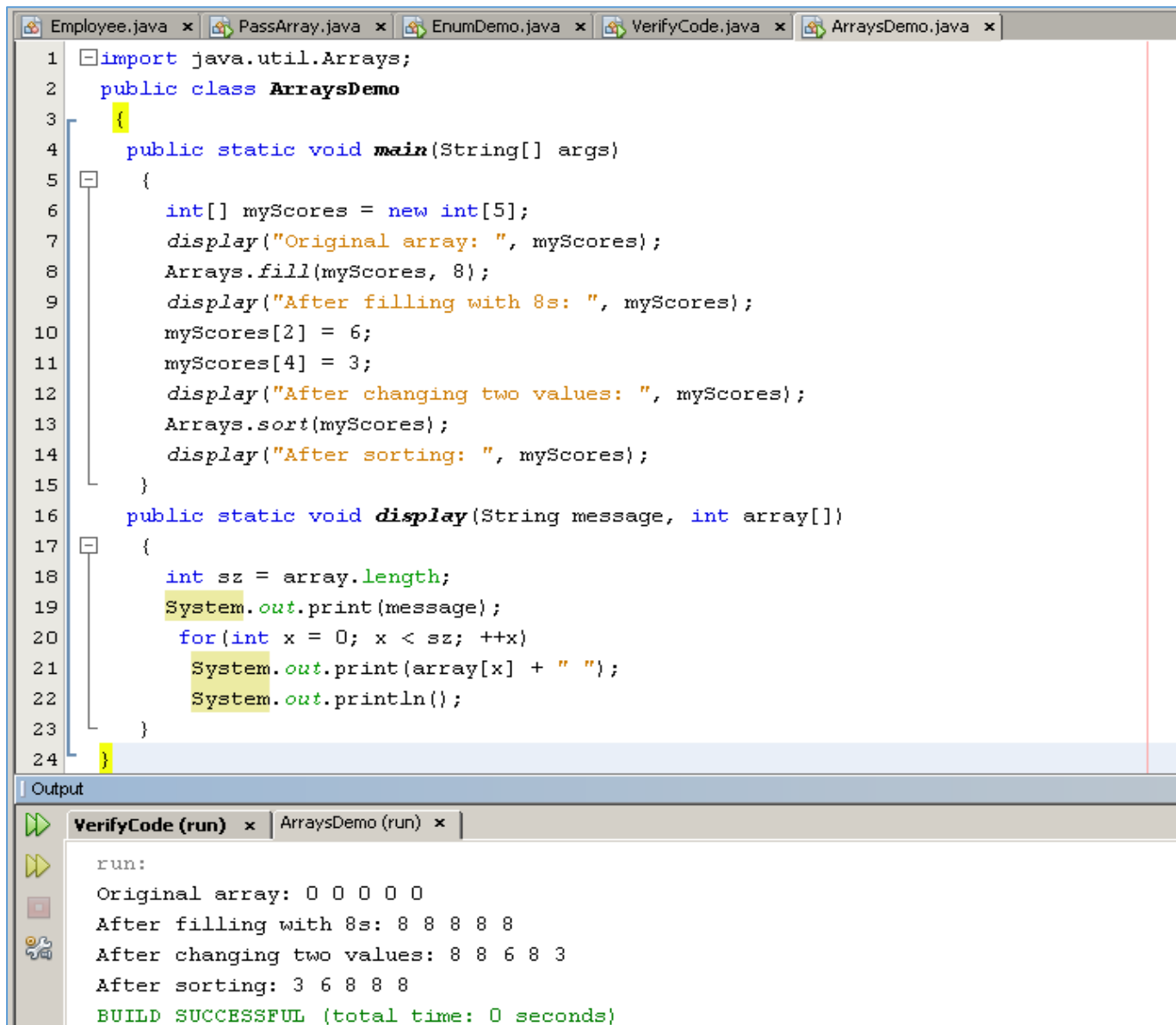
1.5 Using the Arrays Class

When you fully understand the power of arrays, you will want to use them to store all kinds of objects. Frequently, you will want to perform similar tasks with different arrays for example, filling them with values and sorting their elements. Java provides an `Arrays` class, which contains many useful methods for manipulating arrays. For each method listed in the left column of the table, type stands for a data type; an overloaded version of each method exists for each appropriate data type. For example, there is a version of the `sort()` method to sort `int`, `double`, `char`, `byte`, `float`, `long`, `short`, and `Object` arrays.

The methods in the `Arrays` class are a static method, which means you use them with the class name without instantiating an `Arrays` object. The `Arrays` class is located in the `java.util` package, so you can use the statement `import java.util.*;` to access it. In the `ArraysDemo` class, the `myScores` array is created to hold five integers. Then, a message and the array reference are passed to a `display()` method. The first line of the output in the example shows that the original array is filled with 0s at creation. After the first display, the `Arrays.fill()` method is called in the first shaded statement in the example. Because the arguments are the name of the array and the number 8, when the array is displayed a second time the output is all 8s.

In the application, two of the array elements are changed to 6 and 3, and the array is displayed again. Finally, in the second shaded statement, the `Arrays.sort()` method is called.

The output in example shows that when the `display()` method executes the fourth time, the array elements have been sorted in ascending order.



```
1  import java.util.Arrays;
2  public class ArraysDemo
3  {
4      public static void main(String[] args)
5      {
6          int[] myScores = new int[5];
7          display("Original array: ", myScores);
8          Arrays.fill(myScores, 8);
9          display("After filling with 8s: ", myScores);
10         myScores[2] = 6;
11         myScores[4] = 3;
12         display("After changing two values: ", myScores);
13         Arrays.sort(myScores);
14         display("After sorting: ", myScores);
15     }
16     public static void display(String message, int array[])
17     {
18         int sz = array.length;
19         System.out.print(message);
20         for(int x = 0; x < sz; ++x)
21             System.out.print(array[x] + " ");
22         System.out.println();
23     }
24 }
```

Output

```
run:
Original array: 0 0 0 0 0
After filling with 8s: 8 8 8 8 8
After changing two values: 8 8 6 8 3
After sorting: 3 6 8 8 8
BUILD SUCCESSFUL (total time: 0 seconds)
```

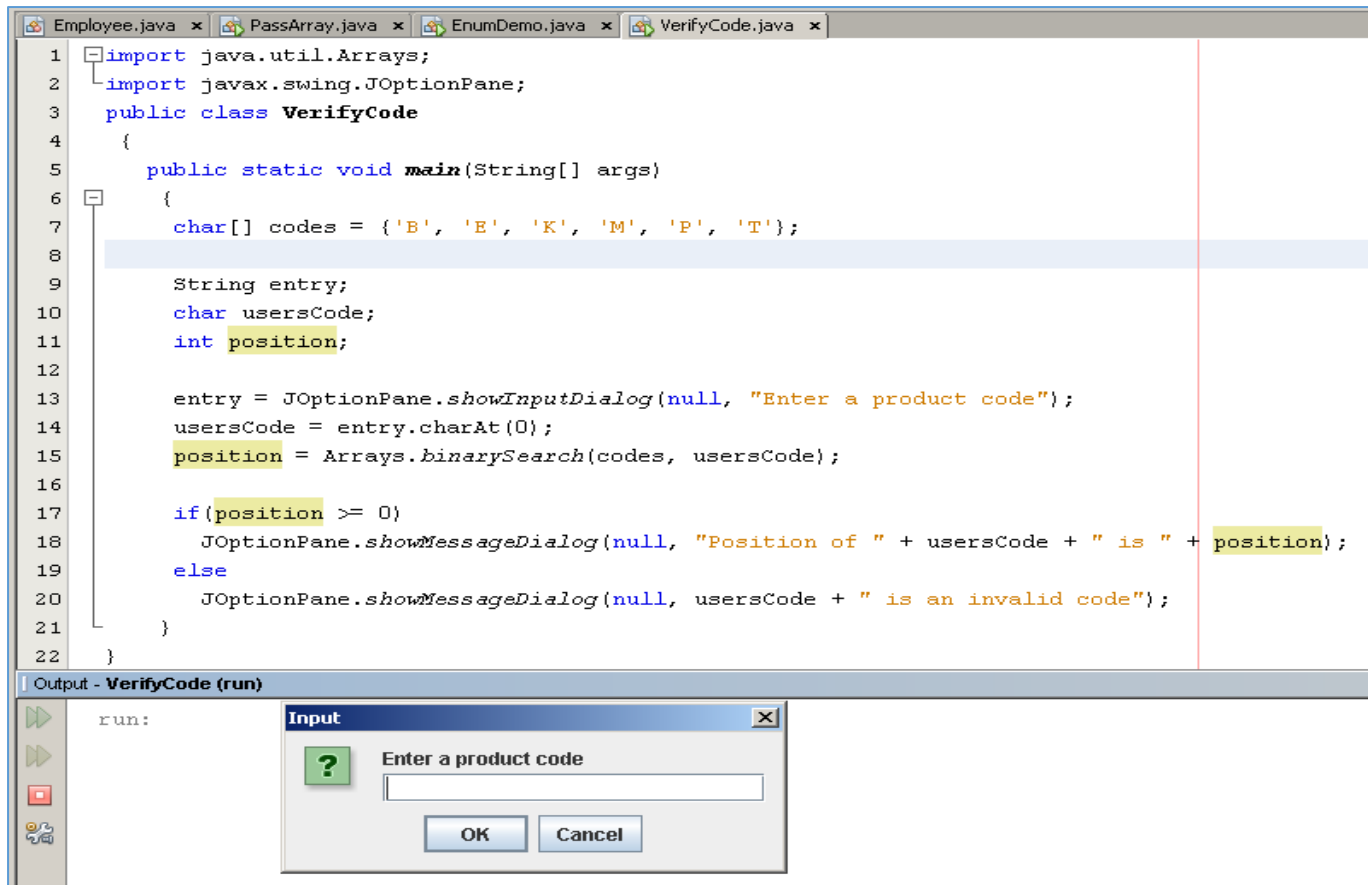
The `Arrays` class `binarySearch()` methods provide convenient ways to search through sorted lists of values of various data types. It is important that the list be in order before you use it in a call to `binarySearch()`; otherwise, the results are unpredictable. You do not have to understand how a binary search works to use the `binarySearch()` method, but basically the operation takes place as follows:

You have a sorted array and an item for which you are searching within the array. Based on the array size, you determine the middle position. (In an array with an even number of elements, this can be either of the two middle positions.). You compare the item you are looking for with the element in the middle position of the array and decide whether your item is above that point in the array that is, whether your item's value is less than the middle-point value.

If it is above that point in the array, you next find the middle position of the top half of the array; if it is not above that point, you find the middle position of the bottom half. Either way, you compare your item with that of the new middle position and divide the search area in half again. Ultimately, you find the element or determine that it is not in the array.

Suppose your organization uses six single-character product codes. The example below contains a VerifyCode application that verifies a product code entered by the user. The array codes hold six values in ascending order. The user enters a code that is extracted from the first String position using the String class `charAt()` method. Next, the array of valid characters and the user-entered character are passed to the `Arrays.binarySearch()` method. If the character is found in the array, its position is returned. If the character is not found in the array, a negative integer is returned and the application displays an error message.

The example shows the program's execution when the user enters K; the character is found in position 2 (the third position) in the array.



Using the ArrayList Class

In addition to the Arrays class, Java provides an ArrayList class which can be used to create containers that store lists of objects. The ArrayList class provides some advantages over the Arrays class. Specifically, an ArrayList is dynamically resizable, meaning that its size can change during program execution. This means that:

- You can add an item at any point in an ArrayList container, and the array size expands automatically to accommodate the new item.
- You can remove an item at any point in an ArrayList container, and the array size contracts automatically.

To use the ArrayList class, you must use one of the following import statements:

```
import java.util.ArrayList;
```

```
import java.util.*;
```

Then, to declare an ArrayList, you can use the default constructor, as in the following example:

```
ArrayList names = new ArrayList();
```

The default constructor creates an ArrayList with a capacity of 10 items. An ArrayList's capacity is the number of items it can hold without having to increase its size. An ArrayList's

capacity is greater than or equal to its size. You can also specify a capacity if you like. For example, the following statement declares an ArrayList that can hold 20 names:

```
ArrayList names = new ArrayList(20);
```

If you know you will need more than 10 items at the outset, it is more efficient to create an ArrayList with a larger capacity. To add an item to the end of an ArrayList, you can use the `add()` method. For example, to add the name Abigail to an ArrayList named `names`, you can make the following statement:

```
names.add("Abigail");
```

You can insert an item into a specific position in an ArrayList by using an overloaded version of the `add()` method that includes the position. For example, to insert the name Bob in the first position of the `names` ArrayList, you use the following statement:

```
names.add(0, "Bob");
```

Understanding the Limitations of the ArrayList Class

An ArrayList can be used to store any type of object reference. In fact, one ArrayList can store multiple types. However, this creates two drawbacks:

- You cannot use an ArrayList to store primitive types such as `int`, `double`, or `char` because those types are not references. If you want to work with primitive types, you can create an array or use the `Arrays` class, but you cannot use the `ArrayList` class.
- When you want to store ArrayList elements, you must cast them to the appropriate reference type before you can do so, or you must declare a reference type in the ArrayList declaration.

For example, if you want to declare a `String` to hold the first name in the `names` ArrayList, you must make statements such as the following:

```
String firstName;
```

```
firstName = (String)names.get(0);
```

The cast operator (`String`) converts the generic returned object from the `get()` method to a `String`. If you do not perform this cast, you receive an error message indicating that you are using incompatible types. You can eliminate the need to perform a cast with ArrayList objects by specifying the type that an ArrayList can hold. For example, you can declare an ArrayList of names as follows:

```
ArrayList<String> names = new ArrayList<String>();
```

Creating an ArrayList declaration with a specified type provides several advantages:

- You no longer must use the cast operator when retrieving an item from the ArrayList.

- Java checks to make sure that only items of the appropriate type are added to the list.
- The compiler warning that indicates your program uses an unchecked or unsafe operation is eliminated.

1.6 Creating Enumerations

Data types have a specific set of values. For example, you learned that a byte cannot hold a value larger than 127 and an int cannot hold a value larger than 2,147,483,647. You can also create your own data types that have a finite set of legal values. A programmer created data type with a fixed set of values is an enumerated data type. In Java, you create an enumerated data type in a statement that uses the keyword `enum`, an identifier for the type, and a pair of curly braces that contain a list of the enum constants, which are the allowed values for the type. For example, the following code creates an enumerated type named `Month` that contains 12 values:

```
enum Month {JAN, FEB, MAR, APR, MAY, JUN, JUL, AUG, SEP, OCT, NOV, DEC};
```

By convention, the identifier for an enumerated type begins with an uppercase letter. This convention makes sense because an enumerated type is a class. Also, by convention, the enum constants, like other constants, appear in all uppercase letters. The constants are not strings and they are not enclosed in quotes; they are Java identifiers.

After you create an enumerated data type, you can declare variables of that type. For example, you might declare the following:

```
Month birthMonth;
```

You can assign any of the enum constants to the variable. Therefore, you can code a statement such as the following:

```
birthMonth = Month.MAY;
```

An enumeration type like `Month` is a class, and its enum constants act like objects instantiated from the class, including having access to the methods of the class. Each of these methods is nonstatic; that is, each is used with an enum object. You can declare an enumerated type in its own file, in which case the filename matches the type name and has a `.java` extension. You will use this approach in a You Do It exercise later in this chapter. Alternatively, you can declare an enumerated type within a class, but not within a method.

```
Employee.java x PassArray.java x EnumDemo.java x
1 package enumdemo;
2 import java.util.Scanner;
3 public class EnumDemo
4 {
5     enum Month {JAN, FEB, MAR, APR, MAY, JUN, JUL, AUG, SEP, OCT, NOV, DEC};
6     public static void main(String[] args)
7     {
8         Month birthMonth;
9         String userEntry;
10        int position;
11        int comparison;
12        Scanner input = new Scanner(System.in);
13        System.out.println("The months are:");
14        for(Month mon : Month.values())
15            System.out.print(mon + " ");
16        System.out.print("\n\nEnter the first three letters of " + "your birth month >> ");
17        userEntry = input.nextLine().toUpperCase();
18        birthMonth = Month.valueOf(userEntry);
19        System.out.println("You entered " + birthMonth);
20        position = birthMonth.ordinal();
21        System.out.println(birthMonth + " is in position " + position);
22        System.out.println("So its month number is " + (position + 1));
23        comparison = birthMonth.compareTo(Month.JUN);
24        if(comparison < 0)
25            System.out.println(birthMonth + " is earlier in the year than " + Month.JUN);
26        else
27            if(comparison > 0)
28                System.out.println(birthMonth + " is later in the year than " + Month.JUN);
29            else
30                System.out.println(birthMonth + " is " + Month.JUN);
31        }
32    }
```

```
Output - EnumDemo (run)
run:
The months are:
JAN FEB MAR APR MAY JUN JUL AUG SEP OCT NOV DEC

Enter the first three letters of your birth month >> jul
You entered JUL
JUL is in position 6
So its month number is 7
JUL is later in the year than JUN
BUILD SUCCESSFUL (total time: 19 seconds)
```

In the application in the above example, a Month enumeration is declared; in the main() method, a Month variable is declared in the first shaded statement. The enhanced for loop declares a local Month variable named mon that takes on the value of each element in the Month.value() array in turn so it can be displayed. In the program in the user then is prompted to enter the first three letters for a month, which are converted to their uppercase equivalents. The third shaded statement in the figure uses the valueOf() method to convert the user's string to an enumeration value.

The fourth shaded statement gets the position of the month in the enumeration list. The last shaded statement compares the entered month to the JUN constant. This is followed by an if statement that displays whether the user's entered month comes before or after JUN in the list, or is equivalent to it.

Starting with Java 7, you can use comparison operators with enumeration constants instead of using the compareTo() method to return a number. For example, you can write the following:

```
if(birthMonth < Month.JUN)
```

```
    System.out.println(birthMonth + " is earlier in the year than " + Month.JUN);
```

You can use enumerations to control a switch structure. The example contains a class that declares a Property enumeration for a real estate company. The program assigns one of the values to a Property variable and then uses a switch structure to display an appropriate message.

1.7 Summary

Sorting is the process of arranging a series of objects in some logical order. Two-dimensional arrays have both rows and columns. A programmer-created data type with a fixed set of values is an enumerated data type.



Self-Evaluation Questions

- 1) Create a java console program which accepts numbers from a user and then sorts them in ascending order.
-

STUDY UNIT 2 INTRODUCTION TO INHERITANCE

2.1 Introduction

Students will learn to derive classes and to access constructors, data fields, and methods of a superclass. After studying this unit, you should be able to:

- Learn about the concept of inheritance
- Extend classes
- Override superclass methods
- Call constructors during inheritance
- Access superclass methods
- Employ information hiding
- Learn which methods you cannot override

2.2 Learning About the Concept of Inheritance

In Java and all object-oriented languages, inheritance is a mechanism that enables one class to inherit, or assume, both the behavior and the attributes of another class. Inheritance is the principle that allows you to apply your knowledge of a general category to more specific objects. A class can inherit all the attributes of an existing class, meaning that you can create a new class simply by indicating the ways in which it differs from an already existing class. You are familiar with the concept of inheritance from all sorts of non-programming situations.

When you use the term inheritance, you might think of genetic inheritance. You know from biology that your blood type and eye colour are the product of inherited genes; you can say that many facts about you—your attributes, or “data fields” are inherited. Similarly, you often can credit your behavior to inheritance. For example, your attitude toward saving money might be the same as your grandmother’s, and the odd way that you pull on your ear when you are tired might match what your Uncle Steve does; thus, your methods are inherited, too.

You also might choose plants and animals based on inheritance. You plant impatiens next to your house because of your shady street location; you adopt a Doberman pinscher because you need a watchdog. Every individual plant and pet has slightly different characteristics, but within a species, you can count on many consistent inherited attributes and behaviors. Similarly, the classes you create in object-oriented programming languages can inherit data and methods from existing classes. When you create a class by making it inherit from another class, you are provided with data fields and methods automatically.

Beginning with the first chapter of this book, you have been creating classes and instantiating objects that are members of those classes. Programmers and analysts sometimes use a graphical language to describe classes and object-oriented processes; this Unified Modeling Language (UML) consists of many types of diagrams. UML diagrams can help illustrate inheritance.

For example, consider the simple Employee class shown below. The class contains two data fields, `empNum` and `empSal`, and four methods, a get and set method for each field. The example shows a UML class diagram for the Employee class. A class diagram is a visual tool

that provides you with an overview of a class. It consists of a rectangle divided into three sections the top section contains the name of the class, the middle section contains the names and data types of the attributes, and the bottom section contains the methods. Only the method return type, name, and arguments are provided in the diagram the instructions that make up the method body are omitted.

```
public class Employee
{
private int empNum;
private double empSal;
public int getEmpNum()
{
return empNum;
}
public double getEmpSal()
{
return empSal;
}
public void setEmpNum(int num)
{
empNum = num;
}
public void setEmpSal(double sal)
{
empSal = sal;
}
}
```

Employee
-empNum : int -empSal : double
+getEmpNum : int +getEmpSal : double

+setEmpNum(int num) : void +setEmpSal(double sal) : void

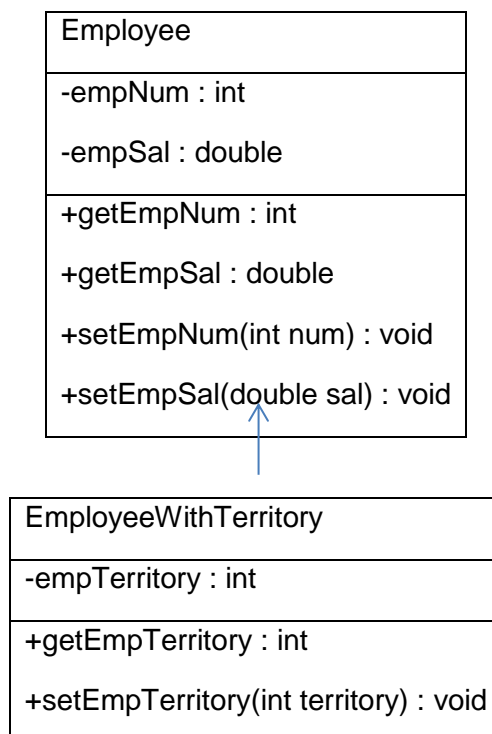
Class Diagram

After you create the Employee class, you can create specific Employee objects, such as the following:

```
Employee receptionist = new Employee();
```

```
Employee deliveryPerson = new Employee();
```

These Employee objects can eventually possess different numbers and salaries, but because they are Employee objects, you know that each Employee has some number and salary. Suppose you hire a new type of Employee named serviceRep, and that a serviceRep object requires not only an employee number and a salary, but also a data field to indicate territory served. You can create a class with a name such as EmployeeWithTerritory, and provide the class three fields (empNum, empSal, and empTerritory) and six methods (get and set methods for each of the three fields). However, when you do this, you are duplicating much of the work that you have already done for the Employee class. The wise, efficient alternative is to create the class EmployeeWithTerritory so it inherits all the attributes and methods of Employee. Then, you can add just the one field and two methods that are new within EmployeeWithTerritory objects.



Class diagram showing the relationship between Employee and EmployeeWithTerritory

When you use inheritance to create the EmployeeWithTerritory class, you:

- Save time because the Employee fields and methods already exist
- Reduce errors because the Employee methods already have been used and tested
- Reduce the amount of new learning required to use the new class, because you have used the Employee methods on simpler objects and already understand how they work

The ability to use inheritance in Java makes programs easier to write, less error prone, and more quickly understood. Besides creating `EmployeeWithTerritory`, you also can create several other specific Employee classes (perhaps `EmployeeEarningCommission`, including a commission rate, or `DismissedEmployee`, including a reason for dismissal). By using inheritance, you can develop each new class correctly and more quickly. The concept of inheritance is useful because it makes class code reusable. Each method already written and tested in the original class becomes part of the new class that inherits it.

Inheritance Terminology

A class that is used as a basis for inheritance, such as `Employee`, is a base class. When you create a class that inherits from a base class (such as `EmployeeWithTerritory`), it is a derived class. When considering two classes that inherit from each other, you can tell which is the base class and which is the derived class by using the two classes in a sentence with the phrase “is a(n).” A derived class always “is a” case or example of the more general base class. For example, a `Tree` class can be a base class to an `Evergreen` class. An `Evergreen` “is a” `Tree`, so `Tree` is the base class; however, it is not true for all `Trees` that “a `Tree` is an `Evergreen`.” Similarly, an `EmployeeWithTerritory` “is an” `Employee` but not the other way around so `Employee` is the base class.

You can use the terms superclass and subclass as synonyms for base class and derived class, respectively. Thus, `Evergreen` can be called a subclass of the `Tree` superclass. You can also use the terms parent class and child class. An `EmployeeWithTerritory` is a child to the `Employee` parent. Use the pair of terms with which you are most comfortable; these terms are used interchangeably throughout this book.

As an alternative way to discover which of two classes is the base class or subclass, you can try saying the two class names together. When people say their names together, they state the more specific name before the all-encompassing family name, as in “Ginny Kroening.” Similarly, with classes, the order that “makes more sense” is the child-parent order. “`Evergreen Tree`” makes more sense than “`Tree Evergreen`,” so `Evergreen` is the child class. Finally, you usually can distinguish super classes from their subclasses by size. Although it is not required, in general a subclass is larger than a superclass because it usually has additional fields and methods. A subclass description might look small, but any subclass contains all the fields and methods of its superclass, as well as the new, more specific fields and methods you add to that subclass.

2.3 Extending Classes

You use the keyword `extends` to achieve inheritance in Java. For example, the following class header creates a superclass-subclass relationship between `Employee` and `EmployeeWithTerritory`:

```
public class EmployeeWithTerritory extends Employee
```

Each `EmployeeWithTerritory` automatically receives the data fields and methods of the superclass `Employee`; you then add new fields and methods to the newly created subclass.

```
public class EmployeeWithTerritory extends Employee
```

```
{
```

```
private int empTerritory;
```

```
public int getEmpTerritory()
```

```
{
```

```
return empTerritory;
```

```
}
```

```
public void setEmpTerritory(int num)
```

```
{
```

```
empTerritory = num;
```

```
}
```

```
}
```

You can write a statement that instantiates a derived class object, such as the following:

```
EmployeeWithTerritory northernRep = new EmployeeWithTerritory();
```

Then you can use any of the next statements to get field values for the `northernRep` object:

```
northernRep.getEmpNum();
```

```
northernRep.getEmpSal();
```

```
northernRep.getEmpTerritory();
```

The `northernRep` object has access to all three get methods two methods that it inherits from `Employee` and one method that belongs to `EmployeeWithTerritory`. Similarly, after the `northernRep` object is declared, any of the following statements are legal:

```
northernRep.setEmpNum(915);
```

```
northernRep.setEmpSal(210.00);
```

```
northernRep.setEmpTerritory(5);
```

The northernRep object has access to all the parent Employee class set methods, as well as its own class's new set method. Inheritance is a one-way proposition; a child inherits from a parent, not the other way around. When you instantiate an Employee object, it does not have access to the EmployeeWithTerritory methods. It makes sense that a parent class object does not have access to its child's data and methods. When you create the parent class, you do not know how many future subclasses it might have or what their data or methods might look like.

In addition, subclasses are more specific than the superclass they extend. An Orthodontist class and Periodontist class are children of the Dentist parent class. You do not expect all members of the general parent class Dentist to have the Orthodontist's applyBraces() method or the Periodontist's deepClean() method. However, Orthodontist objects and Periodontist objects have access to the more general Dentist methods conductExam() and billPatients().

You can use the instanceof operator to determine whether an object is a member or descendant of a class. For example, if northernRep is an EmployeeWithTerritory object, then the value of each of the following expressions is true:

```
northernRep instanceof EmployeeWithTerritory
```

```
northernRep instanceof Employee
```

If aClerk is an Employee object, then the following is true:

```
aClerk instanceof Employee
```

```
aClerk instanceof EmployeeWithTerritory
```

Programmers say that instanceof yields true if the operand on the left can be upcast to the operand on the right.

Overriding Superclass Methods

When you create a subclass by extending an existing class, the new subclass contains data and methods that were defined in the original superclass. In other words, any child class object has all the attributes of its parent. Sometimes, however, the superclass data fields and methods are not entirely appropriate for the subclass objects; in these cases, you want to override the parent class members. When you use the English language, you often use the same method name to indicate diverse meanings. For example, if you think of MusicalInstrument as a class, you can think of play() as a method of that class. If you think of various subclasses such as Guitar and Drum, you know that you carry out the play() method quite differently for each subclass.

Using the same method name to indicate different implementations is called polymorphism, a term meaning "many forms" many different forms of action take place, even though you use the same word to describe the action. In other words, many forms of the same word exist, depending on the object associated with the word. For example, suppose you create an Employee superclass containing data fields such as firstName, lastName,

socialSecurityNumber, dateOfHire, rateOfPay, and so on, and the methods contained in the Employee class include the usual collection of get and set methods.

If your usual time period for payment to each Employee object is weekly, your displayRateOfPay() method might include a statement such as:

```
System.out.println("Pay is " + rateOfPay + " per week ");
```

Imagine your company has a few Employees who are not paid weekly. Maybe some are paid by the hour, and others are Employees whose work is contracted on a job-to-job basis. Because each Employee type requires different paycheck-calculating procedures, you might want to create subclasses of Employee, such as HourlyEmployee and ContractEmployee.

When you call the displayRateOfPay() method for an HourlyEmployee object, you want the display to include the phrase “per hour”, as in “Pay is \$8.75 per hour.” When you call the displayRateOfPay() method for a ContractEmployee, you want to include “per contract”, as in “Pay is \$2000 per contract.” Each class the Employee superclass and the two subclasses requires its own displayRateOfPay() method. Fortunately, if you create separate displayRateOfPay() methods for each class, the objects of each class use the appropriate method for that class. When you create a method in a child class that has the same name and parameter list as a method in its parent class, you override the method in the parent class. When you use the method name with a child object, the child’s version of the method is used.

If you could not override superclass methods, you could always create a unique name for each subclass method, such as displayRateOfPayForHourly(), but the classes you create are easier to write and understand if you use one reasonable name for methods that do essentially the same thing. Because you are attempting to display the rate of pay for each object, displayRateOfPay() is a clear and appropriate method name for all the object types.

Object-oriented programmers use the term polymorphism when discussing any operation that has multiple meanings. For example, the plus sign (+) is polymorphic because it operates differently depending on its operands. You can use the plus sign to add integers or doubles, to concatenate strings, or to indicate a positive value. As another example, methods with the same name but different parameter lists are polymorphic because the method call operates differently depending on its arguments. When Java developers refer to polymorphism, they most often mean subtype polymorphism the ability of one method name to work appropriately for different subclass objects of the same parent class.

Calling Constructors During Inheritance

When you create any object, as in the following statement, you are calling a class constructor method that has the same name as the class itself:

```
SomeClass anObject = new SomeClass();
```

When you instantiate an object that is a member of a subclass, you are calling at least two constructors: the constructor for the base class and the constructor for the extended, derived class. When you create any subclass object, the superclass constructor must execute first, and then the subclass constructor executes. When a superclass contains a default constructor and you instantiate a subclass object, the execution of the superclass constructor often is transparent that is, nothing calls attention to the fact that the superclass constructor is

executing. However, you should realize that when you create an object such as the following (where `HourlyEmployee` is a subclass of `Employee`), both the `Employee()` and `HourlyEmployee()` constructors execute.

```
HourlyEmployee clerk = new HourlyEmployee();
```

The class named `ASuperClass` has a constructor that displays a message. The class named `ASubClass` descends from `ASuperClass`, and its constructor displays a different message. The `DemoConstructors` class contains just one statement that instantiates one object of type `ASubClass`.

```
public class ASuperClass
{
public ASuperClass()
{
    System.out.println("In superclass constructor");
}
}

public class ASubClass extends ASuperClass
{
public ASubClass()
{
    System.out.println("In subclass constructor");
}
}

public class DemoConstructors
{
    public static void main(String[] args)
    {
        ASubClass child = new ASubClass();
    }
}
```

Three classes that demonstrate constructor calling when a subclass object is instantiated

Using Superclass Constructors That Require Arguments

When you create a class and do not provide a constructor, Java automatically supplies you with a default constructor—one that never requires arguments. When you write your own constructor, you replace the automatically supplied version. Depending on your needs, a constructor you create for a class might be a default constructor or might require arguments. When you use a class as a superclass and the class has only constructors that require arguments, you must be certain that any subclasses provide the superclass constructor with the arguments it needs.

When a superclass has a default constructor, you can create a subclass with or without its own constructor. This is true whether the default superclass constructor is the automatically supplied one or one you have written. However, when a superclass contains only constructors that require arguments, you must include at least one constructor for each subclass you create. Your subclass constructors can contain any number of statements, but if all superclass constructors require arguments, then the first statement within each subclass constructor must call one of the superclass constructors. When a superclass requires constructor arguments upon object instantiation, even if you have no other reason to create a subclass constructor, you must write the subclass constructor so it can call its superclass's constructor. If a superclass has multiple constructors but one is a default constructor, you do not have to create a subclass constructor unless you want to. If the subclass contains no constructor, all subclass objects use the superclass default constructor when they are instantiated.

The format of the statement that calls a superclass constructor from the subclass constructor is:

super(list of arguments);

The keyword `super` always refers to the superclass of the class in which you use it. If a superclass contains only constructors that require arguments, you must create a subclass constructor, but the subclass constructor does not necessarily have to have parameters of its own. For example, suppose that you create an `Employee` class with a constructor that requires three arguments: a character, a double, and an integer and you create an `HourlyEmployee` class that is a subclass of `Employee`. The following code shows a valid constructor for

`HourlyEmployee`:

```
public HourlyEmployee()
{
    super('P', 12.35, 40);
    // Other statements can go here
}
```

This version of the `HourlyEmployee` constructor requires no arguments, but it passes three constant arguments to its superclass constructor. A different, overloaded version of the `HourlyEmployee` constructor can require arguments. It could then pass the appropriate arguments to the superclass constructor.

For example:

```
public HourlyEmployee(char dept, double rate, int hours)
{
super(dept, rate, hours);
// Other statements can go here
}
```

Except for any comments, the `super()` statement must be the first statement in any subclass constructor that uses it. Not even data field definitions can precede it. Although it seems that you should be able to use the superclass constructor name to call the superclass constructor—for example, `Employee()`—Java does not allow this. You must use the keyword `super`.

Accessing Superclass Methods

Earlier in this chapter, you learned that a subclass can contain a method with the same name and arguments (the same signature) as a method in its parent class. When this happens, using the subclass method overrides the superclass method. However, instead of overriding the superclass method, you might want to use it within a subclass. If so, you can use the keyword `super` to access the parent class method.

For example, examine the `Customer` class in the example and the `PreferredCustomer` class in the example. A `Customer` has an `idNumber` and `balanceOwed`. In addition to these fields, a `PreferredCustomer` receives a `discountRate`. In the `PreferredCustomer display()` method, you want to display all three fields—`idNumber`, `balanceOwed`, and `discountRate`. Because two-thirds of the code to accomplish the display has already been written for the `Customer` class, it is convenient to have the `PreferredCustomer display()` method use its parent's version of the `display()` method before displaying its own discount rate.

```
public class Customer
{
private int idNumber;
private double balanceOwed;
public Customer(int id, double bal)
{
    idNumber = id;
    balanceOwed = bal;
}.
public void display()
```

```

{
System.out.println("Customer #" + idNumber + " Balance $" + balanceOwed);
}
}

```

The Customer class

```

public class PreferredCustomer extends Customer
{
double discountRate;
public PreferredCustomer(int id, double bal, double rate)
{
super(id, bal);
discountRate = rate;
}
public void display()
{
super.display();
System.out.println(" Discount rate is " + discountRate);
}
}

```

The PreferredCustomer class

```

public class TestCustomers
{
public static void main(String[] args)
{
Customer oneCust = new Customer(124, 123.45);
PreferredCustomer onePCust = new
PreferredCustomer(125, 3456.78, 0.15);
oneCust.display();
onePCust.display();
}
}

```

Comparing this and super

Within a subclass, you can think of the keyword `this` as the opposite of `super`. For example, if a subclass has overridden a superclass method named `someMethod()`, then within the subclass, `super.someMethod()` refers to the superclass version of the method, and both `someMethod()` and `this.someMethod()` refer to the subclass version. On the other hand, when a parent class contains a method that is not overridden within its child, the child can use the method name with `super` (because the method is a member of the superclass), with `this` (because the method is a member of the subclass by virtue of inheritance), or alone (again, because the method is a member of the subclass).

Methods You Cannot Override

Sometimes when you create a class, you might choose not to allow subclasses to override some of the superclass methods. For example, an `Employee` class might contain a method that calculates each `Employee`'s ID number based on specific `Employee` attributes, and you might not want any derived classes to be able to alter this method. As another example, perhaps a class contains a statement that displays legal restrictions to using the class. You might decide that no derived class should be able to alter the statement that is displayed.

The three types of methods that you cannot override in a subclass are:

- static methods
- final methods
- Methods within final classes

A Subclass Cannot Override static Methods in Its Superclass

A subclass cannot override methods that are declared static in the superclass. In other words, a subclass cannot override a class method a method you use without instantiating an object. A subclass can hide a static method in the superclass by declaring a static method in the subclass with the same signature as the static method in the superclass; then, you can call the new static method from within the subclass or in another class by using a subclass object. However, this static method that hides the superclass static method cannot access the parent method using the `super` object.

The example shows a `BaseballPlayer` class that contains a single static method named `showOrigins()`. The example shows a `ProfessionalBaseballPlayer` class that extends the `BaseballPlayer` class to provide a salary. Within the `ProfessionalBaseballPlayer` class, an attempt is made to override the `showOrigins()` method to display the general Abner Doubleday message about baseball as well as the more specific message about professional baseball. However, the compiler returns the error message.

```
public class BaseballPlayer
```

```
{
```

```

private int jerseyNumber; private double battingAvg;

public static void showOrigins()
{
    System.out.println("Abner Doubleday is often " + "credited with inventing baseball");
}
}

public class ProfessionalBaseballPlayer extends BaseballPlayer
{
    double salary;

    public void showOrigins() // You cannot override a static member of a parent class
    {
        super.showOrigins();
        System.out.println("The first professional " +
            "major league baseball game was played in 1871");
    }
}

```

The example below shows a second version of the ProfessionalBaseballPlayer class. In this version, the showOrigins() method has been changed to static the example shows the error message that appears, proving that the parent class method is not overridden.

```

public class ProfessionalBaseballPlayer extends BaseballPlayer
{
    double salary;

    public static void showOrigins()// You cannot override a static member of a parent class
    {
        super.showOrigins();
        System.out.println("The first professional " + "major league baseball game was played
in 1871");
    }
}

```

Finally, the example shows a ProfessionalBaseballPlayer class that compiles without error. Its showOrigins() method is static. Because this method has the same name as the parent class method, when you use the name with a child class object, this method hides the original.

However, it does not override the original, or the super call in the version of the method in the example we would have compiled without error. If you want the ProfessionalBaseballPlayer class to display information about baseball in general as well as professional baseball, you can do either of the following:

- You can display both messages from within a child class method with println() statements.
- You can use the parent class name, a dot, and the method name. Although a child class cannot inherit its parent's static

public class ProfessionalBaseballPlayer **extends** BaseballPlayer

```
{
    double salary;
    public static void showOrigins()
    {
        BaseballPlayer.showOrigins();
        System.out.println("The first professional " + "major league baseball game was played in
1871");
    }
}
```

The ProfessionalBaseballPlayer class

public class TestProPlayer

```
{
    public static void main(String[] args)
    {
        ProfessionalBaseballPlayer aYankee = new ProfessionalBaseballPlayer();
        aYankee.showOrigins();
    }
}
```

The TestProPlayer class

A Subclass Cannot Override final Methods in Its Superclass

A subclass cannot override methods that are declared final in the superclass. For example, consider the `BasketballPlayer` and `ProfessionalBasketballPlayer` classes. When you attempt to compile the `ProfessionalBasketballPlayer` class, you receive the error message in shown, because the class cannot override the final `displayMessage()` method in the parent class.

```
public class BasketballPlayer
{
    private int jerseyNumber;

    public final void displayMessage()
    {
        System.out.println("Michael Jordan is the " + "greatest basketball player - and that is final");
    }
}
```

The BasketballPlayer class

```
public class ProfessionalBasketballPlayer extends BasketballPlayer
{
    double salary;

    public void displayMessage() //A child class method cannot override a parent final method
    {
        System.out.println("I have nothing to say");
    }
}
```

The ProfessionalBasketballPlayer class that attempts to override a final method

You learnt that you can use the keyword `final` when you want to create a constant, as in `final double TAXRATE = 0.065`; You can also use the `final` modifier with methods when you don't want the method to be overridden that is, when you want every child class to use the original parent class version of a method.

In Java, all instance method calls are virtual method calls by default that is, the method used is determined when the program runs because the type of the object used might not be known until the method executes. For example, with the following method you can pass in a `BasketballPlayer` object, or any object that is a child of `BasketballPlayer`, so the "actual" type of the argument `bbplayer`, and which version of `displayMessage()` to use, is not known until the method executes.

```
public void display(BasketballPlayer bbplayer)
{
bbplayer.displayMessage();
}
```

In other words, the version of the method used is not determined when the program is compiled; it is determined when the method call is made. Determining the correct method takes a small amount of time. An advantage to making a method final is that the compiler knows there will be only one version of the method the parent class version. Therefore, the compiler does know which method version will be used the only version and the program is more efficient. Because a final met the compiler can optimize a program's performance by removing the calls to final methods and replacing them with the expanded code of their definitions at each method call location. This process is called inlining the code. When a program executes, you are never aware that inlining is taking place; the compiler chooses to use this procedure to save the overhead of calling a method, and the program runs faster. The compiler chooses to inline a final method only if it is a small method that contains just one or two lines of code.

2.4 Summary

Inheritance is a mechanism that enables one class to inherit both the behavior and the attributes of another class. The keyword *extends* is used to achieve inheritance in Java. Polymorphism is the act of using the same method name to indicate different implementations. To use a superclass method within a subclass, you use the keyword *super* to access it. Information hiding is the concept of keeping data private. The keyword *protected* provides an intermediate level of security between public and private access. A subclass cannot override methods that are: Declared static in a superclass and Declared final or declared within a final class.



Self-Evaluation Questions

1. Should variables that contain numbers always be declared as integer or floating-point data types? Why or why not? Name potential examples.
 2. Write a Java program that asks your name and prints the following message:
Hello, yourName!
 3. Write a Java program that asks for three integers and prints the average value as a double.
Example $(3 + 6 + 7) / 3 = 5.33333333333333$
-

STUDY UNIT 3 ADVANCED INHERITANCE CONCEPTS

3.1 Introduction

Study unit 3 introduces students to more advanced inheritance topics. Students will learn to create abstract classes. Dynamic method binding enables programmers to declare an object using the name of a superclass and then instantiate a subclass. A superclass can also be used as a method argument or as an array type. Students will also learn about the Object class and how to use the toString() and equals() methods. And Finally, learners will learn to create interfaces and packages.

On completion of this course the students should be able to:

- Create and use abstract classes
- Use dynamic method binding
- Create arrays of subclass objects
- Use the Object class and its methods
- Use inheritance to achieve good software design
- Create and use interfaces
- Create and use packages

3.2 Creating and Using Abstract Classes

Developing new classes is easier after you understand the concept of inheritance. When you use a class as a basis from which to create extended child classes, the child classes are more specific than their parent. When you create a child class, it inherits all the general attributes you need; thus, you must create only the new, more specific attributes. For example, a SalariedEmployee and an HourlyEmployee are more specific than an Employee. They can inherit general Employee attributes, such as an employee number, but they add specific attributes, such as pay-calculating methods.

Notice that a superclass contains the features that are shared by its subclasses. For example, the attributes of the Dog class are shared by every Poodle and Spaniel. The subclasses are more specific examples of the superclass type; they add more features to the shared, general features. Conversely, when you examine a subclass, you see that its parent is more general and less specific; for example, Animal is more general than Dog.

A concrete class is one from which you can instantiate objects. Sometimes, a class is so general that you never intend to create any specific instances of the class. For example, you might never create an object that is “just” an Employee; each Employee is more specifically a SalariedEmployee, HourlyEmployee, or ContractEmployee. A class such as Employee that you create only to extend from is not a concrete class; it is an abstract class. In the last chapter, you learned that you can create final classes if you do not want other classes to be able to extend them. Classes that you declare to be abstract are the opposite; your only purpose in creating them is to enable other classes to extend them. If you attempt to instantiate an object

from an abstract class, you receive an error message from the compiler that you have committed an `InstantiationError`. You use the keyword `abstract` when you declare an abstract class. (In other programming languages, such as C++, abstract classes are known as virtual classes.)

3.3 Programmers of an abstract class can include two method types

- Non-abstract methods, like those you can create in any class, are implemented in the abstract class and are simply inherited by its children.
- Abstract methods have no body and must be implemented in child classes.

Abstract classes usually contain at least one abstract method. When you create an abstract method, you provide the keyword `abstract` and the rest of the method header, including the method type, name, and parameters. However, the declaration ends there: you do not provide curly braces or any statements within the method just a semicolon at the end of the declaration. If you create an empty method within an abstract class, the method is an abstract method even if you do not explicitly use the keyword `abstract` when defining the method, but programmers often include the keyword for clarity. If you declare a class to be abstract, its methods can be abstract or not, but if you declare a method to be abstract, you must also declare its class to be abstract.

When you create a subclass that inherits an abstract method, you write a method with the same signature. You are required to code a subclass method to override every empty, abstract superclass method that is inherited. Either the child class method must itself be abstract, or you must provide a body, or implementation, for the inherited method. Suppose you want to create classes to represent different animals, such as `Dog` and `Cow`. You can create a generic abstract class named `Animal` so you can provide generic data fields, such as the animal's name, only once. An `Animal` is generic, but all specific `Animals` make a sound; the actual sound differs from `Animal` to `Animal`. If you code an empty `speak()` method in the abstract `Animal` class, you require all future `Animal` subclasses to code a `speak()` method that is specific to the subclass. The example shows an abstract `Animal` class containing a data field for the name, `getName()` and `setName()` methods, and an abstract `speak()` method.

```
public abstract class Animal
{
    private String name;
    public abstract void speak();
    public String getName()
    {
        return name;
    }
    public void setName(String animalName)
    {
```

```

name = animalName;
}
}

```

The Animal class in the example is declared as abstract; the keyword is shaded. You cannot create a class in which you declare an Animal object with a statement such as `Animal myPet = new Animal("Murphy");`, because a class that attempts to instantiate an Animal object does not compile. Animal is an abstract class, so no Animal objects can exist. You create an abstract class such as Animal only so you can extend it.

For example, because a dog is an animal, you can create a Dog class as a child class of Animal.

Class Dog extends Animal.

```

{
    public void speak()
    {
        System.out.println("Woof!");
    }
}

```

The abstract, parent Animal class contains an abstract `speak()` method. You can code any statements you want within the Dog `speak()` method, but the `speak()` method must exist. If you do not want to create Dog objects but want the Dog class to be a parent to further subclasses, then the Dog class must also be abstract. In that case, you can write code for the `speak()` method within the subclasses of Dog.

If Animal is an abstract class, you cannot instantiate an Animal object; however, if Dog is a concrete class, instantiating a Dog object is perfectly legal. When you code the following, you create a Dog object:

```
Dog myPet = new Dog("Murphy");
```

Then, when you code `myPet.speak();`, the correct Dog `speak()` method executes. The classes in examples below also inherit from the Animal class and implement `speak()`

public class Cow extends Animal

```

{
    public void speak()
    {

```

```

System.out.println("Moo!");
    }
}

public class Snake extends Animal
{
    public void speak()
    {
        System.out.println("Ssss!");
    }
}

public class UseAnimals
{
    public static void main(String[] args)
    {
        Dog myDog = new Dog();
        Cow myCow = new Cow();
        Snake mySnake = new Snake();
        myDog.setName("My dog Murphy");
        myCow.setName("My cow Elsie");
        mySnake.setName("My snake Sammy");
        System.out.print(myDog.getName() + " says ");
        myDog.speak();
        System.out.print(myCow.getName() + " says ");
        myCow.speak();
        System.out.print(mySnake.getName() + " says ");
        mySnake.speak();
    }
}

```

3.4 Using Dynamic Method Binding

When you create a superclass and one or more subclasses, each object of each subclass “is a” superclass object. Every SalariedEmployee “is an” Employee; every Dog “is an” Animal.

(The opposite is not true. Superclass objects are not members of any of their subclasses. An Employee is not a SalariedEmployee. An Animal is not a Dog.) Because every subclass object “is a” superclass member, you can convert subclass objects to superclass objects. As you are aware, when a superclass is abstract, you cannot instantiate objects of the superclass; however, you can indirectly create a reference to a superclass abstract object. A reference is not an object, but it points to a memory address. When you create a reference, you do not use the keyword `new` to create a concrete object; instead, you create a variable name in which you can hold the memory address of a concrete object. So, although a reference to an abstract superclass object is not concrete, you can store a concrete subclass object reference there.

3.5 Using a Superclass as a Method Parameter Type

Dynamic method binding is most useful when you want to create a method that has one or more parameters that might be one of several types. The `talkingAnimal()` method can be used in programs that contain Dog objects, Cow objects, or objects of any other class that descends from Animal. The application in this example passes first a Dog and then a Cow to the method.

```
public class TalkingAnimalDemo
{
    public static void main(String[] args)
    {
        Dog dog = new Dog();
        Cow cow = new Cow();
        dog.setName("Ginger");
        cow.setName("Molly");
        talkingAnimal(dog);
        talkingAnimal(cow);
    }

    public static void talkingAnimal(Animal animal)
    {
        System.out.println("Come one. Come all.");
        System.out.println("See the amazing talking animal!");
        System.out.println(animal.getName() + " says");
        animal.speak();
        System.out.println("*****");
    }
}
```

```
}
```

3.6 Creating Arrays of Subclass Objects

Recall that every array element must be the same data type, which can be a primitive, built-in type or a type based on a more complex class. When you create an array in Java, you are not constructing objects. Instead, you are creating space for references to objects. In other words, although it is convenient to refer to “an array of objects,” every array of objects is really an array of object references. When you create an array of superclass references, it can hold subclass references. This is true whether the superclass in question is abstract or concrete.

For example, even though `Employee` is an abstract class, and every `Employee` object is either a `SalariedEmployee` or an `HourlyEmployee` subclass object, it can be convenient to create an array of generic `Employee` references. Likewise, an `Animal` array might contain individual elements that are `Dog`, `Cow`, or `Snake` objects. As long as every `Employee` subclass has access to a `calculatePay()` method, and every `Animal` subclass has access to a `speak()` method, you can manipulate an array of superclass objects and invoke the appropriate method for each subclass member.

The following statement creates an array of three `Animal` references:

```
Animal[] animalRef = new Animal[3];
```

The statement reserves enough computer memory for three `Animal` objects named `animalRef[0]`, `animalRef[1]`, and `animalRef[2]`. The statement does not actually instantiate `Animals`; `Animal` is an abstract class and cannot be instantiated. The `Animal` array declaration simply reserves memory for three object references. If you instantiate objects from `Animal` subclasses, you can place references to those objects in the `Animal` array. The array of three references is used to access each appropriate `speak()` method.

```
public class AnimalArrayDemo
{
    public static void main(String[] args)
    {
        Animal[] animalRef = new Animal[3];
        animalRef[0] = new Dog();
        animalRef[1] = new Cow();
        animalRef[2] = new Snake();
        for(int x = 0; x < 3; ++x)
```

```
animalRef[x].speak();  
  
}  
  
}
```

In the `AnimalArrayDemo` application, a reference to an instance of the `Dog` class is assigned to the first `Animal` reference, and then references to `Cow` and `Snake` objects are assigned to the second and third array elements. After the object references are in the array, you can manipulate them like any other array elements. The application uses a `for` loop and a subscript to get each individual reference to `speak()`.

3.7 Using the Object Class and Its Methods

Every class in Java is a subclass, except one. When you define a class, if you do not explicitly extend another class, your class implicitly is an extension of the `Object` class. The `Object` class is defined in the `java.lang` package, which is imported automatically every time you write a program; in other words, the following two class declarations have identical outcomes:

```
public class Animal  
{  
  
}  
  
public class Animal extends Object  
{  
  
}
```

When you declare a class that does not extend any other class, you always are extending the `Object` class. The `Object` class includes methods that descendant classes can use or override as you see fit.

3.8 Using the toString() Method

The `Object` class `toString()` method converts an `Object` into a `String` that contains information about the `Object`. Within a class, if you do not create a `toString()` method that overrides the version in the `Object` class, you can use the superclass version of the `toString()` method. Notice that it does not contain a `toString()` method and that it extends the

`Animal` class.

```
public abstract class Animal  
{  
  
private String name;  
  
public abstract void speak();  
  
public String getName()
```

```

    {
        return name;
    }

    public void setName(String animalName)
    {
        name = animalName;
    }
}

public class Dog extends Animal
{
    public void speak()
    {
        System.out.println("Woof!");
    }
}

public class DisplayDog
{
    public static void main(String[] args)
    {
        Dog myDog = new Dog();
        String dogString = myDog.toString();
        System.out.println(dogString);
    }
}

```

3.9 Using the equals() Method

The Object class also contains an equals() method that takes a single argument, which must be the same type as the type of the invoking object, as in the following example:

```

if(someObject.equals(someOtherObjectOfTheSameType))
    System.out.println("The objects are equal ");

```


The `Object` class `equals()` method returns a boolean value indicating whether the objects are equal. This `equals()` method considers two objects of the same class to be equal only if they have the same hash code; in other words, they are equal only if one is a reference to the other. For example, two `BankAccount` objects named `myAccount` and `yourAccount` are not automatically equal, even if they have the same account numbers and balances; they are equal only if they have the same memory address. If you want to consider two objects to be equal only when one is a reference to the other, you can use the built-in `Object` class `equals()` method. However, if you want to consider objects to be equal based on their contents, you must write your own `equals()` method for your classes.

Even though the two `BankAccount` objects have the same account numbers and balances, the `BankAccounts` are not considered equal because they do not have the same memory address.

```
public class CompareAccounts
{
    public static void main(String[] args)
    {
        BankAccount acct1 = new BankAccount(1234, 500.00);
        BankAccount acct2 = new BankAccount(1234, 500.00);

        if(acct1.equals(acct2))
            System.out.println("Accounts are equal");
        else
            System.out.println("Accounts are not equal");
    }
}
```

3.10 Using Inheritance to Achieve Good Software Design

When an automobile company designs a new car model, the company does not build every component of the new car from scratch. The company might design a new feature completely from scratch; for example, at some point someone designed the first air bag. However, many of a new car's features are simply modifications of existing features. The manufacturer might create a larger gas tank or more comfortable seats, but even these new features still possess many properties of their predecessors in the older models. Most features of new car models are not even modified; instead, existing components, such as air filters and windshield wipers, are included on the new model without any changes.

Similarly, you can create powerful computer programs more easily if many of their components are used either “as is” or with slight modifications. Inheritance does not give you the ability to write programs that you could not write otherwise. If Java did not allow you to extend classes, you could create every part of a program from scratch. Inheritance simply makes your job easier. Professional programmers constantly create new class libraries for use with Java programs. Having these classes available makes programming large systems more manageable. You have already used many “as is” classes, such as System and String. In these cases, your programs were easier to write than if you had to write these classes yourself. Now that you have learned about inheritance, you have gained the ability to modify existing classes.

When you create a useful, extendable superclass, you and other future programmers gain several advantages:

- Subclass creators save development time because much of the code needed for the class has already been written.
- Subclass creators save testing time because the superclass code has already been tested and probably used in a variety of situations. In other words, the superclass code is reliable.
- Programmers who create or use new subclasses already understand how the superclass works, so the time it takes to learn the new class features is reduced.
- When you create a new subclass in Java, neither the superclass source code nor the superclass bytecode is changed. The superclass maintains its integrity.
- When you consider classes, you must think about the commonalities among them; then you can create superclasses from which to inherit. You might be rewarded professionally when you see your own superclasses extended by others in the future.

Creating and Using Interfaces

Some object-oriented programming languages, such as C++, allow a subclass to inherit from more than one parent class. For example, you might create an InsuredItem class that contains data fields pertaining to each possession for which you have insurance. Data fields might include the name of the item, its value, the insurance policy type, and so on. You might also create an Automobile class that contains data fields such as vehicle identification number, make, model, and year. When you create an InsuredAutomobile class for a car rental agency, you might want to include InsuredItem information and methods, as well as Automobile information and methods. It would be convenient to inherit from both the InsuredItem and Automobile classes.

The capability to inherit from more than one class is called multiple inheritance. Many programmers consider multiple inheritance to be a difficult concept, and when inexperienced programmers use it they encounter many problems. Programmers have to deal with the possibility that variables and methods in the parent classes might have identical names, which

creates conflict when the child class uses one of the names. Also, you have already learned that a child class constructor must call its parent class constructor. When there are two or more parents, this task becomes more complicated—to which class should `super()` refer when a child class has multiple parents? For all of these reasons, multiple inheritance is prohibited in Java.

Java, however, does provide an alternative to multiple inheritance: an interface. An interface looks much like a class, except that all of its methods (if any) are implicitly public and abstract, and all of its data items (if any) are implicitly public, static, and final. An interface is a description of what a class does, but not how it is done; it declares method headers, but not the instructions within those methods. When you create a class that uses an interface, you include the keyword `implements` and the interface name in the class header. This notation requires the class to include an implementation for every method named in the interface. Whereas using `extends` allows a subclass to use nonprivate, nonoverridden members of its parent's class, `implements` requires the subclass to implement its own version of each method.

```
public abstract class Animal
{
    private String name;
    public abstract void speak();
    public String getName()
    {
        return name;
    }
    public void setName(String animalName)
    {
        name = animalName;
    }
}

public class Dog extends Animal
{
    public void speak()
    {
        System.out.println("Woof!");
    }
}
```

```
}
```

For simplicity, this example gives the Worker interface a single method named work(). When any class implements Worker, it must either include a work() method or the new class must be declared abstract, and then its descendants must implement the method.

```
public interface Worker
```

```
{
```

```
public void work();
```

```
}
```

The WorkingDog class extends Dog and implements Worker. A WorkingDog contains a data field that a “regular” Dog does not have: an integer that holds hours of training received. The WorkingDog class also contains get and set methods for this field. Because the WorkingDog class implements the Worker interface, it also must contain a work() method that calls the Dog speak() method, and then produces two more lines of output: a statement about working and the number of training hours.

```
public class WorkingDog extends Dog implements Worker
```

```
{
```

```
private int hoursOfTraining;
```

```
public void setHoursOfTraining(int hrs)
```

```
{
```

```
    hoursOfTraining = hrs;
```

```
}
```

```
public int getHoursOfTraining()
```

```
{
```

```
    return hoursOfTraining;
```

```
}
```

```
public void work()
```

```
{
```

```
speak();
```

```
System.out.println("I am a dog who works");
```

```
System.out.println("I have " + hoursOfTraining + " hours of professional training!");
```

```
}
```

```
}
```

Each object can use the following methods:

- The setName() and getName() methods that WorkingDog inherits from the Animal class
- The speak() method that WorkingDog inherits from the Dog class
- The setHoursOfTraining() and getHoursOfTraining() methods contained within the WorkingDog class
- The work() method that the WorkingDog class was required to contain when it used the phrase implements Worker; the work() method also calls the speak() method contained in the Dog class

```
public class DemoWorkingDogs
{
public static void main(String[] args)
{
    WorkingDog aSheepHerder = new WorkingDog();
    WorkingDog aSeeingEyeDog = new WorkingDog();
    aSheepHerder.setName("Simon, the Border Collie");
    aSeeingEyeDog.setName("Sophie, the German Shepherd");
    aSheepHerder.setHoursOfTraining(40);
    aSeeingEyeDog.setHoursOfTraining(300);
    System.out.println(aSheepHerder.getName() + " says ");
    aSheepHerder.speak();
    aSheepHerder.work();
    System.out.println(); // outputs a blank line for readability
    System.out.println(aSeeingEyeDog.getName() + " says ");
    aSeeingEyeDog.speak();
    aSeeingEyeDog.work();
}
}
```

You can compare abstract classes and interfaces as follows:

- Abstract classes and interfaces are similar in that you cannot instantiate concrete objects from either one.
- Abstract classes differ from interfaces because abstract classes can contain non-abstract methods, but all methods within an interface must be abstract.
- A class can inherit from only one abstract superclass, but it can implement any number of interfaces.

Beginning programmers sometimes find it difficult to decide when to create an abstract superclass and when to create an interface. Remember, you create an abstract class when you want to provide data or methods that subclasses can inherit, but at the same time these subclasses maintain the ability to override the inherited methods. Suppose you create a `CardGame` class to use as a base class for different card games. It contains four methods named `shuffle()`, `deal()`, `displayRules()`, and `keepScore()`. The `shuffle()` method works the same way for every `CardGame`, so you write the statements for `shuffle()` within the superclass, and any `CardGame` objects you create later inherit `shuffle()`.

The methods `deal()`, `displayRules()`, and `keepScore()` operate differently for every subclass (for example, for `TwoPlayerCardGames`, `FourPlayerCardGames`, `BettingCardGames`, and so on), so you force `CardGame` children to contain instructions for those methods by leaving them empty in the superclass. The `CardGame` class, therefore, should be an abstract superclass. When you write classes that extend the `CardGame` parent class, you inherit the `shuffle()` method, and write code within the `deal()`, `displayRules()`, and `keepScore()` methods for each specific child. You create an interface when you know what actions you want to include, but you also want every user to separately define the behavior that must occur when the method executes. Suppose you create a `MusicalInstrument` class to use as a base for different musical instrument object classes such as `Piano`, `Violin`, and `Drum`. The parent `MusicalInstrument` class contains methods such as `playNote()` and `outputSound()` that apply to every instrument, but you want to implement these methods differently for each type of instrument. By making `MusicalInstrument` an interface, you require every nonabstract subclass to code all the methods.

3.11 Creating Interfaces to Store Related Constants

Interfaces can contain data fields, but they must be public, static, and final. It makes sense that interface data must not be private because interface methods cannot contain method bodies; without public method bodies, you have no way to retrieve private data. It also makes sense that the data fields in an interface are static because you cannot create interface objects. Finally, it makes sense that interface data fields are final because, without methods containing bodies, you have no way, other than at declaration, to set the data fields' values, and you have no way to change them. Your purpose in creating an interface containing constants is to provide a set of data that several classes can use without having to re-declare the values

```

public interface PizzaConstants
{
public static final int SMALL_DIAMETER = 12;
public static final int LARGE_DIAMETER = 16;
public static final double TAX_RATE = 0.07;
public static final String COMPANY = "Antonio's Pizzeria";
}

```

//The PizzaConstants interface

```

public class PizzaDemo implements PizzaConstants
{
public static void main(String[] args)
{
double specialPrice = 11.25;
System.out.println("Welcome to " + COMPANY);
System.out.println("We are having a special offer:\na " +
SMALL_DIAMETER + " inch pizza with four toppings\nor a " +
LARGE_DIAMETER +
" inch pizza with one topping\nfor only $" + specialPrice);
System.out.println("With tax, that is only $" +
(specialPrice + specialPrice * TAX_RATE));
}
}

```

3.12 Creating and Using Packages

A package is a named collection of classes; for example, the `java.lang` package contains fundamental classes and is automatically imported into every program you write. You also have created classes into which you explicitly imported optional packages such as `java.util` and `javax.swing`. When you create classes, you can place them in packages so that you or other programmers can easily import your related classes into new programs. Placing classes in packages for other programmers increases the classes' reusability. When you create many classes that inherit from each other, as well as multiple interfaces that you want to implement with these classes, you often will find it convenient to place these related classes in a package.

When you create professional classes for others to use, you most often do not want to provide the users with your source code in the files that have .java extensions. You expend significant effort developing workable code for your programs, and you do not want other programmers to be able to copy your programs, make minor changes, and market the new product themselves. Rather, you want to provide users with the compiled files that have .class extensions. These are the files the user needs to run the program you have developed. Likewise, when other programmers use the classes you have developed, they need only the completed compiled code to import into their programs. The .class files are the files you place in a package so that other programmers can import them.

If you do not specify a package for a class, it is placed in an unnamed default package. A class that will be placed in a non-default package for others to use must be public. If a class is not public, it can be used only by other classes within the same package. To place a class in a package, you include a package declaration at the beginning of the source code file that indicates the folder into which the compiled code will be placed. When a file contains a package declaration, it must be the first statement in the file (excluding comments). If there are import declarations, they follow the package declaration. Within the file, the package statement must appear outside the class definition. The package statement, import statements, and comments are the only statements that appear outside class definitions in Java program files.

For example, the following statement indicates that the compiled file should be placed in a folder named com.course.animals:

```
package com.course.animals;
```

That is, the compiled file should be stored in the animals' subfolder inside the course subfolder inside the com subfolder (or com\course\animals). The pathname can contain as many levels as you want. When you compile a file that you want to place in a package, you can copy or move the compiled .class file to the appropriate folder. Alternatively, you can use a compiler option with the javac command. The -d (for directory) option indicates that you want to place the generated .class file in a folder. For example, the following command indicates that the compiled Animal.java file should be placed in the directory indicated by the import statement within the Animal.java file:

```
javac -d . Animal.java
```

The dot (period) in the compiler command indicates that the path shown in the package statement in the file should be created within the current directory. If the Animal class file contains the statement package com.course.animals;, the Animal.class file is placed in C:\com\course\animals. If any of these subfolders do not exist, Java creates them. Similarly, if you package the compiled files for Dog.java, Cow.java, and so on, future programs need only use the following statements to be able to use all the related classes:

```
import com.course.animals.Dog;
```

```
import com.course.animals.Cow;
```


Because Java is used extensively on the Internet, it is important to give every package a unique name. Sun Microsystems, the creator of Java, has defined a package-naming convention in which you use your Internet domain name in reverse order. For example, if your domain name is course.com, you begin all of your package names with com.course. Subsequently, you organize your packages into reasonable subfolders.

Creating packages using Java's naming convention helps avoid naming conflicts different programmers might create classes with the same name, but they are contained in different packages. Class naming conflicts are sometimes called collisions. Because of packages, you can create a class without worrying that its name already exists in Java or in packages distributed by another organization. For example, if your domain name is course.com, then you might want to create a class named Scanner and place it in a package named com.course.input. The fully qualified name of your Scanner class is com.course.input.Scanner, and the fully qualified name of the built-in Scanner class is java.util.Scanner.

Summary

An abstract class is a class that you create only to extend from, but not to instantiate from and usually contains abstract methods (Methods with no method statements). One can convert subclass objects to superclass objects. Dynamic method binding can create a method that has one or more parameters that might be one of several types and can create an array of superclass object references but store subclass instances in it. An Interface is like a class, all methods are implicitly public and abstract, all its data fields are implicitly public, static, and final. To create a class that uses an interface, include the keyword implements and the interface name in the class header. One must place classes in packages which are convention using Internet domain names in reverse order.



Self-Evaluation Questions

1. What is the difference between reference parameters and output parameters? When should you use each?
 2. What do you think are the advantages of using methods in your programs? Is it possible to write a complete large-scale program without methods?
-

STUDY UNIT 4 EXCEPTION HANDLING

4.1 Introduction

This module covers the important topic of error handling. You will learn to use the exception-handling mechanisms built into the Java language, including the try, catch, finally, and throws keywords. You will learn to catch exceptions, obtain information about the errors, and find out about the different types of exceptions that may occur. And then concludes with a description of the assert statement, which can be used to verify the logical correctness of a program.

On completion of this course the students should be able to:

- Learn about exceptions
- Try code and catch exceptions
- Throw and catch multiple exceptions
- Use the finally block
- Understand the advantages of exception handling
- Specify the exceptions that a method can throw
- Trace exceptions through the call stack
- Create your own Exception classes
- Use an assertion

4.2 Learning About Exceptions

An exception is an unexpected or error condition. The programs you write can generate many types of potential exceptions, such as when you do the following:

- You issue a command to read a file from a disk, but the file does not exist there.
- You attempt to write data to a disk, but the disk is full or unformatted.
- Your program asks for user input, but the user enters invalid data.
- The program attempts to divide a value by 0.
- The program tries to access an array with a subscript that is too large or too small.

These errors are called exceptions because, presumably, they are not usual occurrences; they are “exceptional.” Exception handling is the name for the object-oriented techniques to manage such errors. Unplanned exceptions that occur during a program’s execution are also called runtime exceptions. Java has two basic classes of errors: Error and Exception. Both classes descend from the Throwable class. Like instantiations of all other classes in Java, exceptions originally descend from Object.

The Error class represents more serious errors from which your program usually cannot recover. Usually, you do not use or implement Error objects in your programs. A program cannot recover from Error conditions on its own. The Exception class comprises less serious

errors that represent unusual conditions that arise while a program is running and from which the program can recover. Some examples of Exception class errors include using an invalid array subscript or performing certain illegal arithmetic operations.

Java displays an Exception message when the program code could have prevented an error. The method declares three integers, prompts the user for values for two of them, and calculates the value of the third integer by dividing the first two values.

```
import java.util.Scanner;

public class Division
{
    public static void main(String[] args)
    {
        Scanner input = new Scanner(System.in);

        int numerator, denominator, result;

        System.out.print("Enter numerator >> ");

        numerator = input.nextInt();

        System.out.print("Enter denominator >> ");

        denominator = input.nextInt();

        result = numerator / denominator;

        System.out.println(numerator + " / " + denominator + " = " + result);
    }
}
```

In the first execution, the user enters two usable values and the program executes normally. In the second execution, the user enters 0 as the value for the denominator and an Exception message is displayed. (Java does not allow integer division by 0, but floating-point division by 0 is allowed the result displays as Infinity. You also get some information about the error (“/ by zero”), the method that generated the error (Division.main), and the file and line number for the error (Division.java, line 12). This example shows two more executions of the Division class. In each execution, the user has entered noninteger data for the denominator first a string of characters, and second, a floating-point value. In each case, a different type of Exception occurs.

You can see from either set of error messages that the Exception is an InputMismatchException. The last line of the messages indicates that the problem occurred in line 11 of the Division program, and the second-to-last error message shows that the problem occurred within the call to nextInt().

Because the user did not enter an integer, the nextInt() method failed. The second-to-last message also shows that the error occurred in line 2050 of the nextInt() method, but clearly you do not want to alter the nextInt() method that resides in the Scanner class you either want

to rerun the program and enter an integer, or alter the program so that these errors cannot occur in subsequent executions. trace history list, or more simply, a stack trace. (You might also hear the terms stack backtrace or stack traceback.) The list shows each method that was called as the program ran.

Just because an Exception occurs, you don't necessarily have to deal with it. In the Division class, you can simply let the offending program. However, the program termination is abrupt and unforgiving. When a program divides two numbers (or performs a less trivial task such as balancing a checkbook), the user might be annoyed if the program ends abruptly. However, if the program is used for a mission critical task such as air traffic control or to monitor a patient's vital statistics during surgery, an abrupt conclusion could be disastrous. (The term mission critical refers to any process that is crucial to an organization.)

Object-oriented error-handling techniques provide more elegant and safer solutions for errors. Of course, you can write programs without using exception-handling techniques you have already written many such programs as you have worked through this book. Programmers had to deal with error conditions long before object-oriented methods were conceived. Probably the most common error-handling solution has been to use a decision to avoid an error. For example, you can change the main() method of the Division class to avoid dividing by 0 by adding the decision shown in the bold portion in the example below.

```
import java.util.Scanner;

public class Division2
{
    public static void main(String[] args)
    {
        Scanner input = new Scanner(System.in);
        int numerator, denominator, result;
        System.out.print("Enter numerator >> ");
        numerator = input.nextInt();
        System.out.print("Enter denominator >> ");
        denominator = input.nextInt();
        if(denominator == 0)
            System.out.println("Cannot divide by 0");
        else
        {
            result = numerator / denominator;
            System.out.println(numerator + " / " + denominator + " = " + result);
        }
    }
}
```

```
}  
}
```

The application in example displays a message to the user when 0 is entered for a denominator value, but it is not able to recover when noninteger data such as a string or floating-point value is entered. Object-oriented exception handling provides a more elegant solution for handling error conditions. Programs that can handle exceptions appropriately are said to be more fault tolerant and robust. Fault-tolerant applications are designed so that they continue to operate, possibly at a reduced level, when some part of the system fails. Robustness represents the degree to which a system is resilient to stress, maintaining correct functioning.

Even if you choose never to use exception-handling techniques in your own programs, you must understand them because built-in Java methods will throw exceptions to your programs.

4.3 Trying Code and Catching Exceptions

In object-oriented terminology, you “try” a procedure that might cause an error. A method that detects an error condition or Exception “throws an exception,” and the block of code that processes the error “catches the exception.” When you create a segment of code in which something might go wrong, you place the code in a try block, which is a block of code you attempt to execute while acknowledging that an exception might occur. A try block consists of the following elements:

- The keyword try
- An opening curly brace
- Executable statements, including some that might cause exceptions
- A closing curly brace

You usually code at least one catch block immediately following a try block. A catch block is a segment of code that can handle an exception that might be thrown by the try block that precedes it. The exception might be one that is thrown automatically, or you might explicitly write a throw statement. A throw statement is one that sends an Exception out of a block or a method so it can be handled elsewhere. A thrown Exception can be caught by a catch block. Each catch block can “catch” one type of exception that is, one object that is an object of type Exception or one of its child classes. You create a catch block by typing the following elements:

- The keyword catch
- An opening parenthesis
- An Exception type
- A name for an instance of the Exception type
- A closing parenthesis
- An opening curly brace
- The statements that take the action you want to use to handle the error condition
- A closing curly brace

```

returnType methodName(optional arguments)
{ // optional statements prior to code that is tried
    try
    {
        // statement or statements that might generate an exception
    }
    catch(Exception someException)
    {
        // actions to take if exception occurs
    }
    // optional statements that occur after try,
// whether catch block executes or not
}

```

4.4 Throwing and Catching Multiple Exceptions

You can place as many statements as you need within a try block, and you can catch as many Exceptions as you want. If you try more than one statement, only the first error-generating statement throws an Exception. As soon as the Exception occurs, the logic transfers to the catch block, which leaves the rest of the statements in the try block unexecuted. When a program contains multiple catch blocks, they are examined in sequence until a match is found for the type of Exception that occurred. Then, the matching catch block executes, and each remaining catch block is bypassed.

The main() method in the DivisionMistakeCaught3 class throws two types of Exceptions: an ArithmeticException and an InputMismatchException. The try block in the application surrounds all the statements in which the exceptions might occur.

```

import java.util.*;

public class DivisionMistakeCaught3
{
    public static void main(String[] args)
    {
        Scanner input = new Scanner(System.in);
        int numerator, denominator, result;

        try
        {
            System.out.print("Enter numerator >> ");

```

```

numerator = input.nextInt();
System.out.print("Enter denominator >> ");
denominator = input.nextInt();
result = numerator / denominator;
System.out.println(numerator + " / " + denominator + " = " + result);
}
catch(ArithmeticException mistake)
{
System.out.println(mistake.getMessage());
}
catch(InputMismatchException mistake)
{
System.out.println("Wrong data type");
}
}
}
}

```

Sometimes, you want to execute the same code no matter which Exception type occurs. For example, within the DivisionMistakeCaught3 application, each of the two catch blocks displays a unique message. Instead, you might want both catch blocks to display the same message. Because `ArithmeticExceptions` and `InputMismatchExceptions` are both subclasses of `Exception`, you can rewrite the program as shown in the example, using a single generic catch block (shaded) that can catch any type of `Exception`.

```

import java.util.*;

public class DivisionMistakeCaught4
{
    public static void main(String[] args)
    {
        Scanner input = new Scanner(System.in);
        int numerator, denominator, result;
        try
        {
            System.out.print("Enter numerator >> ");
            numerator = input.nextInt();

```

```

System.out.print("Enter denominator >> ");
denominator = input.nextInt();
result = numerator / denominator;
System.out.println(numerator + " / " + denominator + " = " + result);
}
catch(Exception mistake)
{
    System.out.println("Operation unsuccessful");
}
}
}

```

The catch block in the example accepts a more generic Exception argument type than that thrown by either of the potentially error-causing try statements, so the generic catch block can act as a “catch-all” block. When either an arithmetic error or incorrect input type error occurs, the thrown exception is “promoted” to an Exception error in the catch block. The example shows several executions of the DivisionMistakeCaught4 application. Notice that no matter which type of mistake occurs during execution, the general “Operation unsuccessful” message is displayed from the generic catch block.

4.5 Using the finally Block

When you have actions, you must perform at the end of a try...catch sequence, you can use a finally block. The code within a finally block executes regardless of whether the preceding try block identifies an Exception. Usually, you use a finally block to perform cleanup tasks that must happen if any Exceptions occurred, and whether or not any Exceptions that occurred were caught.

```

try
{
    // statements to try
}
catch(Exception e)
{
    // actions that occur if exception was thrown
}
finally
{
    // actions that occur whether catch block executed or not
}

```



```
}
```

When the try code works without, control passes to the statements at the end of the method. Also, when the try code fails and throws an Exception, and the Exception is caught, the catch block executes and control again passes to the statements at the end of the method. At first glance, it appears the statements at the end of the method always execute. However, the final set of statements might never execute for at least two reasons:

- An unplanned Exception might occur.
- The try or catch block might contain a `System.exit();` statement.

Any try block might throw an Exception for which you did not provide a catch block. After all, Exceptions occur all the time without your handling them, as one did in the first Division application. In the case of an unhandled Exception, program execution stops immediately, the Exception is sent to the operating system for handling, and the current method is abandoned. Likewise, if the try block contains an `exit()` statement, execution stops immediately.

When you include a finally block, you are assured that the finally statements will execute before the method is abandoned, even if the method concludes prematurely. For example, programmers often use a finally block when the program uses data files that must be closed.

try

```
{  
// Open the file  
// Read the file  
// Place the file data in an array  
// Calculate an average from the data  
// Display the average  
}
```

catch(IOException e)

```
{  
// Issue an error message  
// System exit  
}
```

finally

```
{  
// If the file is open, close it  
}
```

The pseudocode in the example represents an application that opens a file; in Java, if a file does not exist when you open it, an input/output exception, or `IOException`, is thrown and a catch block can handle the error. However, because the application uses an array, an uncaught `IndexOutOfBoundsException` might occur even though the file opened successfully. (Such an Exception occurs, as its name implies, when a subscript is not in the range of valid subscripts for an array.) The `IndexOutOfBoundsException` would not be caught by the existing catch block. Also, because the application calculates an average, it might divide by 0 and an `ArithmeticException` might occur; it also would not be caught. In any of these events, you might want to close the file before proceeding. By using the finally block, you ensure that the file is closed because the code in the finally block executes before control returns to the operating system.

The code in the finally block executes no matter which of the following outcomes of the try block occurs:

- The try ends normally.
- The catch executes.
- An uncaught Exception causes the method to abandon prematurely. An uncaught Exception does not allow the try block to finish, nor does it cause the catch block to execute.

4.6 Understanding the Advantages of Exception Handling

Before the inception of object-oriented programming languages, potential program errors were handled using somewhat confusing, error-prone methods. For example, a traditional, non-object-oriented, procedural program might perform three methods that depend on each other using code that provides error checking like the pseudocode.

```
call methodA()
if methodA() worked
{
    call methodB()
    if methodB() worked
    {
        call methodC()
        if methodC() worked
            everything's okay, so display finalResult
        else
            set errorCode to 'C'
    }
else
```

```

        set errorCode to 'B'
    }
else
    set errorCode to 'A'

```

The pseudocode represents an application in which the logic must pass three tests before `finalResult` can be displayed. The program executes `methodA()`; it then calls `methodB()` only if `methodA()` is successful. Similarly, `methodC()` executes only when `methodA()` and `methodB()` are both successful. When any method fails, the program sets an appropriate `errorCode` to 'A', 'B', or 'C'. (Presumably, the `errorCode` is used later in the application.) The logic is difficult to follow, and the application's purpose and intended usual outcome to display `finalResult` is lost in the maze of if statements. Also, you can easily make coding mistakes within such a program because of the complicated nesting, indenting, and opening and closing of curly braces.

Compare the same program logic using Java's object-oriented, error-handling technique shown in example. Using the try...catch object-oriented technique provides the same results as the traditional method, but the statements of the program that do the "real" work (calling methods A, B, and C and displaying `finalResult`) are placed together, where their logic is easy to follow. The try steps should usually work without generating errors; after all, the errors are "exceptions." It is convenient to see these business-as-usual steps in one location. The unusual, exceptional events are grouped and moved out of the way of the primary action.

```

try
{
    call methodA() and maybe throw an exception
    call methodB() and maybe throw an exception
    call methodC() and maybe throw an exception
    everything's okay, so display finalResult
}
catch(methodA()'s error)
{
    set errorCode to "A"
}
catch(methodB()'s error)
{
    set errorCode to "B"
}

```

```

catch(methodC()'s error)
{
    set errorCode to "C"
}

```

Besides clarity, an advantage to object-oriented exception handling is the flexibility it allows in the handling of error situations. When a method you write throws an Exception, the same method can catch the Exception, although it is not required to do so, and in most object-oriented programs it does not. Often, you don't want a method to handle its own Exception. In many cases, you want the method to check for errors, but you do not want to require a method to handle an error if it finds one. Another advantage to object-oriented exception handling is that you gain the ability to appropriately deal with Exceptions as you decide how to handle them.

When you write a method, it can call another, catch a thrown Exception, and you can decide what you want to do. Just as a police officer has leeway to deal with a speeding driver differently depending on circumstances, programs can react to Exceptions specifically for their current purposes. Methods are flexible partly because they are reusable that is, a well-written method might be used by any number of applications. Each calling application might need to handle a thrown error differently, depending on its purpose. For example, an application that uses a method that divides values might need to terminate if division by 0 occurs. A different program simply might want the user to reenter the data to be used, and a third program might want to force division by 1. The method that contains the division statement can throw the error, but each calling program can assume responsibility for handling the error detected by the method in an appropriate way.

4.7 Specifying the Exceptions, a Method Can Throw

If a method throws an Exception that it will not catch but that will be caught by a different method, you must use the keyword `throws` followed by an Exception type in the method header. This practice is known as exception specification. For example, a `PriceList` class used by a company to hold a list of prices for items it sells. For simplicity, there are only four prices and a single method that displays the price of a single item. The `displayPrice()` method accepts a parameter to use as the array subscript, but because the subscript could be out of bounds, the method contains a shaded `throws` clause, acknowledging it could throw an exception.

```

public class PriceList
{
    private static final double[] price = {15.99, 27.88, 34.56, 45.89};

    public static void displayPrice(int item) throws IndexOutOfBoundsException
    {
        System.out.println("The price is R " + price[item]);
    }
}

```

For most Java methods that you write, you do not use a throws clause. For example, you have not needed to use a throws clause in any of the many programs you have written while working through this book; however, in those methods, if you divided by 0 or went beyond an array's bounds, an Exception was thrown nevertheless. Most of the time, you let Java handle any Exception by shutting down the program. Imagine how unwieldy your programs would become if you were required to provide instructions for handling every possible error, including equipment failures and memory problems. Most exceptions never have to be explicitly thrown or caught, nor do you have to include a throws clause in the headers of methods that automatically throw these Exceptions. The only exceptions that must be caught or named in a throws clause are the type known as checked exceptions. Java's exceptions can be categorized into two types:

- Unchecked exceptions

These exceptions inherit from the Error class or the RuntimeException class. Although you can handle these exceptions in your programs, you are not required to do so. For example, dividing by zero is a type of RuntimeException, and you are not required to handle this exception you can simply let the program terminate.

- Checked exceptions

These exceptions are the type that programmers should anticipate and from which programs should be able to recover. All exceptions that you explicitly throw and that descend from the Exception class are checked exceptions.

Java programmers say that checked exceptions are subject to the catch or specify requirement, which means if you throw a checked exception from a method, you must do one of the following:

- Catch it within the method.
- Specify the exception in your method header's throws clause.

Code that uses a checked exception will not compile if the catch or specify rule is not followed. If you write a method with a throws clause in the header, then any method that uses your method must do one of the following:

- Catch and handle the possible exception.
- Declare the exception in its throws clause. The called method can then rethrow the exception to yet another method that might either catch it or throw it yet again.

In other words, when an exception is a checked exception, client programmers are forced to deal with the possibility that an exception will be thrown. If you write a method that explicitly throws a checked Exception that is not caught within the method, Java requires that you use the throws clause in the header of the method. Using the throws clause does not mean that the method will throw an Exception everything might go smoothly. Instead, it means the method might throw an Exception. You include the throws clause in the method header so applications that use your methods are notified of the potential for an Exception.

To be able to use a method to its full potential, you must know the method's name and three additional pieces of information:

- The method's return type
- The type and number of arguments the method requires
- The type and number of Exceptions the method throws

To use a method, you must know what types of arguments are required. You can call a method without knowing its return type, but if you do, you can't benefit from any value that the method returns. (Also, if you use a method without knowing its return type, you probably don't understand the purpose of the method.) Likewise, you can't make sound method might throw. When a method might throw more than one Exception type, you can specify a list of potential Exceptions in the method header by separating them with commas. As an alternative, if all the exceptions descend from the same parent, you can specify the more general parent class. For example, if your method might throw either an `ArithmeticException` or an `ArrayIndexOutOfBoundsException`, you can just specify that your method throws a `RuntimeException`. One advantage to this technique is that when your method is modified to include more specific `RuntimeException`s in the future, the method header will not change. This saves time and money for users of your methods, who will not have to modify their own methods to accommodate new `RuntimeException` types.

An extreme alternative is simply to specify that your method throws a general Exception, so that all exceptions are included in one clause. Doing this simplifies the exception specification you write. However, using this technique disguises information about the specific types of exceptions that might occur, and such information usually has value to users of your methods.

4.8 Tracing Exceptions Through the Call Stack

When one method calls another, the computer's operating system must keep track of where the method call came from, and program control must return to the calling method when the called method is completed. For example, if `methodA()` calls `methodB()`, the operating system has to "remember" to return to `methodA()` when `methodB()` ends. Likewise, if `methodB()` calls `methodC()`, the computer must "remember" while `methodC()` executes to return to `methodB()` and eventually to `methodA()`. The memory location known as the call stack is where the computer stores the list of method locations to which the system must return. When a method throws an Exception and the method does not catch it, the Exception is thrown to the next method up the call stack, or in other words, to the method that called the offending method. Figure 12-24 shows how the call stack works. If `methodA()` calls `methodB()`, and `methodB()` calls `methodC()`, and `methodC()` throws an Exception, Java first looks for a catch block in `methodC()`. If none exists, Java looks for the same thing in `methodB()`. If `methodB()` does not have a catch block, Java looks to `methodA()`. If `methodA()` cannot catch the Exception, it is thrown to the Java Virtual Machine, which displays a message at the command prompt.

4.9 Creating Your Own Exceptions

Java provides over 40 categories of Exceptions that you can use in your programs. However, Java's creators could not predict every condition that might be an Exception in your applications. For example, you might want to declare an Exception when your bank balance is negative or when an outside party attempts to access your e-mail account. Most organizations have specific rules for exceptional data; for example, an employee number must not exceed three digits, or an hourly salary must not be less than the legal minimum wage. Of course, you can handle these potential error situations with if statements, but Java also allows you to create your own Exceptions.

To create your own throwable Exception, you must extend a subclass of Throwable. Recall that Throwable has two subclasses, Exception and Error, which are used to distinguish between recoverable and nonrecoverable errors. Because you always want to create your own Exceptions for recoverable errors, you should extend your Exceptions from the Exception class. You can extend any existing Exception subclass, such as ArithmeticException or NullPointerException, but usually you want to inherit directly from Exception. When you create an Exception subclass, it's conventional to end its name with Exception.

The Exception class contains four constructors as follows:

- Exception() - Constructs a new exception with null as its detail message
- Exception(String message) - Constructs a new exception with the specified detail message
- Exception(String message, Throwable cause) - Constructs a new exception with the specified detail message and cause
- Exception(Throwable cause) - Constructs a new exception with the specified cause and a detail message of cause.toString(), which typically contains the class and the detail message of cause, or null if the cause argument is null

4.10 Using Assertions

Previously, you learned that you might inadvertently create syntax or logical errors when you write a program. Syntax errors are mistakes using the Java language; they are compile-time errors that prevent a program from compiling and creating an executable file with a .class extension. A program might contain logical errors even though it is free of syntax errors. Some logical errors cause runtime errors, or errors that cause a program to terminate. In this chapter, you learned how to use Exceptions to handle many of these kinds of errors. Some logical errors do not cause a program to terminate, but nevertheless produce incorrect results. For example, if a payroll program should determine gross pay by multiplying hours worked by hourly pay rate, but you inadvertently divide the numbers, no runtime error occurs and no Exception is thrown, but the output is wrong. An assertion is a Java language feature that can help you detect such logic errors and debug a program. You use an assert statement to create an assertion; when you use an assert statement, you state a condition that should be true, and Java throws an AssertionError when it is not.

The syntax of an assert statement is:

Assert booleanExpression : optionalErrorMessage.

The Boolean expression in the assert statement should always be true if the program is working correctly. The optionalErrorMessage is displayed if the booleanExpression is false.

Summary

An Exception is an unexpected or error condition. Exception handling is an Object-oriented technique to manage errors. Basic classes of errors: Error and Exception. Exception-handling code consist of *try block*, *catch block* or *finally block*. Use the throws <name>Exception clause after the method header indicates the type of exception that might be thrown. Call stack is a list of method locations to which the system must return. Java provides over 40 categories of Exceptions. Assertion is used to state a condition that should be true and whether Java throws an AssertionError when it is not.



Self-Evaluation Questions

1. What is the difference between reference parameters and output parameters? When should you use each?
 2. What do you think are the advantages of using methods in your programs? Is it possible to write a complete large-scale program without methods?
-

STUDY UNIT 5 FILE INPUT AND OUTPUT

5.1 Introduction

This module introduces you to input and output in Java. You will discover that files are treated as streams of bytes. The Java language provides many built-in classes for working with files. You will learn about sequential access and random-access files, and how to work with each of them.

After studying this unit, you should be able to:

- Learn about computer files
- Use the Path and Files classes
- Learn about file organization, streams, and buffers
- Use Java's IO classes to write to and read from a file
- Create and use sequential data files
- Learn about random access files
- Write records to a random-access data file
- Read records from a random-access data file

5.2 Understanding Computer Files

Data items can be stored on two broad types of storage devices in a computer system:

Volatile storage is temporary; values that are volatile, such as those stored in variables, are lost when a computer loses power. Random access memory (RAM) is the temporary storage within a computer. When you write a Java program that stores a value in a variable, you are using RAM. Most computer professionals simply call nonvolatile storage memory.

Nonvolatile storage is permanent storage; it is not lost when a computer loses power. When you write a Java program and save it to a disk, you are using permanent storage.

A computer file is a collection of data stored on a nonvolatile device. Files exist on permanent storage devices, such as hard disks, Zip disks, USB drives, reels or cassettes of magnetic tape, and compact discs.

You can categorize files by the way they store data:

- Text files contain data that can be read in a text editor because the data has been encoded using a scheme such as ASCII or Unicode. (See Appendix B for more information on Unicode.) Some text files are data files that contain facts and figures, such as a payroll file that contains employee numbers, names, and salaries; some files are program files or application files that store software instructions. You have created many such files throughout this book.
- Binary files contain data that has not been encoded as text. Their contents are in binary format, which means that you cannot understand them by viewing them in a text editor. Examples include images, music, and the compiled program files with a .class extension that you have created.

Although their contents vary, files have many common characteristics each file has a size that specifies the space it occupies on a section of disk or other storage device, and each file has a name and a specific time of creation. Computer files are the electronic equivalent of paper files stored in file cabinets. When you store a permanent file, you can place it in the main or root directory of your storage device. If you compare computer storage to using a file cabinet drawer, saving to the root directory is equivalent to tossing a loose document into the drawer. However, for better organization, most office clerks place documents in folders, and most computer users organize their files into folders or directories. Users also can create folders within folders to form a hierarchy.

A complete list of the disk drive plus the hierarchy of directories in which a file resides is its path. For example, the following is the complete path for a Windows file named Data.txt, which is saved on the C drive in a folder named Dilo within a folder named Java:

```
C:\Java\Dilo\Data.txt
```

In the Windows operating system, the backslash (\) is the path delimiter—the character used to separate path components. In the Solaris (UNIX) operating system, a slash (/) is used as the delimiter.

When you work with stored files in an application, you typically perform the following tasks:

- Determining whether and where a path or file exists
- Opening a file
- Writing to a file
- Reading from a file
- Closing a file
- Deleting a file

5.3 Java provides built-in classes that contain methods to help you with these tasks.

Using the Path Class

You use the Path class to create objects that contain information about files or directories, such as their locations, sizes, creation dates, and whether they even exist. You can include either of the following statements in a Java program to use the Path class:

```
import java.nio.file.Path;
```

The nio in java.nio stands for new input/output because its classes are “new” in that they were developed long after the first Java versions. The Path class is brand new in Java 7; it replaces the functionality of the File class used in older Java versions. If you search the Web for Java file-handling programs, you will find many that use the older File class.

Creating a Path

To create a Path, you first determine the default file system on the host computer by using a statement such as the following:

```
FileSystem fs = FileSystems.getDefault();
```

Notice that this statement creates a FileSystem object using the `getDefault()` method in the `FileSystems` class. The statement uses two different classes. The `FileSystem` class, without an ending s, is used to instantiate the object. `FileSystems`, with an ending s, is a class that contains factory methods, which assist in object creation. After you create a `FileSystem` object, you can define a `Path` using the `getPath()` method with it:

```
Path path = fs.getPath("C:\\Java\\Chapter.13\\Data.txt");
```

Recall that the backslash is used as part of an escape sequence in Java. (For example, `'\n'` represents a newline character.) So, to enter a backslash as a path delimiter within a string, you must type two backslashes to indicate a single backslash. An alternative is to use the `FileSystem` method `getSeparator()`. This method returns the correct separator for the current operating system. For example, you can create a `Path` that is identical to the previous one:

```
Path filePath = fs.getPath("C:" + fs.getSeparator() + "Java" + fs.getSeparator() + "Dilo" + fs.getSeparator() + "Data.txt");
```

Another way to create a `Path` is to use the `Paths` class (notice the name ends with s). The `Paths` class is a helper class that eliminates the need to create a `FileSystem` object. The `Paths` class `get()` method calls the `getPath()` method of the default file system without requiring you to instantiate a `FileSystem` object. You can create a `Path` object by using the following statement:

```
Path filePath = Paths.get("C:\\Java\\Chapter.13\\SampleFile.txt");
```

After the `Path` is created, you use its identifier (in this case, `filePath`) to refer to the file and perform operations on it. `C:\\Java\\Chapter.13\\SampleFile.txt` is the full name of a stored file when the operating system refers to it, but the path is known as `filePath` within the application. The idea of a file having one name when referenced by the operating system and a different name within an application is like the way a student known as “Arthur” in school might be “Junior” at home. When an application declares a path and you want to use the application with a different file, you would change only the `String` passed to the instantiating method. Every `Path` is either absolute or relative. An absolute path is a complete path; it does not need any other information to locate a file on a system. A relative path depends on another path information. A full path such as:

`C:\\Java\\Dilo\\SampleFile.txt` is an absolute path.

A path such as `SampleFile.txt` is relative. When you work with a path that contains only a filename, the file is assumed to be in the same folder as the program using it. Similarly, when you refer to a relative path such as `Dilo\\SampleFile.txt` (without the drive letter or the top-level Java folder), the `Dilo` folder is assumed to be a subfolder of the current directory, and the `SampleFile.txt` file is assumed to be within the folder. As you have learnt with other classes, the `toString()` method is overridden from the `Object` class; it returns a `String` representation of the `Path`. Basically, this is the list of path elements separated by copies of the default separator for the operating system. The `getName()` method returns the last element in a list of path

names; frequently this is a filename, but it might be a folder name. A Path's elements are accessed using an index. The top-level element in the directory structure is located at index 0; the lowest element in the structure is accessed by the `getName()` method, and has an index that is one less than the number of items on the list. You can use the `getNameCount()` method to retrieve the number of names in the list, and you can use the `getName(int)` method to retrieve the name in the position specified by the argument.

Stream Classes

Java's stream-based I/O is built upon four abstract classes: `InputStream`, `OutputStream`, `Reader`, and `Writer`. They are used to create several concrete stream subclasses. Although your programs perform their I/O operations through concrete subclasses, the top-level classes define the basic functionality common to all stream classes.

`InputStream` and `OutputStream` are designed for byte streams. `Reader` and `Writer` are designed for character streams. The byte stream classes and the character stream classes form separate hierarchies. In general, you should use the character stream classes when working with characters or strings, and use the byte stream classes when working with bytes or other binary objects. In the remainder of this guide, both the byte- and character-oriented streams are examined.

The Byte Streams

The byte stream classes provide a rich environment for handling byte-oriented I/O. A byte stream can be used with any type of object, including binary data. This versatility makes byte streams important to many types of programs. Since the byte stream classes are topped by `InputStream` and `OutputStream`, our discussion will begin with them.

InputStream

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

OutputStream

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

PrintStream

The `PrintStream` class provides all the formatting capabilities we have been using from the `System` file handle, `System.out`, since the beginning of the book. Here are two of its constructors:

`PrintStream(OutputStream outputStream)`

`PrintStream(OutputStream outputStream, boolean flushOnNewline)` where *flushOnNewline* controls whether Java flushes the output stream every time a newline (`\n`) character is output.

If *flushOnNewline* is true, flushing automatically takes place. If it is false, flushing is not automatic. The first constructor does not automatically flush.

Java's `PrintStream` objects support the `print()` and `println()` methods for all types, including `Object`. If an argument is not a simple type, the `PrintStream` methods will call the object's `toString()` method and then print the result.

RandomAccessFile

`RandomAccessFile` encapsulates a random-access file. It is not derived from `InputStream` or `OutputStream`. Instead, it implements the interfaces `DataInput` and `DataOutput`, which define the basic I/O methods. It also supports positioning requests—that is, you can position the *file pointer* within the file. It has these two constructors:

`RandomAccessFile(File fileObj, String access)`

throws `FileNotFoundException`

`RandomAccessFile(String filename, String access)`

throws `FileNotFoundException`

In the first form, *fileObj* specifies the name of the file to open as a `File` object. In the second form, the name of the file is passed in *filename*. In both cases, *access* determines what type of file access is permitted. If it is "r", then the file can be read, but not written. If it is "rw", then the file is opened in read-write mode. If it is "rws", the file is opened for read-write operations and every change to the file's data or metadata will be immediately written to the physical device. If it is "rwd", the file is opened for read-write operations and every change to the file's data will be immediately written to the physical device. The method `seek()`, shown here, is used to set the current position of the file pointer within the file:

`void seek(long newPos) throws IOException`

Here, *newPos* specifies the new position, in bytes, of the file pointer from the beginning of the file. After a call to `seek()`, the next read or write operation will occur at the new file position.

`RandomAccessFile` implements the standard input and output methods, which you can use to read and write to random access files. It also includes some additional methods. One is `setLength()`. It has this signature:

`void setLength(long len) throws IOException`

This method sets the length of the invoking file to that specified by *len*. This method can be used to lengthen or shorten a file. If the file is lengthened, the added portion is undefined.

Java 2, version 1.4 added the `getChannel()` method to `RandomAccessFile`. This method returns a channel connected to the `RandomAccessFile` object. Channels are used by the new I/O classes contained in `java.nio`.

ObjectOutput

The `ObjectOutput` interface extends the `DataOutput` interface and supports object serialization. It defines the methods shown in Table 17-5. Note especially the `writeObject()`

method. This is called to serialize an object. These methods will throw an IOException on error conditions.

5.4 ObjectOutputStream

The `ObjectOutputStream` class extends the `OutputStream` class and implements the `ObjectOutput` interface. It is responsible for writing objects to a stream. A constructor of this class is:

`ObjectOutputStream(OutputStream outStream)` throws `IOException`

The argument *outStream* is the output stream to which serialized objects will be written. The most commonly used methods in this class are shown in figure. They will throw an `IOException` on error conditions. Java 2 added an inner class to `ObjectOutputStream` called `PutField`. It facilitates the writing of persistent fields and its use is beyond the scope of this book.

Summary

Files are objects stored on nonvolatile, permanent storage. The `File` and `Files` class gather file information. Java views file as a series of bytes. Views a stream as an object through which input and output data flows.



Self-Evaluation Questions

1. Write a small Java program using the `Scanner` class to ask people to enter their age. Use Conditional Logic to test how old they are. Display the following messages, depending on how old they are and print to text file:
Less than 16: "You're still a youngster."
Over 16 but under 25: "Fame beckons!"
Over 25 but under 40: "There's still time."
Over 40: "Oh dear, you've probably missed it!"
Only one message box should display, when you Run the main project.
-

STUDY UNIT 6 INTRODUCTION TO SWING COMPONENTS

6.1 Introduction

This module introduces creating user interfaces with the Swing components. You will learn to use JFrames, JLabels, JTextFields, JButtons, JCheckBoxes, ButtonGroups, and JComboBoxes. Rudimentary event handling is covered with an emphasis on the ActionListener and ItemListener components.

After studying this unit, you should be able to:

- Understand Swing components
- Use the JFrame class
- Use the JLabel class
- Use a layout manager
- Extend the JFrame class
- Add JTextFields, JButtons, and tool tips to a JFrame
- Learn about event-driven programming
- Understand Swing event listeners
- Use the JCheckBox, ButtonGroup, and JComboBox classes

6.2 Understanding Swing Components

Computer programs usually are more user friendly (and more fun to use) when they contain user interface (UI) components. UI components are buttons, text fields, and other components with which the user can interact. Java's creators have packaged many prewritten components in the Swing package. Swing components are UI elements such as dialog boxes and buttons; you can usually recognize their names because they begin with J. UI components are also called controls or widgets. Each Swing component is a descendant of a JComponent, which in turn inherits from the java.awt.Container class. You can insert the statement `import javax.swing.*;` at the beginning of your Java program files so you can take advantage of the Swing UI components and their methods. When you import Swing classes, you use the `javax.swing` package instead of `java.swing`. The x originally stood for extension, so named because the Swing classes were an extension of the original Java language specifications.

When you use Swing components, you usually place them in containers. A container is a type of component that holds other components so you can treat a group of them as a single entity. Containers are defined in the Container class. Often, a container takes the form of a window that you can drag, resize, minimize, restore, and close. As you know from reading about inheritance all Java classes descend from the Object class. The Component class is a child of the Object class, and the Container class is a child of the Component class. Therefore, every Container object "is a" Component, and every Component object (including every Container) "is an" Object. The Container class is also a parent class, and the Window class is a child of Container. However, Java programmers rarely use Window objects because the Window subclass Frame and its child, the Swing component JFrame, both allow you to create more useful objects. Window objects do not have title bars or borders, but JFrame objects do.

6.3 Using the JFrame Class

You usually create a JFrame so that you can place other objects within it for display. Recall that the Object class is defined in the java.lang package, which is imported automatically every time you write a Java program. However, Object's descendants are not automatically imported.

The JFrame class has four constructors:

- JFrame() constructs a new frame that initially is invisible and has no title.
- JFrame(String title) creates a new, initially invisible JFrame with the specified title.
- JFrame(GraphicsConfiguration gc) creates a JFrame in the specified

You can construct a JFrame as you do other objects, using the class name, an identifier, the assignment operator, the new operator, and a constructor call. For example, the following two statements construct two JFrames: one with the title "Hello" and another with no title:

```
JFrame firstFrame = new JFrame("Hello");
```

```
JFrame secondFrame = new JFrame();
```

After you create a JFrame object, you can use the now-familiar object-dot-method format you have used with other objects to call methods that manipulate a JFrame's features. Assuming you have declared a JFrame named firstFrame, you can use the following statements to set the firstFrame object's size to 250 pixels horizontally by 100 pixels vertically and set the JFrame's title to display a String argument. Pixels are the picture elements, or tiny dots of light, that make up the image on your computer monitor. firstFrame.setSize(250, 100); firstFrame.setTitle("My frame"); When you set a JFrame's size, you do not have the full area available to use because part of the area is consumed by the JFrame's title bar and borders.

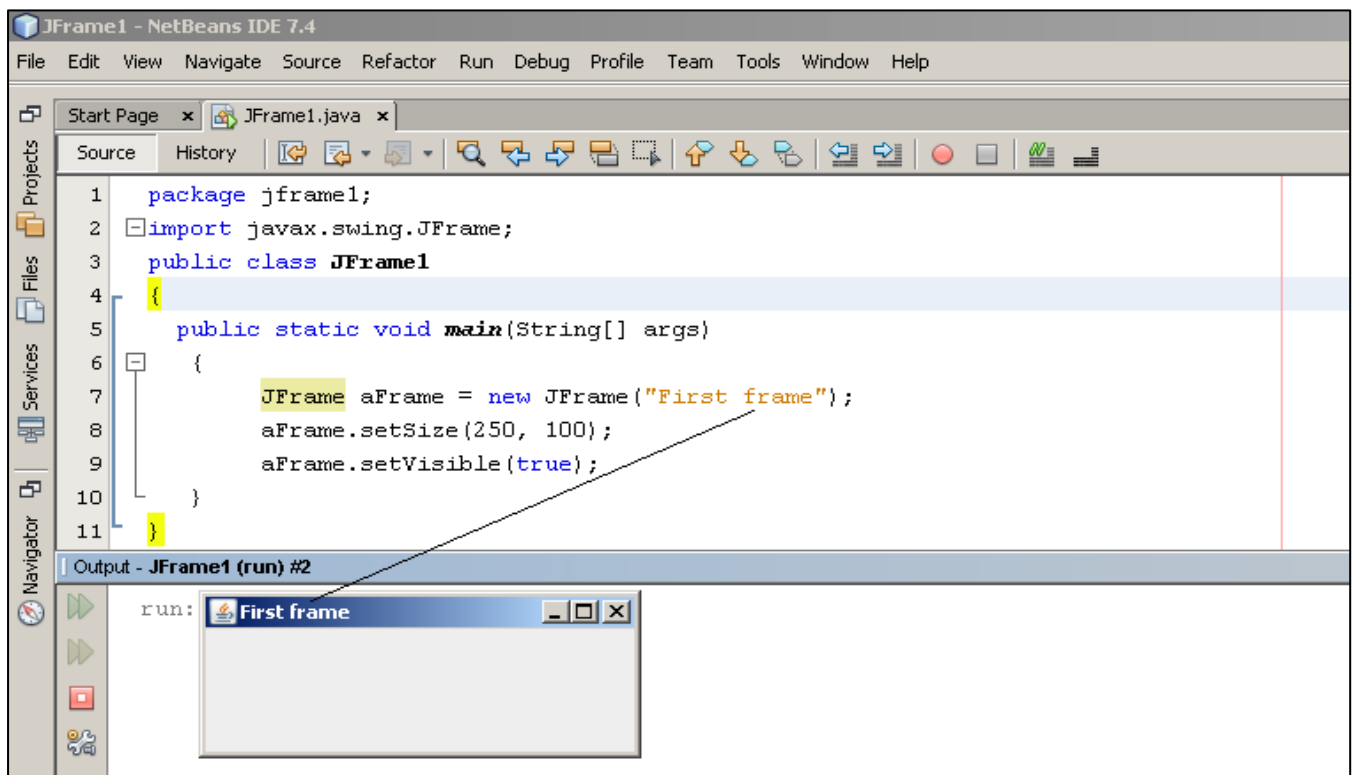
```
import javax.swing.*;

public class JFrame1
{
    public static void main(String[] args)
    {
        JFrame aFrame = new JFrame("First frame");
        aFrame.setSize(250, 100);
        aFrame.setVisible(true);
    }
}
```

The application produces the JFrame resembles frames that you have probably seen when using different UI programs, you have downloaded or purchased. One reason to use similar frame objects in your own programs is that users are already familiar with the frame environment. When users see frames on their computer screens, they expect to see a title bar at the top containing text information (such as "First frame"). Users also expect to see

Minimize, Maximize or Restore, and Close buttons in the frame's upper-right corner. Most users assume that they can change a frame's size by dragging its border or reposition the frame on their screen by dragging the frame's title bar to a new location. The JFrame has these capabilities. In the application all three statements in the main() method are important. After you instantiate aFrame, if you do not use setVisible(true), you do not see the JFrame, and if you do not set its size, you see only the title bar of the JFrame because the JFrame size is 0 x 0 by default. It might seem unusual that the default state for a JFrame is invisible. However, consider that you might want to construct a JFrame in the background while other actions are occurring and that you might want to make it visible later, when appropriate (for example, after the user has taken an action such as selecting an option). To make a frame visible, some Java programmers use the show() method instead of the setVisible() method.

Figure 6.1: Frame



When a user closes a JFrame by clicking the Close button in the upper-right corner, the default behavior is for the JFrame to become hidden and for the application to keep running. This makes sense when there are other tasks for the program to complete after the main frame is closed for example, displaying additional frames, closing open data files, or printing an activity report. However, when a JFrame serves as a Swing application's main user interface (as happens frequently in interactive programs), you usually want the program to exit when the user clicks.

Using the JFrame Class Close. To change this behavior, you can call a JFrame's setDefaultCloseOperation() method and use one of the following four values as an argument:

- JFrame.EXIT_ON_CLOSE exits the program when the JFrame is closed.
- WindowConstants.DISPOSE_ON_CLOSE closes the frame, disposes of the JFrame object, and keeps running the application.

- `WindowConstants.DO_NOTHING_ON_CLOSE` keeps the `JFrame` open and continues running. In other words, it disables the Close button.
- `WindowConstants.HIDE_ON_CLOSE` closes the `JFrame` and continues running; this is the default operation that you frequently want to override.

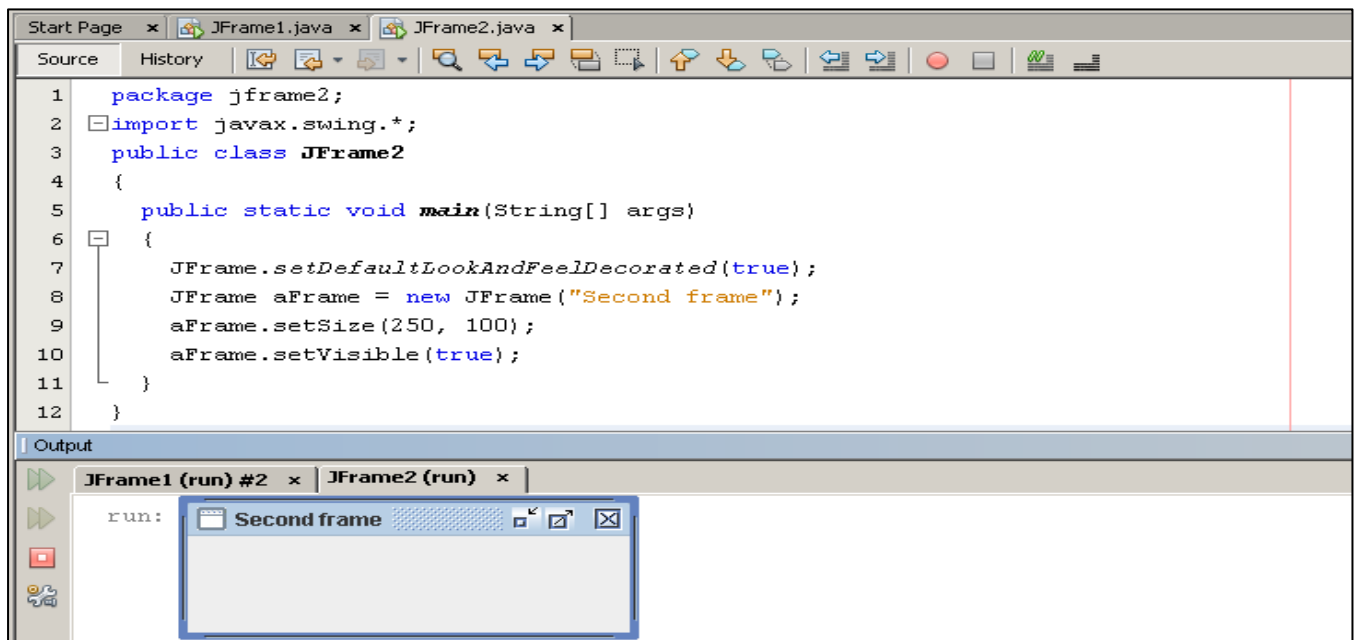
When you execute an application in which you have forgotten to exit when the `JFrame` is closed, you can end the program by typing `Ctrl+C`.

6.4 Customizing a `JFrame`'s Appearance

The appearance of the `JFrame` provided by the operating system in which the program is running (in this case, Windows). For example, the coffee-cup icon in the frame's title bar and the Minimize, Restore, and Close buttons look and act as they do in other Windows applications. The icon and buttons are known as window decorations; by default, window decorations are supplied by the operating system. However, you can request that Java's look and feel provide the decorations for a frame. A look and feel is the default appearance and behavior of any user interface.

Optionally, you can set a `JFrame`'s look and feel using the `setDefaultLookAndFeelDecorated()` method. **For example, an application that calls this method.**

Figure 6.2: an application that calls this method.



Using the `JLabel` Class

One of the components you might want to place on a `JFrame` is a `JLabel`. `JLabel` is a built-in Java Swing class that holds text you can display. Available constructors for the `JLabel` class include the following:

- `JLabel()` creates a `JLabel` instance with no image and with an empty string for the title.

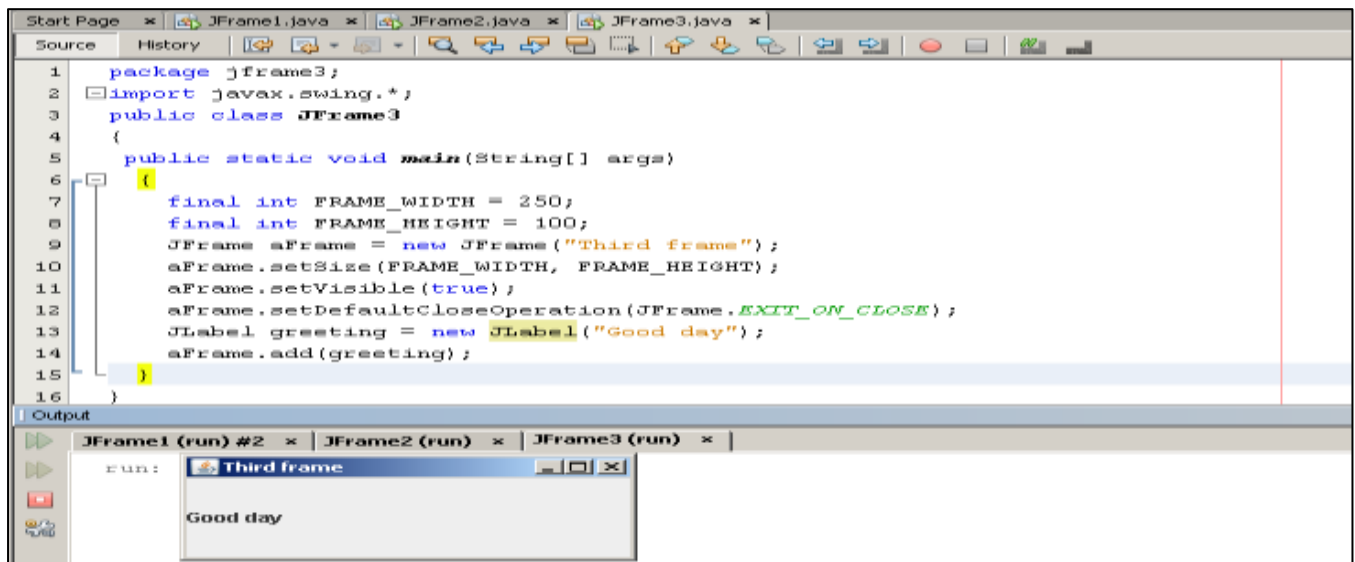
- JLabel(Icon image) creates a JLabel instance with the specified image.
- JLabel(Icon image, int horizontalAlignment) creates a JLabel instance with the specified image and horizontal alignment.
- JLabel(String text) creates a JLabel instance with the specified text.
- JLabel(String text, Icon icon, int horizontalAlignment) creates a JLabel instance with the specified text, image, and horizontal alignment.
- JLabel(String text, int horizontalAlignment) creates a JLabel instance with the specified text and horizontal alignment.

For example, you can create a JLabel named greeting that holds the words “Good day” by writing the following statement: `JLabel greeting = new JLabel("Good day");`

You then can add the greeting object to the JFrame object named aFrame using the `add()` method as follows: `aFrame.add(greeting);`

The example in the next page shows an application in which a JFrame is created and its size, visibility, and close operation are set. Then a JLabel is created and added to the JFrame. The counterpart to the `add()` method is the `remove()` method. The following statement removes greeting from aFrame: `aFrame.remove(greeting);`

Figure 6.3: Creating the JFrame



If you add or remove a component from a container after it has been made visible, you should also call the `invalidate()`, `validate()`, and `repaint()` methods, or else you will not see the results of your actions. Each performs slightly different functions, but all three together guarantee that the results of changes in your layout will take effect. The `invalidate()` and `validate()` methods are part of the Container class, and the `repaint()` method is part of the Component class. You

can change the text in a JLabel by using the Component class `setText()` method with the JLabel object and passing a String to it. For example, the following code changes the value displayed in the greeting JLabel: `greeting.setText("Howdy");`

You can retrieve the text in a JLabel (or other Component) by using the `getText()` method, which returns the currently stored String.

Changing a JLabel's Font

If you use the Internet and a Web browser to visit Web sites, you probably are not very impressed with the simple application. You might think that the string "Good day" is plain and lackluster. Fortunately, Java provides you with a Font class from which you can create an object that holds typeface and size information. The `setFont()` method requires a Font object argument. To construct a Font object, you need three arguments:

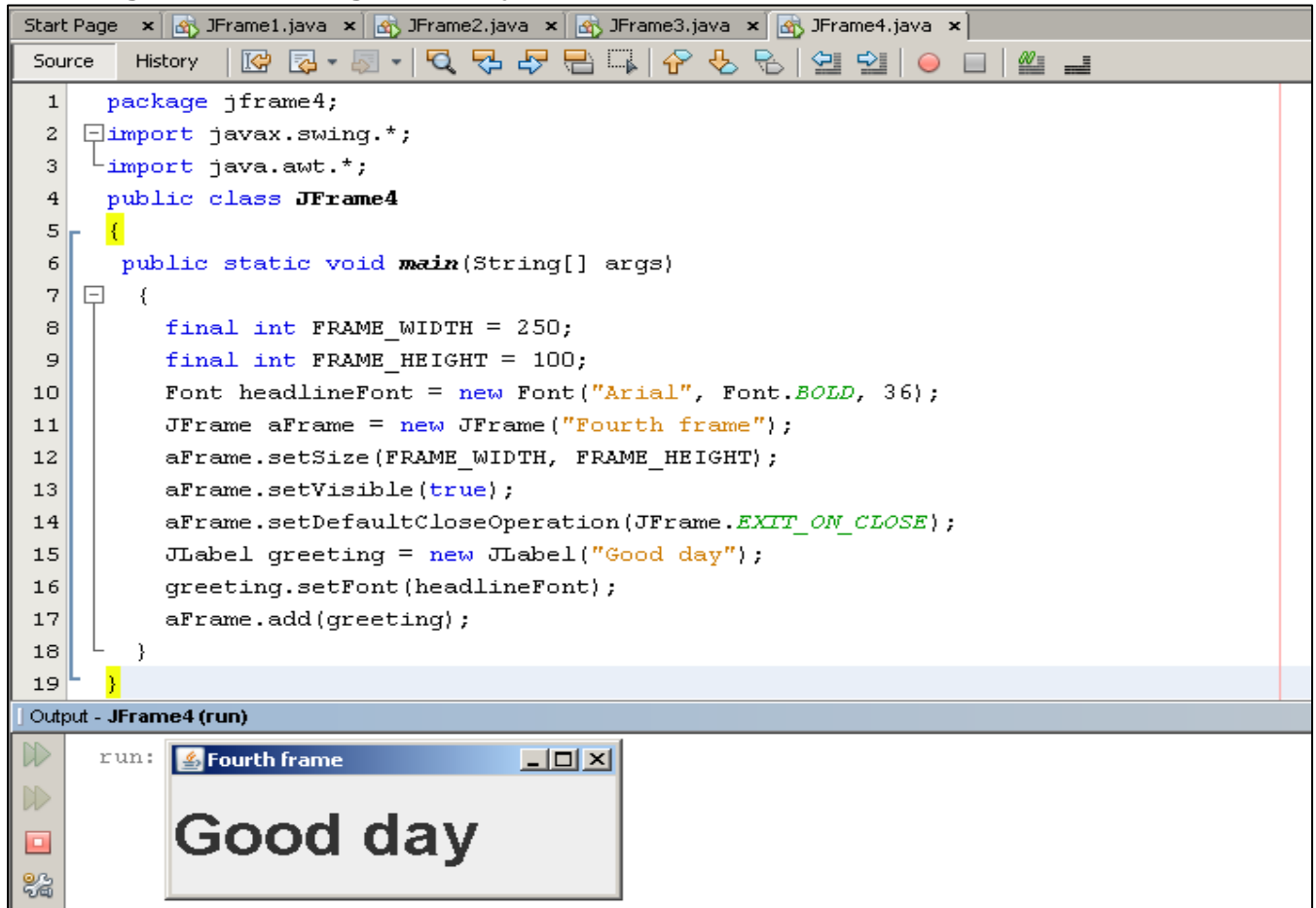
1. Typeface
 2. Style
 3. point size.
- The typeface argument to the Font constructor is a String representing a font. Common fonts have names such as Arial, Century, Monospaced, and Times New Roman. The typeface argument in the Font constructor is only a request; the system on which your program runs might not have access to the requested font, and if necessary, it substitutes a default font.
 - The style argument applies an attribute to displayed text and is one of three values: `Font.PLAIN`, `Font.BOLD`, or `Font.ITALIC`.
 - The point size argument is an integer that represents about 1/72 of an inch. Printed text is commonly 12 points; a headline might be 30 points.

To give a JLabel object a new font, you can create a Font object, as in the following:

`Font headlineFont = new Font("Monospaced", Font.BOLD, 36);` The typeface name is a String, so you must enclose it in double quotation marks. You can use the `setFont()` method to assign the Font to a JLabel with a statement such as:

`greeting.setFont(headlineFont);`

Figure 6.4: Creating a Font object



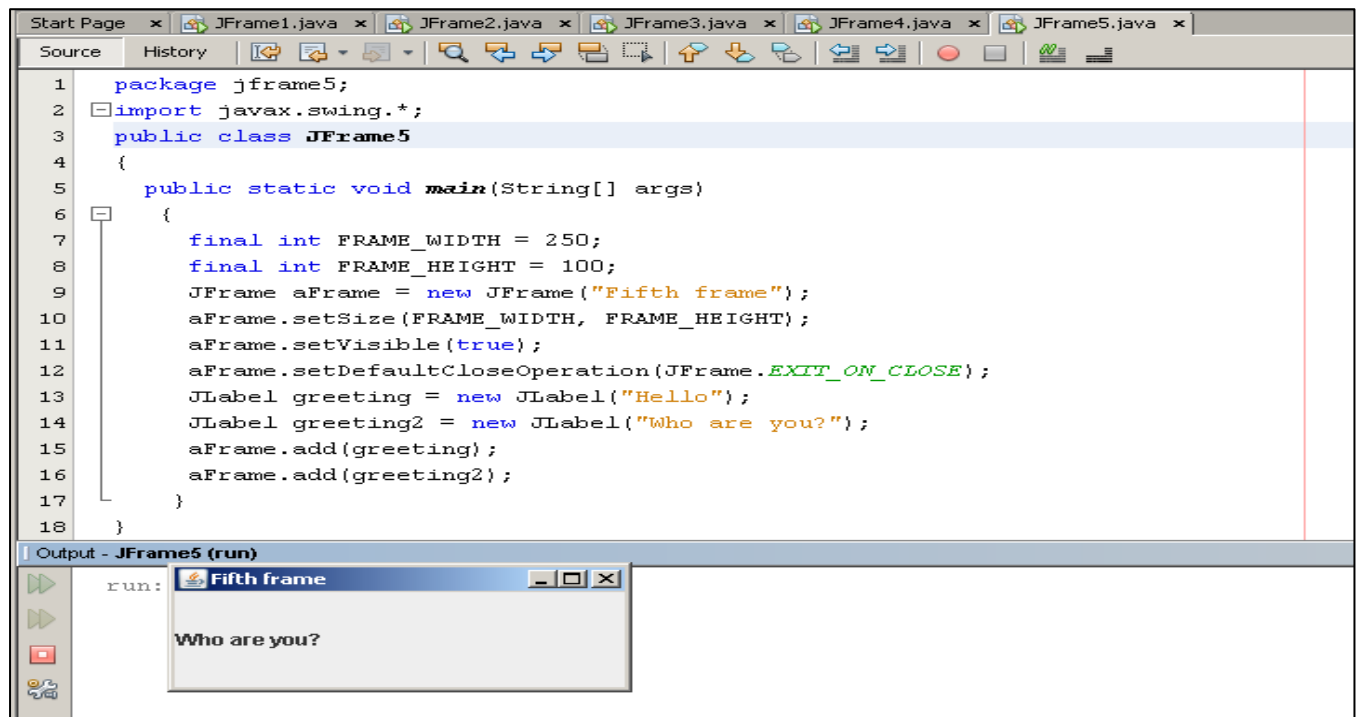
The program contains a Font object named `headlineFont` that is applied to the greeting. The example shows the execution of the `JFrame4` program; the greeting appears in a 36-point, bold, Arial font. You are not required to provide an identifier for a Font. For example, you could omit the shaded statement that declares `headlineFont` and set the greeting Font with the following statement that uses an anonymous Font object:

```
greeting.setFont(new Font("Arial", Font.BOLD, 36));
```

Using a Layout Manager

When you want to add multiple components to a JFrame or other container, you usually need to provide instructions for the layout of the components. For example, an application in which two JLabels are created and added to a JFrame in the final shaded statements.

Figure 6.5: Using a Lay out Manager



Although two JLabels are added to the frame, only the last one added is visible. The second JLabel has been placed on top of the first one, totally obscuring it. If you continued to add more JLabels to the program, only the last one added to the JFrame would be visible. To place multiple components at specified positions in a container so they do not hide each other, you must explicitly use a layout manager a class that controls component positioning. The normal (default) behavior of a JFrame is to use a layout format named BorderLayout. A BorderLayout, created by using the BorderLayout class, divides a container into regions. When you do not specify a region in which to place a component (as the JFrame5 program fail When you use a FlowLayout instead of a BorderLayout, components do not lie on top of each other. Instead, the flow layout manager places components in a row, and when a row is filled, components automatically spill into the next row.

Three constants are defined in the FlowLayout class that specify how components are positioned in each row of their container. These constants are FlowLayout.LEFT, FlowLayout.RIGHT, and FlowLayout.CENTER. For example, to create a layout manager named flow that positions components to the right, you can use the following statement:

```
FlowLayout flow = new FlowLayout(FlowLayout.RIGHT);
```

If you do not specify how components are laid out, by default they are centered in each row. Suppose you create a FlowLayout object named flow as follows:

```
FlowLayout flow = new FlowLayout();
```

Then the layout of a JFrame named aFrame can be set to the newly created FlowLayout using the statement:

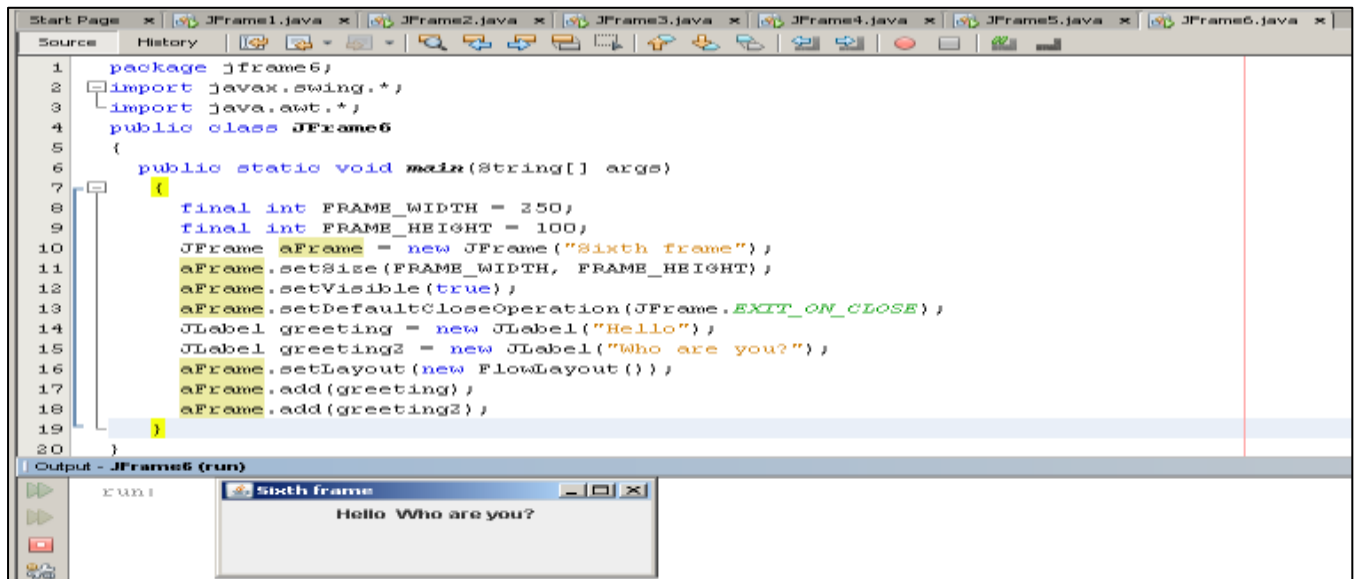
```
aFrame.setLayout(flow);
```

A more compact syntax that uses an anonymous FlowLayout object is:

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

The example below shows an application in which the JFrame's layout manager has been set so that multiple components are visible.

Figure 6.6: JFrame's layout manager with multiple components



6.5 Extending the JFrame Class

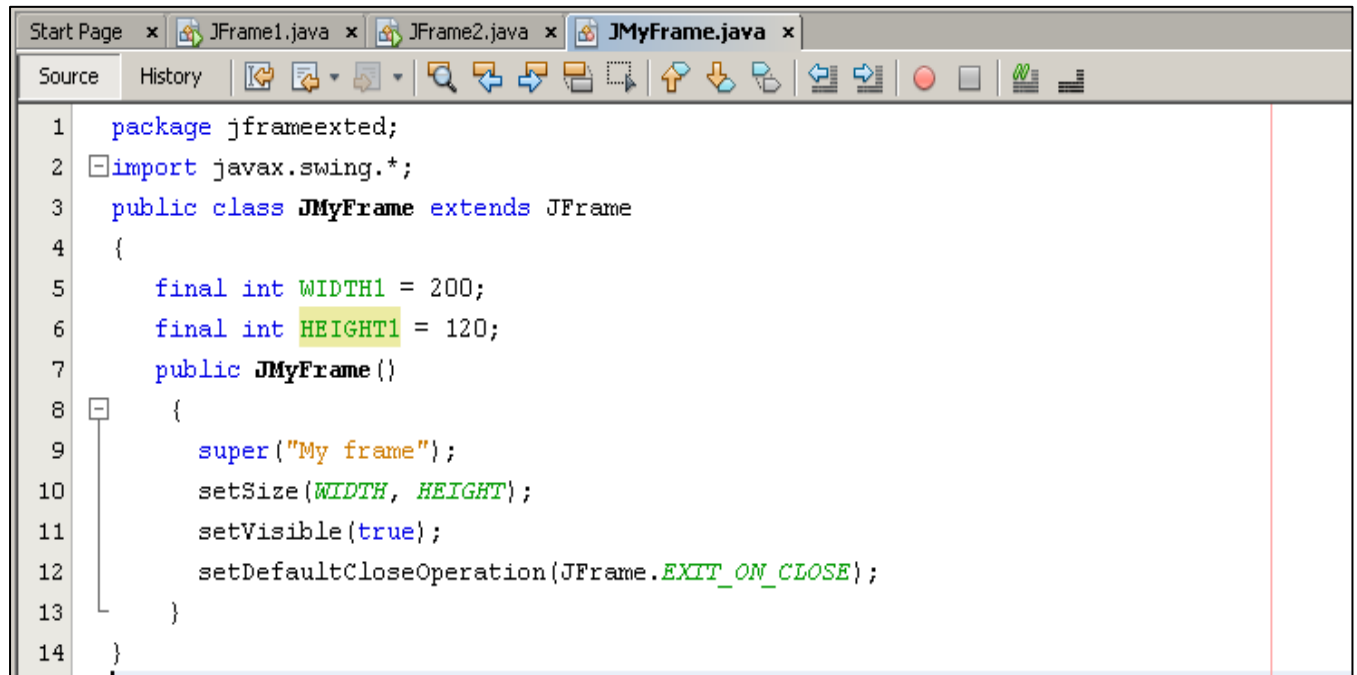
You can instantiate a simple JFrame object within an application's main() method or with any other method of any class you write. Alternatively, you can create your own class that descends from the JFrame class. The advantage of creating a child class of JFrame is that you can set the JFrame's properties within your object's constructor; then, when you create your JFrame child object, it is automatically endowed with the features you have specified, such as title, size, and default close operation.

You already know that you create a child class by using the keyword extends in the class header, followed by the parent class name. You also know that you can call the parent class's constructor using the keyword super, and that when you call super(), the call must be the first statement in the constructor. For example, the JMyFrame class extends JFrame. Within the JMyFrame constructor, the super() JFrame constructor is called; it accepts a String argument to use as the JFrame's title. (Alternatively, the setTitle() method could have been used.) The JMyFrame constructor also sets the size, visibility, and default close operation for every JMyFrame. Each of the methods setSize(), setVisible(), and setDefaultCloseOperation() appears in the constructor without an object, because the object is the current JMyFrame being constructed. Each of the three methods could be preceded with a this reference with exactly the same meaning. That is, within the JMyFrame constructor, the following two statements have identical meanings:

```
setSize(WIDTH, HEIGHT);  
this.setSize(WIDTH, HEIGHT);
```

Each statement sets the size of “this” current JMyFrame instance.

Figure 6.7: Current JMyFrame instance



The JMyFrame class

The example below shows an application that declares two JMyFrame objects. Each has the same set of attributes, determined by the JMyFrame constructor.

```
public class CreateTwoJMyFrameObjects  
{  
    public static void main(String[] args)  
    {  
        JMyFrame myFrame = new JMyFrame();  
        JMyFrame mySecondFrame = new JMyFrame();  
    }  
}
```

The CreateTwoJMyFrameObjects application

Adding JTextFields, JButtons, and Tool Tips to a JFrame

In addition to including JLabel objects, JFrames often contain other window features, such as JTextFields, JButtons, and tool tips.

Adding JTextFields

A JTextField is a component into which a user can type a single line of text data. (Text data comprises any characters you can enter from the keyboard, including numbers and punctuation.) The example shows the inheritance hierarchy of the JTextField class. Typically, a user types a line into a JTextField and then presses Enter on the keyboard or clicks a button with the mouse to enter the data.

You can construct a JTextField object using one of several constructors:

- `public JTextField()` constructs a new JTextField.
- `public JTextField(int columns)` constructs a new, empty JTextField with a specified number of columns.
- `public JTextField(String text)` constructs a new JTextField initialized with the specified text.
- `public JTextField(String text, int columns)` constructs a new JTextField initialized with the specified text and columns.

For example, to provide a JTextField that allows enough room for a user to enter approximately 10 characters, you can code the following:

```
JTextField response = new JTextField(10);
```

To add the JTextField named response to a JFrame named frame, you write:

```
frame.add(response);
```

The number of characters a JTextField can display depends on the font being used and the actual characters typed. For example, in most fonts, 'w' is wider than 'i', so a JTextField of size 10 using the Arial font can display 24 'i' characters, but only eight 'w' characters. Try to anticipate how many characters your users might enter when you create a JTextField. The user can enter more characters than those that display, but the extra characters scroll out of view. It can be disconcerting to try to enter data into a field that is not large enough. It is usually better to overestimate than underestimate the size of a text field.

Several other methods are available for use with JTextFields. The `setText()` method allows you to change the text in a JTextField (or other Component) that has already been created, as in the following:

```
response.setText("Thank you");
```

After a user has entered text in a `JTextField`, you can clear it out with a statement such as the following, which assigns an empty string to the text:

```
response.setText("");
```

The `getText()` method allows you to retrieve the `String` of text in a `JTextField` (or other `Component`), as in:

```
String whatUserTyped = response.getText();
```

When a `JTextField` has the capability of accepting keystrokes, the `JTextField` is editable. A `JTextField` is editable by default. If you do not want the user to be able to enter data in a `JTextField`, you can send a boolean value to the `setEditable()` method to change the `JTextField`'s editable status. For example, if you want to give a user a limited number of chances to answer a question correctly, you can count data entry attempts and then prevent the user from replacing or editing the characters in the `JTextField` by using a statement similar to the following:

```
if(attempts > LIMIT)
response.setEditable(false);
```

Adding JButtons

A `JButton` is a `Component` the user can click with a mouse to select. A `JButton` is even easier to create than a `JTextField`. There are five `JButton` constructors:

- `public JButton()` creates a button with no set text.
- `public JButton(Icon icon)` creates a button with an icon of type `Icon` or `ImageIcon`.
- `public JButton(String text)` creates a button with text.
- `public JButton(String text, Icon icon)` creates a button with initial text and an icon of type `Icon` or `ImageIcon`.
- `public JButton(Action a)` creates a button in which properties are taken from the `Action` supplied. (`Action` is a Java class.)

To create a `JButton` with the label "Press when ready", you can write the following:

```
JButton readyJButton = new JButton("Press when ready");
```

You can add a `JButton` to a `JFrame` (or other container) using the `add()` method. You can change a `JButton`'s label with the `setText()` method, as in:

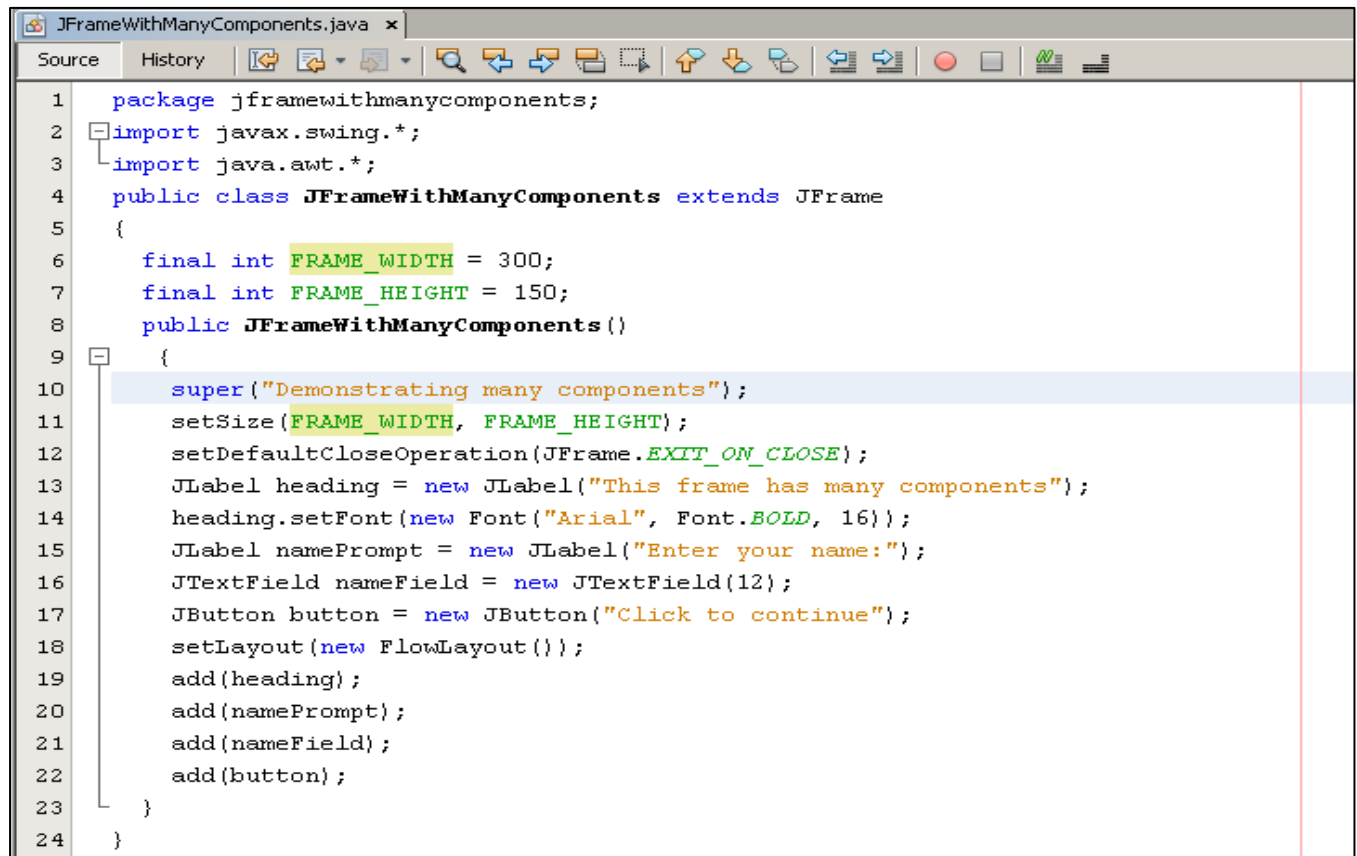
```
readyJButton.setText("Don't press me again!");
```

You can retrieve the text from a JButton and assign it to a String object with the getText() method, as in:

```
String whatsOnJButton = readyJButton.getText();
```

The example in the next page shows a class that extends JFrame and holds several components. As the components (two JLabels, a JTextField, and a JButton) are added to the JFrame, they are placed from left to right in horizontal rows across the JFrame's surface. The program that instantiates an instance of the JFrame.

Figure 6.8: Extending the JFrame



The JFrameWithManyComponents class

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

6.6 Learning About Event-Driven Programming

An event occurs when a user acts on a component, such as clicking the mouse on a JButton object. In an event-driven program, the user might initiate any number of events in any order. For example, if you use a word-processing program, you have dozens of choices at your disposal at any time. You can type words, select text with the mouse, click a button to change text to bold, click a button to change text to italic, choose a menu item, and so on. With each word-processing document you create, you choose options in any order that seems appropriate at the time. The word-processing program must be ready to respond to any event you initiate. Within an event-driven program, a component on which an event is generated is the source of the event. A button that a user can click is an example of a source; a text field that a user can use to enter text is another source. An object that is interested in an event is a listener. Not all objects listen for all possible events you probably have used programs in which clicking many areas of the screen has no effect. If you want an object to be a listener for an event, you must register the object as a listener for the source.

Social networking sites maintain lists of people in whom you are interested and notify you each time a person on your list posts a comment or picture. Similarly, a Java component source object (such as a button) maintains a list of registered listeners and notifies all of them when any event occurs. For example, a JFrame might want to be notified of any mouse click on its surface. When the listener “receives the news,” an event-handling method contained in the listener object responds to the event. To respond to user events within any class you create, you must do the following:

- Prepare your class to accept event messages.
- Tell your class to expect events to happen.
- Tell your class how to respond to events.

Preparing Your Class to Accept Event Messages

You prepare your class to accept button-press events by importing the `java.awt.event` package into your program and adding the phrase `implements ActionListener` to the class header. The `java.awt.event` package includes event classes with names such as `ActionEvent`, `ComponentEvent`, and `TextEvent`. `ActionListener` is an interface a class containing a set of specifications for methods that you can use. Implementing `ActionListener` provides you with standard event method specifications that allow your listener to work with `ActionEvents`, which are the types of events that occur when a user clicks a button.

Telling Your Class to Expect Events to Happen

You tell your class to expect `ActionEvents` with the `addActionListener()` method. If you have declared a JButton named `aButton`, and you want to perform an action when a user clicks `aButton`, `aButton` is the source of a message, and you can think of your class as a target to which to send a message. You learned in Chapter 4 that the `this` reference means “this current

object,” so the code `aButton.addActionListener(this);` causes any `ActionEvent` messages (button clicks) that come from `aButton` to be sent to “this current object.”

Telling Your Class How to Respond to Events

The `ActionListener` interface contains the `actionPerformed(ActionEvent e)` method specification. When a class, such as a `JFrame`, has registered as a listener with a Component such as a `JButton`, and a user clicks the `JButton`, the `actionPerformed()` method executes. You implement the `actionPerformed()` method, which contains a header and a body, like all methods. You use the following header, in which `e` represents any name you choose for the Event (the `JButton` click) that initiated the notification of the `ActionListener` (which is the `JFrame`):

```
public void actionPerformed (ActionEvent e)
```

The body of the method contains any statements that you want to execute when the action occurs. You might want to perform mathematical calculations, construct new objects, produce output, or execute any other operation. For example, in the next page shows a `JFrame` containing a `JLabel` that prompts the user for a name, a `JTextField` into which the user can type a response, a `JButton` to click, and a second `JLabel` that displays the name entered by the user. The `actionPerformed()` method executes when the user clicks the `pressMe` `JButton`; within the method, the `String` that a user has typed into the `JTextField` is retrieved and stored in the `name` variable. The name is then used as part of a `String` that alters the second `JLabel` on the `JFrame`.

```
import javax.swing.*;

import java.awt.*;

import java.awt.event.*;

public class JHelloFrame extends JFrame implements ActionListener
{
    JLabel question = new JLabel("What is your name?");
    Font bigFont = new Font("Arial", Font.BOLD, 16);
    JTextField answer = new JTextField(10);
    JButton pressMe = new JButton("Press me");
    JLabel greeting = new JLabel("");
    final int WIDTH = 175;
    final int HEIGHT = 225;
    public JHelloFrame()
    {
        super("Hello Frame");
```

```

setSize(WIDTH, HEIGHT);
setLayout(new FlowLayout());
question.setFont(bigFont);
greeting.setFont(bigFont);
add(question);
add(answer);
add(pressMe);
add(greeting);
setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
pressMe.addActionListener(this);
}
public void actionPerformed(ActionEvent e)
{
    String name = answer.getText();
    String greet = "Hello, " + name;
    greeting.setText(greet);
}
}

```

The JHelloFrame class that produces output when the user clicks the JButton

```

public class JHelloDemo
{
    public static void main(String[] args)
    {
        JHelloFrame frame = new JHelloFrame();
        frame.setVisible(true);
    }
}

```

An application that instantiates a JHelloFrame object

A typical execution of the JHelloDemo program is shown below. The user enters Lindsey into the JTextField, and the greeting with the name is displayed after the user clicks the button.



When more than one component is added and registered to a JFrame, it might be necessary to determine which component was used to initiate an event. For example, in the JHelloFrame class in Figure 14-24, you might want the user to be able to see the message after either clicking the button or pressing Enter in the JTextField. In that case, you would designate both the pressMe button and the answer text field to be message sources by using the addActionListener() method with each, as follows:

```
pressMe.addActionListener(this);  
answer.addActionListener(this);
```

These two statements make the JFrame (this) the receiver of messages from either object. The JFrame has only one actionPerformed() method, so it is the method that executes when either the pressMe button or the answer text field sends a message. If you want different actions to occur depending on whether the user clicks the button or presses Enter, you must determine the source of the event. Within the actionPerformed() method, you can use the getSource() method of the object sent to determine which component generated the event. For example, within a method with the header public void actionPerformed(ActionEvent e), e is an ActionEvent. ActionEvent and other event classes are part of the java.awt.event package and are subclasses of the EventObject class. To determine what object generated the ActionEvent, you can use the following statement:

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

For example, if a JFrame contains two JButtons named option1 and option2, you can use the decision structure in the method to take different courses of action based on the button that is clicked. Whether an event's source is a JButton, JTextField, or other Component, it can be assigned to an Object because all components descend from Object.

```
void actionPerformed(ActionEvent e)  
{  
    Object source = e.getSource();  
    if(source == option1)  
        //execute these statements when user clicks option1  
    else
```

```
//execute these statements when user clicks any other option
```

```
}
```

An actionPerformed() method that takes one of two possible actions

Alternatively, you can also use the instanceof keyword to determine the source of the event. The instanceof keyword is used when it is necessary to know only the component's type, rather than what component triggered the event. For example, if you want to take some action when a user enters data into any JTextField, but not when an event is generated by a different Component type, you could use the method format.

```
void actionPerformed(ActionEvent e)
{
Object source = e.getSource();
if(source instanceof JTextField)
{
// execute these statements when any JTextField
// generates the event
// but not when a JButton or other Component does
}
}
```

An actionPerformed() method that executes a block of statements when a user generates an event from any JTextField

Using the setEnabled() Method You probably have used computer programs in which a component becomes disabled or unusable. For example, a JButton might become dim and unresponsive when the programmer no longer wants you to have access to the JButton's functionality. Components are enabled by default, but you can use the setEnabled() method to make a component available or unavailable by passing true or false to it, respectively. For example, a JFrame with two JButton objects. The one on top is enabled, but the one on the bottom has been disabled



Understanding Swing Event Listeners

Many types of listeners exist in Java, and each of these listeners can handle a specific event type. A class can implement as many event listeners as it needs for example, a class might need to respond to both a mouse button press and a keyboard key press, so you might implement both `ActionListener` and `KeyListener` interfaces. An event occurs every time a user types a character or clicks a mouse button. Any object can be notified of an event as long as it implements the appropriate interface and is registered as an event listener on the appropriate event source. You already know that you establish a relationship between a `JButton` and a `JFrame` that contains it by using the `addActionListener()` method. Similarly, you can create relationships between other Swing components and the classes that react to users' manipulations of them.

When you want a `JCheckBox` to respond to a user's clicks, you can use the `addItemListener()` method to register the `JCheckBox` as the type of object that can create an `ItemEvent`. The argument you place within the parentheses of the call to the `addItemListener()` method is the object that should respond to the event perhaps a `JFrame` that contains the eventgenerating `JCheckBox`. The format is:

```
theSourceOfTheEvent.addListenerMethod(theClassThatShouldRespond);
```

The class of the object that responds to an event must contain a method that accepts the event object created by the user's action. A method that executes because it is called automatically when an appropriate event occurs is an event handler. In other words, when you register a component (such as a `JFrame`) to be a listener for events generated by another component (such as a `JCheckBox`), you must write an event handler method. You cannot choose your own name for event handlers specific method identifiers react to specific event types.

Until you become familiar with the event-handling model, it can seem quite confusing. For now, remember these tasks you must perform when you declare a class that handles an event:

- The class that handles an event must either implement a listener interface or extend a class that implements a listener interface. For example, if a `JFrame` named `MyFrame` needs to respond to a user's clicks on a `JCheckBox`, you would write the following class header:

public class MyFrame extends JFrame implements ItemListener

If you then declare a class that extends MyFrame, you need not include implements ItemListener in its header. The new class inherits the implementation.

- You must register each instance of the event-handling class as a listener for one or more components. For example, if MyFrame contains a JCheckBox named myCheckBox, then within the MyFrame class you would code:

```
myCheckBox.addItemListener(this);
```

The this reference is to the class in which myCheckBox is declared in this case, MyFrame.

Using the JCheckBox, ButtonGroup, and JComboBox Classes

Besides JButtons and JTextFields, several other Java components allow a user to make selections in a UI environment. These include JCheckBoxes, ButtonGroups, and JComboBoxes.

The JCheckBox Class

A JCheckBox consists of a label positioned beside a square; you can click the square to display or remove a check mark. Usually, you use a JCheckBox to allow the user to turn an option on or off. For example, the code for an application that uses four JCheckBoxes.

The ButtonGroup Class

Sometimes, you want options to be mutually exclusive that is, you want the user to be able to select only one of several choices. When you create a ButtonGroup, you can group several components, such as JCheckBoxes, so a user can select only one at a time. When you group JCheckBox objects, all the other JCheckBoxes are automatically turned off when the user selects any one check box. You can see that ButtonGroup descends directly from the Object class. Even though it does not begin with a J, the ButtonGroup class is part of the javax.swing package.

To create a ButtonGroup in a JFrame and then add a JCheckBox, you must perform four steps:

- Create a ButtonGroup, such as ButtonGroup aGroup = new ButtonGroup();.
- Create a JCheckBox, such as JCheckBox aBox = new JCheckBox();.
- Add aBox to aGroup with aGroup.add(aBox);.

- Add aBox to the JFrame with add(aBox);.

You can create a ButtonGroup and then create the individual JCheckBox objects, or you can create the JCheckBoxes and then create the ButtonGroup. If you create a ButtonGroup but forget to add any JCheckBox objects to it, then the JCheckBoxes act as individual, nonexclusive check boxes. A user can set one of the JCheckBoxes within a group to “on” by clicking it with the mouse, or the programmer can select a JCheckBox within a ButtonGroup with a statement such as the following:

```
aGroup.setSelected(aBox);
```

Only one JCheckBox can be selected within a group. If you assign the selected state to a JCheckBox within a group, any previous assignment is negated. You can determine which, if any, of the JCheckBoxes in a ButtonGroup is selected using the isSelected() method. After a JCheckBox in a ButtonGroup has been selected, one in the group will always be selected. In other words, you cannot “clear the slate” for all the items that are members of a ButtonGroup. You could cause all the JCheckBoxes in a ButtonGroup to initially appear unselected by adding one JCheckBox that is not visible (using the setVisible() method). Then, you could use the setSelected() method to select the invisible JCheckBox, and all the others would appear to be deselected.

The JComboBox Class

A JComboBox is a component that combines two features: a display area showing a default option and a list box that contains additional, alternate options. (A list box is also known as a combo box or a drop-down list.) The display area contains either a button that a user can click or an editable field into which the user can type. When a JComboBox appears on the screen, the default option is displayed. When the user clicks the JComboBox, a list of alternative items drops down; if the user selects one, it replaces the box’s displayed item. A JComboBox as it looks when first displayed and as it looks after a user clicks it.

You can build a JComboBox by using a constructor with no arguments and then adding items (for example, Strings) to the list with the addItem() method. The following statements create a JComboBox named majorChoice that contains three options from which a user can choose:

```
JComboBox majorChoice = new JComboBox();  
majorChoice.addItem("English");  
majorChoice.addItem("Math");  
majorChoice.addItem("Sociology");
```

Alternatively, you can construct a JComboBox using an array of Objects as the constructor argument; the Objects in the array become the listed items within the JComboBox. For example, the following code creates the same majorChoice JComboBox as the preceding code:

```
String[] majorArray = {"English", "Math", "Sociology"};
JComboBox majorChoice = new JComboBox(majorArray);
```

You can use the `setSelectedItem()` or `setSelectedIndex()` method to choose one of the items in a `JComboBox` to be the initially selected item. You also can use the `getSelectedItem()` or `getSelectedIndex()` method to discover which item is currently selected. You can treat the list of items in a `JComboBox` object as an array; the first item is at position 0, the second is at position 1, and so on. It is convenient to use the `getSelectedIndex()` method to determine the list position of the currently selected item; then you can use the index to access corresponding information stored in a parallel array. For example, if a `JComboBox` named `historyChoice` has been filled with a list of historical events, such as "Declaration of Independence," "Pearl Harbor," and "Man walks on moon," you can code the following to retrieve the user's choice:

```
int positionOfSelection = historyChoice.getSelectedIndex();
```

The variable `positionOfSelection` now holds the position of the selected item, and you can use the variable to access an array of dates so you can display the date that corresponds to the selected historical event. For example, if you declare the following, then `dates[positionOfSelection]` holds the year for the selected historical event:

```
int[] dates = {1776, 1941, 1969};
```

In addition to `JComboBoxes` for which users click items presented in a list, you can create `JComboBoxes` into which users' type text. To do this, you use the `setEditable()` method. A drawback to using an editable `JComboBox` is that the text a user types must exactly match an item in the list box. If the user misspells the selection or uses the wrong case, no valid value is returned from the `getSelectedIndex()` method. You can use an if statement to test the value returned from `getSelectedIndex()`; if it is negative, the selection did not match any items in the `JComboBox`, and you can issue an appropriate error message.

Summary

`JFrame` is a Swing container that resembles a window. Has a title bar and borders, and the ability to be resized, minimized, restored, and closed. `JCheckBox` Consists of a label positioned beside a square. `ButtonGroup` groups several components so the user can select only one at a time. `JComboBox` displays an area showing an option combined with a list box containing additional options.



Self-Evaluation Questions

1. Create a java form with all the controls you have learnt.

STUDY UNIT 7 ADVANCED GUI TOPICS

7.1 Introduction

Study unit 7 deals with advanced user interface topics: layout managers and event handling. Students have been introduced to both subjects in previous chapters. The event handling topics focus on handling keyboard and mouse events. Students will also learn to work with menus, colour, and scroll panes.

After studying this unit, you should be able to:

- Use content panes
- Use colour
- Learn more about layout managers
- Use JPanels to increase layout options
- Create JScrollPanels
- Understand events and event handling more thoroughly
- Use the AWTEvent class methods
- Handle mouse events
- Use menus

7.2 Understanding the Content Pane

The JFrame class is a top-level container Swing class. (The other two top-level container classes are JDialog and JApplet.) Every GUI component that appears on the screen must be part of a containment hierarchy. A containment hierarchy is a tree of components that has a top-level container as its root (that is, at its uppermost level). Every top-level container has a content pane that contains all the visible components in the container's user interface. The content pane can contain components like JButtons directly, or it can hold other containers, like JPanels, that in turn contain such components.

A top-level container can contain a menu bar. Conventionally, a menu bar is a horizontal strip that is placed at the top of a container and that contains user options. The menu bar, if there is one, is just above (and separate from) the content pane. A glass pane resides above the content pane. The relationship between a JFrame and its root, content, and glass panes. Whenever you create a JFrame (or other top-level container), you can get a reference to its content pane using the getContentPane() method. You added and removed components from JFrames and set their layout managers without understanding you were using the content pane. You had this ability because Java automatically converts add(), remove(), and setLayoutManager() statements to more complete versions. For example, the following three statements are equivalent within a class that descends from JFrame:

```
this.getContentPane().add(aButton);
```

```
getContentPane().add(aButton);
```

```
add(aButton);
```

In the first statement, this refers to the JFrame class in which the statement appears, and getContentPane() provides a reference to the content pane. In the second statement, the this reference is implied. In the third statement, both the this reference and the getContentPane() call are implied. Although you do not need to worry about the content pane if you only add components to, remove components from, or set the layout manager of a JFrame, you must refer to the content pane for all other actions, such as setting the background colour.

When you write an application that adds multiple components to a content pane, it is more efficient to declare an object that represents the content pane than to keep calling the getContentPane() method. For example, consider the following code in a JFrame class that adds three buttons:

```
getContentPane().add(button1);
getContentPane().add(button2);
getContentPane().add(button3);
```

You might prefer to write the following statements. The call to getContentPane() is made once, its reference is stored in a variable, and the reference name is used repeatedly with the call to the add() method:

```
Container con = getContentPane();
con.add(button1);
con.add(button2);
con.add(button3);
```

As an example, the class creates a JFrame like the ones you created throughout although to keep the example simple, the default close operation was not set and the button was not assigned any tasks.

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

public class JFrameWithExplicitContentPane extends JFrame
{
    private final int SIZE = 180;
    private Container con = getContentPane();
    private JButton button = new JButton("Press Me");
    public JFrameWithExplicitContentPane()
```

```

{
super("Frame");
setSize(SIZE, SIZE);
con.setLayout(new FlowLayout());
con.add(button);
}
public static void main(String[] args)
{
JFrameWithExplicitContentPane frame = new JFrameWithExplicitContentPane();
frame.setVisible(true);
}
}

```

The JFrameWithExplicitContentPane class

7.3 Using Colour

The Colour class defines colours for you to use in your applications. The Colour class can be used with the setBackground() and setForeground() methods of the Component class to make your applications more attractive and interesting. When you use the Colour class, you include the statement import java.awt.Colour; at the top of your class file. The Colour class defines named constants that represent 13 colours. Java constants are usually written in all uppercase letters, however, Java's creators declared two constants for every colour in the Colour class an uppercase version, such as BLUE, and a lowercase version, such as blue. Earlier versions of Java contained only the lowercase Colour constants. (The uppercase Colour constants use an underscore in DARK_GRAY and LIGHT_GRAY; the lowercase versions are a single word: darkgray and lightgray.)

You can also create your own Colour object with the following statement:

```
Colour someColour = new Colour(r, g, b);
```

In this statement, r, g, and b are numbers representing the intensities of red, green, and blue you want in your colour. The numbers can range from 0 to 255. For example, the colour black is created using r, g, and b values 0, 0, 0, and white is created by 255, 255, 255. The following statement produces a dark purple colour that has red and blue components, but no green.

```
Colour darkPurple = new Colour(100, 0, 100);
```

You can create more than 16 million custom colours using this approach. Some computers cannot display each of the 16 million possible colours; each computer displays the closest colour it can to the requested colour. You can discover the red, green, or blue components of any existing colour with the methods `getRed()`, `getGreen()`, and `getBlue()`. Each of these methods returns an integer. For example, you can discover the amount of red in MAGENTA by displaying the value of `Colour.MAGENTA.getRed()`; A short application that sets the background colour of a `JFrame`'s content pane and sets both the foreground and background colours of a `JButton`.

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

public class JFrameWithColour extends JFrame
{
    private final int SIZE = 180;
    private Container con = getContentPane();
    private JButton button =
        new JButton("Press Me");
    public JFrameWithColour()
    {
        super("Frame");
        setSize(SIZE, SIZE);
        con.setLayout(new FlowLayout());
        con.add(button);
        con.setBackground(Colour.YELLOW);
        button.setBackground(Colour.RED);
        button.setForeground(Colour.WHITE);
    }
    public static void main(String[] args)
    {
        JFrameWithColour frame =
            new JFrameWithColour();
        frame.setVisible(true);
    }
}
```



```
}  
  
}
```



7.4 Learning More About Layout Managers

A layout manager is an object that controls the size and position (that is, the layout) of components inside a Container object. The layout manager that you assign to the Container determines how the components are sized and positioned within it. Layout managers are interface classes that are part of the JDK; they align your components so the components neither crowd each other nor overlap. For example, you have already learned that the `FlowLayout` layout manager positions components in rows from left to right across their container. Other layout managers arrange components in equally spaced columns and rows or center components within their container. Each component you place within a Container can also be a Container itself, so you can assign layout managers within layout managers. The Java platform supplies layout managers that range from the very simple (`FlowLayout` and `GridLayout`) to the special purpose (`BorderLayout` and `CardLayout`) to the very flexible (`GridBagLayout` and `BoxLayout`).

7.5 Using BorderLayout

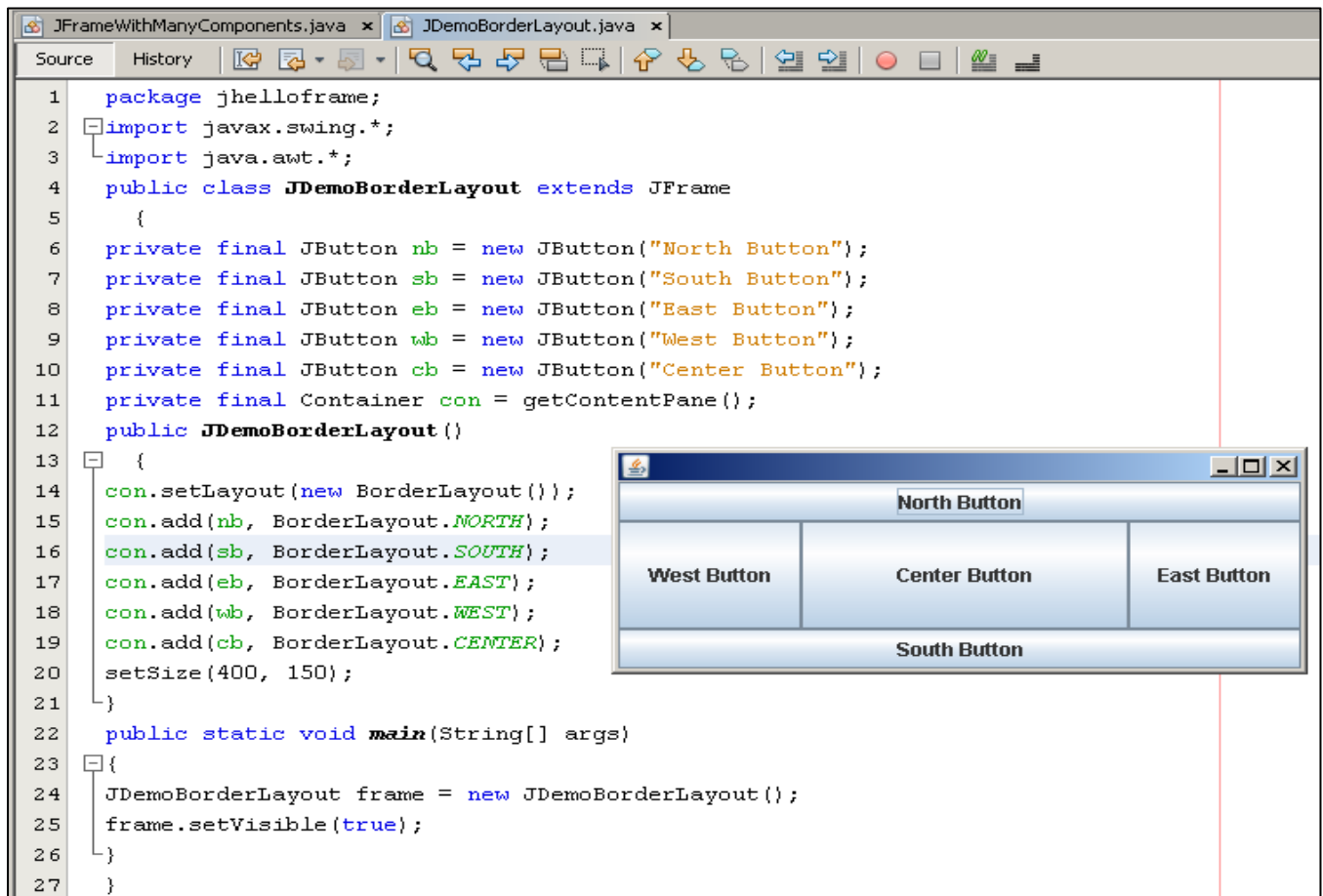
The `BorderLayout` manager is the default manager class for all content panes. You can use the `BorderLayout` class with any container that has five or fewer components. (However, any of the components could be a container that holds even more components.) When you use the `BorderLayout` manager, the components fill the screen in five regions: north, south, east, west, and center. A `JFrame` that contains five `JButton` objects that fill the five regions in a content pane that uses `BorderLayout`.



When you add a component to a container that uses `BorderLayout`, the `add()` method uses two arguments: the component and the region to which the component is added. The `BorderLayout` class provides five named constants for the regions `BorderLayout.NORTH`, `BorderLayout.SOUTH`, `BorderLayout.EAST`, `BorderLayout.WEST`, and `BorderLayout.CENTER`—or you can use the Strings those constants represent:

“North”, “South”, “East”, “West”, or “Center”.

Figure 7.1: BorderLayout



The JDemoBorderLayout class

When you place exactly five components in a container and use BorderLayout, each component fills one entire region. When the application runs, Java determines the exact size of each component based on the component's contents. When you resize a Container that uses BorderLayout, the regions also change in size. If you drag the Container's border to make it wider, the north, south, and center regions become wider, but the east and west regions do not change. If you increase the Container's height, the east, west, and center regions become taller, but the north and south regions do not change.

An anonymous BorderLayout object is created when the constructor is called from within the `setLayout()` method. Instead, you could declare a named BorderLayout object and use its identifier in the `setLayout()` call. However, it's not necessary to use either technique to specify BorderLayout because it is the default layout manager for all content panes; that's why, in many examples in the last chapter, you had to specify FlowLayout to acquire the easier-to-use manager. When you use BorderLayout, you are not required to add components into each of the five regions. If you add fewer components, any empty component regions disappear, and the remaining components expand to fill the available space. If any or all of the north, south, east, or west areas are left out, the center area spreads into the missing area or areas. However, if the center area is left out, the north, south, east, or west areas do not change. A

common mistake when using BorderLayout is to add a Component to a content pane or frame without naming a region. This can result in some of the components not being visible.

Using FlowLayout

Recall from the last chapter, Introduction to Swing Components, that you can use the FlowLayout manager class to arrange components in rows across the width of a Container. With FlowLayout, each Component that you add is placed to the right of previously added components in a row; or, if the current row is filled, the Component is placed to start a new row. When you use BorderLayout, the Components you add fill their regions that is, each Component expands or contracts based on its region's size. However, when you use FlowLayout, each Component retains its default size, or preferred size. For example, a JButton's preferred size is the size that is large enough to hold the JButton's text. When you use BorderLayout and then resize the window, the components change size accordingly because their regions change. When you use FlowLayout and then resize the window, each component retains its size, but it might become partially obscured or change position.

The FlowLayout class contains three constants you can use to align Components with a Container:

- FlowLayout.LEFT
- FlowLayout.CENTER
- FlowLayout.RIGHT

If you do not specify alignment, Components are center-aligned in a FlowLayout Container by default. An application that uses the FlowLayout.LEFT and FlowLayout.RIGHT constants to reposition JButtons. In this example, a FlowLayout object named layout is used to set the layout of the content pane. When the user clicks a button, the shaded code in the actionPerformed() method changes the alignment to left or right using the FlowLayout class setAlignment() method. The application when it starts, how the JButton Components are repositioned after the user clicks the L button, and how the Components are repositioned after the user clicks the R button.

```

1  package javaapplication90;
2  import javax.swing.*;
3  import java.awt.*;
4  import java.awt.event.*;
5  public class JDemoFlowLayout extends JFrame implements ActionListener
6  {
7      private final JButton lb = new JButton("L Button");
8      private final JButton rb = new JButton("R Button");
9      private final Container con = getContentPane();
10     private final FlowLayout layout = new FlowLayout();
11     public JDemoFlowLayout()
12     {
13         con.setLayout(layout);
14         con.add(lb);
15         con.add(rb);
16         lb.addActionListener(this);
17         rb.addActionListener(this);
18         setSize(500, 100);
19     }
20     public void actionPerformed(ActionEvent event)
21     {
22         Object source = event.getSource();
23         if(source == lb)
24             layout.setAlignment(FlowLayout.LEFT);
25         else
26             layout.setAlignment(FlowLayout.RIGHT);
27         con.invalidate();
28         con.validate();
29     }
30     public static void main(String[] args)
31     {
32         JDemoFlowLayout frame = new JDemoFlowLayout();
33         frame.setVisible(true);

```

The JDemoFlowLayout application



The JDemoFlowLayout application as it first appears on the screen, after the user chooses the L button, and after the user chooses the R button

Using GridLayout

If you want to arrange components into equal rows and columns, you can use the `GridLayout` manager class. When you create a `GridLayout` object, you indicate the numbers of rows and columns you want, and then the container surface is divided into a grid, much like the screen you see when using a spreadsheet program. For example, the following statement establishes an anonymous `GridLayout` with four horizontal rows and five vertical columns in a `Container` named `con`:

```
con.setLayout(new GridLayout(4, 5));
```

Specifying rows and then columns when you use `GridLayout` might seem natural to you, because this is the approach you take when defining two-dimensional arrays. As you add new `Components` to a `GridLayout`, they are positioned in sequence from left to right across each row. Unfortunately, you can't skip a position or specify an exact position for a component. (However, you can add a blank label to a grid position to give the illusion of skipping a position.) You also can specify a vertical and horizontal gap measured in pixels, using two additional arguments. For example, a `JDemoGridLayout` program that uses the shaded statement to establish a `GridLayout` with three horizontal rows and two vertical columns, and horizontal and vertical gaps of five pixels each. Five `JButton` `Components` are added to the `JFrame`'s automatically retrieved content pane.

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

public class JDemoGridLayout extends JFrame
{
    private JButton b1 = new JButton("Button 1");
    private JButton b2 = new JButton("Button 2");
    private JButton b3 = new JButton("Button 3");
    private JButton b4 = new JButton("Button 4");
    private JButton b5 = new JButton("Button 5");
    private GridLayout layout = new GridLayout(3, 2, 5, 5);
    private Container con = getContentPane();

    public JDemoGridLayout()
    {
        con.setLayout(layout);
        con.add(b1);
        con.add(b2);
```

```

con.add(b3);
con.add(b4);
con.add(b5);
setSize(200, 200);
}

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

```

The JDemoGridLayout class

The Components are placed into the pane across the three rows. Because there are six positions but only five Components, one spot remains unused. With GridLayout, you can specify the number of rows and use 0 for the number of columns to let the layout manager determine the number of columns, or you can use 0 for the number of rows, specify the number of columns, and let the layout manager calculate the number of rows. When trying to decide whether to use GridLayout or FlowLayout, remember the following:



Output of the JDemoGridLayout program

Use GridLayout when you want components in fixed rows and columns and you want the components' size to fill the available space.

- Use FlowLayout if you want Java to determine the rows and columns, do not want a rigid row and column layout, and want components to retain their “natural” size so their contents are fully visible.

Using CardLayout

The CardLayout manager generates a stack of containers or components, one on top of another, much like a blackjack dealer reveals playing cards one at a time from the top of a deck. Each component in the group is referred to as a card, and each card can be any component type for example, a JButton, JLabel, or JPanel. You use a CardLayout when you want multiple components to share the same display space. A card layout is created from the CardLayout class using one of two constructors:

- CardLayout() creates a card layout without a horizontal or vertical gap.
- CardLayout(int hgap, int vgap) creates a card layout with the specified horizontal and vertical gaps. The horizontal gaps are placed at the left and right edges. The vertical gaps are placed at the top and bottom edges.

For example, a JDemoCardLayout class that uses a CardLayout manager to create a stack of JButtons that contain the labels “Ace of Hearts”, “Three of Spades”, and “Queen of Clubs”. In the class constructor, you need a slightly different version of the add() method to add a component to a content pane whose layout manager is CardLayout. The format of the method is:

```
add(aString, aContainer);
```

In this statement, aString represents a name you want to use to identify the Component card that is added. In a program that has a CardLayout manager, a change of cards is usually triggered by a user's action. For example, in the JDemoCardLayout program, each JButton can trigger the actionPerformed() method. Within this method, the statement next(getContentPane()) flips to the next card of the container. (The order of the cards depends on the order in which you add them to the container.) You also can use previous(getContentPane());, first(getContentPane());, and last(getContentPane()); to flip to the previous, first, and last card, respectively. You can go to a specific card by using the String name assigned in the add() method call. For example, in the application the following statement would display the “Three of Spades” because “three” is used as the first argument when the b2 object is added to the content pane in the JDemoCardLayout constructor:

```
cards.show(getContentPane(), "three");
```

```
import javax.swing.*;
```

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

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

```
public class JDemoCardLayout extends JFrame
```

```
implements ActionListener
```

```

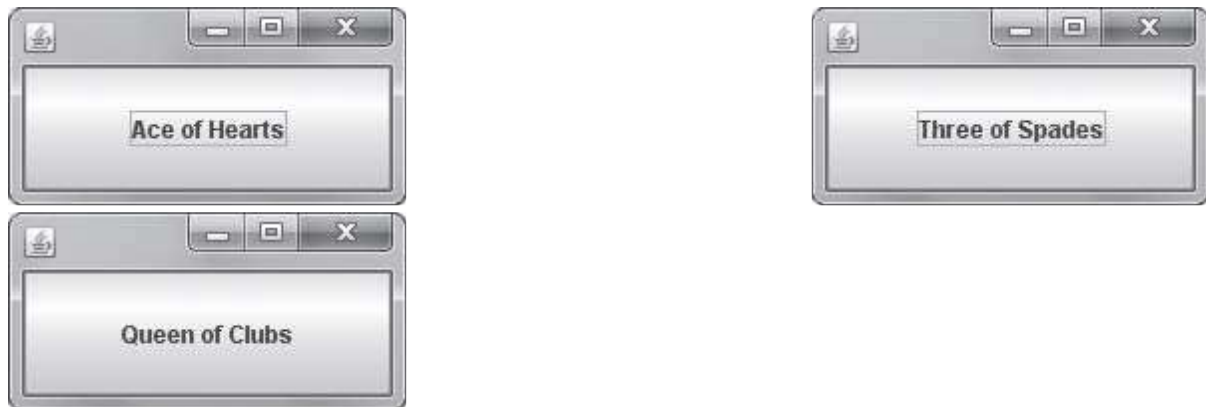
{
private CardLayout cards = new CardLayout();
private JButton b1 = new JButton("Ace of Hearts");
private JButton b2 = new JButton("Three of Spades");
private JButton b3 = new JButton("Queen of Clubs");
private Container con = getContentPane();
public JDemoCardLayout()
{
con.setLayout(cards);
con.add("ace", b1);
b1.addActionListener(this);
con.add("three", b2);
b2.addActionListener(this);
con.add("queen", b3);
b3.addActionListener(this);
setSize(200, 100);
}
public void actionPerformed(ActionEvent e)
{
cards.next(getContentPane());
}
public static void main(String[] args)
{
JDemoCardLayout frame = new JDemoCardLayout();
frame.setVisible(true);
}
}

```

The JDemoCardLayout class

The JDemoCardLayout program when it first appears on the screen, after the user clicks the button once, and after the user clicks the button a second time. Because each JButton is a card, each JButton consumes the entire viewing area in the container that uses the

CardLayout manager. If the user continued to click the card buttons, the cards would continue to cycle in order.



Using Advanced Layout Managers

Just as professional Java programmers are constantly creating new Components, they also create new layout managers. You are certain to encounter new and interesting layout managers during your programming career; you might even create your own. For example, when GridLayout is not sophisticated enough for your purposes, you can use GridBagLayout. The GridBagLayout manager allows you to add Components to precise locations within the grid, as well as to indicate that specific Components should span multiple rows or columns within the grid. For example, if you want to create a JPanel with six JButtons, in which two of the JButtons are twice as wide as the others, you can use GridBagLayout. This class is difficult to use because you must set the position and size for each component, and more than 20 methods are associated with the class. Visit <http://java.sun.com> for details on how to use this class.

Another layout manager option is the BoxLayout manager, which allows multiple components to be laid out either vertically or horizontally. The components do not wrap, so a vertical arrangement of components, for example, stays vertically arranged when the frame is resized. The Java Web site can provide you with details.

Using the JPanel Class

Using the BorderLayout, FlowLayout, GridLayout, and CardLayout managers would provide a limited number of screen arrangements if you could place only one Component in a section of the layout. Fortunately, you can greatly increase the number of possible component arrangements by using the JPanel class. A JPanel is a plain, borderless surface that can hold lightweight UI components. The inheritance hierarchy of the JPanel class. You can see that every JPanel is a Container; you use a JPanel to hold other UI components, such as JButtons, JCheckBoxes, or even other JPanels. By using JPanels within JPanels, you can create an infinite variety of screen layouts. The default layout manager for every JPanel is FlowLayout.

To add a component to a JPanel, you call the container's add() method, using the component as the argument. For example, the code that creates a JFrameWithPanels class that extends JFrame. A JButton is added to a JPanel named panel1, and two more JButtons are added to another JPanel named panel2. Then panel1 and panel2 are added to the JFrame's content pane.

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

public class JFrameWithPanels extends JFrame
{
    private final int WIDTH = 250;
    private final int HEIGHT = 120;
    private JButton button1 = new JButton("One");
    private JButton button2 = new JButton("Two");
    private JButton button3 = new JButton("Three");
    public JFrameWithPanels()
    {
        super("JFrame with Panels");
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        JPanel panel1 = new JPanel();
        JPanel panel2 = new JPanel();
        Container con = getContentPane();
        con.setLayout(new FlowLayout());
        con.add(panel1);
        con.add(panel2);
        panel1.add(button1);
        panel1.setBackground(Color.BLUE);
        panel2.add(button2);
        panel2.add(button3);
        panel2.setBackground(Color.BLUE);
        setSize(WIDTH, HEIGHT);
    }
}
```

```

}

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

```

The JFrameWithPanels class

The output of the JFrameWithPanels program. Two JPanel's have been added to the JFrame. Because this application uses the setBackground() method to make each JPanel's background blue, you can see where one panel ends and the other begins. The first JPanel contains a single JButton and the second one contains two JButtons. When you create a JPanel object, you can use one of four constructors. The different constructors allow you to use default values or to specify a layout manager and whether the JPanel is double buffered. If you indicate double buffering, which is the default buffering strategy, you specify that additional memory space will be used to draw the JPanel off screen when it is updated. With double buffering, a redrawn JPanel is displayed only when it is complete; this provides the viewer with updated screens that do not flicker while being redrawn. The four constructors are as follows:

- JPanel() creates a JPanel with double buffering and a flow layout.
- JPanel(LayoutManager layout) creates a JPanel with the specified layout manager and double buffering.
- JPanel(boolean isDoubleBuffered) creates a JPanel with a flow layout and the specified double-buffering strategy.
- JPanel(LayoutManager layout, boolean isDoubleBuffered) creates a JPanel with the specified layout manager and the specified buffering strategy.



Output of the JFrameWithPanels application

A Closer Look at Events and Event Handling

In the unit Introduction to Swing Components, you worked with ActionEvents and ItemEvents that are generated when a user works with a control that is included in one of your programs.

The parent class for all events is `EventObject`, which descends from the `Object` class. `EventObject` is the parent of `AWTEvent`, which in turn is the parent of specific event classes such as `ActionEvent` and `ItemEvent`. The abstract class `AWTEvent` is contained in the package `java.awt.event`. Although you might think it would have been logical for the developers to name the event base class `Event`, there is no currently active, built-in Java class named `Event` (although there was one in Java 1.0).

An Event-Handling Example: `KeyListener`

You use the `KeyListener` interface when you are interested in actions the user initiates from the keyboard. The `KeyListener` interface contains three methods:

`keyPressed()`, `keyTyped()`, and `keyReleased()`.

For most keyboard applications in which the user must press a keyboard key, it is probably not important whether you take resulting action when a user first presses a key, during the key press, or upon the key's release; most likely, these events occur in quick sequence. However, on those occasions when you don't want to take action while the user holds down the key, you can place the actions in the `keyReleased()` method. It is best to use the `keyTyped()` method when you want to discover which character was typed. When the user presses a key that does not generate a character, such as a function key (sometimes called an action key), `keyTyped()` does not execute. The methods `keyPressed()` and `keyReleased()` provide the only ways to get information about keys that don't generate characters. The `KeyEvent` class contains constants known as virtual key codes that represent keyboard keys that have been pressed. For example, when you type A, two virtual key codes are generated: Shift and "a". The virtual key code constants have names such as `VK_SHIFT` and `VK_ALT`. See the Java Web site for a complete list of virtual key codes. The example shows a `JDemoKeyFrame` class that uses the `keyTyped()` method to discover which key the user typed last.

```
import javax.swing.*;

import java.awt.*;

import java.awt.event.*;

public class JDemoKeyFrame extends JFrame
implements KeyListener
{
    private JLabel prompt = new JLabel("Type keys below:");
    private JLabel outputLabel = new JLabel();
    private JTextField textField = new JTextField(10);

    public JDemoKeyFrame()
    {
```

```

setTitle("JKey Frame");
setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
setLayout(new BorderLayout());
add(prompt, BorderLayout.NORTH);
add(textField, BorderLayout.CENTER);
add(outputLabel, BorderLayout.SOUTH);
addKeyListener(this);
textField.addKeyListener(this);
}

public void keyTyped(KeyEvent e)
{
    char c = e.getKeyChar();
    outputLabel.setText("Last key typed: " + c);
}

public void keyPressed(KeyEvent e)
{
}

public void keyReleased(KeyEvent e)
{
}

public static void main(String[] args)
{
    JDemoKeyFrame keyFrame = new JDemoKeyFrame();
    final int WIDTH = 250;
    final int HEIGHT = 100;
    keyFrame.setSize(WIDTH, HEIGHT);
    keyFrame.setVisible(true);
}
}

```

Using AWT Event Class Methods

In addition to the handler methods included with the event listener interfaces, the AWT Event classes themselves contain many other methods that return information about an event. For example, the `ComponentEvent` class contains a `getComponent()` method that allows you to determine which of multiple Components generates an event. The `WindowEvent` class contains a

similar method, `getWindow()`, that returns the Window that is the source of an event. All Components have these methods:

- `addComponentListener()`
- `addFocusListener()`
- `addMouseListener()`
- `addMouseMotionListener()`

Handling Mouse Events

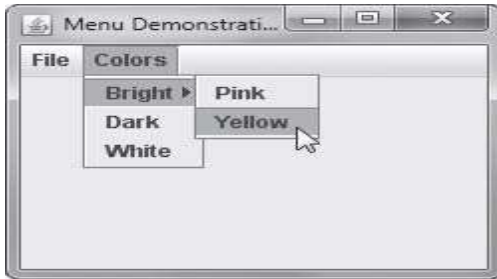
Even though Java program users sometimes type characters from a keyboard, when you write GUI programs you probably expect users to spend most of their time operating a mouse. The `MouseMotionListener` interface provides you with methods named `mouseDragged()` and `mouseMoved()` that detect the mouse being rolled or dragged across a component surface. The `MouseListener` interface provides you with methods named `mousePressed()`, `mouseClicked()`, and `mouseReleased()` that are analogous to the keyboard event methods `keyPressed()`, `keyTyped()`, and `keyReleased()`. With a mouse, however, you are interested in more than its button presses; you sometimes simply want to know where a mouse is pointing. The additional interface methods `mouseEntered()` and `mouseExited()` inform you when the user positions the mouse over a component (entered) or moves the mouse off a component (exited). The `MouseListener` interface implements all the methods in both the `MouseListener` and `MouseMotionListener` interfaces; although it has no methods of its own, it is a convenience when you want to handle many different types of mouse events.

Using Menus

Menus are lists of user options; they are commonly added features in GUI programs. Application users are accustomed to seeing horizontal menu bars across the tops of frames, and they expect to be able to click those options to produce drop-down lists that display more choices. The horizontal list of JMenus is a `JMenuBar`. Each `JMenu` can contain options, called `JMenuItems`, or can contain submenus that also are `JMenus`. A `JFrame` that illustrates the use of the following components:

- A `JMenuBar` that contains two `JMenus` named File and Colours.
- Three items within the Colours `JMenu`: Bright, Dark, and White. Dark and White are `JMenuItems`. Bright is a `JMenu` that holds a submenu. You can tell that Bright is a

submenu because an arrow sits to the right of its name, and when the mouse hovers over Bright, two additional JMenuItems appear: Pink and Yellow.



To create the output a series of JMenuBar, JMenu, and JMenuItem objects were created and put together in stages. You can create each of the components you see in the menus as follows:

- You can create a JMenuBar much like other objects—by using the new operator and a call to the constructor, as follows:

```
JMenuBar mainBar = new JMenuBar();
```

- You can create the two JMenus that are part of the JMenuBar:

```
JMenu menu1 = new JMenu("File");
```

```
JMenu menu2 = new JMenu("Colours");
```

- The three components within the Colours JMenu are created as follows:

```
JMenu bright = new JMenu("Bright");
```

```
JMenuItem dark = new JMenuItem("Dark");
```

```
JMenuItem white = new JMenuItem("White");
```

- The two JMenuItem objects that are part of the Bright JMenu are created as follows:

```
JMenuItem pink = new JMenuItem("Pink");
```

```
JMenuItem yellow = new JMenuItem("Yellow");
```

Once all the components are created, you assemble them.

- You add the JMenuBar to a JFrame using the setJMenuBar() method as follows:

```
setJMenuBar(mainBar);
```

Using the setJMenuBar() method assures that the menu bar is anchored to the top of the frame and looks like a conventional menu bar. Notice that the JMenuBar is not added to a JFrame's content pane; it is added to the JFrame itself.

- The JMenus are added to the JMenuBar using the add() method. For example:

```
mainBar.add(menu1);
```

```
mainBar.add(menu2);
```

- A submenu and two JMenuItem objects are added to the Colours menu as follows:

```
menu2.add(bright);
```

```
menu2.add(dark);
```

```
menu2.add(white);
```

- A submenu can contain its own JMenuItem objects. For example, the Bright JMenu that is part of the Colours menu contains its own two JMenuItem objects:

```
bright.add(pink);
```

```
bright.add(yellow);
```

A complete working program that creates a frame with a greeting and the JMenu .

```
import javax.swing.*;
```

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

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

```
import java.awt.Color;
```

```
public class JMenuFrame extends JFrame implements ActionListener
```

```
{
```

```
private JMenuBar mainBar = new JMenuBar();
```

```
private JMenu menu1 = new JMenu("File");
```



```

private JMenu menu2 = new JMenu("Colours");
private JMenuItem exit = new JMenuItem("Exit");
private JMenu bright = new JMenu("Bright");
private JMenuItem dark = new JMenuItem("Dark");
private JMenuItem white = new JMenuItem("White");
private JMenuItem pink = new JMenuItem("Pink");
private JMenuItem yellow = new JMenuItem("Yellow");
private JLabel label = new JLabel("Hello");
public JMenuFrame()
{
setTitle("Menu Demonstration");
setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
setLayout(new FlowLayout());
setJMenuBar(mainBar);
mainBar.add(menu1);
mainBar.add(menu2);
menu1.add(exit);
menu2.add(bright);
menu2.add(dark);
menu2.add(white);
bright.add(pink);
bright.add(yellow);
exit.addActionListener(this);
dark.addActionListener(this);
white.addActionListener(this);
pink.addActionListener(this);
yellow.addActionListener(this);
add(label);
label.setFont(new Font("Arial", Font.BOLD, 26));
}
public void actionPerformed(ActionEvent e)

```

```

{
Object source = e.getSource();
Container con = getContentPane();
if(source == exit)
System.exit(0);
else if(source == dark)
con.setBackground(Colour.BLACK);
else if(source == white)
con.setBackground(Colour.WHITE);
else if(source == pink)
con.setBackground(Colour.PINK);
else con.setBackground(Colour.YELLOW);
}

    public static void main(String[] args)
{
JMenuFrame mFrame = new JMenuFrame();
final int WIDTH = 250;
final int HEIGHT = 200;
mFrame.setSize(WIDTH, HEIGHT);
mFrame.setVisible(true);
}
}

```

The JMenuFrame class

Using JCheckBoxMenuItem and JRadioButtonMenuItem Objects

The JCheckBoxMenuItem and JRadioButtonMenuItem classes derive from the JMenuItem class. Each provides more specific menu items as follows:

- JCheckBoxMenuItem objects appear with a check box next to them. An item can be selected (displaying a check mark in the box) or not. Usually, you use check box items to turn options on or off.

- JRadioButtonMenuItem objects appear with a round radio button next to them. Users usually expect radio buttons to be mutually exclusive, so you usually make radio buttons part of a ButtonGroup. Then, when any radio button is selected, the others are all deselected.

The state of a JCheckBoxMenuItem or JRadioButtonMenuItem can be determined with the isSelected() method, and you can alter the state of the check box with the setSelected() method.

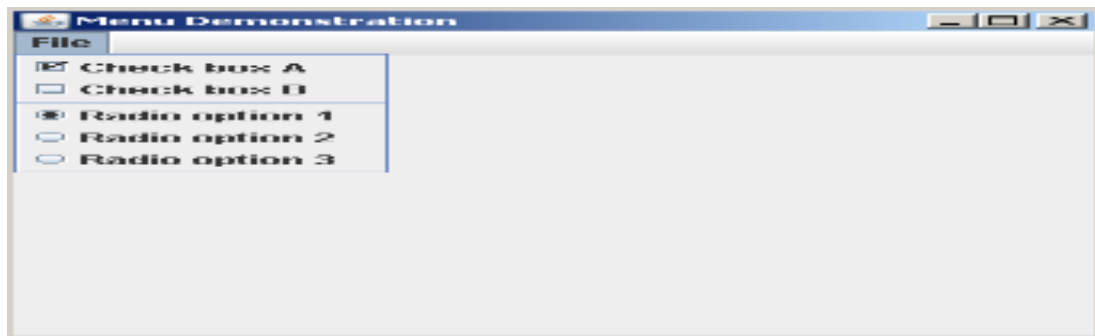
A JMenuFrame2 application in which two JCheckBoxMenuItems and three JRadioButtonMenuItems have been added to a JMenu. The controls have not yet been assigned any tasks, but the menu looks when the application executes.

```

1  package javamenuframes;
2  import javax.swing.*;
3  import java.awt.*;
4  public class JMenuFrame2 extends JFrame
5  {
6      private final JMenuBar mainBar = new JMenuBar();
7      private final JMenu menu1 = new JMenu("File");
8      private final JCheckBoxMenuItem check1 = new
9      JCheckBoxMenuItem("Check box A");
10     private final JCheckBoxMenuItem check2 = new
11     JCheckBoxMenuItem("Check box B");
12     private final JRadioButtonMenuItem radio1 = new
13     JRadioButtonMenuItem("Radio option 1");
14     private final JRadioButtonMenuItem radio2 = new
15     JRadioButtonMenuItem("Radio option 2");
16     private final JRadioButtonMenuItem radio3 = new
17     JRadioButtonMenuItem("Radio option 3");
18     private final ButtonGroup group = new ButtonGroup();
19     public JMenuFrame2()
20     {
21         setTitle("Menu Demonstration");
22         setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
23         setLayout(new FlowLayout());
24         setJMenuBar(mainBar);
25         mainBar.add(menu1);
26         menu1.add(check1);
27         menu1.add(check2);
28         menu1.addSeparator();
29         menu1.add(radio1);
30         menu1.add(radio2);
31         menu1.add(radio3);
32         group.add(radio1);
33         group.add(radio2);
34         group.add(radio3);
35     }
36     public static void main(String[] args)
37     {
38         JMenuFrame2 frame = new JMenuFrame2();
39         final int WIDTH = 150;
40         final int HEIGHT = 200;
41         frame.setSize(WIDTH, HEIGHT);
42         frame.setVisible(true);
43     }
44 }

```

The JMenuFrame2 application



The **output** for *JMenuFrame2* application above

Summary

The event handling topics focus on handling keyboard and mouse events. Students were taught to learn to work with menus, colour, and scroll panes. The `JFrame` class is a top-level container Swing class. (The other two top-level container classes are `JDialog` and `JApplet`.) Every GUI component that appears on the screen must be part of a containment hierarchy

References

- Bishop, J. & Horspool, N., 2006. Cross-platform development: Software that lasts. *Computer*, 39(10), pp. 26-35.
- Budd, T., 2002. *Understanding Object-oriented programming with Java..* s.l.:Pearson Education.
- Cifuentes, C., Simon, D. & Fraboulet, A., 1998. *Assembly to high-level language translation..* IEEE.
- Farell, J., 2016. *Microsoft Visual C#: 2015 An Introduction to Object.* 6th ed. Chicago: cengage.
- Farell, J., 2018. *Microsoft Visual C#: 2017 An Introduction to Object-Oriented Programming.* Chicago: Cengage.