Growing Algorithms And Data Structures

(Fourth Edition)

GROWING ALGORITHMS AND DATA STRUCTURES

INTRODUCTION

Courses such as COMP 1010 and COMP 1012 focus on developing algorithms using primitive data types (ints, doubles, etc.), conditions, loops, arrays, and methods. These are the basic building blocks of computer programming and this style of programming is referred to as "**procedural programming**". In COMP 1020, we examine how these building blocks can be put together to solve more complex problems. At the same time, we also make the transition from procedural programming to object-oriented programming. Writing good object-oriented programs is not easy, it takes some time to stop thinking in a procedural manner and start thinking in an object-oriented manner.

Our study of algorithms concentrates on the development of algorithms that work correctly and are easy to understand. We will not be concerned with developing algorithms that are as efficient as possible. We will note where certain algorithms are inefficient and could be improved but we will not attempt to make each algorithm optimal in terms of its memory requirements and/or execution time.

In COMP 1010 and COMP 1012, the organization of each program in an assignment was normally described in class. In this course, the organization of solution for assignments will not be given; instead, the student is expected to develop the structure of each program by him/herself (with some hints from the instructor). Instead of developing the entire program at once, the program will be developed ("grown") slowly. We will concentrate on "**doing the simplest thing that works**", in other words, develop a simple, incomplete version of the algorithm that executes correctly and then gradually improve the algorithm until it provides the required functionality (**start small and grow slowly**). We will not add functionality that is not explicitly required!

> "Simple systems are easier to build, easier to maintain, smaller, and faster than complex ones. A simple system 'maximizes the work done', by increasing 'the amount of work not done'. A program must do exactly what's required by the user. Adding unasked-for-functionality dramatically increases the development time and decreases stability."
>
> Allen Holub, *Holub on Patterns*, p. 6.

David Scuse
Department of Computer Science
University of Manitoba

June, 2011

1 PROCEDURAL PROGRAMMING

1.1 Introduction

In this chapter, we review the basic constructs used to build procedural (no objects) programs. This chapter includes both Python and Java examples to enable students who were introduced to procedural programming using the Python language to make the transition to Java. The programs in these notes were developed using Python version 3.2 and Java version 6.

In the following sections, we develop programs in both Python and Java – these programs are stored in source-code files; although Python statements may also be entered at a Python command prompt; we will not take advantage of this technique.

1.2 Programming

A program is a collection of statements in a programming language. The statements are executed sequentially, beginning with the first statement and continuing statement by statement until the last statement in the program is encountered (although control structures may be used to change the order in which statements are executed). The process of developing programs is called programming (naturally) and the process of removing errors in programs is referred to as "debugging".

In the following sections, we will briefly review the programming constructs used to build procedural programs and examine how programs can be built from these fairly simple constructs. The following comment from the book *Think Python*, *How to Think Like a Computer Scientist* provides a good description of the thought processes used while developing programs:

> "The goal of this book is to teach you to think like a computer scientist. This way of thinking combines some of the best features of mathematics, engineering, and natural science. Like mathematicians, computer scientists use formal languages to denote ideas (specifically computations). Like engineers, they design things, assembling components into systems and evaluating tradeoffs among alternatives. Like scientists, they observe the behavior of complex systems, form hypotheses, and test predictions.
>
> The single most important skill for a computer scientist is **problem solving**. Problem solving means the ability to formulate problems, think creatively about solutions, and express a solution clearly and accurately. As it turns out, the process of learning to program is an excellent opportunity to practice problem-solving skills."
>
> Allen Downey, *Think Python*, pp. 1.
>
> http://greenteapress.com/thinkpython

1.3 Hello World

The "Hello World" program is the program that is most frequently used to illustrate a new programming language. Hello World is an important program not only because it provides a starting point when learning a new language but also because it ensures that the necessary infrastructure (compiler, paths to libraries, etc.) is set up correctly. The following **Python** program prints `Hello World!`.

```
print("Hello World!")
```

In these notes, the output from most programs or program segments is shown directly after the program. So the program above generates the output:

```
Hello World!
```

If any error messages are generated as the program is executed, the error messages are displayed for the programmer. The following program generates an error because the statement `printer` is not defined in Python.

```
printer("Hello World!")
```

```
Traceback (most recent call last):
File "C:\HelloWorld.py", line 1, in <module> printer("Hello World!")
NameError: name 'printer' is not defined
```

The following **Java method** prints `Hello World!`. Note that Java statements must be terminated by a semi-colon (`;`):

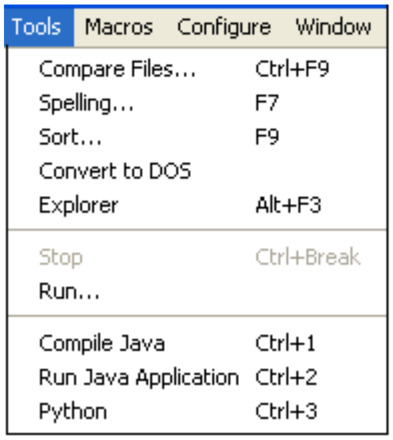
```
public static void main(String[] parms)
{
     System.out.println("Hello World!");
}
```

However, this method is not a complete Java program; Java methods must be defined within a class before they can be compiled and executed correctly. The following program defines a class named `HelloWorld` that is placed around the method. By convention, the first letter of the name of each Java class is defined in upper case. Note that in Java, the name of the class must be the same as the name of the file in which the class is defined. So the program below would be defined in a file named `HelloWorld.java`.

```
public class HelloWorld
{
     public static void main(String[] parms)
     {
        System.out.println("Hello World!");
     }
}
```

Java uses a two-step process in which a program is first compiled and then, if there are no compilation errors, the program can be executed. The compilation and the execution are separate steps so two commands must be issued to compile and execute a Java program. When a Java source file is compiled, Java creates a corresponding file called a "class" file (in the same directory as the source file). If the source file is named `HelloWorld.java`, the class file will be named `HelloWorld.class`. The class file is used when the program is executed.

In these notes, all programs are run using the (Windows) TextPad environment; if you are using a different system, your output may be displayed in a slightly different manner. As can be seen from the following TextPad Tools menu, a Java program is first compiled using `Ctrl+1` and is then executed using `Ctrl+2`. The Python tool was manually added to TextPad so depending on how you configured Python, the Python entry is not necessarily in the TextPad Tools menu, or, if it is, it is not necessary Tool 3.



1.4 Comments

The Python language uses the `#` character to mark the beginning of a comment. All characters on the current line that follow the `#` character are ignored by Python.

```
# This is my first Python program

print("Hello World!")
```

The Java language supports two types of comments – the single-line comment that begins with `//` and a multiline comment that is surrounded by the characters `/*` and `*/`

```
// This is my first Java program
public class HelloWorld
{
     public static void main(String[] parms)
     {
        System.out.println("Hello World!");
     }
}
```

```
/*
This is my first Java program
*/

public class HelloWorld
{
     public static void main(String[] parms)
     {
         System.out.println("Hello World!");
     }
}
```

1.5 Assignment Statement

The assignment statement is one of the fundamental constructs in programming languages. It is used to assign a value to a variable. A variable in Python is a name that begins with an alphabetic character and is not one of the Python reserved keywords. A variable in Java follows the same rules (except that the variable must not be the same as one of the Java reserved keywords).

The following Python program assigns the integer value `25` to the variable `i`. The program then prints the value of `i`. (Again, remember that the output of the program is shown after the program – the line that contains `25` is the output of the program, not part of the program.)

```
i = 25
print(i)

25
```

The two Python statements do not contain a statement terminator. However, the semi-colon (`;`) is used in Python as a statement delimiter (when more than one Python statement is on the same line); so the following Python program is equivalent to the one above.)

```
i = 25; print(i)

25
```

In Java, the equivalent statements are:

```
int i;
i = 25;
System.out.println(i);
```

Again, unlike Python, Java statements must be defined within a method which is defined within a class. So the complete Java program is:

```
public class Program1
{
     public static void main(String[] parms)
```

```
      {
         int i;
         i = 25;
         System.out.println(i);
      }
}

25
```

In the remainder of this chapter, we will not include the Java method declarations or Java class declarations unless it is helpful to do so.

In addition to the method and class information that must be defined in a Java program, the Java program above also includes the statement

```
int i;
```

This statement declares the variable/identifier `i` to be of type `int`. The data type `int` is common to both Python and Java; the difference is that in Python, variables are not declared before they are used; in Java, variables must be declared before they are used. Also, in Java, variables declared to be of a specific type may only be assigned values of that type; in Python, since variables are not declared to be of a specific type, they may be assigned a value of any type (this is referred to as "dynamic typing").

Variables in both Python and Java are **case sensitive** which means that the variable `apple` is not the same variable as `Apple`.

The **arithmetic** operators that manipulate **integers** in Python are the same as those in Java:

Python	**Java**	**Meaning**
+	+	addition
-	-	subtraction
*	*	multiplication
// (Python 3)	/	integer (truncating) division
/	(must cast to a real value)	real division
%	%	modulo division (remainder)
**	Math.pow(base,exponent)	exponentiation

Arithmetic expressions are formed in the same manner in both languages. The following arithmetic expressions are the same in both Python and Java. Parentheses may be used to force the order of the evaluation of expressions.

```
2 + 3
(4 * 5) + 10
(25 - 3) * (-10 +100)
```

Both Python and Java support shortcuts when performing some of the basic assignment operations.

Python	**Java**	**Meaning**
NA	`i++`	`i = i + 1`
NA	`i--`	`i = i - 1`
`i += j`	`i += j`	`i = i + j`
`i -= j`	`i -= j`	`i = i - j`
`i *= j`	`i *= j`	`i = i * j`
`i //= j`	`i /= j`	`i = i / j`

We will examine data types in more detail later in this chapter but for now, we will work with simple integers.

1.6 Conditional Execution

In Python, conditional execution is defined with an `if` statement. Indentation is used to indicate that one or more statements (a "suite" in Python terms) are within the true portion of the if statement or within the false/else portion of the if statement. Note that there is a colon `:` at the end of the `if` header and also at the end of the `else` header.

```
i = 2
if i > 0:
     print(i, "is greater than zero")
else:
     print(i, "is not greater than zero")

2 is greater than zero
```

In Java, conditional execution is similar but requires different delimiters: curly brackets `{` and `}` are used indicate that statements are within the true portion of the if statement or within the false portion. The indentation of statements within the curly brackets is a programming convention but is not strictly necessary.

```
int i;
i = 2;
if (i > 0)
{
     System.out.println(i +" is greater than zero");
}
else
{
     System.out.println(i +" is not greater than zero");
}

2 is greater than zero
```

Note that the Java `println` statement does not add a blank character between parameters; if a blank is required, it must be included by the programmer.

In Python, multiple statements may be included inside the true portion of an if statement and/or inside the false/else portion of an if statement. Indentation is used to indicate that statements are contained within the true portion or the false portion of the if statement.

```
i = 2
if i > 0:
     print(i, end=" ")
     print("is greater than zero")
else:
     print(i, end=" ")
     print("is not greater than zero")

2 is greater than zero
```

In Python, the `print` function causes a newline character to be generated after the last character is printed unless the parameter `end=" "` is included at the end of the parameter list.

In Java, the statements are very similar. Multiple statements may be included within the curly brackets that enclose the true portion and/or the false/else portion of the if statement.

```
int i;
i = 2;
if (i > 0)
{
     System.out.print(i);
     System.out.println(" is greater than zero");
}
else
{
     System.out.print(i);
     System.out.println(" is not greater than zero");
}
```

In Java, the `print` statement does not generate a newline character at the end of the characters that are printed, while the `println` statement does generate a newline character at the end of the printed characters.

The logical expression (or Boolean expression) in the if statement is an expression that evaluates to either true or false. In the programs above, a simple logical expression that compares integer values illustrates the use of the numeric comparison operators.

The numeric **comparison** operators in Python are the same as those in Java:

Python	**Java**	**Meaning**
==	==	equal
!=	!=	not equal
<	<	less than

`>`	`>`	`greater than`
`<=`	`<=`	`less than or equal`
`>=`	`>=`	`greater than or equal`

More complex logical expressions can be created by including logical operators that join numeric (relational) expressions.

The **logical** operators in Python use the name of the operator while those in Java use symbols:

Python	**Java**	**Meaning**
`and`	`&&`	`and`
`or`	`||`	`or`
`not`	`!`	`not`

The result of evaluating a logical expression is a Boolean value (see the Section: Data Types):

Python	**Java**	**Meaning**
`True`	`true`	`true`
`False`	`false`	`false`

Python supports some shortcuts when defining logical expressions. For example, the expression below is a valid Python expression.

```
3 <= i <= 12
```

The expression above is not valid in Java but the statement below is equivalent and is also valid in Python:

```
(3 <= i) && (i <= 12)
```

At times it is necessary to have more than 2 conditions. Python supports this with the `elif` statement.

```
i = 2
if i > 0:
     print(i, "is a positive number")
elif i < 0:
     print(i, "is a negative number")
else:
     print(i, "is zero")

2 is a positive number
```

There may be any number of `elif` statements and the `else` statement is optional.

Java does not support the `elif` statement; instead, additional `if` statements are used.

```
int i;
i = 2;
if (i > 0)
{
     System.out.println(i +" is a positive number");
}
else if (i < 0)
{
     System.out.println(i +" is a negative number");
}
else
{
     System.out.println(i +" is equal to zero");
}

2 is a positive number
```

`if` statements may be nested within other control structures (such as `if` statements). Again, the indentation of the statements in Python is important.

```
i = -20000000
if i > 0:
     if i > 1000000:
         print(i, "is a big positive number")
     else:
         print(i, "is a positive number")
elif i < 0:
     if i < -1000000:
         print(i, "is a big negative number")
     else:
         print(i, "is a negative number")
else:
     print(i, "is zero")

-20000000 is a big negative number
```

In Java, `if` statements may also be nested within other control structures.

```
i = -20000000;
if (i > 0)
{
     if (i > 1000000)
     {
         System.out.println(i +" is a big positive number");
     }
     else
     {
         System.out.println(i +" is a positive number");
     }
}
else if (i < 0)
{
     if (i < -1000000)
     {
         System.out.println(i +" is a big negative number");
     }
     else
     {
        System.out.println(i +" is a negative number");
     }
}
else
{
     System.out.println(i +" is equal to zero");
}

-20000000 is a big negative number
```

1.7 Loops

A loop is a control structure that causes the statements inside the loop to be executed a specified number of times. For example, the following Python loop causes the `print` statement inside the loop to be executed 5 times, with the variable `i` being assigned the values `0, 1, 2, 3,` and `4`. Again, note that the `for` statement header is terminated with a colon.

```
for i in range(0, 5):
     print(i)

0
1
2
3
4
```

The corresponding program in Java is very similar.

```
int i;
for (i=0; i<5; i++)
{
     System.out.println(i);
}
```

Both languages also support a while loop.

```
# Python while loop
i = 0
while i < 5:
     print(i)
     i += 1
```

```
// Java while loop
i = 0;
while (i < 5)
{
     System.out.println(i);
     i += 1;  // or i++;
}
```

In Python, if a loop may terminate before all of the values in the for statement are exhausted, the loop must be written using a `while` statement. The following program is to print all values beginning with `n` and continuing until a value that is a multiple of `20` is found. If such a value is found, the loop is to terminate without printing any additional values.

```
n = 25
i = n
stop = 0
while (i < 50) and (stop==0):
     print(i, end=" ")
     if ((i//20)*20)==i:
        stop = 1
     i += 1
print()

25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40
```

Although Python (and Java) supports a `break` statement that permits the programmer to exit from a loop at any time, using a `break` statement is a poor programming practice and it will not be used in these notes.

In Java, the `for` statement supports the addition of extra clauses that determine when the loop is to terminate so it is not necessary to write the loop using a `while` statement.

```
int n;
int i;
int stop;

n = 25;
stop = 0;
for (i=n; (i<50)&&(stop==0); i++)
{
     System.out.print(i +" ");
     if ((i/20*20)==i)
     {
        stop = 1;
     }
}
System.out.println();

25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40
```

Both Python and Java support control structures nested within other control structures so it is valid to have a loop within a loop.

1.8 Functions/Methods

Python and Java both support functions that are very similar in appearance. In Python, the function is declared before it is referred to. The definition of a function may include any number of parameters passed from the calling function.

The following Python program shows how a function that prints the integer values from `0 to (n-1)` is defined and then called from the main function.

```
def print_loop(n):
     for i in range(0, n):
        print(i)
     return

print_loop(5)
```

If a function does not return a value, it is often referred to as a “void function”. A void function may include a `return` statement that does not include a return value; however, the return statement is not required.

In Java, functions are referred to as “methods”. For now, all Java methods will be specified with the keyword `static`. The type of value returned by a method is either `void` (meaning the method does not return a value) or a valid Java data type (such as `int`). A `void` method in Java is the same as a Python void function – both may include a `return` statement that does not specify a value to be returned. In the following example, the class definition is included around the two methods.

```
public class FunctionTest
{
     public static void main(String[] parms)
     {
        print_loop(5);
     }

     public static void print_loop(int n)
     {
        int i;

        for (i=0; i<n; i++)
        {
           System.out.println(i);
        }
     }
}
```

In Java, a method may be defined either before or after it is referred to.

1.9 Lists/Arrays

An array is a programming construct that can store a collection of values. In Python, the more generic term “collection” is used instead of array. Python has different types of collections but we will focus on the “list” collection in this chapter. The following Python statement creates a Python list that contains the integer values `1, 2, 3,` and `4`.

```
my_list = [1, 2, 3, 4]
```

The advantage of using a list (instead of individual integer values) is that the collection may be passed to a function by specifying only the name of the list (instead of having to pass all of the values individually).

The following Python program creates a list and then passes the list to the function `print_list` which prints the elements of the list on the same line.

```
def print_list(my_list):
     for i in range(0, len(my_list)):
        print(my_list[i], end=" ")
     print()
     return

my_list = [1, 2, 3, 4]
print_list(my_list)

1 2 3 4
```

Python lists are "zero-based" (or "0-based"), meaning that the first element in the list is element zero and is accessed by `my_list[0]` in the program above. The variable `i` in the program above is used as a "subscript" – a value that specifies the position of an element in a list that is to be accessed.

Python provides a variation of the `for` statement that makes the manipulation of lists easier. The version below uses the modified `for` statement that specifies that the variable `i` is to be assigned one value from the list each time through the loop. In this program, the variable `i` is no longer a subscript, `i` contains an element of the list.

```
def print_list(my_list):
     for i in my_list:
        print(i, end=" ")
     print()
     return
```

The following Java methods illustrate the statements required to initialize a Java array and then print the contents of the array using a Java method.

```
public static void main(String[] parms)
{
     int[] my_list;
     my_list = new int[] {1, 2, 3, 4};
     print_list(my_list);
}

public static void print_list(int[] my_list)
{
     int i;

     for (i=0; i<my_list.length; i++)
     {
        System.out.print(my_list[i] +" ");
     }
     System.out.println();
}

1 2 3 4
```

As can be seen, Java requires more effort to declare the array, to pass the array as a parameter, and to manipulate the array, but the structure of the program is very similar to the structure of the Python program.

An array variable is declared in Java by placing square brackets `[` and `]` after the data type that the array is to contain. The parameter of the `print_list` function must be declared to be of the desired type, in this case, an integer array `int[]`. The number of elements in the array is determined in a manner similar to that of Python; in Python, `len(my_list)` is used while in Java, `my_list.length` is used.

Lists/arrays can be modified in both Python and Java but Python provides signiciantly more flexibility than does Java.

The following Python program creates a new list and initializes the elements to the values `0, 1, 2, …, n-1` where `n` is passed as a parameter to the `create_list` function.

```
def print_list(my_list):
     for i in my_list:
        print(i, end=" ")
     print()
     return

def create_list(n):
     my_list = []
     for i in range(0, n):
        my_list.append(i)
     return my_list

my_list = create_list(5)
print_list(my_list)

0 1 2 3 4
```

The statement `my_list = []` is used to create a new, empty list. The Python `append` function is used to append an element to the end of an existing list. So the effect of this program is to create a list that consists of the values `0, 1, 2, 3, 4` and then print that list.

The equivalent Java program is shown below.

```
public static void main(String[] parms)
{
     int[] my_list;

     my_list = create_list(5);
     print_list(my_list);
}

public static int[] create_list(int n)
{
     int[] list;
     int i;
     list = new int[n];
     for (i=0; i<n; i++)
     {
        list[i] = i;
     }
     return list;
}


public static void print_list(int[] list)
{
     int i;

     for (i=0; i<list.length; i++)
     {
        System.out.print(list[i] +" ");
     }
     System.out.println();
}

0 1 2 3 4
```

Although the two programs are very similar (ignoring the syntactic differences), there is one significant difference. In Python, a list is a dynamic structure: a list is typically created as an empty list and then elements are added to the end of the list using the `append` function. In Java, an array is a static structure: it is given an initial size and this size can not be modified. Once a Java array has been created, the elements of the array are modified by replacing their initial values. In the Java program above, the array is declared to be of size `n` (which is `5` in this example). At this point, the array contains 5 elements (which are initialized to zero by Java). Any of these 5 elements may be modified using an assignment statement such as: `list[3] = 4;` However, the size of the array is set to 5 and this can not be changed.

The following function illustrates how the maximum value in a Python list can be determined.

```
def max_list(my_list):
     max = my_list[0]
     for i in my_list:
        if i > max:
           max = i
     return max
```

The program does have some warts in it – if the list is empty, the program terminates abnormally; also, the first element in the list is processed twice. However, these problems can be fixed with a little effort.

The equivalent Java method is shown below.

```
public static int max_list(int[] my_list)
{
     int max;
     int i;

     max = my_list[0];
     for (i=1; i<my_list.length; i++)
     {
        if (my_list[i] > max)
        {
           max = my_list[i];
        }
     }
     return max;
}
```

Python contains functions that permit the programmer to insert a new element at any location in an existing list, delete an element from a list, and append to the end of a list. These functions do not exist with arrays in Java but can be written by the programmer. The following Java method `remove_element` removes an element from an existing array (assuming that the element exists in the array). It is important to note that the parameter passed to `remove_element` is the value of the element to be deleted, not the position of the element to be deleted. The method `System.arraycopy` is described in more detail in the next chapter.

```
public static void main(String[] parms)
{
     int[] my_list;

     my_list = create_list(5);
     print_list(my_list);
     System.out.println();

     my_list = remove_element(my_list, 0);
     print_list(my_list);

     my_list = remove_element(my_list, 1);
     print_list(my_list);

     my_list = remove_element(my_list, 4);
     print_list(my_list);

     my_list = remove_element(my_list, 3);
     print_list(my_list);
}

public static int[] create_list(int n)
{
     int[] list;
     int i;

     list = new int[n];
     for (i=0; i<n; i++)
     {
        list[i] = i;
     }
     return list;
}

public static int[] remove_element(int[] list, int element)
{
     int[] new_list;
     int position;
     int i;
     position = -1;
     for (i=0; i<list.length; i++)
     {
        if (list[i] == element)
        {
           position = i;
        }
     }
     if (position == -1)
     {
        new_list = list;  // the element was not found, so return the original list
     }
     else
     {
        new_list = new int[list.length-1];
        System.arraycopy(list, 0, new_list, 0, position);
        System.arraycopy(list, position+1, new_list, position, new_list.length-position);
     }
     return new_list;
}

public static void print_list(int[] list)
{
     int i;

     for (i=0; i<list.length; i++)
     {
        System.out.print(list[i] +" ");
     }
     System.out.println();
}

0 1 2 3 4

1 2 3 4
2 3 4
2 3
2
```

As can be seen, deleting an element from an array is not a trivial exercise. However, Java does contain a more sophisticated collection mechanism which is discussed in Chapter 9 – Object Collections.

1.10 Passing Parameters

Parameters of functions are treated in essentially the same manner in both Python and Java. If a numeric value is passed to a function, the value may be modified within the function but the modified value does not replace the original value in the calling function. In the following example, the value of `n` is passed to `function1`. `n` is modified within the function but this modified value does not replace the original value of `n` in the calling function when `function1` returns.

```
def function1(n):
     print(n)
     n += 1
     print(n)
     return

n = 10;
function1(n)
print(n)

10
11
10
```

Java works in exactly the same manner. Numeric values that are passed to a method may be modified in the method but the modifications are not visible in the calling method.

In Python, if a list is passed to a function and an element of the list is modified in the function, the modification is reflected back (or is visible) in the calling function. In the following example, the third element of `list` is modified in `function2`. As can be seen in the output, this modification is visible in the calling function.

```
def function2(list):
     print(list)
     list[2] += 10
     print(list)
     return

list = [1, 2, 3, 4]
function2(list)  # list is updated by function call
print(list)

[1, 2, 3, 4]
[1, 2, 13, 4]
[1, 2, 13, 4]
```

In Java, the elements of an array may be modified in a method call and the changes are visible in the calling method.

In Python, if the number of elements in a list is modified in a function (for example, by appending a new element to the end of the list), the change is visible in the calling function. If this operation is performed in a Java method, the changes are not visible in the calling method. The following Java program illustrates this issue.

```
public static void main(String[] parms)
{
     int[] my_list;
     my_list = new int[] {1, 2, 3, 4, 5};
     print_list(my_list);
     add_element(my_list, 200);
     print_list(my_list);
}
```

```
public static void add_element(int[] my_list, int element)
{
     int[] new_list;

     new_list = new int[my_list.length+1];
     System.arraycopy(my_list, 0, new_list, 0, my_list.length);
     new_list[new_list.length-1] = element;
     my_list = new_list;
     print_list(my_list);
}

public static void print_list(int[] my_list)
{
     int i;

     for (i=0; i<my_list.length; i++)
     {
        System.out.print(my_list[i] +" ");
     }
     System.out.println();
}

1 2 3 4 5
1 2 3 4 5 200
1 2 3 4 5
```

In Java, if the size of an array in the calling method must be modified, the called method must return the modified array and the calling method must assign the modified array to the original array.

```
public static void main(String[] parms)
{
     int[] my_list;
     my_list = new int[] {1, 2, 3, 4, 5};
     print_list(my_list);

     my_list = add_element(my_list, 200);
     print_list(my_list);
}

public static int[] add_element(int[] my_list, int element)
{
     int[] new_list;

     new_list = new int[my_list.length+1];
     System.arraycopy(my_list, 0, new_list, 0, my_list.length);
     new_list[new_list.length-1] = element;
     my_list = new_list;
     print_list(my_list);
     return my_list;
}

public static void print_list(int[] my_list)
{
     int i;

     for (i=0; i<my_list.length; i++)
     {
        System.out.print(my_list[i] +" ");
     }
     System.out.println();
}

1 2 3 4 5
1 2 3 4 5 200
1 2 3 4 5 200
```

Python supports keyword parameters but since this type of parameter is not supported in Java, they will not be examined in these notes.

1.11 Scope

The scope of variables in Python is straightforward – variables used in a function are local to that function. When the function terminates, the variables are destroyed. If a variable in a calling function is required in a called function, the variable should be passed to the called function as a parameter. The value of the variable can then be accessed without any problems. Depending on the type of the variable, it may also be possible to modify the variable in the called function and have the modification visible in the calling function (see the previous Section: Passing Parameters).

Python permits variables that are given values in the calling function to be accessed (but not modified) in a called function (without passing the variables as parameters). We will not examine this technique since it is not a standard programming construct.

In Java, variables are local to the method in which they are declared. Java does not permit variables that are declared in one method to be accessed in a called method. So, if the value of a variable in a calling method is required in a called method, the variable must be passed to the called method.

1.12 Data Types

Java implements a collection of data types that are referred to as "primitive data types". As we shall see later in these notes, the primitive data types are stored in a different manner than non-primitive data types (which are objects). Java's primitive data types are: `byte`, `short`, `int`, `long`, `float`, `double`, `char`, and `boolean`. Each variable declared in a Java program is given a specific type. The types `byte`, `short`, `int`, and `long` are used to represent integer values of varying sizes. The types `float` and `double` are used to represent floating-point values of different sizes. The type `boolean` is used to store a Boolean value (true or false). The type `char` is used to store a single character. A variable that is given one type can not be assigned a value of another type (unless the two types are compatiable or some additional work is performed).

Java's primitive data types are shown below. Python does not have "primitive data types" but it does have some core data types that are similar to Java's primitive data types.

Python	**Java**	**Meaning**
int		an integer of any size
	byte	-128 <= integer <= 127
	short	-32768 <= integer <= 32767
	int	-2147483648 <= integer <= 2147483647
	long	approximately 18-digit integer
	float	floating-point value with 7-digit accuracy
float	double	floating-point value with 15-digit accuracy
bool	boolean	Boolean (true/false)
	char	one character

Python uses dynamic typing for its variables. This means that Python variables are not declared to have a specific type; instead, when a Python variable is assigned a value, the type of the value is stored with the actual value. So it is perfectly valid in Python to assign an integer value to a variable, use the variable in statements that make sense for integers, and then at a later time, assign a boolean value to the variable. As long as the variable is then used in a boolean context, the program will work correctly.

Python permits the programmer to determine the type of the value currently stored in a variable using the `type()` function (remember that a variable does not have a type in Python).

```
i = 25;
j = 25.5
print(type(i))
print(type(j))

<class 'int'>
<class 'float'>
```

To make the type output slightly nicer, the following function `get_type` can be used. (The _ _ used in the function is 2 underscore characters typed side-by-side.)

```
def get_type (var):
     return var.__class__.__name__

i = 25
j = 25.5
print(get_type(i))
print(get_type(j))

int
float
```

If it is necessary to convert a value from one type to another, Python provides a collection of functions to perform the conversion. The `int()` function converts its parameter to an integer value; the `float()` function converts its parameter to a floating-point value.

The following statements illustrate the conversion between the integer and floating-point data types in Python.

```
i = 25;
print(i)
j = float(i)
j += 0.5
print(j)
i = int(j)
print(i)

25
25.5
25
```

In Java, data types are converted from one type to another using Java's "cast" mechanism. A cast involves placing the desired type in parentheses before an expression or variable. The following Java statements have the same effect as in the Python statements above.

```
int i;
float j;

i = 25;
System.out.println(i);
j = (float) i;
j += 0.5;
System.out.println(j);
i = (int) j;
System.out.println(i);

25
25.5
25
```

1.13 Strings

Python supports the data type `str` which is used to store a collection of characters (or a character string). The following statements illustrate the use of character strings in Python.

```
s = "abc"
print(s)
print(type(s))

abc
<class 'str'>
```

In Python, strings may be delimited either by single quote marks (`'`) or by double quote marks (`"`). As long as a string begins and ends with the same delimiter, it doesn't matter which is used.

The type conversion functions in Python can be used to convert between numeric values and strings. For example, the statements illustrate the conversions between an integer and a string.

```
i = int("123")
s = str(i)
```

A subset of the contents of a string can be extracted using the square brackets notation (this is referred to as "slicing" in Python). The following example illustrates this process.

```
s = "abc"
t = s[1:2]
print(t)

b
```

The notation `[from, to]` means that the characters beginning with character `from` and continuing up to but not including character `to` are extracted. As with arrays, character indexing is zero-based. If only the `from` value is included, only the one corresponding character is extracted. If the notation `[from:]` is used, this means extract characters beginning at the `from` character and continuing until the end of the string. (The slicing notation supported in Python contains additional options that we will not examine.)

Strings in Python are compared using the relational operators (<, >, ==, etc.) When strings are compared for equality in Python, it is important that the == operator be used. As we shall see, this is not the case in Java.

Python contains a rich collection of functions that manipulate character strings. For more information, take a look at the web or a Python textbook.

Java supports the primitive data type `char` which can be used to store one character. If you want to store a collection of characters (or a character string) in Java, you could use a `char[]` (array) but a more convenient mechanism for manipulating character strings in Java is the data type `String` (which is not a primitive data type).

```
String s;

s = "abc";
System.out.println(s);

abc
```

Java also permits the programmer to extract a subset of the characters in a character string. However, in Java, a method named `substring` is used to perform the processing. The parameters of `substring` are almost the same as Python's slicing notation. The only difference is that in Java, if only one value is supplied, all characters beginning at that position and continuing until the end of the string are extracted.

```
String s;
String t;

s = "abc";
System.out.println(s);
System.out.println(s.substring(1,2));
System.out.println(s.substring(1));

abc
b
bc
```

In Java, when two strings are compared for equality, a special method, the `equals` method, must be used instead of the relational operator ==.

```
String s;
String t;

s = "abc";
t = "abc";
if (t.equals(s))
{
     System.out.println("The strings are equal");
}
else
{
     System.out.println("The strings are not equal");
}
```

1.14 Modules/Classes

The statements and functions defined inside a Python file (a `.py` file) are referred to as a Python module. Python includes many modules that provide useful functionality (so that the programmer does not have to write all of the functions him/herself). A simple example is the `math` module that contains many mathematical functions. Before the functions in a module may be used, the module must be made available (imported).

```
import math

i = math.sqrt(25.0)
print(i)

5.0
```

Java also contains a variety of predefined methods and its equivalent of modules. For example, the mathematical functions are also collected together into a `Math` "class". This class can be imported using the statement shown at the beginning of the following program. To make life a little easier for the programmer, Java automatically imports the `Math` class (along with a variety of other classes) so the `import` statement below is not actually necessary.

```
import java.lang.Math;

public class TestModules
{
     public static void main(String[] parms)
     {
        double d;

        d = Math.sqrt(25.0);
        System.out.println(d);
     }
}

5.0
```

1.15 Determining Prime Numbers

The following program determines the prime numbers that are less than `n`. A number is prime if it does not have any integer divisors greater than 1 other than itself.

```
def isPrime(value):
     result = True
     count = 2
     while (count<value) and (result):
        if (((value//count)*count)==value):
           result = False;
        count += 1
     return result

def generatePrimes(n):
     list = []
     for i in range(2, n):
        if isPrime(i):
           list.append(i)
     return list

list = generatePrimes(40)
print(list)

[2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37]
```

The corresponding Java program is shown below.

```
public class Primes
{
     public static void main(String[] parms)
     {
        int[] list;

        list = generatePrimes(40);
        print_list(list);
     }

     public static int[] generatePrimes(int max)
     {
        int[] list;
        int[] new_list;
        int numPrimes;
        int count;
        list = new int[max];
        numPrimes = 0;
        for (count=2; count<max; count++)
        {
           if (isPrime(count))
           {
              list[numPrimes] = count;
              numPrimes++;
           }
        }

        new_list = new int[numPrimes];
        System.arraycopy(list, 0, new_list, 0, numPrimes);
        return new_list;
     }

     public static boolean isPrime(int value)
     {
        int count;
        int sqrt;
        boolean result;

        result = true;
        for (count=2; (count<value) && result; count++)
        {
           if (((value/count)*count)==value)
           {
              result = false;
           }
        }
        return result;
     }

     public static void print_list(int[] my_list)
     {
        int i;

        for (i=0; i<my_list.length; i++)
        {
           System.out.print(my_list[i] +" ");
        }
        System.out.println();
     }
}

2 3 5 7 11 13 17 19 23 29 31 37
```

As can be seen, the Java program is more "wordy" than the Python program although the Java program does not require more thought than the Python program (with the possible exception of shrinking the array of primes to the exact size needed).

The prime number generation programs are presented without any discussion of how such programs would be developed. The process of developing programs is discussed in the next Chapter.

1.16 Differences between Python 2 and Python 3

The `print` function has been added to Python 3; in Python 2, `print` was a statement that did not require parentheses.

```
print 1, 2, 3      # Python 2
print(1, 2, 3)     # Python 3
```

There have also been some changes to the way that integers are defined and manipulated.

```
long               # Python 2: the long type was removed in Python 3
a / b              # Python 2: integer result
a / b              # Python 3: real (floating point) result
a // b             # Python 3: integer result
```

1.17 Differences between Python and Java

Java uses brace/curly brackets to start and end blocks, while Python uses indentation.

Java employs static typing, while Python is dynamically typed.

Java methods are explicitly declared to be static.

Java uses `array.length` while Python uses `len(list)`.

There are several types of integer values in Java (depending on the maximum integer to be manipulated); Python only has one type which can manipulate integers of any size.

The representation of Python data values is different from the representation of Java values, particularly Java primitive data types.

1.18 Summary

In this chapter, we examined the fundamental constructs used in procedural programming. This chapter has only scratched the surface of the Python language – Python contains many elegant constructs and also some very powerful libraries of predefined functions.

2 GROWING ALGORITHMS

2.1 Introduction

In this chapter, we introduce the process of "growing" software, that is, instead of writing an entire program at once, the program is **grown** incrementally from a small, **working** portion of code, adding functionality in small pieces until the desired result is achieved. This process is also known as **iterative development**.

2.2 Searching a List

Searching the elements in a list for a specific element is one of the standard computer science algorithms and is used in a wide variety of programs. In the following example, an array of integers is searched for a specific integer. We begin by defining a simple main method that indicates the parts of the program that must be defined. We will use this program organization throughout these notes – the main method will not perform any explicit processing; instead, it will call appropriate methods that perform the processing.

```
public static void main(String[] parms)
{
     int[] list;
     int searchValue = 10;
     int result;

     list = createList();
     result = searchList(list, searchValue);
     System.out.println("Result is: " +result);
}
```

The main method shown above defines the structure of the program that is to be created. A method "createList" is used to initialize the array of integers and a method "searchList" is used to examine the elements in the array of integers, looking for an element that is identical to the value in "searchValue". By defining the main method in this manner, we have reduced the problem to smaller components and we can now examine these components individually.

We need a method createList that stores a collection of integer values in an array and returns the array to the main program. Following our rule of doing the simplest thing that could work, the createList method is initially defined with a "**hard coded**" array of integers. (Hard coding means that the values are specified in the program instead of being supplied by the user or read from a file.)

```
public static int[] createList()
{
     int[] list = {10, 20, 30, 40, 50, 60, 70, 80, 90, 100};

     return list;
}
```

Note that since createList returns an array of integers, its return type must be declared as int[] instead of void. While this version of createList is quite limited, it does provide a working version that can be improved upon later. A method that works correctly but is not complete is often referred to as a “stub” method.

Now, we need a searchList method that accepts the array and a simple integer value and determines whether or not the integer is an element of the array. The searchList method is initially created to return 0, the position of the first element in the array.

```
public static int searchList(int[] list, int searchValue)
{
     int result;

     result = 0;
     return result;
}
```

At this point, the program actually works correctly – it creates an array of integers and identifies the position of the integer that is being searched for in the array. While the program works only for the array that is hard coded in the createList method, the program does identify all of the necessary components. By first breaking the program down into smaller pieces and writing very simple instructions that work (in a very limited manner), we can now concentrate on improving the individual methods one-at-a-time.

Of course, if you know exactly what is required in each method, it is not necessary to grow the program so slowly; instead, you can take bigger steps towards the solution. However, the advantage of developing a program slowly is that if something goes wrong between one iteration and the next, it is easier to identify the cause of the problem.

The goal of approaching program development in this manner is to ensure that the overall design of the program is correct before adding the details. The **design** of a program is similar to the blueprint of a building that is to be constructed. However, there are significant differences between developing software and building (or engineering) a physical structure – these issues will be addressed from time to time in these notes.

Now we improve the methods so that each becomes more general.

We initially hard coded the array in the createList method.

```
public static int[] createList()
{
     int[] list;

     list = new int[] {10, 20, 30, 40, 50, 60, 70, 80, 90, 100};
     return list;
}
```

An alternative is to generate random values using the Math.random method.

```
public static int[] createList()
{
     int[] list;
     int count;
     int numElements = 10;

     list = new int[numElements];
     for (count=0; count<numElements; count++)
     {
        list[count] = (int) (Math.random()*101.0);
     }
     return list;
}
```

The array is constructed to contain 10 random integer values in the range 0 to 100. (Recall that Math.random returns a double value (in the range: `0 ≤ value < 1`) so the random value must be cast to an int before it can be stored in the array.) This version is a definite improvement over the previous version but we will see that other versions of createList can also be defined.

2.2.1 Linear Search

In our initial version of searchList, we simply returned the position of the first element in the array. This allowed us to get the program working but this method now must be improved so that it works correctly with all data.

```
public static int searchList(int[] list, int searchValue)
{
     int result;

     result = 0;
     return result;
}
```

A simple linear search of the elements in the array can be used to locate the value.

```
public static int searchList(int[] list, int searchValue)
{
     int count;
     int result;

     result = -1;
     for (count=0; count<list.length; count++)
     {
        if (list[count] == searchValue)
        {
           result = count;
        }
     }
     return result;
}
```

The searchList method now returns the location of the desired element in the array if the element is found or returns -1 if the element is not found. **Returning a value of -1 to indicate that a method was not able to accomplish its task is a common technique.**

In the method above, the loop continues until the end of the array even if the desired element is located. While we are not going to attempt to optimize each program as much as is possible, this is an example that is particularly easy to improve by modifying only the loop condition so that the loop terminates as soon as the desired element is located.

```
public static int searchList(int[] list, int searchValue)
{
     int count;
     int result;

     result = -1;
     for (count=0; result==-1 && count<list.length; count++)
     {
        if (list[count] == searchValue)
        {
           result = count;
        }
     }
     return result;
}
```

Now the loop continues as long as the desired value has not been located and there are still more elements to be examined in the array.

If data are to be provided by a user, the createList method becomes somewhat more complex. `JOptionPane` can be used to obtain each value from the user, but we don't know in advance how many elements will be entered. The simplest solution is to ask the user to specify the number of elements that will be entered. Then, the array can be created so that it is exactly the right size.

```
public static int[] createList()
{
     int[] list;
     int numElements;
     int count;
     int value;
     String string;

     string = JOptionPane.showInputDialog("Please enter the number of values:");
     numElements = Integer.parseInt(string);
     list = new int[numElements];

     for (count=0; count<list.length; count++)
     {
        string = JOptionPane.showInputDialog("Please enter a number:");
        value = Integer.parseInt(string);
        list[count] = value;
     }
     return list;
}
```

Note that the array was declared at the beginning of the method but the array was not instantiated (with an `= new()` statement) until later in the method when the actual number of elements to be stored in the array was known. This is a common Java programming technique.

There are other ways in which an unknown number of elements can be stored in an array and we will examine some of these techniques later in this chapter.

Recall that when Java's Graphical User Interface (GUI) components are used, the program must import the GUI component(s) from the Swing package.

```
import javax.swing.JOptionPane;
```

Similarly, to ensure that the program terminates correctly when GUI components are used, the statement below must be included at the end of the main method.

```
System.exit(0);
```

Having to type values into the console each time that the program is tested is annoying after the first couple of iterations. It is often easier to generate random data and then switch to user-supplied data only when all parts of the program are working.

An alternative to using JOptionPane is to read the data from a (disk) file that is provided by the user. We will examine how to read information from files in Chapter 4 – File Input and Output.

2.2.2 Linear Search of an Ordered List

If an array is sorted into ascending order, then the contents of the array are arranged so that the value of the first element is less than or equal to the value of the second element, the value of the second element is less than or equal to the value of the third element, and so on. The following diagram shows the contents of an array that consists of 5 elements, with each element less than or equal to all following elements.

10	20	30	40	60

We will examine techniques that sort the contents of an array into a particular order in Chapter 13 – Sorting but for now we will assume that the integers are provided in sorted order.

If the elements in an array have been sorted into ascending order, there are several ways in which the array can be searched that make the search process more efficient. Our original linear search algorithm searches the array until either the desired element is found or until the end of the array is found.

```
public static int searchList(int[] list, int searchValue)
{
     int count;
     int result;

     result = -1;
     for (count=0; result==-1 && count<list.length; count++)
     {
        if (list[count] == searchValue)
        {
           result = count;
        }
     }
     return result;
}
```

This search process can be improved slightly by stopping the search as soon as the desired element is found **or** when an element with a value that is larger than the desired element is found. The search is still a linear search but it stops earlier if the desired element is not found.

```
public static int searchList(int[] list, int searchValue)
{
     int count;
     int result;

     result = -1;
     for (count=0; searchValue>=list[count] && result==-1
                                            && count<list.length; count++)
     {
        if (list[count] == searchValue)
        {
           result = count;
        }
     }
     return result;
}
```

The example works correctly in most cases but has a problem (or bug) in one particular situation: if the desired element is greater than the largest element in the array, a subscript-out-of-bounds exception is generated. This problem can be eliminated by reordering the `for` condition to ensure that position is still within the array before evaluating the condition `searchValue>=list[count]`.

It is important to understand why the program above does not work correctly in all situations. When learning to program, it is much more important to write code that is correct than to be concerned about efficiency.

The correct version of the program is shown below.

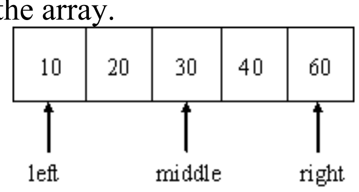
```
public static int searchList(int[] list, int searchValue)
{
     int count;
     int result;

     result = -1;
     for (count=0; result==-1 && count<list.length && searchValue>=list[count]; count++)
     {
        if (list[count] == searchValue)
        {
           result = count;
        }
     }
     return result;
}
```

This example illustrates how easily a bug that occurs only occasionally can be introduced into a program. Many programmers do not perform sufficient testing of their programs and problems such as this one may slip through the development process and then cause the system to crash when the system is put into production.

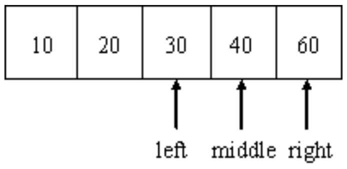
2.2.3 Binary Search

If the elements in the array are sorted in ascending order, the search process can be further improved by using a binary search.

A binary search involves accessing the elements of the array in a non-linear manner. The search begins by noting the position of the first element in the array, the last element in the array, and determining the middle element in the array. If the middle element is less than the desired element, then we know that the desired element can not be below the middle element and must be in the upper half of the array.



If we know that the desired element can not be to the left of the current middle element, we can move the position of the left-most element to middle and then recalculate the new middle element.



This process continues until the desired element is located or until left and right have moved past each other so we know that the desired element is not in the array.

```
public static int searchList(int[] list, int searchValue)
{
      int left;
      int right;
      int middle;
      int result;
      int middleElement;

      result = -1;
      left = 0;
      right = list.length;

      while ((left <= right) && (result == -1))
      {
          middle = (left + right) / 2;
          middleElement = list[middle];
          if (middleElement == searchValue)
          {
              result = middle;
          }
          else if (middleElement < searchValue)
          {
              left = middle;
          }
          else if (middleElement > searchValue)
          {
              right = middle;
```

```
            }
        }
        return result;
}
```

The algorithm above also contains a subtle bug: when an element does not exist in the array, the algorithm never terminates. The reason for this problem is that the variables "left" and "right" are not modified correctly in the program. Again, it is important that you understand why this bug occurs. The modified version below does work correctly.

```
public static int searchList(int[] list, int searchValue)
{
        int left;
        int right;
        int middle;
        int result;
        int middleElement;

        result = -1;
        left = 0;
        right = list.length-1;

        while ((left <= right) && (result == -1))
        {
            middle = (left + right) / 2;
            middleElement = list[middle];
            if (middleElement == searchValue)
            {
                result = middle;
            }
            else if (middleElement < searchValue)
            {
                left = middle + 1;
            }
            else if (middleElement > searchValue)
            {
                right = middle - 1;
            }
        }
        return result;
}
```

A binary search repeatedly reduces the number of elements that must be examined by half. As a result, a binary search is significantly faster than a linear search.

The binary search algorithm described above has recently been found to contain a very subtle bug that has only emerged as the amount of data being processed has become larger and larger. (Imagine performing a search of the Google database of terms.) The problem occurs when approximately 1 billion elements are being searched. The statement

```
middle = (left + right) / 2;
```

causes an overflow when left and right are added together since int variables can store a maximum positive value of 2^{31}-1 (2,147,483,647). When the two int values are added together, the result overflows into the sign position and the result is a negative value. For example, computing 2,147,483,647 + 1 generates a result of -2,147,483,648. While this situation is unlikely to happen in most binary searches, the solution to the problem is quite simple – use the following statement instead:

```
middle = left + ((right-left)/2);
```

Alternatively, if very large values must be manipulated, use `long` variables instead of `int` variables.

2.3 Searching a List of Strings

Although all of the methods developed in the previous section used simple integer values, any valid Java data type can be used instead of integers with very few changes required to the programs. The example below illustrates how the program can be modified to search a list of Strings (the list of Strings is hard-coded in the createList method.) Strings are examined in detail in Chapter 5 – Strings.

```
public static void testStrings()
{
     String[] list;
     String searchValue = "Fred";
     int result;

     list = createList();
     printList(list);
     result = searchList(list, searchValue);
     System.out.println("Result is: " +result);
}

public static String[] createList()
{
     String[] list;
     list = new String[] {"Joe", "BillyBob", "Fred", "Homer"};
     return list;
}

public static int searchList(String[] list, String searchValue)
{
     int count;
     int result;

     result = -1;
     for (count=0; count<list.length && result == -1; count++)
     {
         if (list[count].equals(searchValue))
         {
             result = count;
         }
     }
     return result;
}

public static void printList(String[] list)
{
     int count;
```

```
      System.out.print("\nList elements: ");
      for (count=0; count<list.length; count++)
      {
         System.out.print(list[count] +"  ");
      }
      System.out.println();
}
```

Alternatively, Java provides a utility method named `Arrays.toString` that can be used when printing the contents of any array. The following statement prints the contents of the array `myArray`. The elements of the array are separated by commas and the collection of elements is surrounded by square brackets.

```
System.out.println(Arrays.toString(myArray));

[1, 2, 3, 4, 5]
```

2.4 Recap

In the sections above, we developed methods that created an array of integers in several different ways and methods that searched the array for a specific element in different ways. We began with a simple pair of methods that worked only because we hard coded the data. By making small modifications, each of the methods was improved until we had several sophisticated ways of both creating an array and searching an array.

The examples above illustrate how a program can be "grown" from some small, not particularly functional methods to a complete program without requiring a lot of effort. This process of growing code also makes debugging the code much easier since only small changes to one method are made at any particular point in time.

2.5 Array Initialization

The previous sections used several techniques to initialize an array. The following statements illustrate the ways in which an array can be initialized.

```
int[] ints1 = {10, 20, 30, 40, 60};

int[] ints2 = new int[] {10, 20, 30, 40, 60};

ints1 = new int[] {10, 20, 30, 40, 60};


boolean[] bools1 = {true, false, true};

boolean[] bools2 = new boolean[] {true, false, true};

bools1 = new boolean[] {true, false, true};


String[] strings1 = {"Jamie", "Joel"};

String[] strings2 = new String[] {"Katie", "Ashley"};

strings1 = new String[] {"Jamie", "Joel"};
```

2.6 Conditional Operator

Java includes a special operator, the conditional operator, that is very useful in certain situations. For example, the following statements determine the larger of two values.

```
if (value1 > value2)
{
     max =  value1;
}
else
{
     max = value2;
}
```

While the purpose of the statements is perfectly clear, the statements are spread over 8 lines. The conditional operator is used to perform the same function in one line of code. The following statement is equivalent to the statements above.

```
max = (value1 > value2) ? value1 : value2;
```

The conditional operator (?:) consists of a logical expression which is evaluated. If the expression is true, the result of the expression that follows the "?" is returned by the operator; if the expression is false, the result of the expression that follows the ":" is returned.

The following statement includes the use of the conditional operator in a print statement (note that the result returned may be any expression, including a method call):

```
System.out.println( (value1 > value2) ? process(value1) : process(value2) );
```

2.7 The For Statement

The for statement is a statement that you have probably used in most programs but there are variations that are useful to know. The basic for statement is shown below (the output generated is shown after the program):

```
max = 5;
for (count=0; count<max; count++)
{
     System.out.print(count +" ");
}

0 1 2 3 4
```

The general form of the for statement is shown below:

```
for (initialization; test; nextIteration)
{
     statements within the for statement
}
```

The for statement is executed as follows: the first time that the statement is executed, the initialization is performed; then the test condition is evaluated; if the test condition is true, the statements within the for statement are executed. At the end of the for statement, the nextIteration statements are executed; then the test is evaluated, and, if the test condition is true, the statements within the for statement are executed again. This process continues until the value of the test condition is false.

The initialization portion of the statement consists of zero or more initialization statements. If there is more than one initialization statement, the statements are separated by commas.

```
for (count1=0, count2=0; test; nextIteration)
{

}
```

The test consists of a logical expression that evaluates to true or false (the expression may be as complex as is required). In the following example, the for statement continues as long as the value of count is less than the value of max and the boolean variable done is false.

```
for (initialization; (count<max) && (!done); nextIteration)
{

}
```

The nextIteration portion of the statement modifies any variables prior to the next iteration. A common modification involves incrementing a counter to its next value.

```
for (initialization; test; count++)
{

}
```

Often, a loop begins with the small values and continues until the largest value has been processed.

```
max = 5;
for (count=0; count<max; count++)
{
     System.out.print(count +" ");
}
```

However, there are times when it is necessary to work backwards, beginning with the largest value and continuing until the smallest value has been processed.

```
max = 5;
for (count=max-1; count>=0; count--)
{
     System.out.print(count +" ");
}

4 3 2 1 0
```

While the nextIteration increment `count++` is frequently used, the for statement may include any valid assignment statement as part of the nextIteration.

```
max = 5;
for (count=0; count<max; count+=2)
{
     System.out.print(count +" ");
}

0 2 4
```

The following assignment is also valid and has the same effect as the increment in the previous example.

```
max = 5;
for (count=0; count<max; count=count+2)
{
     System.out.print(count +" ");
}

0 2 4
```

Be careful with the nextIteration clause – using `count+2` (instead of `count=count+2`) is not correct.

Both the initialization and the nextIteration clauses are optional in the for statement (as indicated by the square brackets below). If both the initialization and the nextIteration clauses are omitted, then the for statement becomes equivalent to a while statement. The test clause is also optional but leaving it out leads to bad programming practices so we will require an explicit test in all for statements.

```
for ([initialization]; test; [nextIteration])
{
     statements within the for statement
}
```

2.8 System.arraycopy

When the contents of one array must be copied to another array, writing a simple loop works well. However, Java provides a very useful method, `System.arraycopy`, that performs the same processing. The parameters used in System.arraycopy are as follows:

- first parameter: the source array;
- second parameter: the starting position in the source array;
- third parameter: the destination array;
- fourth parameter: the starting position in the destination array;
- fifth parameter: the number of elements to be copied from the source array to the destination array.

System.arraycopy generates an error **if either array has not yet been instantiated** or if the range of elements being copied exceeds the number of elements in either array.

The source and destination arrays may be the same array. In this case, elements in the array are copied from one location to another location. For example, the following statement copies the elements at positions 0, 1, and 2 in array 1 into positions 5, 6, and 7.

```
System.arraycopy(array1, 0, array1, 5, 3);
```

When the source and destination arrays are the same array, the range of elements being copied may overlap. For example, the following statement copies the 3 elements that begin at position 0 one position to the right (this is sometimes referred to as a "right shift" of the elements).

```
System.arraycopy(array1, 0, array1, 1, 3);
```

For example, if array1 contains the following elements before the preceding statement is executed:

```
10, 20, 30, 40, 50
```

the array will contain the following elements after the statement is executed:

```
10, 10, 20, 30, 50
```

2.9 Rotating the Elements in an Array

Occasionally, it is necessary to "rotate" the elements in an array. This is similar to a "shift" of the elements as mentioned in the previous section. The difference is that with a rotation, when elements are shifted to the left (off the beginning of the array), those elements are added to the end of the array (hence the term rotate). Similarly, if elements are rotated to the right, any elements that are shifted off the end of the array are added to the beginning of the array. The following method performs a left rotation of the elements of an array. The value n indicates the number of elements that are to be shifted to the left (and rotated around to the end of the array). Rotations to the right are also possible but are not handled by the method below.

```
public static void rotate(int[] list, int n)
{
     int[] temp;

     temp = new int[n];
     System.arraycopy(list,0,temp,0,n);
     System.arraycopy(list,n,list,0,list.length-n);
     System.arraycopy(temp,0,list,n-1,n);
}
```

If you test the method with the array shown below and rotate the contents of the array 3 elements to the left, the correct result is as shown.

```
100 200 300 400 500

Rotating 3 positions to the left

400 500 100 200 300
```

If you stop at this point, you will have a method that works correctly in one specific set of circumstances. You should always test your methods with more than just a given set of data because you may find that your method does not work correctly in all situations. The method above works for the given data but contains a bug that would be found if the method were tested more completely. For example, what happens if 0 elements, 1 element, 5 elements, etc. are rotated? If you test the method, you will find that it generates a subscript out of range error in many situations. (The solution is fairly simple, once you look at the algorithm.)

2.10 Generating Prime Numbers

In this section and the next few sections, we develop some algorithms that solve various problems using procedural algorithms.

The following method determines whether or not an integer value is a prime number. The definition of a prime number is a number that is divisible only by itself and 1. This method uses the simple technique of dividing the value by each integer from 2 up to the value-1. If there is a non-zero remainder, then the value is not divisible by that integer. Note that the mod operator is used to determine whether or not the value is divisible by each integer.

```
public static boolean isPrime(int value)
{
     int count;
     boolean result;

     result = true;
     if ((value % 2)==0)
     {
        result = false;
     }
     for (count=3; count<value; count+=2)
     {
        if ((value%count)==0)
        {
           result = false;
        }
     }
     return result;
}
```

The program above works correctly but it does a significant amount of unnecessary work. (Once a number has been determined not to be prime, there is no reason to perform any additional tests.) Making the simple modification shown below, the algorithm terminates as soon as it is determined that a value is not prime.

```
public static boolean isPrime(int value)
{
     int count;
     boolean result;

     result = true;
     if ((value % 2)==0)
     {
        result = false;
     }
     for (count=3; (count<value) && (result); count+=2)
     {
        if ((value%count)==0)
        {
           result = false;
        }
     }
     return result;
}
```

The method above works correctly but can be improved significantly. For example, it is not necessary to continue past the square root of the value being tested for primality. Also, it is not necessary to divide by all values between 2 and `value` (or the square root of `value`); the fundamental theorem of arithmetic states that every number can be written as the product of prime numbers, so it is only necessary to divide by all prime numbers less than the square root of `value`.

2.11 Generating Sequences of Values

The following program generates all sequences (permutations) of the integers from 1 to n with each sequence consisting of 3 values such that the same value does not occur more than once in a specific sequence. (The output for this program is shown immediately after the program.) While the value of n can be specified in the program, the number of values in each sequence is hard-coded in the generate method (there are exactly 3 for loops).

```
public class Sequences
{
     public static void main(String[] parms)
     {
        generate(3);

        System.out.println("Program completed normally.");
     }

     public static void generate(int value)
     {
        int count1;
        int count2;
        int count3;

        for (count1=1; count1<=value; count1++)
        {
           for (count2=1; count2<=value; count2++)
           {
              for (count3=1; count3<=value; count3++)
              {
                 if ((count1!=count2) && (count2!=count3) && (count1!=count3))
                 {
                    System.out.println(count1 +" " +count2 +" " +count3);
                 }
              }
           }
        }
     }
}

1 2 3
1 3 2
2 1 3
2 3 1
3 1 2
3 2 1
```

An algorithm that can generate sequences of arbitrary length is not significantly more difficult than the algorithm above (but it does require some thought, as opposed to brute force).

2.12 Generating A Frequency Count of Values

The following program determines the frequency with which each of the values from 1 to 10 occurs in an array of values.

```
public class Frequencies
{
     public static void main(String[] parms)
     {
        int[] list = {1, 1, 2, 1, 4, 3, 3, 2, 5, 5, 5, 8, 10};
        int[] freq;
        print(list);
        freq = countFreq(list);
        print(freq);
        System.out.println("Program completed normally.");
     }

     public static int[] countFreq(int[] list)
     {
        int MAX_VALUE = 10;
        int[] freq;
        int count;
        freq = new int[MAX_VALUE+1];
        for (count=0; count<list.length; count++)
        {
           freq[list[count]]++;
        }
        return freq;
     }

     public static void print(int[] list)
     {
        int count;

        for (count=0; count<list.length; count++)
        {
           System.out.print(list[count] +" ");
        }
        System.out.println();
     }
}
```

In the program above, if a value outside of the range 1..10 is included, the program will terminate with a subscript out of range error message. The following improved version keeps track of the number of out of range values in position 0 of the frequency count array.

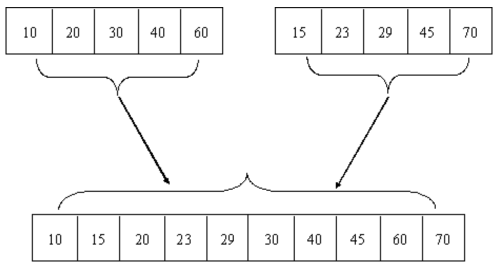
```
public static int[] countFreq(int[] list)
{
     int MAX_VALUE=10;
     int[] freq;
     int count;
     int temp;
     int size;
     freq = new int[MAX_VALUE+1];
     size = freq.length;
     for (count=0; count<list.length; count++)
     {
        temp = list[count];
        if ((temp>0) && (temp<size))
        {
           freq[temp]++;
        }
        else
        {
           freq[0]++;
        }
     }
     return freq;
}
```

2.13 Merge Algorithm

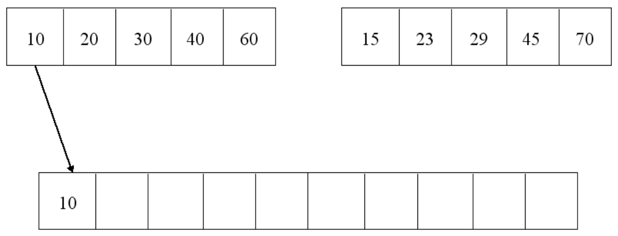
The merge algorithm is one of the fundamental computer science algorithms. A merge involves taking 2 (or possibly more) sets of data that are already sorted in ascending (or descending) order and creating one combined set of data that is also sorted in ascending (or descending) order.

The following example illustrates merging the contents of two arrays in order to create a third arrays. Each array contains zero or more integer values.

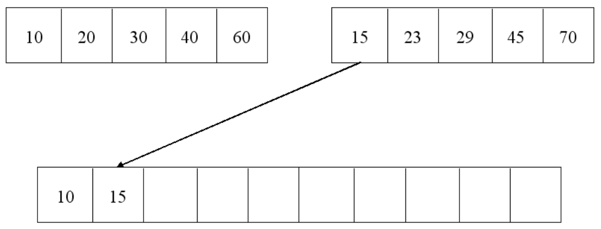
The contents of the arrays have already been sorted into ascending order (somehow) and we want to generate a new array that contains the contents of both arrays with all integers sorted into ascending order. For example, the following diagram illustrates the contents of two arrays and the result after the contents of the arrays have been merged to create a third array.



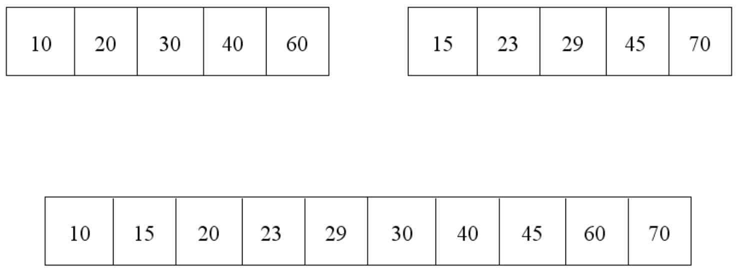
The merge process works by comparing the first integer in the first array with the first integer in the second array. Whichever integer is smaller is copied to the output array.



Then, the second integer in the first array is compared with the first integer in the second array and the smaller integer is copied to the output array.



This process continues until all integers in the two input arrays have been processed.



We will grow the algorithm slowly to ensure that little time is spent debugging errors in the program.

The two arrays passed to the mergeArrays method are the following:

```
int[] array1 = {10, 20, 30, 40, 60};
int[] array2 = {15, 23, 29, 45, 70};
```

Our first version of the program simply reads one of the arrays and copies the contents of that array to the output array. The output array is then printed to ensure that the method worked correctly.

```
public static int[] mergeArrays(int[] array1, int[] array2)
{
     int count;
     int count1;
     int[] array3;

     array3 = new int[array1.length + array2.length];

     count1 = 0;
     for (count=0; count1<array1.length; count++, count1++)
     {
        array3[count] = array1[count1];
     }
     printList(array3);
     return array3;
}
```

The method that prints an array is straightforward.

```
public static void printList(int[] array)
{
     int count;
     for (count=0; count<array.length; count++)
     {
        System.out.print(array[count] + "  ");
     }
     System.out.println();
}

10  20  30  40  60  0  0  0  0  0
```

So we know that the contents of the first array were copied corrected to the output array.

We now add the statements that copy the elements of the second array.

```
public static int[] mergeArrays(int[] array1, int[] array2)
{
     int count;
     int count1;
     int count2;
     int[] array3;

     array3 = new int[array1.length + array2.length];

     count1 = 0;
     count2 = 0;
     for (count=0; count1<array1.length && count2<array2.length; count++)
     {
        array3[count] = array1[count1];
        count1++;
        array3[count] = array2[count2];
        count2++;
     }
     printList(array3);
     return array3;
}

15  23  29  45  70  0  0  0  0  0
```

This looks like a step backwards because now we have only the contents of the second array copied to the output array. (Do you understand why?) However, it is a small step to add an "if" statement to the loop so that the smaller of the two current values is selected and copied to the output array.

```
public static int[] mergeArrays(int[] array1, int[] array2)
{
     int count;
     int count1;
     int count2;
     int[] array3;

     array3 = new int[array1.length + array2.length];

     count1 = 0;
     count2 = 0;
     for (count=0; count1<array1.length && count2<array2.length; count++)
     {
        if (array1[count1] < array2[count2])
        {
           array3[count] = array1[count1];
           count1++;
        }
        else
        {
           array3[count] = array2[count2];
           count2++;
        }
     }
     printList(array3);
     return array3;
}
```

Now things are looking more promising, we copy the smaller value to the output array during each pass through the loop. Unfortunately, when the program is executed, the output is not correct:

```
10  15  20  23  29  30  40  45  60  0
```

All of the elements in the two arrays were correctly merged except the last element in the second array. Can you figure out what the problem is and how to fix it? It is a subtle problem but this type of bug occurs frequently in program development.

The problem is that when all of the elements of one array have been copied to the output array, the loop terminates because the loop continues only as long as there is data in both arrays. This causes any remaining elements in the other array to be ignored. We can fix this problem by adding an additional loop that copies the remaining elements from the second array to the end of the output array.

```
public static int[] mergeArrays(int[] array1, int[] array2)
{
     int count;
     int count1;
     int count2;
     int[] array3;

     array3 = new int[array1.length + array2.length];

     count1 = 0;
     count2 = 0;
     for (count=0; count1<array1.length && count2<array2.length; count++)
     {
        if (array1[count1] < array2[count2])
        {
           array3[count] = array1[count1];
           count1++;
        }
        else
        {
           array3[count] = array2[count2];
           count2++;
        }
     }
     for (count=count; count2<array2.length; count++, count2++)
     {
        array3[count] = array2[count2];
     }
     printList(array3);
     return array3;
}

10  15  20  23  29  30  40  45  60  70
```

Now the output that is generated is correct.

The astute programmer should realize that the second array will not always be the array that has some elements left in it at the end of the primary loop – if the last element in the first array is larger than the last element in the second array, the first array will be the one that still contains some elements. So another loop is required to handle this situation.

```
public static int[] mergeArrays(int[] array1, int[] array2)
{
     int count;
     int count1;
     int count2;
     int[] array3;

     count1 = 0;
     count2 = 0;
     array3 = new int[array1.length + array2.length];
     for (count=0; count1<array1.length && count2<array2.length; count++)
     {
        if (array1[count1] < array2[count2])
        {
           array3[count] = array1[count1];
           count1++;
        }
        else
        {
           array3[count] = array2[count2];
           count2++;
        }
     }
     for (count=count; count1<array1.length; count++, count1++)
     {
        array3[count] = array1[count1];
     }
     for (count=count; count2<array2.length; count++, count2++)
     {
        array3[count] = array2[count2];
     }
     printList(array3);
     return array3;
}
```

In the methods above, a loop similar to the loop below was used to copy elements from one array to another array.

```
for (count=count; count1<array1.length; count++, count1++)
{
     array3[count] = array1[count1];
}
```

This loop uses a somewhat confusing initialization of the variable `count`: `count` is assigned its current value. This is perfectly valid Java and the statement does what is intended but the statement does look somewhat weird. If you prefer, you could add another variable so that there is no confusion about the statement.

```
for (count2=count; count1<array1.length; count2++, count1++)
{
     array3[count2] = array1[count1];
}
```

Alternatively, the initialization portion of the statement could be omitted, as shown below.

```
for ( ; count1<array1.length; count++, count1++)
{
     array3[count] = array1[count1];
}
```

In the variation of the loop shown above, `count` is assumed to have been initialized before the loop. This loop is no different from the version that contained the explicit assignment

```
for (count=count; count1<array1.length; count++) . . .
```

All 3 of the variations work equally well, it is the programmer's choice as to which variation the programmer is more comfortable with.

With the revised algorithm, regardless of which array contains the smaller last element, all elements will be merged correctly into the output array.

So our merge algorithm is now complete and correct. By developing it in small steps, we were able to concentrate on one part of the algorithm at a time. If you understand the processing that is required in such an algorithm, it is not necessary to take such small steps. However, the danger in taking bigger steps is that you may make an error in the design and have to go back and rewrite some (or a lot) of the code that you have written. The advantage in taking small steps is that if you do have to rewrite a portion of the code, it should be a relatively small portion.

To ensure that you can always revert to a previous version of your program, you should make a backup copy of your source file every few minutes.

Once a program is complete, a good programmer reviews the program to see if there are any portions of the program that could be improved or made clearer. (This is referred to as **refactoring**.)

In the program above, the two loops after the primary loop can be eliminated if the primary loop is changed as shown below. The statements that are equivalent to the two loops are moved inside the primary loop and changed from "for" statements to "if" statements.

Note that the logical operator in the primary loop's "for" statement has been changed to "or" instead of "and".

```
public static int[] mergeArrays(int[] array1, int[] array2)
{
     int count;
     int count1;
     int count2;
     int[] array3;

     array3 = new int[array1.length + array2.length];
     count1 = 0;
     count2 = 0;
     for (count=0; count1<array1.length || count2<array2.length; count++)
     {   // This version was suggested by John Braico

         if (count2 >= array2.length)
         {   // if array2 has no more elements, copy the remaining elements in array1
             array3[count] = array1[count1];
             count1++;
         }
         else if (count1 >= array1.length)
         {   // if array1 has no more elements, copy the remaining elements in array2
             array3[count] = array2[count2];
             count2++;
         }
         // to get this far, each array has at least one element in it
         else if (array1[count1] < array2[count2])
         {   // array1 has the smaller element so copy it to array3
             array3[count] = array1[count1];
             count1++;
         }
         else
         {   // array2 has the smaller element so copy it to array3
             array3[count] = array2[count2];
             count2++;
         }
     }
     printList(array3);
     return array3;
}
```

If you examine the loop, there are 4 different conditions (if/else statements) but only two different situations: either copy an element from array1 to array3 or copy an element from array2 to array3. By rearranging the conditions, it is possible to collapse the 4 conditions down to 2 conditions. The following version of the merge algorithm illustrates this refactoring.

```
public static int[] mergeArrays(int[] array1, int[] array2)
{
     int count;
     int count1;
     int count2;
     int[] array3;

     array3 = new int[array1.length + array2.length];

     count1 = 0;
     count2 = 0;
     for (count=0; count1<array1.length || count2<array2.length; count++)
     {
        // copy from array1 to array3 as long as array1 has at least
        // one element remaining and either array2 doesn't have
        // any elements remaining or array2 does have at least one element
        // remaining but it is larger than the element in array1
        if ((count1<array1.length) &&
                  ((count2>=array2.length) || (array1[count1]<array2[count2])))
        {
           array3[count] = array1[count1];
           count1++;
        }
        else
        {
           array3[count] = array2[count2];
           count2++;
        }
     }
     printList(array3);
     return array3;
}
```

The program is now slightly shorter but its complexity has increased slightly because it is not as easy to look at the method and determine quickly whether or not it works correctly. One of the basic principles of refactoring is **don't make a program more complex just to save a few lines of code**. Instead make the program easier to understand, even if it means adding some additional lines of code. So making this change is not necessarily a good idea at this time. (This is an example of a borderline refactoring – it is not obviously better but also not obviously worse.)

However, if more than two arrays must be merged, this refactoring is quite useful, but until we need to merge more than two arrays, the previous algorithm is also perfectly acceptable. (We will take advantage of this refactoring in Chapter 4 – File Input and Output.)

2.14 Merging Strings

The algorithms in the previous section merged arrays of int's. However, any pair of arrays can be merged with the almost the same algorithm. The following method illustrates how two sorted arrays of Strings can be merged into one sorted array of Strings. The only change that must be made is to change the < comparison operator that is used to compare integers to the compareTo method that is used to compare Strings. The compareTo method is described in Chapter 5 – Strings.

```
public static String[] mergeArrays(String[] array1, String[] array2)
{
     int count;
     int count1;
     int count2;
     String[] array3;

     array3 = new String[array1.length + array2.length];

     count1 = 0;
     count2 = 0;
     for (count=0; count1<array1.length || count2<array2.length; count++)
     {
        if ((count1<array1.length) &&
           ((count2>=array2.length)||(array1[count1].compareTo(array2[count2])<0)))
        {
           array3[count] = array1[count1];
           count1++;
        }
        else
        {
           array3[count] = array2[count2];
           count2++;
        }
     }
     printList(array3);
     return array3;
}

public static void printList(String[] list)
{
     int count;

     for (count=0; count<list.length; count++)
     {
        System.out.print(list[count] +"  ");
     }
     System.out.println();
}
```

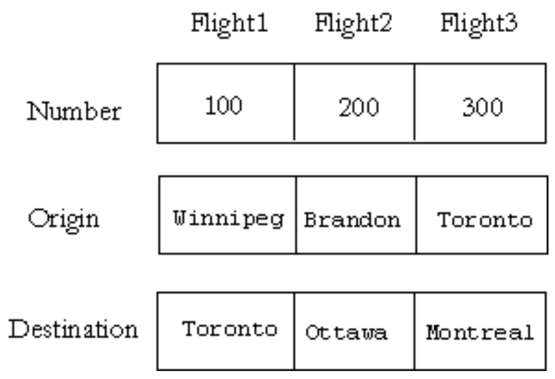
Note that it is valid to have two methods (such as mergeArrays or printList) with the same name in the same program as long as the methods with the same names have different **signatures** (the number, type, and order of parameters). This is referred to as "**overloading**" a method.

2.15 Parallel Arrays

In the earlier examples that dealt with searching and merging, we manipulated only one array of values. As a result, the array could be created in the createList method and returned explicitly (using a "return" statement) by the createList method. If multiple arrays are used to store different pieces of information, then we can not create and return the arrays from createList since only one array can be returned as the result of a method. To get around this problem, we will examine several techniques that can be used. As we will see in Chapter 3 – Objects, the use of parallel arrays is no longer required but the programming techniques used when processing parallel arrays are useful to know.

The following examples maintain information about airline flights. Each flight is represented by 3 values: the flight number (an integer), the flight origin (a string), and the flight destination (a string). Obviously more information could be defined but these 3 values are sufficient for now.

In the following diagram, the information about 3 flights is defined. Flight 100 travels from Winnipeg to Toronto, flight 200 travels from Brandon to Ottawa, and flight 300 travels from Toronto to Montreal. Using the techniques that are available to us now, this information must be defined using 3 different arrays. (In some procedural programming languages, such as C, there is a mechanism – the struct – that permits different pieces of information to be grouped together. In Java, this is not possible unless objects are used.)



2.15.1 Maintaining the Number of Elements in the Arrays

If we don't know in advance how many flights will be created, we could initialize the arrays in the main program and then pass them to the create method. The create method adds flight information to the arrays and returns the actual number of flights that have been stored in the arrays. The following example illustrates this technique.

Note that each of the 3 arrays is instantiated at the beginning of the main program. If the maximum size is too small, the program will not work correctly.

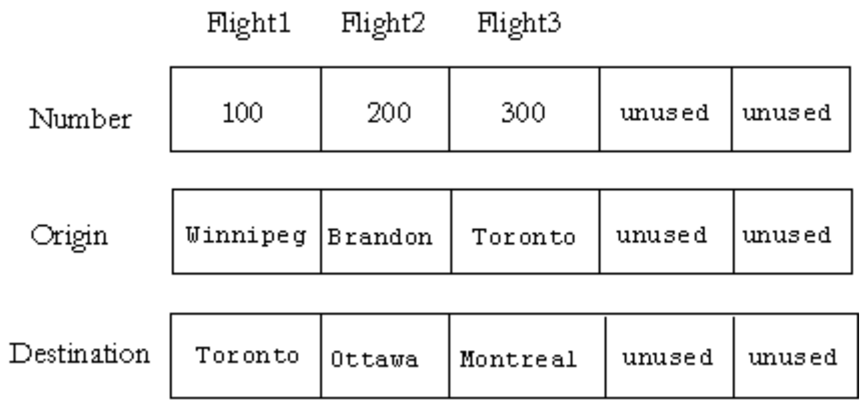
```
public static void main(String[] parms)
{
     int[] flightNumbers = new int[100];
     String[] flightOrigins = new String[100];
     String[] flightDestinations = new String[100];
     String searchValue = "Toronto";
     int result;
     int numFlights;
     numFlights = createFlights(flightNumbers, flightOrigins, flightDestinations);
     result = searchOrigin(numFlights, flightOrigins, searchValue);
     System.out.println("Result is: " +result);
     System.exit(0);
}
```

The createFlights method below has the flight information hard coded in the method. Again, this is a good way to get started growing a program.

```
public static void createFlights(int[] flightNumbers, String[] flightOrigins,
                                                      String[] flightDestinations)
{
     int numFlights;
     int flightCount;
     int[] myFlightNumbers = new int[] {100, 200, 300, 400};
     String[] myFlightOrigins = new String[] {"Winnipeg", "Brandon", "Toronto"};
     String[] myFlightDestinations = new String[] {"Toronto","Ottawa","Montreal"};

     numFlights = myFlightNumbers.length;
     for (flightCount=0; flightCount<numFlights; flightCount++)
     {
         flightNumbers[flightCount] = myFlightNumbers[flightCount];
         flightOrigins[flightCount] = myFlightOrigins[flightCount];
         flightDestinations[flightCount] = myFlightDestinations[flightCount];
     }
}
```

Note that createFlights modifies the contents of its 3 parameters. These modifications to the arrays remain in effect when control returns to the main method. Although the general rule is that modifications made to parameters by a method do not affect the calling method, with arrays that already have been instantiated (using an "= new …" statement) prior to calling a method, changes made to the **contents** of the arrays in the called method remain in effect when control returns to the calling method. (We will examine this topic in more detail in Chapter 7 – Object Representation.)

The following diagram illustrates the contents of the 3 arrays when the arrays are initialized to size 5 and numFlights is equal to 3.



Note that there is some unused space (or **free space**) at the end of each array. This free space does not cause any problems since we know the number of actual flights that are stored and so we will not attempt to process entries that have not been given values.

The following version of the createFlights method allows the user to enter the information using JOptionPane. The method begins by asking the user how many flights will be entered. The values that define each flight are then read into the corresponding locations in the 3 arrays. (Again, this is a particularly time-consuming way to provide information but it is the best that we can do right now.) The method returns the number of elements in the arrays to the main program.

```
public static int createFlights(int[] flightNumbers, String[] flightOrigins,
                                                       String[] flightDestinations)
{
     int numFlights;
     int flightNumber;
     int count;
     String flightOrigin;
     String flightDestination;
     String string;

     string = JOptionPane.showInputDialog("Please enter the number of flights:");
     numFlights = Integer.parseInt(string);
     for (count=0; count<numFlights; count++)
     {
        string = JOptionPane.showInputDialog("Please enter a flight number:");
        flightNumber = Integer.parseInt(string);
        flightNumbers[count] = flightNumber;
        flightOrigin = JOptionPane.showInputDialog("Please enter an origin:");
        flightOrigins[count] = flightOrigin;
        flightDestination=JOptionPane.showInputDialog(
                                                    "Please enter a destination:");
        flightDestinations[count] = flightDestination;
     }
     return numFlights;
}
```

Alternatively, instead of asking the user to enter the number of flights in advance, the method could keep reading flights until a special value is entered. The following version of createFlights generates the same results as the previous version but it accomplishes its task by reading flight information until a flight number of -1 is entered. Note that this version still requires that the arrays be instantiated in the calling method (the main method, in this case).

```
public static int createFlights(int[] flightNumbers, String[] flightOrigins,
                                                     String[] flightDestinations)
{
     int count;
     int flightNumber;
     String flightOrigin;
     String flightDestination;
     String string;

     count = 0;
     string = JOptionPane.showInputDialog("Please enter a flight number:");
     flightNumber = Integer.parseInt(string);
     while (flightNumber != -1)
     {
        flightNumbers[count] = flightNumber;
        flightOrigin = JOptionPane.showInputDialog("Please enter an origin:");
        flightOrigins[count] = flightOrigin;
        flightDestination = JOptionPane.showInputDialog("Enter a destination:");
        flightDestinations[count] = flightDestination;
        count++;
        string = JOptionPane.showInputDialog("Please enter a flight number:");
        flightNumber = Integer.parseInt(string);
     }
     return count;
}
```

In order to process the flights, each processing routine must know how many elements are stored in the arrays. For example, the following method searches the list of flights for a flight that leaves from a specific location.

```
public static int searchOrigin(int numFlights, String[] flightOrigins,
                                                              String searchValue)
{
     int count;
     int result;

     result = -1;
     for (count=0; count<numFlights && result==-1; count++)
     {
        if (flightOrigins[count].equals(searchValue))
        {
           result = count;
        }
     }
     return result;
}
```

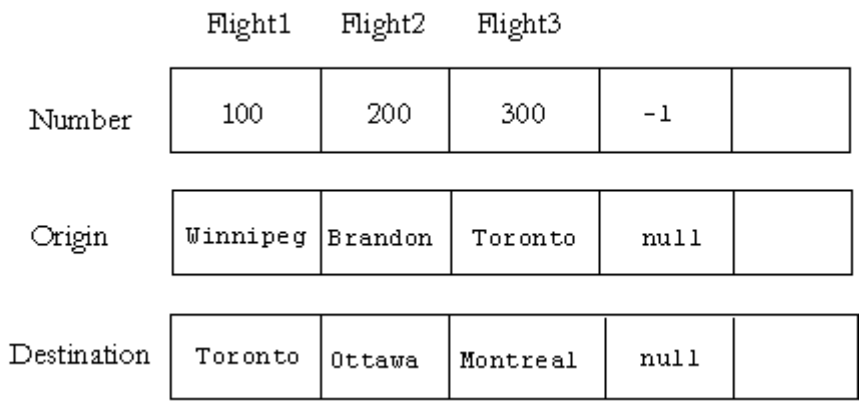
2.15.2 Using a Sentinel Value

An alternative technique to maintaining the number of array elements is to create the arrays in advance and then mark the end of the useful information in each array by a "**sentinel**" value. The following example uses this technique to maintain information about the airline flights.

```
public static void main(String[] parms)
{
     int[] flightNumbers = new int[100];
     String[] flightOrigins = new String[100];
     String[] flightDestinations = new String[100];
     String searchValue = "Toronto";
     int result;

     createFlights(flightNumbers, flightOrigins, flightDestinations);
     result = searchOrigin(flightOrigins, searchValue);
     System.out.println("Result is: " +result);
     System.exit(0);
}
```

In the createFlights method, sentinel values of -1, "", and `null` are used to indicate the end of the flight information in the arrays. By using a sentinel value, we do not need to maintain the number of elements in each array.

The value `null` is a special value that indicates that an object has not yet been given a value. We will examine the `null` value more in subsequent chapters.



```
public static void createFlights(int[] flightNumbers, String[] flightOrigins,
                                                     String[] flightDestinations)
{
     int count;
     int flightNumber;
     String flightOrigin;
     String flightDestination;
     String string;

     count = 0;
     string = JOptionPane.showInputDialog("Please enter a flight number:");
     flightNumber = Integer.parseInt(string);
     while (flightNumber != -1)
     {
        flightNumbers[count] = flightNumber;
        flightOrigin = JOptionPane.showInputDialog("Please enter an origin:");
        flightOrigins[count] = flightOrigin;
        flightDestination = JOptionPane.showInputDialog("Enter a destination:");
        flightDestinations[count] = flightDestination;
        count++;
        string = JOptionPane.showInputDialog("Please enter a flight number:");
        flightNumber = Integer.parseInt(string);
     }
     flightNumbers[count] = -1;
     flightOrigins[count] = null;
     flightDestinations[count] = null;
}
```

After making this change to the createFlights method, we also have to modify each method that uses the flight information so that processing is terminated when the sentinel value is encounted. The change to the searchOrigin method is shown below.

```
public static int searchOrigin(String[] flightOrigins, String searchValue)
{
     int count;
     int result;

     result = -1;
     for (count=0; result==-1 && (flightOrigins[count] != null)); count++)
     {
        if (flightOrigins[count].equals(searchValue))
        {
           result = count;
        }
     }
     return result;
}
```

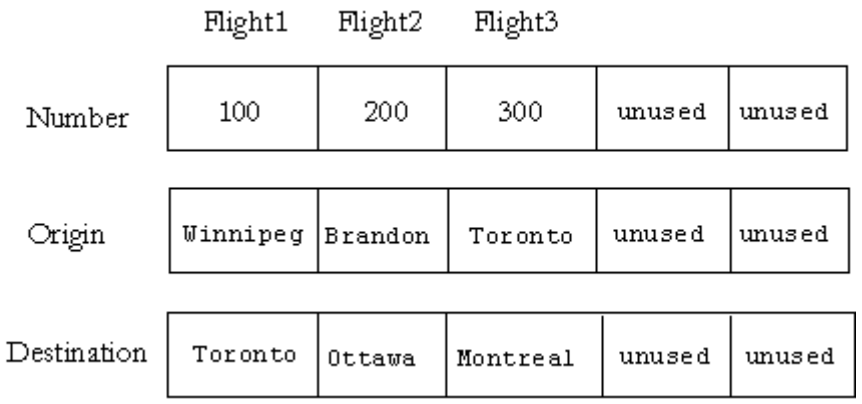
Although we do not have to know how many elements are in each array, we do have to know the sentinel value for each array. If the sentinel value changes, the change could affect a large number of methods that manipulate the arrays.

2.15.3 Copying the Arrays

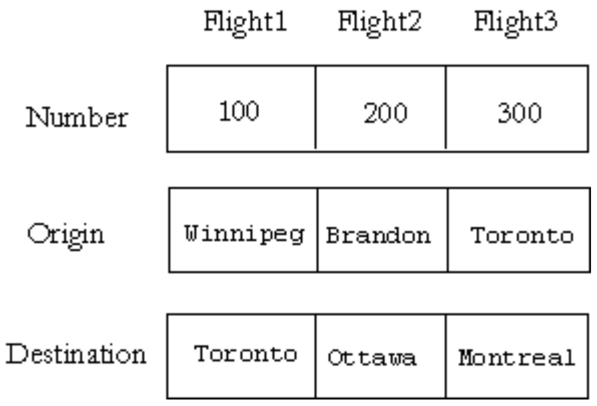
A variation on the first technique that we used involves creating the 3 initial arrays but then copying their contents into arrays that are exactly the right size. This requires a bit of additional work in the main program but all subsequent methods that manipulate the arrays can assume that the arrays are completely full of flight information.

```
public static void main(String[] parms)
{
     int[] initialFlightNumbers = new int[100];
     String[] initialFlightOrigins = new String[100];
     String[] initialFlightDestinations = new String[100];
     int[] flightNumbers;
     String[] flightOrigins;
     String[] flightDestinations;
     String searchValue = "Toronto";
     int result;
     int numFlights;
     int count;

     numFlights = createFlights(initialFlightNumbers, initialFlightOrigins,
                                                  initialFlightDestinations);
     flightNumbers = new int[numFlights];
     flightOrigins = new String[numFlights];
     flightDestinations = new String[numFlights];

     for (count=0; count<numFlights; count++)
     {
        flightNumbers[count] = initialFlightNumbers[count];
        flightOrigins[count] = initialFlightOrigins[count];
        flightDestinations[count] = initialFlightDestinations[count];
     }

     result = searchOrigin(flightOrigins, searchValue);
     System.out.println("Result is: " +result);
     System.exit(0);
}
```

The createFlights method returns the original arrays filled with numFlights flights.



The main program then copies the contents of each of the 3 arrays into an array of the same type that is exactly the size required.



Once the new arrays have been created, the methods that process the flight information can now process all elements in the arrays that are provided instead of having to know the number of elements or know the sentinel value at the end of each array.

```
public static int searchOrigin(String[] flightOrigins, String searchValue)
{
     int count;
     int result;

     result = -1;
     for (count=0; count<flightOrigins.length && result==-1; count++)
     {
        if (flightOrigins[count].equals(searchValue))
        {
           result = count;
        }
     }
     return result;
}
```

Although the loop in the main method above that copies the contents of the 3 initial arrays into the 3 arrays of the correct size is quite simple, Java's System.arraycopy method provides a more convenient mechanism for copying the arrays. Each of the following 3 statements copies the contents of one array into another array. Note that both arrays must be instantiated before they can be used in System.arraycopy.

```
flightNumbers = new int[numFlights];
flightOrigins = new String[numFlights];
flightDestinations = new String[numFlights];

System.arraycopy(initialFlightNumbers, 0, flightNumbers, 0, numFlights);
System.arraycopy(initialFlightOrigins, 0, flightOrigins, 0, numFlights);
System.arraycopy(initialFlightDestinations, 0, flightDestinations, 0, numFlights);
```

2.15.4 Global Variables

There is one final technique that can be used to create the parallel arrays and to make them available to the processing methods. This technique uses "global variables". A global variable is a variable that is defined at the beginning of a class, outside of the methods defined within the class. Such variables can be accessed by any method within the class and so the variables do not have to be passed as parameters from one method to another method. Notice that the variables must be defined as **`static`** variables; this will be discussed more in Chapter 3 – Objects).

As a general rule, global variables should not be used in a program such as the one below, but occasionally, there are times when the use of global variables is appropriate.

```
public class Parallel
{
     static int[] flightNumbers;
     static String[] flightOrigins;
     static String[] flightDestinations;

     public static void main(String[] parms)
     {
        String searchOrigin = "Toronto";
        int result;

        createFlights();
        printFlights();
        result = searchOrigin(searchOrigin);
        System.out.println(result);
     }

     public static void createFlights()
     {
        int numFlights;
        int flightNumber;
        int count;
        String flightOrigin;
        String flightDestination;
        String string;
        flightNumbers = new int[] {100, 200, 300, 400};
        flightOrigins = new String[] {"Winnipeg","Toronto","Winnipeg","Brandon"};
        flightDestinations= new String[] {"Toronto","Montreal","Halifax","ThePas"};
    }

    public static int searchOrigin(String searchOrigin)
    {
        int count;
        int result;

        result = -1;
        for (count=0; count<flightNumbers.length && result == -1; count++)
        {
            if (flightOrigins[count].equals(searchOrigin))
            {
                result = count;
            }
        }

        return result;
    }

    public static void printFlights()
    {
        int count;

        System.out.println("Flights: ");
        for (count=0; count<flightNumbers.length; count++)
        {
            System.out.println(flightNumbers[count] +"  "
                        +flightOrigins[count] +"  " +flightDestinations[count]);
        }
        System.out.println();
    }
}
```

Since the variables are global – accessible by all methods in the class – it is not necessary to pass the variables as parameters.

2.15.5 Recap

A problem still remains with all of the createFlight methods – they may attempt to overfill an array if there are more values provided than there are array locations. This problem can be handled in the same manner as earlier by ensuring that `count` does not run off the end of the array, that is, by changing the while statement from:

```
    while (flightNumber != -1)
```

to:

```
    while (flightNumber != -1 && count<flightNumbers.length)
```

Adding the condition to the loop ensures that a subscript out of range error is not generated. However, the user should still be notified that too many flights are being created.

We will not show appropriate error checking in these notes so that we can focus on the algorithms being used. **However, error checking must be performed in any code that is written and it should be included at the beginning of code development, not added as an afterthought at the end.** At the very least, an error message should be printed on System.out when an error occurs. (We will examine an alternate technique in the section on Exceptions at the end of Chapter 4 – File Input and Output.)

The examples in this section illustrate the use of parallel arrays in which different pieces of information that describe something (in this case, an airline flight) are stored in individual arrays. This technique was used in the "prehistoric" days of computing when there was no alternative. Now, however, object orientation has reduced the need for parallel arrays. We will examine object orientation in Chapter 3 – Objects.

2.16 Growing Algorithms

We have been referring to "growing" software throughout this chapter without defining what growing software means. (For those who don't like the term "growing software", the term "iterative development" can be substituted.) Growing is simply a metaphor that is intended to replace the "engineering" approach to developing software that many academic institutions encourage. The engineering approach involves very carefully designing the functionality of the software before beginning the programming process. Software "engineering" design involves identifying all of the classes, methods, and variables that are required. With this approach, it is assumed that once you have made a design decision, you don't have to change the decision at a later time. While this approach works well when building a large bridge, it does not carry over well into building small to medium-size software systems. When developing software, the external appearance of a system may be well defined at the beginning of a project (or assignment) but the internal details are typically not known in advance. Using a growing approach to developing software frees the developer from having to spend a significant amount of time designing the complete internal workings of a system prior to beginning development. Instead, a small portion of the system is designed, then implemented and tested to ensure that it runs correctly, refined if there are small improvements that can be made, and then extended to include more functionality. (This is referred to as the **design/implement/refactor** process.) Using this approach, the programmer does not write pages and pages of code at one time and then try to get the program working, a very time-consuming process when a problem could be anywhere in the pages of code. So starting small and then gradually increasing the scope and functionality of a program actually allow the programmer to develop code faster than writing large portions of code at one time. This approach also ensures that there is always a system that actually runs, although the system does not provide much functionality at the beginning. In contrast with the engineering approach, nothing works until the entire system has been designed, developed, and debugged.

This is a risky strategy with software since forgetting one aspect of the system when doing the design can lead to a system that has to be extensively rewritten. The engineering approach is necessary when constructing physical structures. For example, if you forget to include the rebar when building a bridge, you don't have the ability to go back and fix the problem without tearing down the bridge. With the growing approach, problems are encounted and solved as the system is developed, not at the end when all the pieces are first put together. Again, the engineering approach is appropriate when building many types of physical structures but is less appropriate when building software systems. Software systems can be extremely complex and the best way to approach a complex system is by starting small and then adding to it slowly.

The growing approach is a foundation of the “agile” software development methodologies and, in particular, eXtreme Programming (XP).

2.17 Debugging

Even if a program is grown slowly, errors inevitably creep into the program. There are 3 primary types of errors that are encounted: compile errors, run-time errors, and logic errors. In the following example, we examine the three types of errors in a simple loop that accumulates the sum of the elements in an array.

Compile errors cause an error message to be generated by the compiler. A compile error is the result of an incorrect statement, that is, the statement does not conform to the Java language specification. Compile errors can be the result of many different types of problems and a good programmer will learn to identify the problems that cause compile errors.

```
int[] array = {1,2,3,4,5};
sum = 0;
for (count=1; count<=array.length(); count++)
{
     sum = sum + array[count];
}
System.out.println("The sum of the array elements is: " +sum);
```

The statements above generate the following compiler message:

```
Debug.java:10: cannot resolve symbol
symbol  : method length ()
location: class int[]
        for (count=1; count<=array.length(); count++)
                                  ^
1 error
```

The message indicates that there is a problem with the “**method** length”. This should spark some level of recognition that “length” is not a method but instead is a variable and that removing the parentheses that follow length should fix the problem. If the solution to the error does not appear obvious, then it is time to consult your class notes, the textbook, or the on-line Java documentation.

Once the parentheses are removed, the program compiles correctly.

```
int[] array = {1,2,3,4,5};
sum = 0;
for (count=1; count<=array.length; count++)
{
     sum = sum + array[count];
}
System.out.println("The sum of the array elements is: " +sum);
```

However, when the program is run, it produces the following run-time error:

```
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 5
        at Debug.main(Debug.java:5)
```

This error is relatively straightforward since Java has told us that the problem occurs on line number 5 (`sum = sum + array[count];`) and the problem is that the array subscript (5) is larger than the largest valid subscript for that array. This error illustrates one problem with debugging: the statement where the error occurs is not necessarily the statement that must be fixed. In this example, the problem is that the for loop varied count up to and including array.length instead of varying count up to but not including array.length.

This problem is also easy to fix so that the program does not attempt to process non-existent elements in the array.

```
int[] array = {1,2,3,4,5};
sum = 0;
for (count=1; count<array.length; count++)
{
     sum = sum + array[count];
}
System.out.println("The sum of the array elements is: " +sum);
```

The program now runs successfully and generates the following result.

```
The sum of the array elements is: 14
```

A quick manual check of the output indicates that it is not correct, the program generates the value 14 when the correct result is 15. Assuming that you have no idea why the correct result was not generated, rather than stare at the program for hours, adding one simple print statement to the program should help to determine the error.

```
int[] array = {1,2,3,4,5};
sum = 0;
for (count=1; count<array.length; count++)
{
     sum = sum + array[count];
     System.out.println(count +" " +array[count] +" " +sum);
}
System.out.println("The sum of the array elements is: " +sum);
```

Adding this print statement to the inside of the loop generates the following output:

```
1 2 2
2 3 5
3 4 9
4 5 14
The sum of the array elements is: 14
```

As can be seen from the output, the program correctly computes the sum of the array elements that are referenced in the loop but the problem is that the program does not process the first element in the array.

Changing the program as shown below causes the correct answer to be generated for this particular array.

```
int[] array = {1,2,3,4,5};
sum = 0;
for (count=1; count<array.length; count++)
{
     sum = sum + array[count];
}
sum = sum + 1;
System.out.println("The sum of the array elements is: " +sum);
```

But the program will not work correctly if the first element in the array is not 1. Changing the program again as shown below causes the correct answer to be generated for all arrays.

```
int[] array = {1,2,3,4,5};
sum = 0;
for (count=1; count<array.length; count++)
{
     sum = sum + array[count];
}
sum = sum + array[0];
System.out.println("The sum of the array elements is: " +sum);
```

However, the solution is not elegant or satisfying. The solution has addressed the problem but not the cause of the problem. A programmer who has to make changes to this routine at a later date will likely be confused by the program's logic. Taking a closer look at the program reveals that the problem is caused in the "for" statement when count is initialized to 1 instead of 0. So the correct change to the program is:

```
int[] array = {1,2,3,4,5};
sum = 0;
for (count=0; count<array.length; count++)
{
     sum = sum + array[count];
}
System.out.println("The sum of the array elements is: " +sum);
```

Now the program works correctly in all situations and there are no strange statements such as "sum = sum + array[0];". (Note that the previous version would not have worked correctly if the array was empty.) This example illustrates how programmers may develop tunnel vision when looking at a problem; as a result, they miss the bigger picture and the more appropriate solution to a problem.

In addition to ensuring that the program works correctly with an array that contains elements, you should also ensure that it works correctly if the array does not contain any elements. Note that it is perfectly valid to declare an array as:

```
int[] array = new int[0];
```

If this array is processed by the code above, sum is given the value 0. This is what would normally be expected although if the array is always supposed to contain at least one value, then a message should be generated for the user if an empty array is encounted.

The process of debugging is not a simple one; debugging involves begin able to trace the logic of a program and ensure that each section of code is performing its task correctly. For novice programmers, the best debugging strategy is to include "print" statements in a program that is not behaving as expected. The print statements are used to display the current values of important variables as the program is executed.

Some other strategies that often help if a problem refuses to go away are:

- make frequent copies of your source code so that you can always revert to an earlier version that was working correctly;
- take a break from programming and do something else; it is common for problems to become obvious after taking a break and not thinking about the program;
- rewrite the problem code in a different way if it resists all efforts at fixing the problem.

As larger and more complicated programs are developed, using a simple editor such as TextPad is not sufficient. A more powerful Java development system such as Eclipse provides the programmer with additional tools that make program development much easier. For example, Eclipse provides a "**debugger**" that can be used to examine the contents of variables as a program is executing. A good debugger allows the programmer to locate bugs much more quickly than adding print statements to a program.

2.18 Testing

Debugging is the processing of determining the source of an error or bug. As was mentioned earlier in this chapter, it is important to perform sufficient testing of code that you are confident that the code works correctly in all situations.

When developing a program, ensure that you test the program with more than one set of data. Failure to do so often leads to bugs in code that do not show up until the code is put into production.

In particular, test the boundary conditions of a method; for example, if an array is being processed, test the method with a null array, an array of length 0, an array with just one element, etc.

2.19 Defensive Programming

Defensive programming involves taking steps to ensure that even in the presence of errors, your code will function correctly or at least, identify the fact there is an error and generate an appropriate (i.e. meaningful) error message. Much of defensive programming involves following standards that define how programs should be written. For example,

- make frequent copies of your source code so that you can always revert to an earlier version that was working correctly;
- keep methods small;
- each method should have one purpose;
- use parentheses, whether they are required or not;
- use parentheses and indentation in a consistent manner;
- use appropriate variable names;
- write only one statement per line;
- refactor code – improve variable names; simplify code; move code into methods.

And, most importantly, **grow code slowly** – you will find that a working program is developed more quickly with this technique than if you try to write the entire program at once.

2.20 Algorithm Efficiency

When developing the algorithms in this chapter (and in subsequent chapters), the focus is on developing algorithms that are correct and easy to understand. Some inefficiencies will be corrected but we will not attempt to make each algorithm optimal in terms of its memory requirements and/or execution time. There are several reasons for this decision. First, most algorithms do not need to be optimal unless they are being executed on a frequent basis and/or for long periods of time. Secondly, most programmers can not identify the "**hot spots**" (areas of a program in which a significant amount of time is spent); most often, programmers' assumptions about hot spots are incorrect. There are tools, referred to as "profilers", that measure the amount of time spent in each section of code – profilers provide an accurate picture of the routines that are using a significant number of computer cycles. Finally, in an attempt to make algorithms more efficient, many programmers make the algorithm significantly more difficult to understand and to maintain. This increases the likelihood that a subsequent programmer who must make changes to the program will "break" the program by making changes incorrectly due to the increased complexity of the "optimized" program.

> "Don Knuth has observed that premature optimization is the root of much programming evil; it can compromise the correctness, functionality and maintainability of programs. Save concern for efficiency when it matters." Jon Bentley, *Programming Pearls, Second Edition,* page 96, Addison Wesley
>
> "Barry Boehm reports that he has measured that 20 percent of the routines consume 80 percent of the execution time (Boehm 1987b). In the classic paper, "An Empirical Study of FORTRAN programs," Donald Knuth found that less than 4 percent of a program usually accounts for more than 50 percent of its run time (Knuth 1971).
>
> "Knuth used a line-count profiler to discover this surprising relationship, and the implications for optimization are clear. **Measure the code to find the hot spots, then put your resources into optimizing the few percent that are used the most**. Knuth profiled his line-count program and found that it was spending half the execution time in two loops. He changed a few lines of code and doubled the speed of the profiler in less than an hour.
>
> "Jon Bentley describes a case in which a thousand-line program spent 80 percent of its time in a five-line square-root routine. By tripling the speed of the square-root routine he doubled the speed of the program (Bentley 1988).
>
> "Bentley also reports the case of a team that discovered that half an operating system's time was spent in a small loop. They rewrote the loop in microcode and made the loop 10 times faster, but it didn't change the system's performance – they had rewritten the system's idle loop.
>
> "The team that designed the Algol language – the precursor to Pascal, C, and Ada and one of the most influential languages ever – received the following advice: The best is the enemy of the good. Working toward perfection may prevent completion. **Complete it first, then perfect it. The part that needs to be perfect is usually small**."
>
> http://www.stevemcconnell.com/cctune.htm
> or Steve McConnell, *Code Complete*, Chapter 58.2: Introduction to Code Tuning

2.21 Summary

There is a significant amount of information about Java available on the web as well as in textbooks. The Java API (application programming interface) defines all of the classes, methods, and variables that are used in Java. The Java 6 API is available at: `http://download.oracle.com/javase/6/docs/api/` The API documentation is not easy to understand but it is worth while taking a look at it if you are having difficulties with a particular Java method or class. The documentation may also be downloaded to your computer.

The complete Java Language Specification, Third Edition, is available for viewing and/or download at the site: `http://java.sun.com/docs/books/jls/` The book is about 680 pages long and is not easy to read but is available for those who want to understand the intricacies of Java.

3 OBJECTS

3.1 Introduction

In this chapter we examine the basic structure of objects. We will not define all aspects of objects – those can be found in any introductory Java text. Instead, we will focus on the appropriate use of objects and some of the features of objects that the programmer must use with care.

3.2 Data Types

Chapter 1 introduced the basics of procedural programming. When variables are required in a program, they are defined with an appropriate declaration statement of the following form:

```
data-type variable;
```

For example,

```
int count;
```

This statement declares `count` to be a variable of type `int`. `int` is one of the primitive data types in Java (with some of the others being `boolean`, `char`, `float`, and `double`). The primitive data types begin with a lower-case letter.

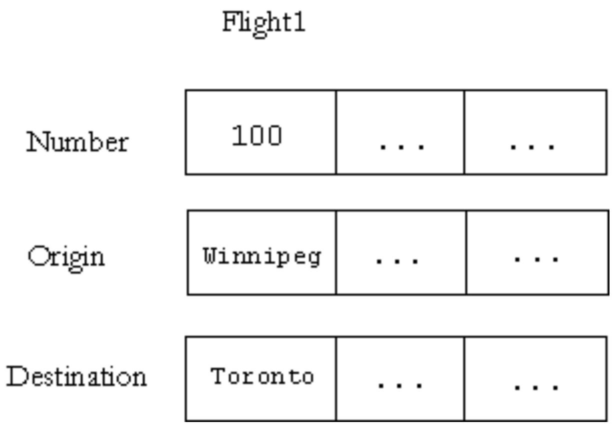
The primitive data types are built in to the Java compiler and can be used by the programmer at any time.

We also used the `String` data type in Chapter 1 but without explaining exactly what Strings are. A `String` is a system-defined data type that is also provided by the Java compiler. Any data type that is not a primitive data type is referred to as an "**object**". The Java system includes a variety of system-defined data types, some of which will be examined in subsequent chapters. The programmer may also define new data types (objects) and that is the focus of this chapter.

An object must be defined before it can be used. For example, the `String` object is defined in one of the Java "packages". (Don't worry about what packages are.)

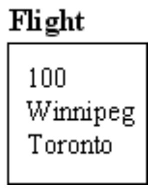
3.3 Basic Object Structure

In Chapter 2 – Growing Algorithms, we examined parallel arrays in which different pieces of information that described airline flights were stored in multiple arrays.



Using parallel arrays to maintain multiple pieces of information is not very convenient in a large program (or even a small program). (Non-object oriented languages often include a **struct** programming construct that permits related pieces of information to be grouped together as a unit.)

Since the development of object-oriented languages, the information that previously was stored in parallel arrays can now be stored in an object. Our diagramming convention (for now) will be to draw each object as a rectangle, place the name of the associated class above the rectangle, and show the data that the object contains inside the rectangle. The object below contains the 3 pieces of data that define a flight (and that would require 3 parallel arrays if procedural programming were used).



The definition of an object is relatively straightforward. Objects are defined using a `class` statement, as shown below.

```
public class Flight
{
}
```

Each class definition is normally placed in its own source file that has the same name and case as the class defined in the source file. So the class defined above would be stored in a file named Flight.java. By convention, the names of Java classes begin with an upper-case letter.

Now that we have defined a class, we can create objects that belong to the class. (Formally, an object is said to be an **instance** of the class.) Objects are **declared** in the same manner as other Java variables. The following statement declares flight1 to be a variable of type Flight.

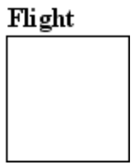
```
Flight flight1;
```

At this point, the variable has not been **instantiated** (assigned a value or initialized); flight1 is instantiated using the statement:

```
flight1 = new Flight();
```

This statement creates a new object of type Flight and assigns the object to the variable flight1 (this process is described more specificially in Chapter 7 – Object Representation).

Although we have created a Flight object, the object is quite useless because it does not contain any information or perform any processing.

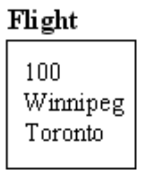


We can define variables inside the Flight class in the same manner that we defined variables in the programs written in the earlier chapters.

```
public class Flight
{
     int flightNumber;
     String flightOrigin;
     String flightDestination;
}
```

Now if we create a Flight object flight1, the object has 3 variables in it but we need a mechanism for accessing those variables. The simplest mechanism is shown in the program segment below.

```
public static void main(String[] parms)
{
     Flight flight1;

     flight1 = new Flight();
     flight1.flightNumber = 100;
     flight1.flightOrigin = "Winnipeg";
     flight1.flightDestination = "Toronto";
}
```



Now flight1 contains values for its 3 variables. The value of any of the object's variables may be accessed in the same manner. (We will soon see that this is not a good programming practice but it is good enough for now.)

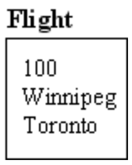
```
public static void main(String[] parms)
{
     Flight flight1;

     flight1 = new Flight();
     flight1.flightNumber = 100;
     flight1.flightOrigin = "Winnipeg";
     flight1.flightDestination = "Toronto";

     System.out.println(flight1.flightNumber +" " + flight1.flightOrigin
                                             +" " + flight1.flightDestination);
}
```

Normally when we instantiate an object, we would like to supply values for some or all of its variables at the same time. This can be accomplished by adding a "**constructor**" to the class. A constructor is a **method** that is executed when an object is created. The constructor has the same name as the class and does not return a value. A constructor may be passed parameters in the same manner that any method may be passed parameters.

```
public class Flight
{
     int flightNumber;
     String flightOrigin;
     String flightDestination;

     public Flight(int number, String origin, String destination)
     {
        flightNumber = number;
        flightOrigin = origin;
        flightDestination = destination;
     }
}
```

Now the object can be created in one statement, with the values of its 3 pieces of information provided to the constructor.

```
public static void main(String[] parms)
{
     Flight flight1;

     flight1 = new Flight(100, "Winnipeg", "Toronto");

     System.out.println(flight1.flightNumber +" " + flight1.flightOrigin
                                             +" " + flight1.flightDestination);
}
```



It is valid to have more than one constructor defined in a class as long as each constructor has a different signature (the number, type, and order of parameters). If a class does not include a constructor, Java adds a **default constructor** which looks like the following:

```
public class Flight
{
     int flightNumber;
     String flightOrigin;
     String flightDestination;

     public Flight()
     {
     }
}
```

The default constructor is not actually added to your source code but is added to the class definition internally by Java. It is perfectly valid for you to define your own default constructor – it would not contain any parameters but may contain statements that are executed when the object is created.

A class may also have methods defined in it. These methods perform work related to the class. For example, most classes include a `toString()` method that returns information about the object. For the Flight class, the `toString()` method could return the flight number, origin, and destination as a single string.

```
public class Flight
{
     int flightNumber;
     String flightOrigin;
     String flightDestination;

     public Flight(int number, String origin, String destination)
     {
        flightNumber = number;
        flightOrigin = origin;
        flightDestination = destination;
     }

     public String toString()
     {
        return flightNumber +" " +flightOrigin +" " +flightDestination;
     }
}
```

Now the contents of the object can be printed using just the `toString()` method.

```
public static void main(String[] parms)
{
     Flight flight1;

     flight1 = new Flight(100, "Toronto", "Winnipeg");

     System.out.println(flight1.toString());
}
```

To summarize, a class definition may include variables, constructors, and/or methods. We will examine the details as we progress in this chapter.

3.4 What is an Object?

In the previous section, we identified the basic components of an object: variables, constructors, and methods. (Note that with the exception of constructors, these are also the basic components of procedural programming.) From this point of view, an object is a programmer-defined data type that contains variables and/or methods. (Or an object is a system-defined data type if the object is included with the Java system; for example, a String is a system-defined data type.)

However, this is a very narrow definition of objects that focuses on implementation details, not on the purpose of objects. An object is better defined by **what it can do**.

For example, a clock object can display the current time. Internally, a real clock could consist of a series of gears or it could consist of some digital circuitry. We don't care how the clock object performs its functions, we just want to know what the current time is. Only the clockmaker (or programmer) needs to be aware of the internals of the object.

Objects communicate with each other by sending messages back and forth. With the clock object, we may be able to send a message to the clock that causes the current time to be modified or a message that causes the object to modify the format in which the time is displayed (digital format, analog format, 12/24 hour display, timezone display, etc.)

Similarly, we have used String objects without having to know how the processing within the object is carried out. All that we need to know is **what** the object is able to do; knowing **how** the object performs its processing is normally irrelevant.

Thinking of objects from the "outside" in terms of what they can do normally results in shorter development time and a higher-quality product. Someone still has to implement the internals of the object but both that person and the people who use the object should be concentrating on what the object will do, not how the object will perform its processing.

We view real-world "objects" from a similar perspective. Unless you are a particularly weird health-care professional, you see your friends as individuals who have differing capabilities, not as collections of internal organs and physiological systems, that is, you see the person, not the components.

We will often refer to the data and behaviours of objects but that is really just another way of describing what the objects can do.

3.5 Simple Class Structure

The class below illustrates the parts of a simple Java class.

```
public class Flight  // Class header
{
     int flightNumber;   // Instance variable
     String flightOrigin;
     String flightDestination;

     public Flight(int number,String origin,String destination)  // Constructor
     {
        flightNumber = number;
        flightOrigin = origin;
        flightDestination = destination;
     }

     public String toString()    // Instance method
     {
        return flightNumber +" " +flightOrigin +" " +flightDestination;
     }
}
```

Each class begins with a class header. By convention, the class name begins with an upper-case letter.

Following the class header are the **instance variables**. A copy of the instance variables is made for each object that is created from the class. Thus, the instance variables have different values for each object of the class. The instance variables define the **state** of an object. The state of an object persists until the object is destroyed.

Instance variables are **global** variables that may be accessed by any method inside the object.

A **constructor** is used to create instances of the class. Typically, parameters are passed to the constructor and the constructor saves the values of the parameters in instance variables. If there is more than one constructor, each constructor must have a unique "signature" (the number, type, and order of parameters). As a general rule, constructors should be used to supply values for some or all of the instance variables. Performing complex processing in a constructor is not a good programming practice.

A new object is created (instantiated) using a `new` statement that calls the constructor:

```
Flight myFlight;
myFlight = new Flight(100, "Winnipeg", "Toronto");
```

This statement creates a new object which is an instance of the class Flight and assigns the values 100, “Winnipeg”, and “Toronto” to the instance variables.

Classes normally include **instance methods** that define the **behaviour** of the objects of that class. For example, the statement:

```
System.out.println(myFlight.toString());
```

causes the **message** “toString()” to be sent to the object myFlight; internally, this statement causes the **method** toString() to be executed within the object myFlight.

If an object is printed without including a specific method, Java assumes that the toString() method should be used to generate the necessary information. So the following statement has the same effect as the statement above.

```
System.out.println(myFlight);
```

A complete main class that uses the Flight class is shown below. (The main class is the class that contains the main method; the main method is the method that is executed when the program begins.)

```
public class TestFlights
{
     public static void main(String[] parms)
     {
        processFlights();
     }

     public static void processFlights()
     {
        Flight flight100;
        Flight flight200;
        Flight flight300;

        flight100 = new Flight(100, "Winnipeg", "Toronto");
        System.out.println(flight100);
        flight200 = new Flight(200, "Toronto", "Winnipeg");
        System.out.println(flight200);
        flight300 = new Flight(300, "Toronto", "Montreal");
        System.out.println(flight300);
     }
}
```

The output of the program is shown below.

```
100 Winnipeg Toronto
200 Toronto Winnipeg
300 Toronto Montreal
```

3.6 Visibility Modifiers

It is a good programming practice to define instance variables as being **private** to the class so that methods outside of the class can not examine or modify the variables. (`public` and `private` are referred to as **visibility modifiers**.)

```
private int flightNumber;
private String flightOrigin;
private String flightDestination;
```

3.7 Manipulating an Object

Since the instance variables are private, if we need to access or modify the value of an instance variable, an appropriate method must be defined. For example, a changeDestination() instance method could be used to modify the destination of myFlight from Toronto to Montreal.

```
myFlight.changeDestination("Montreal");
```

The instance method changeDestination would be defined as follows in the Flight class.

```
public void changeDestination(String newDestination)
{
     flightDestination = newDestination;
}
```

As a general rule, extracting the value of an instance variable using an instance method and then performing some processing of that value outside of the object is a poor programming practice. Instead, try to keep the processing inside the object.

For example, the following statements determine whether or not a flight travels between Winnipeg and Toronto. The processing is performed outside of the object. (The methods getOrigin and getDestination would have to be added to the class.)

```
public static void processFlights()
{
     Flight flight;

     flight = new Flight(. . .);

     if (flight.getOrigin.equals("Winnipeg")
             && flight.getDestination.equals("Toronto"))
     {
        System.out.println(flight);
     }
}
```

While the statements above are correct, they are not well designed. Processing such as this should be performed inside the object and the result returned to the method.

The statements below illustrate a much cleaner version of the processing. A method checkTrip is included in the Flight class and, when given an origin and a destination, this method returns true or false, indicating whether or not the flight travels between the two cities.

```
public static void processFlights()
{
     Flight flight;

     flight = new Flight(. . .);

     if (flight.checkTrip("Winnipeg", "Toronto"))
     {
        System.out.println(flight);
     }
}
```

The following checkTrip method would then be defined in the Flight class:

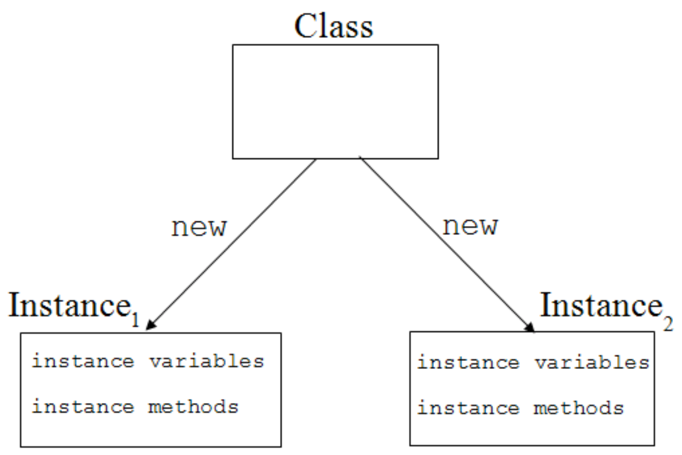
```
public boolean checkTrip(String origin, String destination)
{
     return (flightOrigin.equals(origin) && flightDestination.equals(destination));
}
```

As a result, the processing program is simpler and easier to write.

3.8 Classes and Instances

A class definition contains information about the structure of the class. An instance of a class is created using the class definition. An instance of a class is an object that contains its own copy of any instance variables and methods. There may be any number of instances of a class but there is only one class definition.

The class definition can be viewed as a blueprint for a specific type of object. For example, in construction, a blueprint may define a specific type of house but each instance of the blueprint is an actual house (constructed according to the corresponding blueprint). While many houses many be constructed from the same blueprint, each house can differ from other houses constructed from the same blueprint (carpeting, exterior finish, colour of the walls, etc.).



3.9 State Information

In addition to storing parameters that are passed to an object in instance variables, instance variables in objects can also save state information that is used during multiple requests to the object. In this section we examine how variables can retain their values between calls.

The following procedural code prints the contents of an array of Strings that contains one word in each array element. `printWords` is a simple method that performs all of its processing within the method. The variable MAX_CHARS_LINE specifies the maximum number of characters that are to be printed on each line. Words are separated by exactly one blank/space. (The method actually exceeds MAX_CHARS_LINE by one character if the last character of the last word on a line ends in column MAX_CHARS_LINE because a blank is added after each word – this could be fixed quite easily but isn't fixed in this example in order to avoid obscuring the processing.)

```
public static void printWords(String[] words)
{
      final int MAX_CHARS_LINE = 50;
      String word;
      int numChars;
      int wordCount;

      numChars = 0;
      for (wordCount=0; wordCount<words.length; wordCount++)
      {
          word = words[wordCount];
          if ((numChars+word.length()) > MAX_CHARS_LINE)
          {
              System.out.println();
              numChars = 0;
          }
          System.out.print(word +" ");
          numChars += word.length()+1;
      }
      if (numChars > 0)
      {
          System.out.println();
```

```
     }
}
```

The following processing splits the method `printWords` into two methods. (In this particular example, there is no good reason for splitting a simple method into two methods, but as methods become larger and more complex, splitting a method into multiple methods does become necessary to maintain the readability of each method.) Now, the subordinate method `printWord` needs to retain the information in `MAX_CHARS_LINE` and `numChars` between calls to the method. Saving this information in local variables within `printWord` does not work because variables local to a method are re-initialized each time that the method is called. Thus, it is necessary to save the information in the calling method (`printWords`) and then pass the information to `printWord` each time that it is called. (This is a limitation of Java but is not a limitation of all programming languages.)

```
public static void printWords(String[] words)
{
     final int MAX_CHARS_LINE = 50;
     String word;
     int numChars;
     int wordCount;

     numChars = 0;
     for (wordCount=0; wordCount<words.length; wordCount++)
     {
        word = words[wordCount];
        numChars = printWord(MAX_CHARS_LINE, numChars, word);
     }
     if (numChars > 0)
     {
        System.out.println();
     }
}

public static int printWord(int maxCharsLine, int numChars, String word)
{
     if ((numChars+word.length()) > maxCharsLine)
     {
        System.out.println();
        numChars = 0;
     }
     System.out.print(word +" ");
     numChars += word.length()+1;
     return numChars;
}
```

Also, notice that the calling method contains instructions that terminate the last line if it did not fill the line completely. This processing does not really belong in the calling method and should really be included in the `printWord` method but this would require some additional processing. (Can you figure out how to add this processing to `printWord`?)

The values of instance variables in an object do persist between calls to the object. And so the use of an object makes the problem of having to store information required by the called method in the calling method go away. The following code illustrates the use of an object to handle the printing of each word. Note that the calling method is now much cleaner – it does not have to store information that is not relevant to the method.

```
public static void printWords(String[] words)
{
     final int MAX_CHARS_LINE = 50;
     PrintWord printer;
     String word;
     int wordCount;

     printer = new PrintWord(MAX_CHARS_LINE);
     for (wordCount=0; wordCount<words.length; wordCount++)
     {
        word = words[wordCount];
        printer.addWord(word);
     }
     printer.checkLastLine();
}

public class PrintWord
{
     private int maxCharsLine; // instance variable retains its value between calls
     private int numChars;     // instance variable retains its value between calls

     public PrintWord(int maxCharsLine)
     {
        this. maxCharsLine = maxCharsLine;
        numChars = 0;
     }

     public void addWord(String word)
     {
        if ((numChars+word.length())>maxCharsLine)
        {
           System.out.println();
           numChars = 0;
        }
        System.out.print(word +" ");
        numChars += word.length() + 1;
     }

     public void checkLastLine()
     {
        if (numChars > 0)
        {
           System.out.println();
        }
     }
}
```

3.10 Arrays of Objects

The main class TestFlights defined earlier creates 3 objects and stores the objects in individual variables. However, if a large number of objects must be manipulated, having to reference the objects individually would be very cumbersome. Instead, we can store the objects in an array. Objects can be stored in an array in the same manner as primitive data types.

The program below illustrates the use of an array to maintain a collection of objects. The program hard codes the creation of the objects in the createFlights method. Hard-coding the creation of the objects is easier than obtaining the information from the user or reading the information from a file (which is covered in the next chapter.) Once the objects have been created, the collection of objects can be passed as a single array. This technique is significantly simpler than having to use parallel arrays when programming in a procedural language. The array of objects is first printed and is then searched for all direct flights that travel between a specific origin and destination (Winnipeg and Toronto in this example).

```
public class TestFlights
{
     public static void main(String[] parms)
     {
        Flight[] flights;

        flights = createFlights();
        printFlights(flights);
        searchFlights(flights, "Winnipeg", "Toronto");
     }

     public static Flight[] createFlights()
     {  // take the easy way out for now and "hard code" the flights
        Flight[] flights;
        flights = new Flight[6];

        flights[0] = new Flight(100, "Toronto", "Winnipeg");
        flights[1] = new Flight(200, "Winnipeg", "Toronto");
        flights[2] = new Flight(300, "Toronto", "Montreal");
        flights[3] = new Flight(400, "Ottawa", "Toronto");
        flights[4] = new Flight(500, "Ottawa", "Montreal");
        flights[5] = new Flight(600, "Winnipeg", "Toronto");
        return flights;
     }

     public static void searchFlights(Flight[] myFlights,
                                      String origin, String destination)
     {
        int count;
        Flight currentFlight;

        System.out.println("Flights between " +origin +" and " +destination +"\n");
        for (count=0; count<myFlights.length; count++)
        {
           currentFlight = myFlights[count];
           if (currentFlight.checkTrip(origin, destination))
           {
              System.out.println(currentFlight);
           }
        }
     }

     public static void printFlights(Flight[] myFlights)
     {
        int count;
        Flight currentFlight;

        System.out.println("\nListing of all flights:" +"\n");
        for (count=0; count<myFlights.length; count++)
        {
           currentFlight = myFlights[count];
           System.out.println(currentFlight);
        }
     }
}
```

```
public class Flight
{
     private int flightNumber;
     private String flightOrigin;
     private String flightDestination;

     public Flight(int flightNumber, String flightOrigin, String flightDestination)
     {
        this.flightNumber = flightNumber;
        this.flightOrigin = flightOrigin;
        this.flightDestination = flightDestination;
     }

     public boolean checkTrip(String origin, String destination)
     {
        return (flightOrigin.equals(origin)
               && flightDestination.equals(destination));
     }

     public String toString()
     {
        return "Flight " +flightNumber +" travels from "
                          + flightOrigin +" to " + flightDestination;
     }
}
```

If you look carefully at the Flight class, you will notice that an additional keyword has been used – “**this.**” The variable `flightOrigin` is used both as a parameter for the method and also as an instance variable for the class. When multiple variables with the same name exist, Java uses the most recently (or closely) declared variable. So `flightOrigin` refers to the method parameter. But we also need to be able to refer to the instance variable and Java allows us to do that if we prefix the instance variable with the keyword `this.` The keyword `this` refers to the **current object instance**, so `this.flightOrigin` refers to the variable `flightOrigin` in the current object instance. Now there is no ambiguity – `this.flightOrigin` (the instance variable) is assigned the value of `flightOrigin` (the method parameter). Of course we could have avoided this situation by giving the variable in the parameter list a different name but it is a common programming practice to use the same variable name in a method’s parameter list as the corresponding instance variable.

As we saw in Chapter 2 – Growing Algorithms, an array may be declared and initialized at the same time. The following statements illustrate array initialization with objects.

```
Flight[] flights1 = {new Flight(100), new Flight(200)};

Flight[] flights2 = new Flight[] {new Flight(200), new Flight(400)};

flights1 = new Flight[] {new Flight(500), new Flight(600)};
```

The statements above used a modified Flight class constructor to keep the examples as simple as possible.

```
public Flight(int number)
{
     flightNumber = number;
}
```

3.11 Class Variables and Class Methods

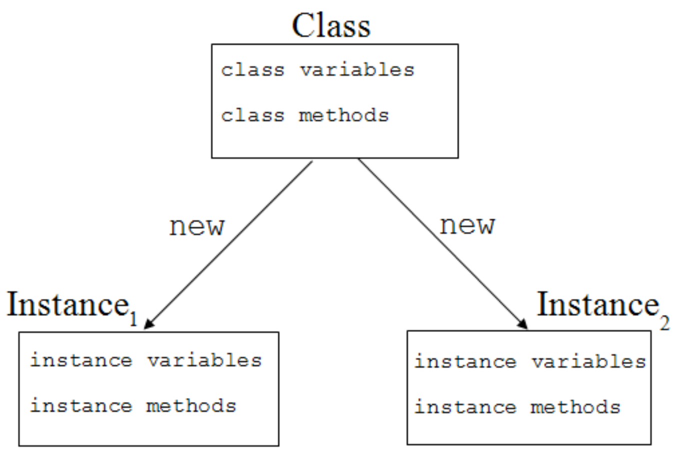
In addition to instance variables, a class may contain **class variables**. Class variables are created once for the entire class and the value of a class variable is the same for all objects of the class. A class variable definition includes the keyword **static**.

Class variables are variables that may be accessed by any method inside the class. If a class variable is also `public`, then it can be accessed by methods outside of the class.

You have probably already used several public class variables such as `Math.PI`.

Class methods are methods that include the keyword **static** in the definition of the method. Class methods are executed by prefixing the method name with the class name.

You have already encounted various class methods such as `Math.sqrt()`, `Math.pow()`, `Double.parseDouble()`, and `System.arraycopy()`.



In the example below, `totalFlights` is a class variable that is used to keep track of the number of flights that have been created. Unlike instance variables, there is only one totalFlights variable in the class.

```
public class Flight
{
     private int flightNumber;
     private String flightOrigin;
     private String flightDestination;

     private static int totalFlights = 0;  // Class variable

     public Flight(int number,String origin,String destination)
     {
        flightNumber = number;
        flightOrigin = origin;
        flightDestination = destination;
        totalFlights++;
     }

     public boolean checkTrip(String origin, String destination)
     {
        return (flightOrigin.equals(origin)
                                        && flightDestination.equals(destination));
     }

     public void changeDestination(String newDestination)
     {
        flightDestination = newDestination;
     }

     public static String howManyFlights()  // Class method
     {
        return "There are currently " +totalFlights +" flights.";
     }

     public String toString()
     {
        return flightNumber +" " +flightOrigin +" " +flightDestination;
     }
}
```

The statement

```
System.out.println(Flight.howManyFlights());
```

prints the number of flights that have been created.

The class variables are stored with the class definition (ensuring that there is only one copy of each class variable).

3.12 Wrapper Classes

The primitive data types (`int`, `char`, `boolean`, `double`, etc.) that were used in simple procedural programs in Chapter 1 are not objects. There are times when it would be convenient to be able to process a primitive value as an object. To permit this type of processing, Java provides a **wrapper** class for each of the primitive data types. For example, the Integer class can be used to create an object that contains an int.

```
Integer myInteger;
myInteger = new Integer(3);
```

The statements above create an object that contains an integer value. Until the arrival of Java 5, the value stored in a wrapper object could not be manipulated directly. If the value in an Integer object had to be processed, it first had to be extracted from the object using the object's accessor method (intValue() in the case of an Integer). Similarly, if two Integer objects are to be compared, they must be compared using the appropriate comparison **methods** for objects (`equals` or `compareTo`), not the comparison **operators** (such as ==) used to compare primitive data types. To compare two Integer objects for equality, either the following comparisons could be used:

```
Integer myInteger1;
Integer myInteger2;
int myInt;
myInteger1 = new Integer(3);
myInteger2 = new Integer(3);

if (myInteger1.intValue() == myInteger2.intValue()) // perform primitive comparison
if (myInteger1.equals(myInteger2))                  // perform object comparison
```

The Integer class also contains some useful (static) **class variables** and (static) **class methods** that can be used without having to create an instance of the Integer class. For example, to convert the contents of a String to an int, the `Integer.parseInt(string)` static class method is used. Similarly, to obtain the value of the largest possible integer on the current machine, the static class variable `Integer.MAX_VALUE` is used.

The other wrapper classes (Double, Character, etc.) perform processing similar to that performed in the Integer class and also contain similar methods (doubleValue, charValue, etc.)

Beginning with Java 5, Java automatically wraps (or boxes or autoboxes) and unwraps (or unboxes or autounboxes) the contents of wrapper objects. Boxing means that, based on the context, Java converts a primitive data value to the equivalent wrapper object. Unboxing means that Java converts the contents of a wrapper object to the equivalent primitive data value.

The program segment below illustrates automatic boxing of int values and unboxing of Integer values.

```
Integer myInteger1;
Integer myInteger2;
int myInt;

myInteger1 = 3;
myInteger2 = 10;
myInt = myInteger1 + myInteger2;
```

3.13 Immutable Objects

The objects created by the wrapper classes are **immutable**, meaning that the value of the object can not be changed once the object has been created. You will notice if you examine the class header of a wrapper class that each header includes the keyword **`final`**, indicating that once a value is assigned, it can not be modified.

```
public final class Integer { … }
```

Similarly, if you examine the methods provided with a wrapper class, you will find accessor methods but you will not find mutator methods. (An accessor method returns the value of an instance variable; a mutator method modifies the value of an instance variable.)

Therefore, if you need to modify the value of a variable, you must create a new instance of the object.

```
Integer myInteger;
myInteger = new Integer(3);
…
myInteger = new Integer(10);
```

However, if Java 5 is being used, you can "apparently" modify a wrapper object, as shown below. (We will see in Chapter 7 – Object Representation what is actually happening in the statements.)

```
Integer myInteger;

myInteger = 3;
myInteger += 1;
System.out.println(myInteger);

4
```

Although we will not examine Strings until Chapter 5 – Strings, Strings are objects that are immutable.

3.14 Comparing Objects

Objects should not be compared using the == comparison operator used to compare primitive data types. Instead, objects are compared using an `equals` method that is defined in the class. For example, two String objects are compared as shown below:

```
if (string1.equals(string2))
{
}
```

If an object may not yet have been instantiated, it can **not** be compared with another object using the equals method. Instead, it should be compared with `null` first and then compared with the other object.

```
result = false;
if (someString != null)
{
     result = someString.equals(someOtherString);
}
```

All system-defined classes contain an `equals` method. Programmer-defined classes do not automatically contain an `equals` method, the programmer must explicitly define the method.

For example, with the Flight class, if two flights are considered equal if they contain the same flight number, the following equals method would be added to the Flight class:

```
public boolean equals(Flight flight)
{
     return this.flightNumber == flight.flightNumber;
}
```

Note that it is valid to refer to the private instance variables of a Flight object within a different Flight object.

The meaning of "equals" depends on the specific class. In the example above, two flights are considered to be the same if they have the same flight number. However, two flights might be considered equal if they have the same origin and destination; in this case, the following definition of equals would be used for the Flight class:

```
public boolean equals(Flight flight)
{
     return (this.flightOrigin.equals(flight.flightOrigin))
        && (this.flightDestination.equals(flight.flightDestination));
}
```

If objects can be ordered according to one or more of the instance variables, the ordering is defined in a `compareTo` method.

System objects in Java that can be ordered already contain a `compareTo` method. This method returns an int value that is less that 0 if the first object (`object1` below) comes before the parameter (`object2` below), returns 0 if the two objects are equal, and returns a value that is greater than 0 if the first object comes after the parameter.

```
int result = object1.compareTo(object2)

result < 0  if object1 comes before object2
result = 0  if object1 is equal to object2
result > 0  if object1 comes after object2
```

The lexicographic ordering of two String objects is defined in the String method `compareTo` (see Chapter 5 – Strings for more information).

```
string1.compareTo(string2)
```

If two Flight objects are ordered according to their flight numbers, the corresponding `compareTo` method is:

```
public int compareTo(Flight flight)
{
     int result;

     if (this.flightNumber < flight.flightNumber)
     {
        result = -1;
     }
     else if (this.flightNumber == flight.flightNumber)
     {
        result = 0;
     }
     else
     {
        result = +1;
     }
     return result;
}
```

If two Flight objects are ordered by their origins and destinations combined, the corresponding `compareTo` method is:

```
public int compareTo(Flight flight)
{
     int result;

     result = this.flightOrigin.compareTo(flight.flightOrigin);
     if (result == 0)
     {
        result = this.flightDestination.compareTo(flight.flightDestination);
     }
     return result;
}
```

Note that when two (or more) instance variables participate in the ordering of objects, you **must not** just concatenate the values of the variables to determine the ordering of objects.

Similarly if objects of type `Person` contain the variables `firstName` and `lastName`, the corresponding `equals` and `compareTo` methods are:

```
public boolean equals(Person person)
{
     return this.lastName.equals(person.lastName) && this.firstName.equals(person.firstName);
}
```

```
public int compareTo(Person person)
{
     int result;

     result = this.lastName.compareTo(person.lastName);
     if (result == 0)
     {
        result = this.firstName.compareTo(person.firstName);
     }
     return result;
}
```

3.15 Encapsulation and Information Hiding

In this chapter we examined the fundamentals of object orientation. Object orientation helps support **encapsulation** and **information hiding**, two features that make programs easier to write and maintain, particularly as programs become larger. Encapsulation refers to the property that all data and behaviours for a particular type of object are collected in the associated class. For example, if we need to know something about a flight, that information will be found in the Flight class.

Information hiding refers to the property that the implementation details about an object are hidden within the object. For example, we don't need to know how two flight objects are compared, we just use the appropriate method that performs the comparison and this method is hidden within the Flight class. All variables in a class should be hidden (declared private).

3.16 Factory Method

When an object is instantiated using `new ClassName()`, it is not possible to return anything other than a valid object of the type defined by `ClassName`. In the following example, if the parameter is not a valid integer (even if the error is caught), there is nothing that can be done about the problem.

```
public class MyObject
{
     private int parm;
     public MyObject(String parm)
     {
        this.parm = Integer.parseInt(parm);
     }
}
```

A factory method is a method within a class that is used to create an object with the option of returning an alternative value (typically a `null` value) if there is a problem with the parameters.

```
public class MyObject
{
     private int parm;

     private MyObject(int parm)
     {
        this.parm = Integer.parseInt(parm);
     }

     public static MyObject create(String parm)
     {
        MyObject object;
        int parm;

        try
        {
           this.parm = Integer.parseInt(parm);
           object = new MyObject(this.parm);
        }
        catch (NumberFormatException ex)
        {  // catch conversion errors
           System.out.println("Invalid parameter value: " +parm);
           object = null;
        }
        return object;
     }
}
```

Since the `create` method must be static, the method is used in the calling method as follows:

```
MyObject.create(parm);
```

The calling method should check the value returned before attempting to process the object. A factory method may be used for other purposes as well but they are beyond the scope of these notes.

3.17 The Main Class

In Chapters 1 and 2, we developed procedural programs without understanding what the `class` statement meant or why the methods had to be declared `static`. By now, the use of these two keywords should be somewhat more obvious. Every Java program is defined in one or more classes. The method that receives control from Java when a program begins is the `main` method and this method must include an array of Strings that will contain the parameters (if there are any) passed to the method. The class that contains the `main` method is often referred to as the main class.

The main method is a static method and it may call any other static methods within the same class or within other classes. If a static method in another class is called, the method must be prefixed with the class name: *`ClassName.method()`*. If a program consists of only static methods, then the program is essentially a procedural program (ignoring the fact that some Java objects will probably be used in the program for printing, string manipulation, and reading/writing files).

Procedural programming does not take advantage of the power of object orientation and so most Java programs will also contain objects, each of which is defined a class. An object is created (instantiated) using the *`new ClassName(…)`* notation. Again, the class definition is a blueprint for creating objects. Each object has its own copy of the instance variables defined in the class and a copy of the instance methods in the class. Once an object has been created, its instance (non-static) methods may be called by prefixing the method name with an object reference: *`object.method()`* . When an instance method is called, other instance methods within the same class may be called without having to prefix the method call with an object reference. (The context in this case is the current object – the current object can be referred to using the keyword *`this`*.)

A class/static method may access any class/static variables in the current class and any public class/static variables in other classes. A class/static method may **not** access instance variables/methods in the same class or instance variables/methods in any other class; if such an attempt is made, a compile error similar to the one below is generated.

```
C:\Test.java:21: non-static method getGPA() cannot be referenced from a static context
         System.out.println(Student.getGPA());
                                          ^
```

3.18 Summary

A class defines a blueprint for objects. Each object is an instance of a particular class. Each object has its own instance variables and instance methods. Any class variables and class methods defined in the class are shared by all instances of that class. The values of instance variables persist between calls to the instance; local variables defined within an instance method do not persist between calls to the method – each time that the method is called, the method variables are initialized to their default values (or to the values specified in the declaration statement).

So, now we have the necessary tools for developing complex programs: data and algorithms that are wrapped inside objects. We have also begun to define a process for developing complex programs – that of growing a program rather than attempting to design the entire program in advance (the engineering approach) and then suffering the consequences when we put all of the pieces together and find out that some of the design decisions were wrong. By growing our data and algorithms, if we determine that a particular part of the program is not well designed, it is much easier to fix it since we have not yet built on top of that particular part of the program. By adding the process of "**refactoring**" (which is described in Chapter 18 – Growing and Refactoring), we will have a process that allows us to grow programs fairly quickly but still make changes to the organization of the program when we identify portions that could be improved.

While basic object orientation is a major improvement over procedural programming, there is still another aspect of object-oriented programming that makes classes even more sophisticated. "**Inheritance**" permits classes to be linked together to make programming even easier. We will examine inheritance in Chapter 10 – Object Hierarchies.

4 FILE INPUT AND OUTPUT

4.1 Introduction

Until now, input data values have been supplied by the user via JoptionPane or were hard coded into the program and output data values have been written to the system console using `System.out.println(…)`. In this chapter we examine how information can be read from and written to (disk) files. Although file processing in Java makes extensive use of Java's classes and objects, you can manipulate files with only a few simple statements that do not require an in-depth knowledge of Java's file-processing classes.

4.2 Scanner Class

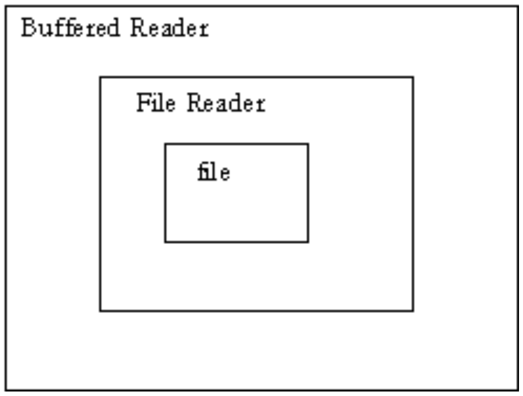
You may have encountered the Scanner class that is used to read from a file. This class is helpful when beginning programming in Java and it is necessary to read from a file.

```
Scanner fileIn;
String inputLine;

try
{
     fileIn = new Scanner(new File("in.txt"));

     while (fileIn.hasNextLine())
     {
         inputLine = fileIn.nextLine();
         System.out.println(inputLine);
     }
     fileIn.close();
}
catch (Exception e)
{
     System.out.println(e.getMessage());
     e.printStackTrace();
}
```

While the Scanner class is useful at times, it does not provide the flexibility that we require and so we will not use the Scanner class – instead, we will use another Java class that also supports file input with only a small additional amount of effort.

4.3 Buffered File Input

Input data can be read from a file using two Java wrapper classes. The `FileReader` class performs low-level processing (connecting to the file and reading the contents of the file one character at a time). The `BufferedReader` class collects the characters read by the `FileReader` class and returns the characters one line at a time. While these two classes perform a significant amount of complex processing, the programmer who uses the `BufferedReader` class does not have to be aware of this processing – the programmer only has to be aware of the public methods that are provided by the `BufferedReader` class and use these methods appropriately. These classes are a good example of object orientation – the programmer who uses the classes does not have to know **how** the processing is performed, only **what** the class does, and **what** the class does is defined by the class API.



```
FileReader fileReaderIn;
BufferedReader fileIn;
String inputLine;
try
{
     fileReaderIn = new FileReader("in.txt");
     fileIn = new BufferedReader(fileReaderIn);

     inputLine = fileIn.readLine();
     while (inputLine != null)
     {
        System.out.println(inputLine);
        inputLine = fileIn.readLine();
     }
     fileIn.close();
}
catch (IOException e)
{
     System.out.println(e.getMessage());
}
```

The program above is correct but the two new Java classes are not included in a program by default; instead, they must be specificially imported at the beginning of the program.

```
import java.io.*;
```

or

```
import java.io.FileReader;
import java.io.BufferedReader;
```

Note that the FileReader object identifies the name of the file to be processed (in this example, “in.txt”). Each line of input from the file is assigned to the String variable `inputLine`. There may be any number of characters on the line, including blanks and special characters (such as control characters). The Buffered Reader object removes the newline and carriage return characters from the end of each line. All files contain an end-of-file marker at the end of the file that is generated by the operating system. When this end-of-file marker is encounted while reading the file, the Buffered Reader returns a null String.

In the example above, the variable `fileReaderIn` is declared but is used only inside the creation of the `BufferedReader` object. Since `fileReaderIn` is not used anywhere else, the variable declaration may be removed and the instantiation may be moved inside the instantiation of the `BufferedReader`.

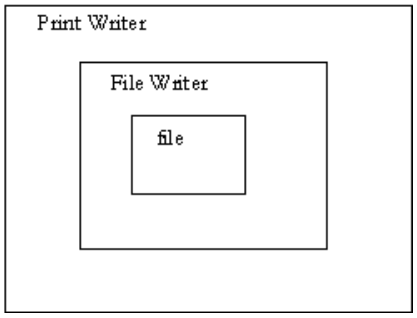
```
BufferedReader fileIn;
try
{
     fileIn = new BufferedReader(new FileReader("in.txt"));
     ...
}
```

This syntax makes the program slightly easier to read because the temporary objects no longer need to be declared and no longer clutter up the program.

4.4 File Output

Writing information to a file requires the use of wrapper classes that are very similar to those used when reading information from a file.

The following program segment writes the contents of an array of strings to a file named "out.txt". First, an object of type FileWriter is wrapped around the output file. Then, an object of type PrintWriter is wrapped around the FileWriter object. Now, the program can write information to the file using the same methods that are used with System.out.



```
PrintWriter fileOut;
String[] outputLines = {"First line.","Second line.","Third line.","Last line."};
int count;

try
{
     fileOut = new PrintWriter(new FileWriter("out.txt"));

     for (count=0; count<outputLines.length; count++)
     {
        fileOut.println(outputLines[count]);
     }
     fileOut.close();
}
catch (IOException ioe)
{
     ioe.printStackTrace();
}
```

As with the file input program shown earlier in this chapter, the two new Java classes are not included in a program by default; instead, they must be specificially imported at the beginning of the program.

```
import java.io.*;
```

or

```
import java.io.FileWriter;
import java.io.PrintWriter;
```

It is important to close each output file when the file is complete. If the file is not closed, information will not be written to the file correctly. When a file is closed, Java writes any information that has not yet been written to the file and then the operating system writes an end-of-file marker at the end of the file.

There is another wrapper class that can be used when writing to files – the `BufferedWriter`. The `BufferedWriter` is also wrapped around a `FileWriter` object. However, the `BufferedWriter` class is not quite as easy to use as the `PrintWriter` class so we will use only the `PrintWriter` class when writing to files.

The following program is a combination of the programs above. This program copies the information from one file to another file.

```
BufferedReader fileIn;
PrintWriter fileOut;
String inputLine = "";
try
{
     fileIn = new BufferedReader(new FileReader("in.txt"));

     fileOut = new PrintWriter(new FileWriter("out.txt"));

     inputLine = fileIn.readLine();
     while (inputLine != null)
     {
        fileOut.println(inputLine);
        inputLine = fileIn.readLine();
     }
     fileIn.close();
     fileOut.close();
}
catch (IOException ioe)
{
     ioe.printStackTrace();
}
```

One useful feature when opening an output file is the ability to **append** (add) to the end of an existing file instead of erasing the current contents of the file. Appending to an output file requires only one simple change when the FileWriter object is created:

```
new FileWriter("out.txt", true)
```

4.5 Recap

Although there is a variety of techniques that can be used when reading from a file and writing to a file, **we will use only `BufferedReader`'s when reading from a file and will use only `PrintWriter`'s when writing to a file**. System.out is already wrapped in a PrintStream object and the methods that are used when writing to System.out are the same as those that are used when writing to a file.

We have assumed that all data read and written is a collection of characters. Java also provides classes that can be used to read **byte** streams, that is, information that is already in its internal representation. For example, an integer byte stream would consist of the internal value of the integer, not a collection of characters. Reading and writing byte streams is more efficient since conversion from one type to another is not necessary. However, we will not examine the processing of byte streams.

One thing to be particularly careful of when processing files is the presence of blank lines in the file, particularly at the end of the file. The last line of a file is usually terminated by a carriage return but if there is an additional line that consists only of a carriage return, this line is returned to the program for processing and will likely cause an error.

4.6 Merging Files

The basic merge algorithm was examined in Chapter 2 – Growing Algorithms. In this section, we examine how the merge algorithm can be applied to merging the contents of two input files together and writing the output to a file.

In this example, instead of merging integer values, each file will contain one string per line and the strings will already have been sorted into ascending order. (See Chapter 5 – Strings for more information about how strings are ordered.) When two Strings are compared, the comparison must be performed using the `compareTo()` method and not the < operator used in Chapter 1 with the integer values.

We begin with the algorithm from Chapter 2 – Growing Algorithms that contains 3 loops: the primary loop merges the contents of the two files as long as there is at least one string remaining in each file. Note that since we do not know in advance how many lines there are in each file, we continue processing until the end-of-file marker (a null string) is encounted.

To make the development of the program a bit easier, we begin by writing the output to System.out instead of to a file. This is a common technique when processing files since it avoids having to open the output file each time that the program is executed to examine the output.

```
public static void mergeFiles()
{
     BufferedReader file1;
     BufferedReader file2;

     String inputLine1;
     String inputLine2;

     try
     {
        file1 = new BufferedReader(new FileReader("file1.txt"));
        file2 = new BufferedReader(new FileReader("file2.txt"));

        inputLine1 = file1.readLine();
        inputLine2 = file2.readLine();

        while ((inputLine1 != null) && (inputLine2 != null))
        {
           if (inputLine1.compareTo(inputLine2) < 0)
           {
              System.out.println(inputLine1);
              inputLine1 = file1.readLine();
           }
           else
           {
              System.out.println(inputLine2);
              inputLine2 = file2.readLine();
           }
        }

        while (inputLine1 != null)
        {
           System.out.println(inputLine1);
           inputLine1 = file1.readLine();
        }

        while (inputLine2 != null)
        {
           System.out.println(inputLine2);
           inputLine2 = file2.readLine();
        }

        file1.close();
        file2.close();
     }
     catch (IOException ioe)
     {
        ioe.printStackTrace();
     }
}
```

Now, the algorithm executes correctly and the only piece that remains is to write the output to a file instead of to System.out. This involves creating the file and then changing references from System.out to the name of the file.

```
public static void mergeFiles()
{
     BufferedReader file1;
     BufferedReader file2;
     PrintWriter file3;
```

```
      String inputLine1;
      String inputLine2;

      try
      {
         file1 = new BufferedReader(new FileReader("file1.txt"));
         file2 = new BufferedReader(new FileReader("file2.txt"));
         file3 = new PrintWriter(new FileWriter("file3.txt"));

         inputLine1 = file1.readLine();
         inputLine2 = file2.readLine();

         while ((inputLine1 != null) && (inputLine2 != null))
         {
            if (inputLine1.compareTo(inputLine2) < 0)
            {
               file3.println(inputLine1);
               inputLine1 = file1.readLine();
            }
            else
            {
               file3.println(inputLine2);
               inputLine2 = file2.readLine();
            }
         }

         while (inputLine1 != null)
         {
            file3.println(inputLine1);
            inputLine1 = file1.readLine();
         }

         while (inputLine2 != null)
         {
            file3.println(inputLine2);
            inputLine2 = file2.readLine();
         }

         file1.close();
         file2.close();
         file3.close();
      }
      catch (IOException ioe)
      {
         ioe.printStackTrace();
      }
}
```

The algorithm is now complete. We saw in Chapter 2 – Growing Algorithms how the program could be made slightly shorter by moving the two trailing loops into the primary loop. The following method illustrates this version of the file-merge algorithm.

```
public static void mergeFiles()
{
     BufferedReader file1;
     BufferedReader file2;
     PrintWriter file3;

     String inputLine1;
     String inputLine2;

     try
     {
        file1 = new BufferedReader(new FileReader("file1.txt"));
        file2 = new BufferedReader(new FileReader("file2.txt"));
        file3 = new PrintWriter(new FileWriter("file3.txt"));

        inputLine1 = file1.readLine();
        inputLine2 = file2.readLine();

        while ((inputLine1 != null) || (inputLine2 != null))
        {
           if ((inputLine1 != null) &&
                   ((inputLine2 == null) || (inputLine1.compareTo(inputLine2) < 0)))
           {
              file3.println(inputLine1);
              inputLine1 = file1.readLine();
           }
           else
           {
              file3.println(inputLine2);
              inputLine2 = file2.readLine();
           }
        }

        file1.close();
        file2.close();
        file3.close();
     }
     catch (IOException ioe)
     {
        ioe.printStackTrace();
     }
}
```

While this refactoring does not make the 2-file merge significantly easier, it makes merging 3 files much easier.

The following method merges the contents of 3 files and writes the result to an output file. Again, each line of each file contains one String and the Strings in each file have already been sorted into ascending order.

```
public static void mergeFiles()
{
     BufferedReader file1;
     BufferedReader file2;
     BufferedReader file3;
     PrintWriter file4;

     String inputLine1;
     String inputLine2;
     String inputLine3;

     try
     {
        file1 = new BufferedReader(new FileReader("file1.txt"));
        file2 = new BufferedReader(new FileReader("file2.txt"));
        file3 = new BufferedReader(new FileReader("file3.txt"));
        file4 = new PrintWriter(new FileWriter("file4.txt"));

        inputLine1 = file1.readLine();
        inputLine2 = file2.readLine();
        inputLine3 = file3.readLine();

        while ((inputLine1!=null) || (inputLine2!=null) || (inputLine3!=null))
        {
           if ((inputLine1!=null) &&
              ((inputLine2==null)||(inputLine1.compareTo(inputLine2)<0)) &&
              ((inputLine3==null)||(inputLine1.compareTo(inputLine3)<0)))
           {
              file4.println(inputLine1);
              inputLine1 = file1.readLine();
           }
           else if ((inputLine2!=null) &&
                  ((inputLine3==null)||(inputLine2.compareTo(inputLine3)<0)))
           {
              file4.println(inputLine2);
              inputLine2 = file2.readLine();
           }
           else
           {
              file4.println(inputLine3);
              inputLine3 = file3.readLine();
           }
        }

        file1.close();
        file2.close();
        file3.close();
        file4.close();
     }
     catch (IOException ioe)
     {
        ioe.printStackTrace();
     }
}
```

Although this program is no longer trivial, it is much simpler than trying to make the program with separate loops (after the first loop) work correctly when 3 files must be merged. If you examine the program carefully, you should notice that there is a pattern to the `if` statements used to determine which String is the smallest. Understanding the pattern makes it possible to merge any number of files at a time. (An alternative to merging many files at once is to merge several files into a temporary file and then merge the remaining files and the temporary file to create the final file.)

4.7 Two Useful TextPad Features

When growing a program, adding a control structure such as an “if” or a “loop” around some existing statements is quite common. In order to maintain proper indentation, TextPad provides two statements that assist in changing the indentation. For example, if we begin with the statements:

```
fileOut.println(inputLine);
inputLine = fileIn.readLine();
```

but we later add a loop to repeat the statements, the indentation will not be correct.

```
while (inputLine != null)
{
fileOut.println(inputLine);
inputLine = fileIn.readLine();
}
```

With TextPad, simply select the statements for which the indentation is to be modified and type **ctrl-i to increase the indentation** or **ctrl-shift-i to decrease the indentation**.

```
while (inputLine != null)
{
     fileOut.println(inputLine);
     inputLine = fileIn.readLine();
}
```

Being able to change the indentation is a very useful TextPad feature as programs are grown. Another useful TextPad feature is the ability to match brackets (either “(“ and “)” or “{“ and “}”). Place the cursor beside a bracket and then type **ctrl-M; this causes TextPad to move the cursor to the matching bracket.** Typing ctrl-M again will cause TextPad to return to the original matching bracket. If you type ctrl-M in the middle of some code, TextPad will search in the forward direction for the nearest bracket.

4.8 Exceptions

Try-catch blocks are required around all input and output statements with the exception of writing to System.out. The reason for this requirement is that with any I/O statement, there is always the possibility that something will go wrong that is beyond the programmer’s control (for example, a file named in the program does not exist or the disk on which the file resides may not be available). Therefore, the Java language specification insists that we recognize the potential for an error and provide a mechanism for handling the error. So statements that might generate an error that the programmer can not check for in advance must be enclosed in a try block and the try block must be immediately followed by a catch block that defines the action that is to take place if an error occurs.

An example of a try-catch block is:

```
try
{
     //statements
}
catch (IOException ioe)
{
     ioe.printStackTrace();
}
```

The statements that may generate an error are placed inside the try block. Any additional statements may also be included in the try block. The statements that perform the processing in the event of an error are defined in the catch block. The parameter of the catch block is the type of error that is to be processed by this catch block. The object passed to the catch block (ioe in this example) can provide information about an error via the methods getMessage() and printStackTrace(). If the try block executes without an error, the catch block is ignored. However, if the catch block is executed, processing will continue with the next statement after the catch block.

While I/O statements must be enclosed in try/catch blocks, many other routine Java statements may also generate an error but do not have to be enclosed in try/catch blocks. For example, converting a string value to an integer may generate an error that causes the Java system to stop execution with a NumberFormatException. To prevent a program from stopping in such a situation, you could include the conversion statement in a try/catch block that handles the error.

```
int value;
String inputLine;

try
{
     value = Integer.parseInt(inputLine);
}
catch (NumberFormatException ex)
{
     System.out.println("Invalid value: " +inputLine);
     // do something to fix the error
}
```

This processing is often included when a user enters an input value and it is important to notify the user of the input error and provide an opportunity to correct the error. This process is referred to as **data validation**.

Multiple catch blocks, each catching a different type of error may be placed one after the other. In this situation, Java matches the exception that occurred to the exceptions defined in the catch blocks, beginning with the first catch block and continuing sequentially until it finds a match. For example, in the following statements, if an error is generated, the type of error is first compared with the type specified in the first catch block (NumberFormatException). If they match, then the first catch block is executed and the remaining catch blocks are ignored. If the error is not a NumberFormatException, then the error is compared with the next catch block. This process is repeated until either there is a match or until all of the catch blocks have failed to provide a match.

```
try
{
     //statements
}
catch (NumberFormatException ex1)
{
     System.out.println(ex1.getMessage());
}
catch (ArithmeticException ex2)
{
     System.out.println(ex2.getMessage());
}
catch (IOException ex3)
{
     System.out.println(ex3.getMessage());
}
```

Exceptions form a hierarchy of objects, with Exception being at the top of the hierarchy. (This will be examined in more detail in Chapter 10 – Object Hierarchies.) The exception type “Exception” matches all possible exceptions and could be used at the end of a list of catch blocks or as the only catch block.

```
try
{
     //statements
}
catch (NumberFormatException ex1)
{
     System.out.println(ex1.getMessage());
}
catch (IOException ex2)
{
     System.out.println(ex2.getMessage());
}
catch (Exception ex3)
{
     System.out.println(ex3.getMessage());
}
```

4.8.1 Thowing Exceptions

At times it is inconvenient to process an exception in the method in which the exception occurred and it would be more convenient to process the exception in the calling method.

Java permits the programmer to "throw" an exception back to the calling method which catches the exception in a try/catch block.

In the following program segment, the calling method (main in this example) calls the fileCopy method. Instead of including a try/catch block around the I/O statements in the fileCopy method, the method simply throws any IOExceptions back to the calling method (main). The main method then catches any IOExceptions and performs the necessary processing.

```
public static void main(String[] parms)
{
     try
     {
        fileCopy();
     }
     catch (IOException ioe)
     {
        System.out.println("Something went wrong in fileCopy.");
     }
}

public static void fileCopy() throws IOException
{
     BufferedReader fileIn;
     PrintWriter fileOut;
     String inputLine = "";
     fileIn = new BufferedReader(new FileReader("missingFile.txt"));
     fileOut = new PrintWriter(new FileWriter("out.txt"));

     inputLine = fileIn.readLine();
     while (inputLine != null)
     {
        fileOut.println(inputLine);
        inputLine = fileIn.readLine();
     }
     fileIn.close();
     fileOut.close();
}
```

If desired, the statement used earlier to identify the methods that were active at the time of the exception may be included in the catch block in the calling method.

```
ioe.printStackTrace();
```

4.8.2 Programmer-Defined Exceptions

When an error occurs, we have been printing a message on System.out (assuming that we did anything at all). The following code segment illustrates a typical error message.

```
int value;
String inputLine;
try
{
     value = Integer.parseInt(inputLine);
}
catch (NumberFormatException ex)
{
     System.out.println("Invalid value : " +inputLine);
}
```

The problem with the approach used above is that the rest of the program is not made aware of the error and the program may continue performing its processing when it really should be terminated. Instead of printing an error message, a catch block may throw its own exception. The following methods illustrate this process.

```
public static void main(String[] parms)
{
     try
     {
        throwTest("abc");
     }
     catch (Exception e)
     {
        System.out.println(e.getMessage());
     }

     System.out.println("\nProgram completed normally.");
}

public static void throwTest(String inputLine) throws Exception
{
     int value;

     try
     {
        value = Integer.parseInt(inputLine);
     }
     catch (NumberFormatException ex)
     {
        throw new Exception("Number format exception in throwTest: " +inputLine);
     }
}
```

These methods are performing data validation and generate the following message if the data are invalid.

```
Number format exception in throwTest: abc
```

4.9 An Input/Output Processing Class

Once you become used to file processing, it is somewhat tedious to have to type the same file manipulation statements over and over again. It would be more convenient if these statements could be defined once and then used in any program. That is exactly the functionality that objects provide. The following class IOProcess illustrates how much of the low-level processing required to read from a file can be moved into an object. Then, simple methods are used to perform the file manipulations. Note that this example uses instance variables to store state information as was demonstrated in the previous chapter.

```
import java.io.*;

public class IOProcess
{
     private BufferedReader fileIn;
     private PrintWriter fileOut;
     private boolean output;

     public IOProcess(String fileName)
     {
        output = false;
        openFile(fileName);
     }

     public IOProcess(String fileName, boolean output)
     {
        this.output = output;
        openFile(fileName);
     }

     private void openFile(String filename)
     {
        if (output)
        {
           openOutputFile(fileName);
        }
        else
        {
           openInputFile(fileName);
        }
     }

     private void openInputFile(String fileName)
     {
        try
        {
           fileIn = new BufferedReader(new FileReader(fileName));
        }
        catch (IOException ioe)
        {
           ioe.printStackTrace();
        }
     }

     private void openOutputFile(String fileName)
     {
        try
        {
           fileOut = new PrintWriter(new FileWriter(fileName));
        }
        catch (IOException ioe)
        {
           ioe.printStackTrace();
        }
     }

     public String readLine()
     {
        String inputLine = null;
        try
        {
           inputLine = fileIn.readLine();
        }
        catch (IOException ioe)
        {
           ioe.printStackTrace();
        }
        return inputLine;
     }

     public void println(String outputLine)
     {
        fileOut.println(outputLine);
     }

     public void close()
     {
        try
        {
           if (output)
           {
              fileOut.close();
           }
           else
           {
              fileIn.close();
           }
        }
        catch (IOException ioe)
        {
           ioe.printStackTrace();
        }
     }
}
```

The following method illustrates the use of the IOProcess class.

```
public static void testIOProcess(String inFileName, String outFileName)
{
     IOProcess inputFile;
     IOProcess outputFile;
     String inputLine;

     inputFile = new IOProcess(inFileName);
     outputFile = new IOProcess(outFileName, true);
     inputLine = inputFile.readFile();
     while (inputLine != null)
     {
        outputFile.println(inputLine);
        inputLine = inputFile.readLine();
     }
     inputFile.close();
     outputFile.close();
}
```

As can be seen in this method, much of the “clutter” that is required to read from a file and/or write to a file has been moved into the IOProcess class. Since the IOProcess class is written only once but may be used by any number of programs, this class is a good example of object orientation.

By creating a new IOProcess object for each file, any number of files may be read at the same time.

5 STRINGS

5.1 Introduction

Most programs process character strings in one way or another. In this chapter we will first examine low-level character manipulation using the `char` primitive data type and then we will examine more sophisticated character manipulation using Java's `String` class.

Strings are good examples of objects. A String object stores a character string and the character string can be manipulated in various ways by sending messages to the object. The programmer can use a String object without being aware of the internal details of how the character string is represented and manipulated.

5.2 Character Manipulation

A `char` is a primitive data type that can store one character. A collection of characters can be stored in a `char` array.

```
public static void main(String[] parms)
{
     char[] chars1;
     char[] chars2;
     char[] chars3;
     char[] chars4;

     chars1 = getChars1();
     printChars(chars1);
     chars2 = getChars2();
     printChars(chars2);
     chars3 = concat(chars1, chars2);
     printChars(chars3);
     chars4 = substring(chars3,3,6);
     printChars(chars4);
     System.out.println("\n" +equals(chars1,chars2));
     System.out.println("\n" +equals(chars4, new char[] {'d','e','H'}));

     System.out.println("\nProgram completed normally.");
}

public static char[] getChars1()
{
     char[] chars;
     int count;

     chars = new char[5];
     for (count=0; count<chars.length; count++)
     {
        chars[count] = (char) (count+97);
     }
     return chars;
}

public static char[] getChars2()
{
     char[] chars = {'H','e','l','l','o'};

     return chars;
}

public static char[] concat(char[] input1, char[] input2)
{
     char[] result;
     int count;
     int length;

     length = input1.length;
     result = new char[length + input2.length];

     for (count=0; count<length; count++)
     {
        result[count] = input1[count];
     }

     for (count=0; count<input2.length; count++)
     {
        result[length+count] = input2[count];
     }
     return result;
}

public static char[] substring(char[] input, int fromPosition, int toPosition)
{
     char[] result;
     int count;

     result = new char[toPosition-fromPosition];
     for (count=fromPosition; count<toPosition; count++)
     {
        result[count-fromPosition] = input[count];
     }
     return result;
}

public static boolean equals(char[] chars1, char[] chars2)
{
     int count;
     boolean result;

     result = false;
     if (chars1.length == chars2.length)
     {
        result = true;
        for (count=0; count<chars1.length&&result; count++)
        {
           if (chars1[count] != chars2[count])
           {
              result = false;
           }
        }
     }
     return result;
}

public static void printChars(char[] chars)
{
     int count;
     System.out.println();
     for (count=0; count<chars.length; count++)
     {
        System.out.print(chars[count]);
     }
     System.out.println();
}
```

The following method performs slightly more complicated processing on an array of `char`'s, it removes all **white space** from a line and separates words by exactly one blank. (White space is one or more characters that separate the words. Normally, blanks, tabs, new-line characters, etc. are all treated as white space.) The method uses the Character class static method `isWhitespace` to determine whether or not a character is a white space character.

```
public static char[] removeWhiteSpace(char[] inputLine)
{
     char[] newLine;
     char[] result;
     int lineLength;
     int count1;
     int count2;
     int currentWord;
     int length;

     length = inputLine.length;
     newLine = new char[length+1]; // +1 because a blank is added after each word
     count1 = 0;
     count2 = 0;
     while (count1<length)
     {
        while ((count1<length)&&(Character.isWhitespace(inputLine[count1])))
        {  // skip over white space
           count1++;
        }

        while ((count1<length)&&(!Character.isWhitespace(inputLine[count1])))
        {  // copy non-white-space characters
           newLine[count2++] = inputLine[count1++];
        }
        newLine[count2++] = ' '; // add a blank after each word
     }
     result = new char[count2];
     System.arraycopy(newLine,0,result,0,count2);
     return result;
}

Original:     To   think     or    not  to    think,     that     is the     question!
Result:   To think or not to think, that is the question!
```

5.3 Character Wrapper Class

A `char` is a primitive data type, not an object. However, Java provides a Character wrapper class that can be wrapped around a single `char` to create an object that contains the character. For example, the following statement creates an object that contains the `char` '`a`'.

```
Character character;
character = new Character('a');
```

The character stored inside the object can be extracted using the statement:

```
char char1 = character.charValue();
```

As we noted earlier, Java 5 now performs automatic boxing of primitive data values and automatic unboxing of wrapper values. So the statement above could be written without the need to extract the value from the wrapper object.

```
char char1 = character;
```

5.4 Strings

As can be seen from the examples in the previous sections, processing individual characters is laborious and quite error-prone. Programmers who have to process characters in this manner often build their own libraries of useful utilities, such as the substring and concatenation methods shown above. However, Java is an object-oriented language and therefore, it is not necessary for programmers to build their own libraries of frequently used methods for character manipulation. Instead, Java provides a class that simplifies character manipulation by treating a collection of characters as a unit instead of as individual characters – the `String` class.

The declaration of a String is the same as the declaration of any other object:

```
String myString;
```

A String is instantiated using a `new` statement. For example,

```
myString = new String("Hello World!");
```

To make life a little easier for the programmer, Java also permits the "`new String`" portion of the command to be omitted, so the following is equivalent to the previous command:

```
myString = "Hello World!";
```

Java can also convert an array of `char`’s to a String and can convert a String to an array of `char`’s.

```
char[] chars1 = {'H', 'e', 'l', 'l', 'o'};
char[] chars2;
String myString;

myString = new String(chars1);
chars2 = myString.toCharArray();
```

The following method illustrates how much easier it is to read each line in a file and store the characters in a String instead of in an array of `char`'s. Remember that a `null` String is used to mark the end of the file when a BufferedReader is used to handle file input.

```
public static void fileDisplay()
{
     BufferedReader fileIn;
     String inputLine;
     try
     {
        fileIn = new BufferedReader(new FileReader("in.txt"));

        inputLine = fileIn.readLine();
        while (inputLine != null)
        {
           System.out.println(inputLine);
           inputLine = fileIn.readLine();
        }
        fileIn.close();
     }
     catch (IOException ioe)
     {
        ioe.printStackTrace();
     }
}
```

The contents of a String can be examined and processed in the same manner as the individual elements of an array of characters can be examined. The following method counts the number of blank characters in a String.

```
public static int countBlanks(String string)
{
     int count;
     int numBlanks;
     numBlanks = 0;
     for (count=0; count<string.length(); count++)
     {
        if (string.charAt(count) == ' ')
        {
           numBlanks++;
        }
     }
     return numBlanks;
}
```

The `length()` **method** is used to determine the length of a String and the `charAt(int)` method is used to extract one `char` from a String. It is important to note that charAt returns a `char` and therefore must be treated as a `char`, not as a String, so the comparison below is **not** valid because the char is being compared with a String (double-quote characters) instead of a char (single-quote characters).

```
     if (string.charAt(count) == " ")
```

The following method accepts a String and returns a String that contains the contents of the original String in reverse order.

```
public static String reverseString(String inputString)
{
      String outputString;
      int count;
      int length;
      length = inputString.length();
      outputString = "";
      for (count=length-1; count>=0; count--)
      {
          outputString += inputString.charAt(count);
      }
      return outputString;
}
```

Note that the statement:

```
            outputString += inputString.charAt(count);
```

is simply an abbreviation for the String concatentation statement:

```
            outputString = outputString + inputString.charAt(count);
```

The following method accepts a String and determines whether or not the contents of the String are a palindrome. (A palindrome is a sequence of characters that read the same both forwards and backwards. A simple palindrome is `"abba"`.) This method is case sensitive so even if the same character appears in two corresponding locations in the String, if the case of the characters is not identical, the method will indicate that the String is not a palindrome.

```
public static boolean isPalindrome(String string)
{
      int count;
      int length;
      boolean result;
      length = string.length();
      result = true;
      for (count=0; count<length/2 && result; count++)
      {
          if (string.charAt(count) != string.charAt(length-count-1))
          {
              result = false;
          }
      }
      return result;
}
```

5.5 String Methods

The following is a list of the more commonly-used String methods. There are additional methods that are described in the Java documentation for the String class.

`length()`	returns the length of the String; will be zero if the String is empty
`substring(start)`	returns characters from the String, beginning at position start and

	continuing until the end of the string
`substring(start,end)`	returns characters from the String; beginning at position start and continuing up to but not including the position indicated by end
`charAt(position)`	returns the `char` at the specified position
`equals(string)`	compares the contents of two strings; returns true or false
`equalsIgnoreCase(string)`	compares the contents of two strings but ignores the case of the strings; returns true or false
`compareTo(string)`	compares the contents of two strings; returns a negative value if the first string comes before the second string; returns 0 if the two strings are equal; returns a positive value if the first string comes after the second string
`compareToIgnoreCase(string)`	compares the contents of two strings but ignores the case of the strings; returns a negative value if the first string comes before the second string; returns 0 if the two strings are equal; returns a positive value if the first string comes after the second string
`concat(string)`	returns the concatenation of the string with the string parameter
`contains(string)`	returns true if the string contains the string parameter
`trim()`	removes white space from the beginning and the end of the string
`toLowerCase()`	converts all characters in the string to lower case
`toUpperCase()`	converts all characters in the string to upper case
`indexOf(string)`	returns the position at which the string parameter occurs in the string; returns -1 if the parameter does not occur in the string; this method is case sensitive
`indexOf(string,start)`	starting at position start in the string, returns the position at which the string parameter occurs in the string; returns -1 if the parameter does not occur in the string; this method is case sensitive
`lastIndexOf(string)`	same as indexOf but starts at the end of the string and works backwards
`lastIndexOf(string,start)`	same as lastIndexOf but starts at position start and works backwards
`startsWith(string)`	returns true if the string begins with the parameter string; returns false otherwise
`startsWith(string,start)`	returns true if the string, starting at position start, begins with the parameter string; returns false otherwise
`replaceFirst(from,to)`	replace the first occurrence of the from-string with the to-string
`replace(from,to)`	replace all occurrences of the from-string with the to-string
`replaceAll(from,to)`	replace all occurrences of the from-string with the to-string
`toCharArray()`	convert a String to the equivalent array of char's
`String.valueOf(value)`	static method used to convert a value to the equivalent String (see below).

It is important to remember that String variables are not updated in place; if the value of a String variable is to be modified, the variable must be assigned a new String value. For example, the statement below extracts the first 5 characters from the String myString and then assigns the result back to myString.

```
myString = myString.substring(0,5);
```

The String class contains a static method, `String.valueOf(value)`, that is used to convert other data types to the equivalent String representation. For example, the statement

```
int int1 = 23;
String string = String.valueOf(int1);
```

converts the integer value 23 to the equivalent string "`23`". The same effect can be obtained by the following statement:

```
int int1 = 23;
String string = "" +int1;
```

The `String.valueOf()` method can process each of the primitive data types (int, char, boolean, double, etc.). This method also can be used with an object: if the object is null, the string "`null`" is returned; otherwise, the value of object.toString() is returned.

A method that can process different types of parameters is said to be **overloaded** – there is a different version of the method for each type of parameter.

5.6 String Comparison

When two integers are compared, the result is obvious. For example, each of the following comparisons produces a result of TRUE.

```
5 < 10
-1 < 1
5 == 5
```

When two Strings are compared, the result is not necessarily obvious. For example, which of the following comparisons is/are true (ignore the fact that Strings are being compared using the standard comparison operators)?

```
"1" < "2"         true?
"abc" == "ABC"    true?
```

In Java, each character in a String is represented by a series of numbers in the Unicode character set. For example, the character `'a'` is represented by the number 0097. (We will ignore the two leading zeroes in the remainder of this section.) The order of the numbers that represent the individual characters is referred to as the “collating sequence” of the character set. The collating sequence of the Unicode character set can be summarized as follows (the Unicode value is shown below each character):

```
' ' < '0' < … < '9' < 'A' < … <  'Z' < 'a' < … < 'z'

32     48         57     65         90     97        122
```

When comparing the contents of two Strings, Java begins comparing the Strings with the first character in each String. If the two characters are equal, comparison proceeds to the next character. When two characters are not equal, the String that contains the character with the smaller Unicode value is considered to be "less than" the other string. If two Strings contain the same number of characters and all characters are identical, the two Strings are identical. If one String is shorter than the other but both Strings contain the same characters (up to the length of the shorter String), the shorter String is less than the longer string

```
"1" < "2"          True
" 1" < "2"         False (note that the first String contains a leading space/blank)
"abc" == "ABC"     False
"ABC" < "ABCD"     True
"abc" < "ABC"      False
```

Although we used the basic comparison operators to illustrate the comparison of Strings in the examples in this section, **Strings should only be compared using the String comparison methods,** `equals` **and** `compareTo` (or the related methods `equalsIgnoreCase` and `compareToIgnoreCase`).

The `equals` method compares two Strings for equality, as defined above. The `compareTo` method compares the collating sequence of two Strings. If the first String is less than the String parameter, `compareTo` returns a negative value. The negative value may be any negative value, so be careful when processing the result. If the two Strings are equal, `compareTo` returns zero. If the first String is greater than the String parameter, `compareTo` returns a positive value. Again, the value returned may be any positive value. The `equalsIgnoreCase` and `compareToIgnoreCase` methods perform exactly the same processing except that the case of the characters in the two Strings is ignored.

5.7 Strings are Objects

Strings are objects but you must be careful with Strings because there are several subtle points with Strings that can be confusing.

When comparing string objects, it is important to remember to use the `equals()` String method to perform the comparison.

```
String string1;
String string2;

if (string1.equals(string2))
{
     ...
}
```

Using `equals()` ensures that the **values** of the two strings are compared. (This topic is discussed in more detail in Chapter 7 – Object Representation.)

As with other objects, Java assigns the value `null` to a String variable to indicate that the String has not yet been instantiated. It is very important to note that `null` is not the same as an empty String. An empty String is a String object that contains no characters but is still a valid object. For example, in the following code, both statements are valid, but only string1 is a valid String object.

```
string1 = "";
string2 = null;
```

The following statements execute correctly because string1 is a valid object.

```
string1 = "";
if (string1.equals(""))
{
     System.out.println("yes");
}

yes
```

If the following statements are executed,

```
string2 = null;
if (string2.equals(""))
{
     System.out.println("yes");
}
```

attempting to use string2 as an object generates the following exception:

```
Exception in thread "main" java.lang.NullPointerException
```

If it is necessary to check for a null value, the comparison operator == must be used:

```
string2 = null;
if (string2 == null)
{
     System.out.println("yes");
}
```

Java permits concatenation to a null String object, as shown by the following code (although the result in string2 is probably not what you want):

```
String string1 = "";
String string2 = null;
string1 = string1 + "abc";
string2 = string2 + "abc";
System.out.println(string1);
System.out.println(string2);
```

```
abc
nullabc
```

5.8 Strings are Immutable

Java's String objects are **immutable**. This means that once a String object has been created, the object can not be modified. However, a different String object can be assigned to the same variable. For example, the following statements are perfectly valid.

```
string1 = "abc";
string1 = string1 + "123";
```

The first assignment statement causes a new string object with the value "`abc`" to be created and assigned to the variable string1.

The second assignment statement causes a new string object that contains the value "`abc123`" to be created and assigned to string1.

It is particularly important to remember that Strings are **immutable** when passing Strings as parameters to a method. A method may modify the value of a String argument in the method but the change will not be reflected back in the calling method.

For example, the following program segment prints the contents of a string and then passes the string to the method test which modifies the string and then prints the modified string. The calling method then prints the string again. Note that only within the test method is the modified value of the string visible.

```
public static void main(String[] parms)
{
     String string1;
     string1 = "Hello World!";
     System.out.println(string1);
     test(string1);
     System.out.println(string1);
}

public static void test(String string)
{
     string = string + " Goodbye World?";
     System.out.println(string);
}
```

```
Hello World!
Hello World! Goodbye World?
Hello World!
```

The **contents** of an array of Strings are mutable when the array is passed as a parameter. For example, the method below modifies one of the elements of an array of Strings and this modification is reflected in the calling method (the printStrings method is not included because it consists of a simple loop).

```
public static void main(String[] parms)
{
     String[] myArray = {"abc", "def", "ghi", "jkl"};

     printStrings("Before:", myArray);
     process(myArray);
     printStrings("After:", myArray);
}

public static void process(String[] strings)
{
     strings[1] = "hello";
}
```

```
Before: abc def ghi jkl

After: abc hello ghi jkl
```

If a new array had been created in the `process` method and assigned to the variable `strings`, this change would not have been reflected back in the calling method.

Note that an array of Strings can be initialized in the declaration statement, as shown above.

5.9 Data Validation

As was mentioned in Chapter 4 – File Input and Output, the try/catch block can be used to perform data validation. For example, if a String is supposed to contain an integer value, we can use the parseInt method to attempt to extract the integer value but if the String contains any invalid characters, Java generates an exception and terminates the program. By including a try/catch block around the parseInt method, we can catch any exceptions that are generated and allow the program to continue (assuming that there is something that we can do to fix the problem).

```
int value;
String string;

try
{
     value = Integer.parseInt(string);
}
catch (NumberFormatException ex)
{
     System.out.println("Invalid value: " +string);
     // do something to fix the error
}
```

This technique may be used during any conversion when the possibility of incorrect data exists. For example, Double.parseDouble can be enclosed in a try/catch block to catch any invalid doubles that were entered by the user.

5.10 String Formatting

Until now, when values are printed, we have not had many options to make the output look "nice" (line up values in columns, left-justify values, etc.). The String class contains a method that provides the programmer with additional flexibility when formatting string values.

The String method `format` takes a parameter that defines the format of the resulting string plus one or more arguments whose values will be formatted.

```
int intval = 10;
String string = String.format("%d", intval);
System.out.println(string);

10
```

If a string is being formatted in preparation for printing, the step that stores the string can be skipped by using the `format` method with `System.out` to format the string and then print the result.

```
int intval = 10;
System.out.format("%d", intval);

10
```

String formatting is identical whether the result is stored in a string (using `String.format`) or is printed (using `System.out.format`). The following examples illustrate the use of the `format` method. Also, the parameters that are to be formatted may be constant values; we will use constants in the following examples to reduce the amount of clutter in the examples.

```
System.out.format("%d", 10);
10
```

The format specifier is a character string that contains format codes plus optional characters that are to be included. In the example above, `%d` is a format code that specifies that an integer value is to be formatted. This format code does not do any special formatting but with the addition of optional parameters, the integer value can be formatted more precisely.

A format code may contain any combination of the following parameters preceding the `type`:

`%[flags][width][.precision]type`

The width parameter specifies the width of the result; however, if using the specified width would cause a loss of precision in the left-most digits, then the width is overridden. (Note that there are two leading blanks before the value `100`.)

```
System.out.format("%5d", 100);

  100
```

In the following example, the value requires more space than is specified so the format method increases the width so that the value can be represented appropriately.

```
System.out.format("%2d", 100);

100
```

A very useful flag is the `"-"` flag – this flag causes the value to be left-justified in the available space.

```
System.out.format("%-5d", 100);

100
```

If a floating-point value is to be generated, the `%f` format code is used. The width parameter defines the size of the entire field; if included, the precision specifies the number of digits to the right of the decimal point (if not specified, the default precision is 6 digits).

```
System.out.format("%5f", 100.0);  // default precision

100.000000
```

```
System.out.format("%6.2f", 100.0); //explicit precision (to right of decimal point)

100.00
```

```
System.out.format("%8.2f", 100.0); //explicit precision (to right of decimal point)

  100.00
```

```
System.out.format("%-8.2f", 100.0); // left justify

100.00
```

```
System.out.format("%06d", 1000);  // add leading zeroes

001000
```

Any characters that are included in the format specifier that are not part of a format code are included in the result.

```
System.out.format("The value is: %6.2f", 100.0);

The value is: 100.00
```

The `format` method does not generate a linefeed character after the output has been generated but the programmer may include the linefeed character format code as shown below.

```
System.out.format("%d %n", 10);

10
```

A string may also be included as a parameter. Strings are formatted using the `%s` format code.

```
System.out.format("%5s", "abc");

  abc
```

```
System.out.format("%-5s", "abc");

abc
```

While it is not common, it is valid to store the format in a String variable and then use the variable in a format method.

```
formatString = "%5s";
System.out.format(formatString, "abc");

  abc
```

The examples above illustrate the most common uses of format but there are many additional format codes and flags. Additional details on formatting can be found at:

http://download.oracle.com/javase/6/docs/api/java/util/Formatter.html

5.11 String Processing

The following methods read individual words from a file and format the words so that as many words as possible (up to a maximum of 50 characters). If an empty line is encounted, this indicates the end of a paragraph so the current output line is terminated and a blank line is generated.

For example, if the input file contains the following words:

```
Introduction

I've
been
involved
with
XP
for
a
couple
of
years
now
and
where
I've
seen
XP
implemented
properly
it
seems
to
have
worked
extremely
well.
```

the following output is generated:

```
Introduction

I've been involved with XP for a couple of years
now and where I've seen XP implemented properly it
seems to have worked extremely well.
```

```
public static void processWords()
{
      BufferedReader inFile;
      PrintWriter outFile;
      String inputLine;
      int position;

      try
      {
          inFile = new BufferedReader(new FileReader("In.txt"));
          outFile = new PrintWriter(new FileWriter("Out.txt"));

          position = 1; // using 1-based counting instead of 0-based
          inputLine = inFile.readLine();
          while (inputLine != null)
          {
              position = processWord(outFile, position, inputLine);
              inputLine = inFile.readLine();
          }
          inFile.close();
          outFile.close();
      }
      catch (IOException ioe)
      {
          System.out.println(ioe.getMessage());
      }
}

public static int processWord(PrintWriter outFile, int position, String word)
{
      final int LINE_SIZE = 50;

      if (word.equals(""))
      {   // an empty line so terminate current output line
          if (position > 1)
          {
              outFile.println();
          }
          outFile.println();
          position = 1;
      }
      else
      {   // ensure that there is sufficient space on the current line for the word
          if ((position+word.length()) > LINE_SIZE)
          {
              outFile.println();
              position = 1;
          }
          if (position > 1)
          {   // add blank before current word (as long as it isn't the first word)
              outFile.print(" ");
              position++;
          }
          outFile.print(word);
          position += word.length();
      }
      return position;
}
```

There are several points that should be noticed about these methods. First, a file object may be passed as a parameter to a method. Secondly, the method processWords needs to know the number of characters that have already been written to the current line. This information can not be saved in the method since the values of all variables are erased at the end of the method. So this "state" information must be passed from the processWords method and the updated state information must be returned by the processWord method.

5.12 String Tokenizing

When input lines are read from a file, it is frequently necessary to identify the pieces of text on each input line – this is referred to as determining the **tokens** on the line. One of the things that makes this processing non-trivial is that there may be white space between tokens (and this white space must be removed prior to processing the strings).

5.12.1 Extracting Tokens

The following method extracts the tokens in a String and returns them as individual tokens in a String array.

```
public static String[] extractTokens(String inputLine)
{
     String[] currentTokens;
     String[] newTokens;
     String string;
     int count;
     int currentToken;
     int length;

     count = 0;
     currentToken = 0;
     currentTokens = new String[1000];
     length = inputLine.length();
     while (count < length)
     {
        while ((count<length)&&(Character.isWhitespace(inputLine.charAt(count))))
        {  // skip over leading white space
           count++;
        }

        string = "";
        while ((count<length)&&(!Character.isWhitespace(inputLine.charAt(count))))
        {  // extract a token and store it in "string"
           string += inputLine.charAt(count);
           count++;
        }

        if (!((count>=length)&&(string.equals(""))))
        {  // store a token as long as it isn't empty and at the end of the string
           currentTokens[currentToken] = string; // copy the token into an array
           currentToken++;
        }
     }

     newTokens = new String[currentToken]; // shrink the array to the exact size
     System.arraycopy(currentTokens, 0, newTokens, 0, currentToken);
     return newTokens;
}
```

5.12.2 String Split Method

Java now includes a String tokenizing **method** named `split()` that performs processing that is similar to the `extractTokens` method shown above. The String method `split` returns an array of String tokens. A token is any set of characters that is surrounded by white space (blanks, tabs, linefeeds, etc.).

The split method uses a **regular expression** to define the delimiters. (Regular expressions are beyond the scope of these notes – we will use only one standard regular expression.) Normally, the regular expression string `"\\s+"` is used with split. This expression causes all white space to be ignored. Other expressions may be used but you must understand regular expressions before you change from the expression shown above.

```
string1 = "This    is    a    set     of     words.";
String[] result = string1.split("\\s+");
for (int count=0; count<result.length; count++)
{
     System.out.println("<" +result[count] +">");
}
```

```
<This>
<is>
<a>
<set>
<of>
<words.>
```

The split method does not skip over leading white space at the beginning of a string and generates an empty string in the first array element if there is leading white space. If this is a potential problem, then the string should be trimmed (using the String method `trim()`) of leading and trailing white space before the split method is used.

If split is used on an empty string, an array that consists of one empty string is generated.

The split method can also be used in an unusual manner to tokenize the contents of a String based on a specific character string. For example, splitting a string using `string.split("Fred")` would cause the string to be tokenized around occurrences of the String `"Fred"`. This is not a normal use of split but it might be useful in certain situations.

5.13 Binary Search of an Array of Strings

The example below is a simple binary search of an array of words that have already been sorted into ascending order. There are only a few differences from the binary search of an array of int's that was examined earlier.

```
public static int searchList(String[] list, String searchValue)
{
     int left;
     int right;
     int middle;
     int result;
     String middleElement;

     result = -1;
     left = 0;
     right = list.length-1;

     while ((left <= right) && (result == -1))
     {
        middle = left + ((right - left) / 2);
        middleElement = list[middle];
        comparison = middleElement.compareTo(searchValue);
        if (comparison == 0)
        {
           result = middle;
        }
        else if (comparison < 0)
        {
           left = middle + 1;
        }
        else if (comparison > 0)
        {
           right = middle - 1;
        }
     }
     return result;
}
```

5.14 Pattern Matching

The following method illustrates how the contents of a file can be changed so that all occurrences of one string (`fromString`) are changed to another string (`toString`). Four different methods that perform the same processing are included. (Of course, an easier way to perform this processing is to use the String method `replace`.)

```
public static void fileChange(String fileName, String fromString, String toString)
{
     FileReader fileReaderIn;
     BufferedReader fileIn;
     String inputLine;
     String result;

     try
     {
        fileReaderIn = new FileReader(fileName);
        fileIn = new BufferedReader(fileReaderIn);

        inputLine = fileIn.readLine();
        while (inputLine != null)
        {
           System.out.println(inputLine);
           result = replace(inputLine, fromString, toString);
           System.out.println(result);
           inputLine = fileIn.readLine();
```

```
         }
         fileIn.close();
      }
      catch (IOException ioe)
      {
         ioe.printStackTrace();
      }
}
```

```
public static String replace(String string, String from, String to)
{
      int count;

      for (count=0; count<string.length()-from.length()+1; )
      {
         if (string.substring(count,count+from.length()).equals(from))
         {
            string = string.substring(0,count) +to +string.substring(count+from.length());
            count += to.length();
         }
         else
         {
            count++;
         }
      }
      return string;
}

public static String replace(String string, String from, String to)
{
      int position;

      position = string.indexOf(from);
      while (position != -1)
      {
         string = string.substring(0,position) +to +string.substring(position+from.length());
         position += to.length();
         position = string.indexOf(from, position);
      }
      return string;
}

public static String replace(String string, String from, String to)
{
      String done;
      String toDo;
      int position;

      toDo = string;
      position = toDo.indexOf(from);
      done =  "";
      while (position != -1)
      {
         done += toDo.substring(0,position) +to;
         toDo = toDo.substring(position+from.length());
         position = toDo.indexOf(from);
      }
      done += toDo.substring(0);
      return done;
}
```

```
public static String replace(String string, String from, String to)
{
     String done;
     int position;
     int previousPosition;

     position = string.indexOf(from);
     previousPosition = 0;
     done = "";
     while (position != -1)
     {
        done += string.substring(previousPosition,position) +to;
        previousPosition = position+from.length();
        position = string.indexOf(from, position+from.length());
     }
     done += string.substring(previousPosition);
     return done;
}
```

Note that this is a very basic version of a pattern-matching method. Some of the issues that are not addressed in this method are: should the method be case sensitive or case insensitive? should only whole words be changed or should portions of a word be changed? should all occurrences of the "from" string be changed or just the first occurrence on each line? Also, in this method the output has been written to System.out; in a more useful version, the output would be written to a file (either the same file as the input file or a different file with a similar name).

5.15 More Pattern Matching

The following method illustrates how more sophisticated pattern-matching could be performed.

```
public static String replace(String string, String from, String to)
{
     int position;
     int fromLength;

     fromLength = from.length();
     for (position=0; position<string.length()-fromLength+1; position++)
     {
        if (match(string.substring(position),from))
        {
           string = string.substring(0, position)
                    +to
                    +string.substring(position+fromLength);
           position += to.length()-1;
        }
     }
     return string;
}

public static boolean match(String string, String matchString)
{
     String s1;
     String s2;
     boolean result;
     int count;

     result = true;
     for (count=0; count<matchString.length()&&result; count++)
     {   // does matchString match the beginning of string??
        s1 = string.substring(count,count+1);
        s2 = matchString.substring(count,count+1);
        if (!(s2.equals(".") || s1.equals(s2)))
        {
           result = false;
        }
     }
     return result;
}
```

The replace method above works correctly but is not a trivial method. The following version is simpler and easier to understand.

```
public static String replace(String string, String from, String to)
{
     String result;
     int position;
     int fromLength;

     fromLength = from.length();
     result = "";
     while (string.length() >= fromLength)
     {
        if (match(string,from))
        {
           result += to;
           string = string.substring(fromLength);
        }
        else
        {
           result += string.substring(0,1);
           string = string.substring(1);
        }
     }
     result += string;
     return result;
}
```

5.16 Counting Characters

The following method counts the frequency of characters in a string of text. The method illustrates various character manipulation techniques that are useful when processing Strings.

```
public static int[] processString(String string)
{
     int[] counts = new int[256];
     char c;
     Character ch;
     int count;

     for (count=0; count<string.length(); count++)
     {
        c = string.charAt(count);
```

```
         counts[c]++;
     }
     return counts;
}
```

The method above counts the frequency of each character in the input string and returns an array that contains the frequencies.

Assume that what we really require is only the frequency of the alphabetic characters and that we do not need to differentiate between lower-case characters and uppercase characters. The following method incorporates these changes into the processing. In this method, we take advantage of the fact that char's can be manipulated in the same manner as int's.

```
public static int[] processString(String string)
{
     int[] counts = new int[26];
     char c;
     int count;

     for (count=0; count<string.length(); count++)
     {
        c = string.charAt(count);
        if (Character.isLowerCase(c))
        {
           counts[c-'a']++;
        }
        else if (Character.isUpperCase(c))
        {
           counts[c-'A']++;
        }
     }
     return counts;
}
```

The following method is a slight improvement over the previous method. This method begins by converting all characters to lower case so that it is not necessary to check the case of each character in the string for both upper and lower case.

```
public static int[] processString(String string)
{
     int[] counts = new int[26];
     char c;
     int count;
     String string2;

     string2 = string.toLowerCase();
     for (count=0; count<string2.length(); count++)
     {
        c = string2.charAt(count);
        if (Character.isLowerCase(c))
        {
           counts[c-'a']++;
        }
     }
     return counts;
}
```

5.17 A Mutable String Class

The following class defines a simple mutable String class named MyString. The methods in this class are not complete but they provide a foundation for adding additional functionality. These methods permit a MyString object to be passed to a method and be modified by the method with the new value being visible in the calling method.

```
class MyString
{
     private String string;

     public MyString(String string)
     {
        this.string = string;
     }

     public String get()
     {
        return string;
     }

     public String set(String string)
     {
        this.string = string;
        return string;
     }

     public String append(String string)
     {
        this.string += string;
        return string;
     }

     public String toString()
     {
        return string;
     }
}
```

As an exercise, rewrite the method above so that it uses a char array instead of a String variable to store the characters.

5.18 StringBuffer Class

When a String object is created, the object is immuatable. If the contents of a variable must be modified, a new String object is created and it replaces the previous String object. If this process occurs repeatedly, then a large number of String objects will have been created and then discarded. (This process is examined in more detail in Chapter 7 – Object Representation.)

Java has recently included a new character string manipulation class, the `StringBuffer`. With this class, the string inside the StringBuffer object is **mutable**, that is, the string may be modified without having to create a new object.

Unfortunately, manipulating a StringBuffer object is slightly more difficult than manipulating a String object because the StringBuffer does not provide the shortcuts that are available with Strings. For example, String concatenation can be performed using the "+" operator but this feature is not available with StringBuffers.

5.18.1 StringBuffer Methods

The basic methods provided by the StringBuffer class are:

`new StringBuffer()`	create a new StringBuffer object; initialize the object to an empty string ("")
`new StringBuffer(length)`	create a new StringBuffer object that has room for length characters; initialize the object to an empty string ("")
`new StringBuffer(string)`	create a new StringBuffer object and initialize the object to the contents of string
`append(string)`	append the string to the end of the current StringBuffer
`length()`	returns the number of characters in the StringBuffer
`substring(start)`	extract the characters beginning at start and continuing until the end of the buffer
`substring(start,end)`	extract the characters beginning at start and continuing up to but not including end
`insert(start,string)`	insert a String into the StringBuffer
`replace(start,end,string)`	replace a portion of the characters in the StringBuffer
`delete(start,end)`	delete a portion of the characters in the StringBuffer

There are other methods that are provided by the StringBuffer class – they can be found in the Java documentation or a recent Java textbook.

5.18.2 Using a StringBuffer Object

The following example compares the use of a String object and a StringBuffer object. Each method creates a String (or StringBuffer) that is 1000 blanks long. This is not a particularly elegant way of performing the processing.

```
public static void main(String[] parms)
{
     createString(1000);
     createStringBuffer(1000);
}

public static String createString(int length)
{
     String result;
     int count;
     result = "";
     for (count=0; count<length; count++)
     {
        result += " ";
     }
     return result;
}

public static StringBuffer createStringBuffer(int length)
{
     StringBuffer result;
     int count;
     result = new StringBuffer(length);
     for (count=0; count<length; count++)
     {
        result.append(" ");
     }
     return result;
}
```

The method checkHeap is defined in Chapter 19 – Miscellaneous Topics. This method prints the amount of available (free) memory. The following 4 lines of output show the amount of free memory before the loop in createString, after the loop in createString, before the loop in createStringBuffer, and after the loop in createStringBuffer. Notice that the createString method uses 120,952 bytes of memory while the createStringBuffer method uses 5,768 bytes. So the memory cost of using a String instead of a StringBuffer is significant; using a String also requires more execution time.

```
Free memory: 1656856
Free memory: 1535904

Free memory: 1535904
Free memory: 1530136
```

Characters that are stored in a String object are actually stored in an array that is exactly the right size for the characters. For example, the string "abc" would be stored in an array of 3 characters. StringBuffer objects also store the characters in an array but the array is normally larger than is necessary to contain the characters. Inside the object, the StringBuffer keeps track of the number of characters that are actually stored in the array. If the array is too small to store all of the characters, the StringBuffer object creates a new, larger array and copies the characters from the original array into the new array. Thus, the array used by a StringBuffer object grows larger as necessary.

6 OBJECT EXAMPLES

6.1 Introduction

In this chapter we examine several simple problems that illustrate the object principles defined in Chapter 3 – Objects. Each of the examples is quite basic but could easily be improved.

6.2 Person Example

We begin with a simple object that maintains information about individuals. The object doesn't know much or do much but it illustrates the basic components of an object: variables, a constructor, and an instance method.

```
public class Person
{
     private String name;
     private String address;
     private int birthYear;

     public Person(String name, String address, int birthYear)
     {
        this.name = name;
        this.address = address;
        this.birthYear = birthYear;
     }

     public String toString()
     {
        return name +" " +address +" " +birthYear;
     }
}
```

Once we know the year in which someone was born, it is natural to want to know how old the person is. For this we need to know the current year and that information can be obtained using the Calendar class in java.util. (The Calendar class is used only to illustrate that there are many helpful classes provided with Java.)

We can now add a getAge method that computes a person's age. (Do you understand why you should not store a person's age in the class instead of computing the age when required?)

The getAge method is not exact since only the birth year is provided and so the age returned may be one year greater than the actual age. In order to determine a person's age exactly, we also need to know the person's birthday and the current day and month of the year.

```
public class Person
{
     private String name;
     private String address;
     private int birthYear;

     public Person(String name, String address, int birthYear)
     {
        this.name = name;
        this.address = address;
        this.birthYear = birthYear;
     }

     public int getAge()
     {
        Calendar calendar;
        int currentYear;

        calendar = Calendar.getInstance();
        currentYear = calendar.get(Calendar.YEAR);
        return currentYear - birthYear;
     }

     public String toString()
     {
        return name +" " +address +" " +birthYear;
     }
}
```

6.3 Video Store Example

In the next example, we want to be able to keep track of the videos (DVD's) that are available at a video store.

The following video class maintains information about a specific title, including the number of copies owned by the video store and the number of copies that are currently available in the store.

When a video is rented, the total revenue generated by that video is increased by the rental cost and the number of copies that are currently available is decreased by 1. If a video is sold/purchased, the total revenue is increased and the number of copies available and the total number of copies are decreased by 1. When a rented video is returned, the number of copies available is increased by 1.

There are 3 pieces of data (title, actor, and genre) that are maintained in the class but are not currently used. However, these values would make it possible to search all of the videos looking for a specific title, actor, and/or genre.

```
public class Video
{
     private String title;
     private String actor;
     private String genre;
     private int totalCopies;
     private int copiesAvailable;
     private double rentalCost;
     private double purchaseCost;
     private double revenue;

     public Video(String title, String actor, String genre,
                             int totalCopies, double rentalCost)
     {
        this.title = title;
        this.actor = actor;
        this.genre = genre;
        this.totalCopies = totalCopies;
        this.copiesAvailable = totalCopies;
        this.rentalCost = rentalCost;
        purchaseCost = 5.0 * rentalCost;
        revenue = 0.0;
     }

     public void rent()
     {
        revenue += rentalCost;
        copiesAvailable--;
     }

     public void purchase()
     {
        revenue += purchaseCost;
        copiesAvailable--;
        totalCopies--;
     }

     public void returnVideo()
     {
        copiesAvailable++;
     }

     public String toString()
     {
        return title +" " +actor +" " +genre +" " +totalCopies
                    +" " +copiesAvailable +" " +rentalCost +" " +revenue;
     }
}
```

6.4 Employee Example

The following example maintains information about the employees in a company. Each employee works on zero or more projects and the name of each project is maintained in the object.

There is a method (addProject) that is used to add a new project to an employee's project list. The addProject method should ensure that the project does not already exist in the list before adding a new project (but in this version, it doesn't). There is not a comparable method to remove a project from an employee's project list. (See the Coffee Shop example in this chapter for a similar `remove` method.)

```
public class Employee
{
     private String number;
     private String name;
     private String address;
     private String[] projects;
     private int numProjects;

     public Employee(String number, String name, String address)
     {
        this.number = number;
        this.name = name;
        this.address = address;
        projects = new String[100];
        numProjects = 0;
     }

     public void addProject(String newProject)
     {
        projects[numProjects] = newProject;
        numProjects++;
     }

     public void getProjects()
     {
        String myProjects;
        int count;

        myProjects = "";
        for (count=0; count<numProjects; count++)
        {
           myProjects += projects[count] +"  ";
        }
        return myProjects;
     }

     public String toString()
     {
        return number +" " +name +" " +address +" " +getProjects();
     }
}
```

6.5 GST Account Example

The following example performs some basic accounting for businesses by maintaining the amount of GST that has been collected. For each service that is provided by a business, the service is identified as GST exempt or not GST exempt. The object maintains the total value of all services that were provided and also computes the amount of GST that is payable on the services that are not GST exempt.

```
public class GSTAccount
{
     private String businessName;
     private String businessNumber;
     private double gstServices;
     private double exemptServices;
     private double totalGST;
     private double GST_RATE = 0.06;

     public GSTAccount(String businessName, String businessNumber)
     {
        this.businessName = businessName;
        this.businessNumber = businessNumber;
        gstServices = 0;
        ememptServices = 0;
        totalGST = 0;
     }

     public void addService(double amount)
     {  // by default, services are not GST exempt
        addService(amount, false);
     }

     public void addService(double amount, boolean exempt)
     {
        if (exempt)
        {
           exemptServices += amount;
        }
        else
        {
           gstServices += amount;
        }
     }

     public double getGST()
     {
        return gstServices * GST_RATE;
     }

     public double getNetRevenue()
     {
        return exemptServices + gstServices - getGST();
     }

     public String toString()
     {
        return businessName +" " +businessNumber
                            +" " +getNetRevenue() +" " + getGST();
     }
}
```

6.6 Coffee Shop Example

The final example creates some objects that would be used if you were running a coffee shop. Since the example is slightly more complex than the previous examples, we will grow this example slowly.

6.6.1 Iteration 1

Initially, we create a Drink class that contains only the name of a drink and the cost of the drink. We can obtain the information about the drink by sending the `toString` message to the Drink object.

```
public class Drink
{
     private String drinkName;
     private double cost;

     public Drink(String drinkName, double cost)
     {
        this.drinkName = drinkName;
        this.cost = cost;
     }

     public String toString()
     {
        return drinkName +" " +cost;
     }
}
```

The Order class maintains the drinks that have been ordered by a customer. When a new order is created, the Order object contains the customer's name but it does not contain any drinks. The String representing a drink and its price are added to the Order object using the addDrink method.

```
public class Order
{
     private String customerName;
     private String[] drinks;
     private int numDrinks;

     public Order(String customerName)
     {
        this.customerName = customerName;
        drinks = new String[100];
        numDrinks = 0;
     }

     public void addDrink(String drink)
     {
        drinks[numDrinks] = drink;
        numDrinks++;
     }

     public void printOrder()
     {
        int count;

        System.out.print(customerName +"'s order: ");
        for (count=0; count<numDrinks; count++)
        {
           System.out.print(drinks[count] +"; ");
        }
        System.out.println();

     }
}
```

6.6.2 Iteration 2

In this iteration we add a facility that determines the cost of an order. Unfortunately, in the first iteration, we stored the Drink information as an array of Strings in the Order object. By storing a drink as a string, it is difficult to obtain the price of a drink. In this iteration we will refactor the Order class slightly by storing Drink objects in the Order object instead of storing the String representation of the drink.

Once addDrink is modified to accept a Drink object, we can add a getCost method to the Drink class that returns the cost of a drink.

Now the printOrder method can be extended to calculate the total cost of the drinks.

```
public class Drink
{
     private String drinkName;
     private double cost;

     public Drink(String drinkName, double cost)
     {
        this.drinkName = drinkName;
        this.cost = cost;
     }

     public double getCost()
     {
        return cost;
     }

     public String toString()
     {
        return drinkName +" " +cost;
     }
}
```

```
public class Order
{
     private String customerName;
     private Drink[] drinks;
     private int numDrinks;

     public Order(String customerName)
     {
        this.customerName = customerName;
        drinks = new Drink[100];
        numDrinks = 0;
     }

     public void addDrink(Drink drink)
     {
        drinks[numDrinks] = drink;
        numDrinks++;
     }

     public void printOrder()
     {
        int count;
        double orderTotal;

        orderTotal = 0.0;
        System.out.print(customerName +"'s order: ");
        for (count=0; count<numDrinks; count++)
        {
           System.out.print(drinks[count] +"; ");
           orderTotal += drinks[count].getCost();
        }
        System.out.println("total cost: " +orderTotal);

     }
}
```

6.6.3 Iteration 3

Since customers occasionally change their minds about their orders, we need to add the facility to remove a drink from an order. For now, the processing implemented to remove a drink involves locating the drink and then setting the object at that location in the array to `null`. Unfortunately, this change has a ripple effect because the printOrder method must now ensure that each object in the `drinks` array is a valid object and has not been set to `null`.

```
public class Drink
{
     private String drinkName;
     private double cost;

     public Drink(String drinkName, double cost)
     {
        this.drinkName = drinkName;
        this.cost = cost;
     }

     public double getCost()
     {
        return cost;
     }

     public String toString()
     {
        return drinkName +" " +cost;
     }
}
```

```
public class Order
{
     private String customerName;
     private Drink[] drinks;
     private int numDrinks;

     public Order(String customerName)
     {
        this.customerName = customerName;
        drinks = new Drink[100];
        numDrinks = 0;
     }

     public void addDrink(Drink drink)
     {
        drinks[numDrinks] = drink;
        numDrinks++;
     }

     public void removeDrink(Drink drink)
     {
        int count;
        boolean found;

        found = false;
        for (count=0; count<numDrinks && !found; count++)
        {
           if (drink.equals(drinks[count]))
           {
              drinks[count] = null;
              found = true;
           }
        }
     }

     public void printOrder()
     {
        int count;
        double orderTotal;

        orderTotal = 0.0;
        System.out.print(customerName +"'s order: ");
        for (count=0; count<numDrinks; count++)
        {
           if (drinks[count] != null)
           {
              System.out.print(drinks[count] +"; ");
              orderTotal += drinks[count].getCost();
           }
        }
        System.out.println("total cost: " +orderTotal);
     }
}
```

6.6.4 Iteration 4

The use of a `null` value to indicate that a location in an array is no longer used is somewhat problematic. If the only method that will ever process the `drinks` array is printOrder, then we could probably live with this organization. However, it is likely that other methods will also have to process the array and they would also have to be aware of the potential for `null` values. So, at this time, it is a good idea to perform a simple refactoring so that when a drink object is removed, the objects that come after the deleted object are moved to the left instead of inserting the null value. System.arraycopy can be used to move the contents of an array to the left (or to the right). The check for a null value can now be removed from `printOrder`.

```
public class Drink
{
      private String drinkName;
      private double cost;

      public Drink(String drinkName, double cost)
      {
          this.drinkName = drinkName;
          this.cost = cost;
      }
      public double getCost()
      {
          return cost;
      }

      public String toString()
      {
          return drinkName +" " +cost;
      }
}
```

```
public class Order
{
      private String customerName;
      private Drink[] drinks;
      private int numDrinks;

      public Order(String customerName)
      {
          this.customerName = customerName;
          drinks = new Drink[100];
          numDrinks = 0;
      }

      public void addDrink(Drink drink)
      {
          drinks[numDrinks] = drink;
          numDrinks++;
      }

      public void removeDrink(Drink drink)
      {
          int count;
          boolean found;
          found = false;
          for (count=0; count<numDrinks && !found; count++)
          {
              if (drink.equals(drinks[count]))
              {
                  System.arraycopy(drinks,count+1,drinks,count,numDrinks-count);
                  numDrinks--;
                  found = true;
              }
          }
      }

      public void printOrder()
      {
          int count;
          double orderTotal;

          orderTotal = 0.0;
          System.out.print(customerName +"'s order: ");
          for (count=0; count<numDrinks; count++)
          {
              System.out.print(drinks[count] +"; ");
              orderTotal += drinks[count].getCost();
          }
          System.out.println("total cost: " +orderTotal);
      }
}
```

6.6.5 Recap

The Coffee Shop classes still need some additional functionality before they would actually be useful but we have developed a foundation with the Drink class and the Order class.

One of the problems that we encounted during the development of the classes was the difficulty of making changes to the contents of an array. We will examine a more convenient way of making modifications to a collection of objects in Chapter 9 – Object Collections.

6.7 Words Class

The following class Words reads the contents of a file (one line at a time) and returns the words on each line (one at a time).

The constructor is passed a String fileName. To make the processing simpler, the file processing is moved into its own class, Lines.

The method nextWord returns the next word in the file. If the current line contains at least one word that has not yet been processed, that word is returned. If there are no more words on the current line, the next line is read from the file and the first word on that line is returned. When there are no more words in the file, the file is closed and the value **null** is returned.

It is assumed that there is at least one word on each line in the file. The words are separated by whitespace.

```
class Words
{
     private Lines lines;
     private String[] words;
     private String inputLine;
     private int length;
     private int wordCount;

     public Words(String fileName)
     {
        lines = new Lines(fileName);
        wordCount = 0;
        length = 0;
     }

     public String getWords()
     {
        String result;

        if (wordCount>=length)
        {
           inputLine = lines.readLine();
           if (inputLine!=null)
           {
              words = (inputLine.trim()).split("\\s+");
              length = words.length;
              wordCount = 0;
           }
           else
           {
              words = null;
           }
        }
        if (words != null)
        {
           result = words[wordCount];
           wordCount++;
        }
        else
        {
           result = null;
        }
        return result;
     }
}
```

The Lines class below reads one line from a file at a time. This class is a simplified version of the IOProcess class described earlier in these notes.

```
class Lines
```

```
{
     private BufferedReader fileIn;
     private String inputLine;

     public Lines(String fileName)
     {
        fileOpen(fileName);
     }

     private void fileOpen(String fileName)
     {
        try
        {
           fileIn = new BufferedReader(new FileReader(fileName));
        }
        catch (IOException ioe)
        {
           System.out.println("Error");
        }
     }

     public String readLine()
     {
        try
        {
           inputLine = fileIn.readLine();
           if (inputLine == null)
           {
              fileIn.close();
           }
        }
        catch (IOException ioe)
        {
           System.out.println("Error");
        }
        return inputLine;
     }
}
```

7 OBJECT REPRESENTATION

7.1 Introduction

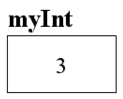
In this chapter we examine how objects are represented. This information has an impact on how objects are passed as parameters and how objects are (or should be) manipulated.

7.2 Object References

With the primitive data types, the value stored in a variable is the value of the primitive data type. For example, in the following instructions

```
int myInt;
myInt = 3;
```

the value 3 is stored in the variable myInt.



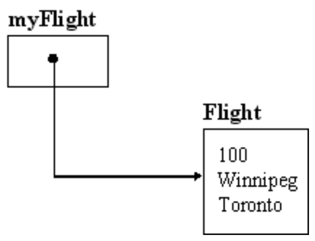
If the value in myInt is assigned to the variable myInt2, a copy of the value in myInt is made and then stored in myInt2. As a result, both variables contain their own copy value 3.

```
int myInt;
int myInt2;
myInt = 3;
myInt2 = myInt;
```

myInt
3
myInt2
3

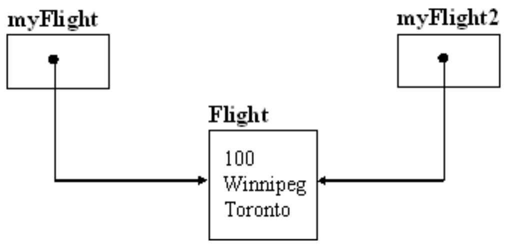
However, when an object is created, the value stored in the variable is a **reference** (or **pointer**) to the object, not the object itself. For example, the following instructions create a new Flight object and store the reference to the object in the variable myFlight.

```
Flight myFlight;
myFlight = new Flight(100, "Winnipeg", "Toronto");
```



If we subsequently assign the value in the variable myFlight to another Flight variable, myFlight2, the contents of the variable myFlight are copied and then assigned to the variable myFlight2. As a result, the variable myFlight2 points to the same object as myFlight (i.e. the object is not copied). This type of copy in which only the object reference is copied is referred to as a **shallow copy**.

```
Flight myFlight;
Flight myFlight2;
myFlight = new Flight(100, "Winnipeg", "Toronto");
myFlight2 = myFlight;
```



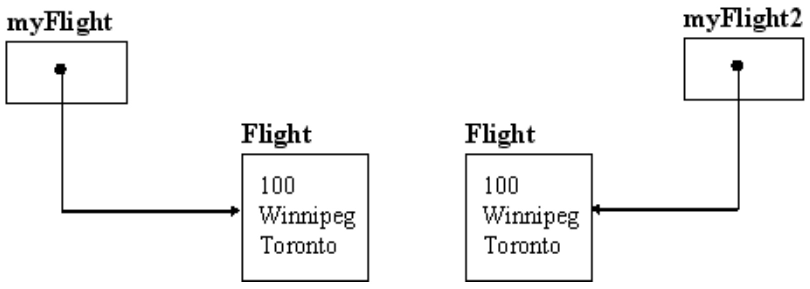
If a change is made to the Flight object via myFlight, myFlight2 will see the change because it refers to (points to) the same object.

If a true copy of an object is required, we could create a method that copies the contents of one object to a new object. (Making a copy of an object is referred to as "**cloning**" the object.) For example, the following method could be added to the Flight class to make a copy of the contents of an existing flight.

```
public Flight clone()
{    // create a new Flight object with the same instance variable values
     Flight newFlight;
     newFlight = new Flight(flightNumber, flightOrigin, flightDestination);
     return newFlight;
}
```

The clone method could then be used as follows:

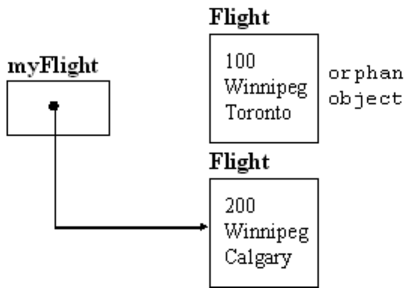
```
Flight myFlight;
Flight myFlight2;
myFlight = new Flight(100, "Winnipeg", "Toronto");
myFlight2 = myFlight.clone();
```



Now there are two identical copies of the object. If one of the objects is modified, that change does not affect the contents of the other object. Making a copy of an object in this manner is referred to as making a **deep copy**.

If a variable contains a reference to an object but later a new object is created and the reference to the object is assigned to the variable, the original object becomes an “**orphan**” – it is not pointed to by any variable.

```
Flight myFlight;
myFlight = new Flight(100, "Winnipeg", "Toronto");

…

myFlight = new Flight(200, "Winnipeg", "Calgary");
```



These orphan objects take up memory that could be used more productively by the program. Java uses a technique referred to as **garbage collection** when a program is running low on memory. Garbage collection involves identifying objects that are no longer referred to by any variable and then reusing the space occupied by those objects. (See Chapter 19 – Miscellaneous Topics for some additional information on garbage collection.)

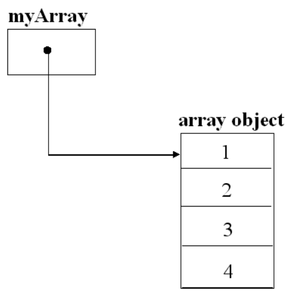
7.3 Arrays are Objects

Arrays look like objects in some ways but not in others. For example, arrays use the [ ] notation to refer to an element of the array – this is not a notation that “normal” objects use. Also, unlike the classes examined so far, Java does not contain a class named “array”. Java does contain an Array class but this class consists primarily of static methods that are used to manipulate arrays.

These differences between arrays and objects are somewhat confusing and it takes a while to become used to the fact that although arrays use a special syntax at times, arrays really are objects.

For example, if an array of 4 int values is created, the array object contains the 4 values and the variable to which the array object is assigned contains a reference to the array object.

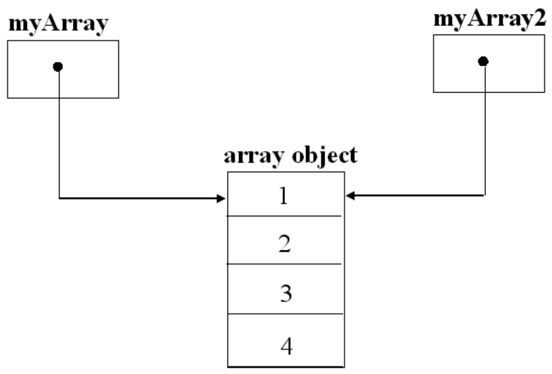
```
int[] myArray = {1, 2, 3, 4};
```



As with other objects, if the array in one variable is assigned to another variable, only the **reference** to the array elements is assigned to the variable.

```
int[] myArray = {1, 2, 3, 4};
int[] myArray2;
myArray2 = myArray;
```

The assignment statement above does **not** cause a copy of the data elements in myArray to be made so changing an element in myArray causes an equivalent change to myArray2 because both myArray and myArray2 reference the same object (i.e. the array elements).

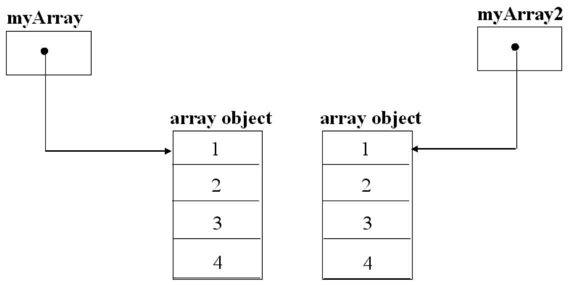


Again, this type of copy is referred to as a **shallow copy** since the actual array elements are not copied during the assignment process.

If you require that two arrays have completely separate copies of the array elements, you must copy the elements one-at-a-time from one array to the other array. This is a **deep copy**. When System.arraycopy() is used to copy array elements, it copies the contents of the variables in the array object – it performs a deep copy **only if** the data types being copied are primitive data types. (More on this below.)

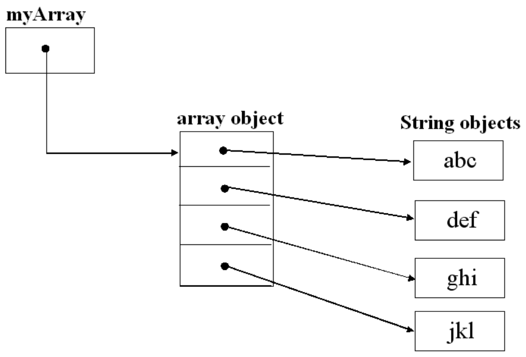
```
int[] myArray = {1, 2, 3, 4};
int[] myArray2;
myArray2 = new int[myArray.length];
System.arraycopy(myArray,0,myArray2,0,myArray.length);
```

After System.arraycopy copies the elements, each variable refers to a different array object and changing the contents of one array does not affect the contents of the other array.



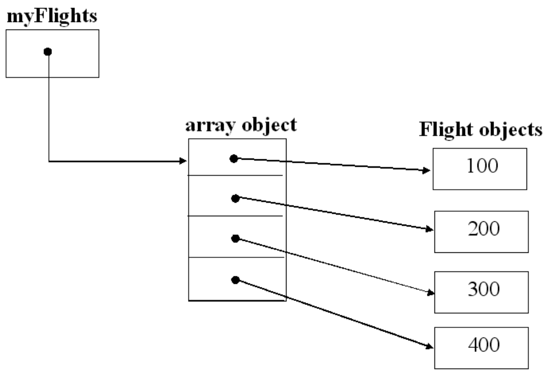
If an array is an array of objects, each element in the array contains a reference to the associated object. For example, the following statement creates an array of String objects.

```
String[] myArray = {"abc", "def", "ghi", "jkl"};
```

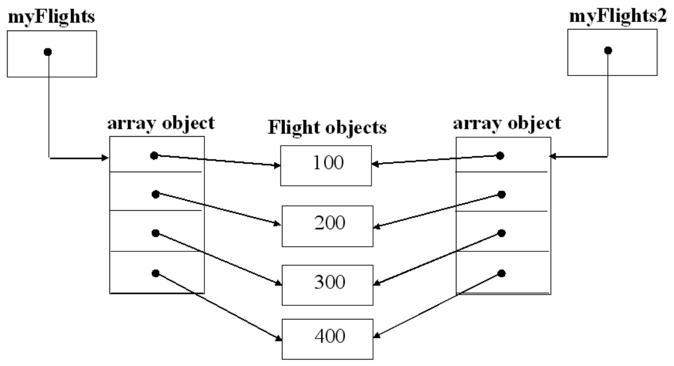


Similarly, if an array contains user-defined objects, such as Flight objects, each array element contains a reference to the associated flight object.

```
Flight[] myFlights = new Flight[4];
```



System.arraycopy does not do what you might expect when copying arrays of such objects. If System.arraycopy is used to copy the contents of the variable myFlights to another variable, myFlights2, the contents of the first array are copied to the second array. However, since the array elements contain object references, each element in myFlights2 points to the same object as the corresponding element in myFlights. As a result, making a change to Flight 100 via myFlights also changes the object that myFlights2 refers to. So, in this case, System.arraycopy does not perform a deep copy, it copies the contents of the array but does not make a copy of whatever the array points to if the array contains object references.



When System.arraycopy is used to copy an array of objects, if the objects are **immutable**, then Java in effect performs a deep copy; if the objects are not immutable, then Java does not perform a deep copy. There is additional information on immutable objects in the next section.

If a deep copy of an object is required, additional programming must be performed to ensure that a true deep copy of the object is made. Even Java's `.clone` method does not perform a true deep copy.

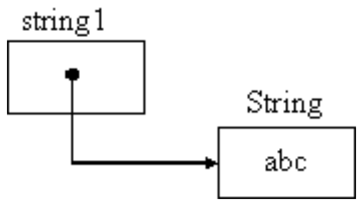
7.4 Strings

Strings are objects but you must be careful with Strings because there are several subtle points with Strings that can confuse the programmer.

Since a String is an object, the value assigned to a String variable is an object reference, not the actual value of a String. Executing the statement:

```
string1 = "abc";
```

causes Java to assign a reference to the String object to the variable string1.



When comparing string objects, it is important to remember to use the `equals()` String method to perform the comparison. If two String variables are compared using the comparison operator ==, the contents of the two variables (which contain the object **references** of the two Strings) are compared, not the contents of the Strings. Since you almost never want to compare object references, you must compare the variables as follows:

```
String string1;
String string2;

if (string1.equals(string2))
{
     ...
}
```

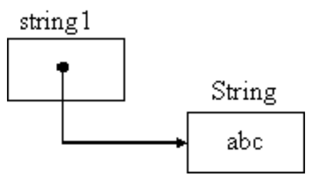
Using `equals()` ensures that the contents of the two strings are compared, not their object references.

7.4.1 Strings are Immutable

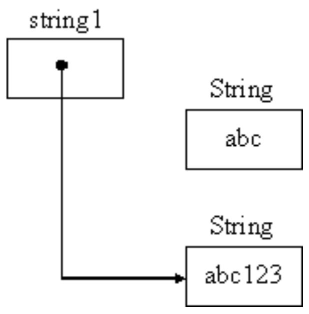
Java's String objects are **immutable**. This means that once a String object has been created, the object itself can not be modified. However, a different String object can be assigned to the same variable. For example, the following statements are perfectly valid.

```
string1 = "abc";
string1 = string1 + "123";
```

The first assignment statement causes a new string object with the value "`abc`" to be created and its object reference is assigned to the variable string1.



The second assignment statement causes a new string object that contains the value "`abc123`" to be created and its object reference assigned to string1.



The object that contains "abc" still exists but has become an orphan object since it is no longer pointed to by any variable. The space occupied by this object will be reclaimed the next time that the garbage collector runs.

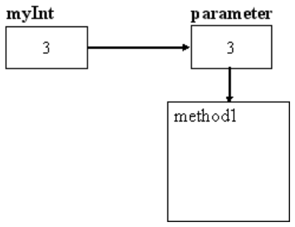
7.5 Passing Parameters

An understanding of how Java represents both primitive data items and objects makes it easier to understand how parameters are passed from a calling method to a called method and to understand the implications of making modifications to the parameters in the called method.

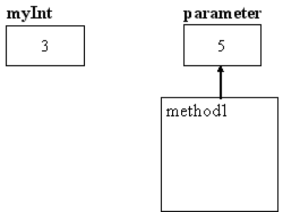
When a variable is passed to a method, the calling method makes a copy of the value in the variable and that copy is passed to the called method. (This process is referred to as "call by value" or "pass by value".) When control returns to the called method, the copies of the contents of the parameters are discarded.

When a primitive data item is passed to a method, a copy of the value of the primitive data item is passed to the method. The method may change the value of the parameter but the modified value is just a copy of the original value and so the original value in the calling method is not modified.

```
myInt = 3;
method1(myInt);
System.out.println(myInt);

public static void method1(int myInt)
{
     myInt = 5;
}

3
```



When myInt is passed to method1, a copy of the value in myInt is made and this value is passed to the method.

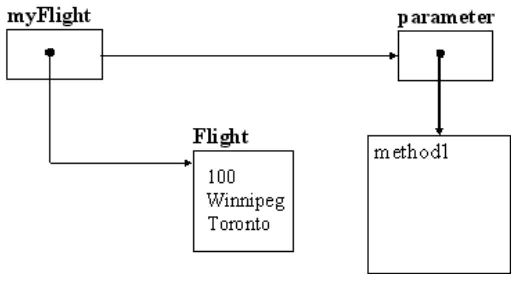


method1 may modify the parameter value but since the value is not copied back to myInt, the modification is not reflected in the calling method (and so the value 3 is printed).

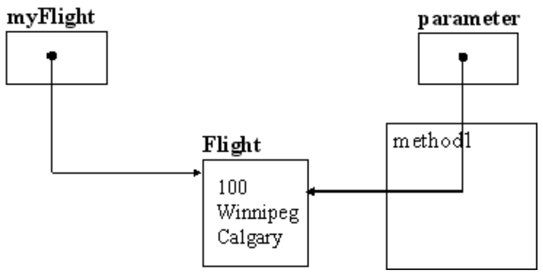
When an object is passed to a method, a copy of the value in the variable is again passed to the method. However, the value in the variable is a pointer to the associated object, not the object itself. So it is the object reference that is passed by to the called method. Therefore, the called method is able to access the object itself (via the copied reference). Unlike passing a primitive data item by value, if the called method modifies the object that is referred to by the pointer, the object is modified and the modification is reflected in the calling method.

```
myFlight = new Flight(100, "Winnipeg", "Toronto");
method1(myFlight);
System.out.println(myFlight);

public static void method1(Flight myFlight)
{
     myFlight.changeDestination("Calgary");
}

100  Winnipeg  Calgary
```

A copy of the pointer to the Flight object is passed as a parameter to the method.



If the method modifies the Flight object, the modification is reflected in the calling method since the parameter points at the original object.

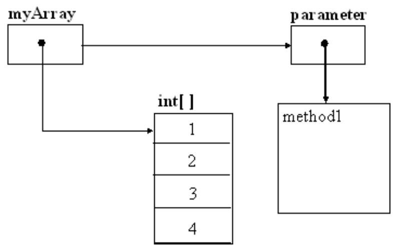


However, if a new Flight object is instantiated in the method and is assigned to the parameter, this new object will not be visible in the calling method.

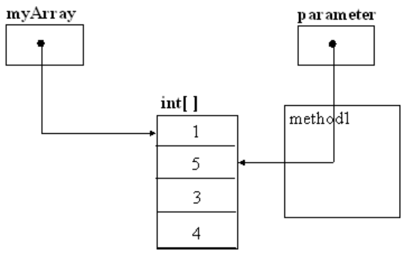
Since an array is an object, it is perfectly valid to modify the contents of an existing (instantitated) array in a called method; the modifications are reflected in the calling method.

```
int[] myInts = {1, 2, 3, 4};
method1(myInts);
printList(myInts);

public static void method1(int[] myInts)
{
     myInts[1] = 5;
}

1 5 3 4
```

A pointer to the array is copied to the parameter that is passed to the method.



If the contents of the array are modified in the method, the modification is visible in the calling method.



However, if a parameter is assigned a new array object by the called method, that modification is not visible in the calling method.

If an immutable object is modified in a called method, the modification is not reflected in the calling method because modifying an immutable object results in the creation of a new object (reference).

For example, Strings are **immutable** so a method may modify the value of a String parameter but the change will not be reflected back in the calling method.

The following program segment prints the contents of a string and then passes the string to the method test which modifies the string and then prints the modified string. The calling method then prints the string again.

```
public static void main(String[] parms)
{
     string1 = "Hello World!";
     System.out.println(string1);
     test(string1);
     System.out.println(string1);
}

public static void test(String string)
{
     string = string + " Goodbye World?";
     System.out.println(string);
}
```

```
Hello World!
Hello World! Goodbye World?
Hello World!
```

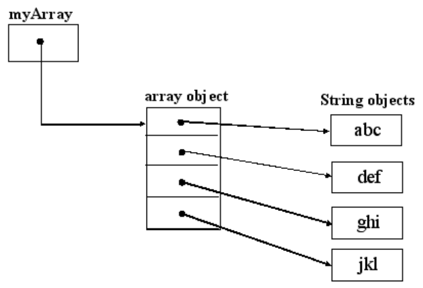
However, the contents of an **array** of Strings are mutable when the array is passed as a parameter. For example, the method below modifies one of the elements of an array of Strings and this modification is reflected in the calling method (the printStrings method is not included because it consists of a simple loop).

```
public static void main(String[] parms)
{
     String[] myArray = {"abc", "def", "ghi", "jkl"};
     printStrings("Before:", myArray);
     process(myArray);
     printStrings("After:", myArray);
}

public static void process(String[] strings)
{
     strings[1] = "hello";
}
```

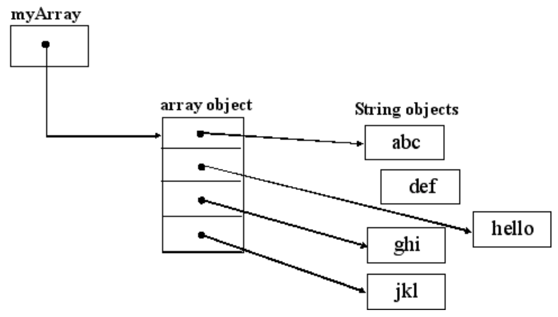
```
Before: abc def ghi jkl

After: abc hello ghi jkl
```

If you examine a diagram below, it should be clear that although the contents of the second element in the array were modified (to point to a different string), the array itself was not modified and so the change is visible in the calling method.

Before:



After:



If a new array had been created in the `process` method and assigned to the variable `strings`, this change would not have been reflected back in the calling method.

7.6 Cloning Objects

It was mentioned briefly at the beginning of this chapter that making a copy of an object only makes a copy of the reference to the object, not the object itself. As a result, both the original reference and the copied reference point to the same object. If this is not what is desired, then the object itself must be copied, this is referred to as cloning the object. A simple method named clone can be added to a class to enable a copy/clone of an object to be made.

```
Flight clonedFlight = existingFlight.clone();
```

```
public class Flight
{
     private int flightNumber;
     private String flightOrigin;
     private String flightDestination;

     public Flight(int number, String origin, String destination)
     {
        flightNumber = number;
        flightOrigin = origin;
        flightDestination = destination;
     }

     public Flight clone()
     {   // create a new object with the values of the current instance variables
        return new Flight(flightNumber, flightOrigin, flightDestination);
     }
}
```

Java provides its own `clone` method but the setup for that method is slightly more complex than the method shown above; also, Java's clone method performs a deep copy only to one level.

7.7 Information About Objects

Java allows the programmer to ask if an object is a member of a particular class using the **`instanceof`** comparison operator. The following instructions generate the expected output:

```
Flight flight1 = new Flight(100, "Winnipeg", "Toronto");
if (flight1 instanceof Flight)
{
     System.out.println("Yes, flight1 is really a Flight object.");
}
```

```
Yes, flight1 is really a Flight object.
```

The instanceof operator is examined in more detail in Chapter 10 – Object Hierarchies.

7.8 Summary

In theory, it is not necessary to understand the details of how objects are represented; however, it practice it is necessary. The representation of objects has an impact on how values are copied and how values are passed as parameters to methods.

8 OBJECT ORIENTATION PRACTICES

8.1 Introduction

In this chapter we examine how good object orientation practices can be used during the development of a program that manages credit cards. The example will be developed (grown) slowly so that we can be comfortable that all code that has been developed at a particular point in time works correctly, thus minimizing the amount of time that is spent debugging. Any debugging that is required should be isolated in a small amount of code and therefore easy to fix.

The initial code will **not** follow good object orientation practices. Instead, some code that works correctly but is not well designed will be developed. This code will then be improved in stages in order to illustrate the difference between code that works and code that works and is also well designed.

8.2 Basic Class Structure

The information that is to be maintained about each credit-card holder is: account number, account name, current credit-card balance, and credit-card limit. (If the credit-card balance is positive, the customer owes money to the credit-card institution, if the balance is negative, the customer has over-paid the account.) If we were to use parallel arrays to manage this information, we would require 4 arrays, one for each piece of information. However, when objects are used, the 4 pieces of information can be defined in one object.

The basic class structure is shown below. The constructor is given all 4 pieces of information when a new object is created.

```
public class CreditCard
{
     String accountNumber;
     String accountName;
     double currentBalance;
     double creditLimit;

     public CreditCard(String number, String name, double balance, double limit)
     {
        accountNumber = number;
        accountName = name;
        currentBalance = balance;
        creditLimit = limit;
     }
}
```

The credit-card objects will be stored in an array of credit-card objects.

```
public static void main(String[] parms)
{
     CreditCard[] creditCards;

     creditCards = createCreditCards();
}
```

To make the creation of the credit-card objects simple, we will hard-code the objects in the `createCreditCards` method. The `createCreditCards` method creates each credit-card object individually and then stores the object in the array of credit-card objects. Normally, the credit-card information would be read from a file but by hard-coding the information, we get the program working more quickly. Once the complete program has been developed, we can always modify this method to read the information from a file.

```
public static CreditCard[] createCreditCards()
{
     CreditCard[] creditCards;
     int count;

     creditCards = new CreditCard[3];
     count = 0;

     creditCards[count++] = new CreditCard("100", "Fred", 5000.00, 10000.00);
     creditCards[count++] = new CreditCard("200", "Mary", 5000.00, 10000.00);
     creditCards[count++] = new CreditCard("300", "Bono", 5000.00, 10000.00);

     return creditCards;
}
```

Now that the objects have been created, we should print the objects to ensure that they are defined correctly. To do this, we will first add a `toString` method to the credit-card class that returns the information about that object.

```
public class CreditCard
{
     String accountNumber;
     String accountName;
     double currentBalance;
     double creditLimit;

     public CreditCard(String number, String name, double balance, double limit)
     {
        accountNumber = number;
        accountName = name;
        currentBalance = balance;
        creditLimit = limit;
     }

     public String toString()
     {
        return accountNumber +" " +accountName +" " +currentBalance
                             +" " +creditLimit;
     }
}
```

Now we can add a `printCreditCards` method to the main class.

```
public static void printCreditCards(CreditCard[] creditCards)
{
     int currentElement;

     for (currentElement=0; currentElement<creditCards.length; currentElement++)
     {
        System.out.println(creditCards[currentElement].toString());
     }
}
```

We can invoke this method by adding the following statement to the end of the main method:

```
printCreditCards(creditCards);
```

Now we have a collection of credit cards and we can print the contents of the collection.

Next we want to be able to process transactions against the credit cards (make purchases, make payments, etc.) We will keep this method quite simple – again hard-coding the transactions to avoid having to complicate the program at this time by adding file processing.

```
public static void processTransactions(CreditCard[] creditCards)
{
     processPurchase(creditCards, "100", 50.00);
}
```

The `processPurchase` method accepts a collection of credit cards, the account number of a specific credit card, and the amount of the transaction (which is to be added to the balance of the corresponding credit card).

```
public static void processPurchase(CreditCard[] creditCards, String accountNumber,
                                                                     double amount)
{
     CreditCard currentCreditCard;
     double balance;
     int found;

     found = locateCreditCard(creditCards, accountNumber);
     if (found != -1)
     {
        currentCreditCard = creditCards[found];
        balance = currentCreditCard.getBalance();
        currentCreditCard.setBalance(balance+amount);
     }
     else
     {
        System.out.println("CreditCard " +accountNumber +" could not be found.");
     }
}
```

Before this method can perform any processing, it must locate the credit card that corresponds to the account number parameter. Since this processing will likely be used in other methods, it is defined in its own method.

```
public static int locateCreditCard(CreditCard[] creditCards, String accountNumber)
{
     CreditCard currentCreditCard;
     int found;
     int currentElement;

     found = -1;
     for (currentElement=0; currentElement<creditCards.length
                                                   && found==-1; currentElement++)
     {
        currentCreditCard = creditCards[currentElement];
        if (currentCreditCard.getAccountNumber().equals(accountNumber))
        {
           found = currentElement;
        }
     }
     return found;
}
```

Note that both the `locateCreditCard` method and the `processPurchase` method require access to information that is in a credit-card object. Therefore, we must add the necessary accessors and mutator to the creditCard class. (This should set off a warning bell; however, we will take a simple approach first and then improve the code.)

```
public String getAccountNumber()
{
     return accountNumber;
}

public double getBalance()
{
     return currentBalance;
}

public void setBalance(double newBalance)
{
     currentBalance = newBalance;
}
```

Now that we have the general infrastructure defined, we can add a `processPayment` method easily since it is identical to `processPurchase` except for the sign change in the calculation of the new balance.

```
public static void processPayment(CreditCard[] creditCards, String accountNumber,
                                                                   double amount)
{
     CreditCard currentCreditCard;
     double balance;
     int found;

     found = locateCreditCard(creditCards, accountNumber);
     if (found != -1)
     {
         currentCreditCard = creditCards[found];
         balance = currentCreditCard.getBalance();
         currentCreditCard.setBalance(balance-amount);
     }
     else
     {
         System.out.println("CreditCard " +accountNumber +" could not be found.");
     }
}
```

In general, when two methods (such as `processPurchase` and `processPayment`) are almost identical, we should consider amalgamating the two methods. However, in this particular example, purchases and payments are not the same and at some point, it may be necessary to define additional statements that are specific to either purchases or payments but not to both. So, while eliminating duplicate code is a good idea, it is not always a good idea to amalgamate two methods that are similar but have different purposes.

Notice that even though the credit limit for each customer is defined in the object, this information is ignored by the `processPurchase` method. The following modification to the method incorporates a credit-card limit check before processing a transaction.

```
public static void processPurchase(CreditCard[] creditCards, String accountNumber,
double amount)
{
     CreditCard currentCreditCard;
     double balance;
     double limit;
     int found;

     found = locateCreditCard(creditCards, accountNumber);
     if (found != -1)
     {
         currentCreditCard = creditCards[found];
         balance = currentCreditCard.getBalance();
         limit = currentCreditCard.getCreditLimit();
         if ((balance+amount) <= limit)
         {
             currentCreditCard.setBalance(balance+amount);
         }
         else
         {
             System.out.println("Transaction rejected -- customer "
                     +accountNumber +" would be over credit limit of " +limit);
         }
     }
     else
     {
        System.out.println("CreditCard " +accountNumber +" could not be found.");
     }
}
```

The `processPurchase` method now verifies each transaction. Note that this method requires that an additional accessor be added to the CreditCard class – `getCreditLimit`.

8.3 Remove Accessors and Mutators

In the previous section, when the transaction-processing methods in the main class required information that was stored in an object, they used accessors or mutators to access or modify the information. **As a general rule, you should avoid retrieving information from an object and performing processing outside of the object.** Instead, the object should perform the processing itself.

For example, in the previous section the method `processPayment` retrieved the current balance and then replaced the current balance with the newly calculated balance. While the processing is correct, the code is an example of performing processing outside of an object that should be performed inside the object.

```
balance = currentCreditCard.getBalance();
currentCreditCard.setBalance(balance-amount);
```

Instead of using `getBalance` and `setBalance`, there should be a method inside the object that accepts an amount parameter and subtracts the amount from the current balance.

```
currentCreditCard.makePayment(amount);
```

Similarly, in the `processPurchase` method, a purchase made by the customer is processed by retrieving the current balance, retrieving the current credit limit, ensuring that the new balance does not exceed the credit limit, and finally updating the balance in the credit-card object.

```
balance = currentCreditCard.getBalance();
limit = currentCreditCard.getCreditLimit();
if ((balance+amount) <= limit)
{
     currentCreditCard.setBalance(balance+amount);
}
else
{
     System.out.println("Transaction rejected -- customer "
                         +accountNumber +" would be over credit limit of " +limit);
}
```

This processing should be defined in the credit-card object, not in the application program.

```
currentCreditCard.processPurchase(amount);
```

The two methods `makePayment` and `processPurchase` now replace the 3 accessors/mutators.

```
public void makePayment(double amount)
{
     currentBalance -= amount;
}
```

```
public void processPurchase(double amount)
{
     if ((currentBalance+amount) <= creditLimit)
     {
        currentBalance += amount;
     }
     else
     {
        System.out.println("Transaction rejected -- customer "
                +accountNumber +" would be over credit limit of " +creditLimit);
     }
}
```

Now the processing is localized in the object instead of in the application program. This is particularly important if there are many application programs that perform similar processing.

Similarly, when the `locateCreditCard` method must compare credit-card account numbers, instead of retrieving the account number from the object and then performing the comparison in the application program, it would be better to have the object perform the comparison itself. So the statement:

```
if (currentCreditCard.getAccountNumber().equals(accountNumber))
```

would be replaced by the statement:

```
if (currentCreditCard.compareAccountNumbers(accountNumber))
```

This change involves replacing the accessor in the object with the following method:

```
public boolean compareAccountNumbers(String number)
{
     return (accountNumber.equals(number));
}
```

Again, the processing is now localized in the object instead of in the application program.

Any time that you find yourself writing an accessor or mutator, ask why processing is being performed outside of the object instead of inside the object? The answer will normally be that there is no good reason and so instead of using an accessor or mutator, define the processing instructions inside the object.

8.4 Move Object Processing into its own Class

The methods in the previous section create the credit-card objects and then illustrate how the credit cards can be manipulated in various ways. With a large system, it is likely that many different programs will have to perform the same manipulations on the collection of credit cards. As a result, the same methods would have to be included in each program. Then, if any of the methods have to be updated, each program that includes the methods would have to be modified. This is obviously not a good programming practice.

Instead, it would make sense to move the creation and manipulation of the credit cards into its own class. By doing so, we isolate all of the credit-card processing in one class, making it easy to locate and/or modify any of the methods that manipulate the credit cards. This is an example of encapsulation.

The following class illustrates this processing.

```
public class CreditCardProcessing
{
     CreditCard[] creditCards;

     public CreditCardProcessing()
     {
        createCreditCards();
     }
     public void createCreditCards()
     {
        int count;

        creditCards = new CreditCard[3];
        count = 0;

        creditCards[count++] = new CreditCard("100", "Fred", 5000.00, 10000.00);
        creditCards[count++] = new CreditCard("200", "Mary", 5000.00, 10000.00);
        creditCards[count++] = new CreditCard("300", "Bono", 5000.00, 10000.00);
     }
```

```
     public void processPurchase(String accountNumber, double amount)
     {
        CreditCard currentCreditCard;
        double balance;
        double limit;
        int found;

        found = locateCreditCard(accountNumber);
        if (found != -1)
        {
           currentCreditCard = creditCards[found];
           currentCreditCard.processPurchase(amount);
        }
        else
        {
           System.out.println("CreditCard " +accountNumber
                                            +" could not be found.");
        }
     }
```

```
    public void processPayment(String accountNumber, double amount)
    {
       CreditCard currentCreditCard;
       double balance;
       int found;

       found = locateCreditCard(accountNumber);
       if (found != -1)
       {
          currentCreditCard = creditCards[found];
          currentCreditCard.makePayment(amount);
       }
       else
       {
          System.out.println("CreditCard " +accountNumber
                                            +" could not be found.");
       }
    }
```

```
    public int locateCreditCard(String accountNumber)
    {
       CreditCard currentCreditCard;
       int found;
       int currentElement;

       found = -1;
       for (currentElement=0; currentElement<creditCards.length
                                    && found==-1; currentElement++)
       {
          currentCreditCard = creditCards[currentElement];
          if (currentCreditCard.compareAccountNumbers(accountNumber))
          {
             found = currentElement;
          }
       }
       return found;
    }
```

```
    public void print()
    {
       int currentElement;

       System.out.println("\n" +"Credit Cards");

       for (currentElement=0; currentElement<creditCards.length; currentElement++)
       {
          System.out.println(creditCards[currentElement]);
       }
       System.out.println();
    }
}
```

The credit-card objects are now hidden within the creditCardProcessing class and the main class that manipulates the credit cards is just a few simple lines of code instead of about 2 pages of code in the previous version.

```
public static void main(String[] parms)
{
     CreditCardProcessing creditCards;

     creditCards = new CreditCardProcessing();
     creditCards.print();
     processTransactions(creditCards);
     creditCards.print();
}
```

```
public static void processTransactions(CreditCardProcessing creditCards)
{
     creditCards.processPurchase("100", 100.00);
     creditCards.processPayment("200", 1000.00);
     creditCards.processPurchase("300", 6000.00);
}
```

Now writing programs that manipulate the objects is significantly easier because the programmer just sends messages to the `CreditCardProcessing` class. Plus, maintaining and extending the methods that process the objects is also easier because the instructions occur in only one place – the `CreditCardProcessing` class. Note that these changes did not require any changes to the `CreditCard` class itself.

8.5 Using File Input

Until now, the program has hard-coded the credit-card data. Since the program is almost complete, we should replace the hard-coded data with the statements that read the data from a file. The original `createCreditCards` method is replaced with the method:

```
public void createCreditCards()
{
     BufferedReader fileIn;
     CreditCard[] newCreditCards;
     String[] strings;
     String inputLine;
     int count;

     count = 0;
     newCreditCards = new CreditCard[100];
     try
     {
        fileIn = new BufferedReader(new FileReader("CreditCards.txt"));

        inputLine = fileIn.readLine();
        while (inputLine != null)
        {
           strings = inputLine.split("\\s+");
           newCreditCards[count] = new CreditCard(strings[0],strings[1],
                Double.parseDouble(strings[2]),Double.parseDouble(strings[3]));
           count++;
           inputLine = fileIn.readLine();
        }
        fileIn.close();
     }
     catch (NumberFormatException ex)
     {  // catch parseDouble errors
        System.out.println("Invalid input value: " +inputLine);
```

```
         // do something to fix the error
      }
      catch (IOException ioe)
      {
         ioe.printStackTrace();
      }

      if (count > 0)
      {
         creditCards = new CreditCard[count];
         System.arraycopy(newCreditCards,0,creditCards,0,count);
      }
      else
      {
         creditCards = new CreditCard[0];
      }
}
```

8.6 Using a Factory Method

The method in the previous section that created the credit cards was responsible for two separate tasks – reading the input file and also parsing (tokenizing) each input line in order to create a new credit card. While this method is not particularly complicated, it could be split into two methods that simplify the program structure slightly. As is shown below, the first method is responsible for reading the lines of input. This method then calls a static method named `create` in the CreditCard class. The create method parses the input string and performs any data validation that is required. If the parameters are valid, a new object is created using the class constructor. If any of the parameters are not valid, a null value is returned. If the value returned is not null, the calling method then stores the new credit card object in the array of credit cards. Moving this processing inside the CreditCard class is an example of good design since it centralizes some of the data validation within the class. (In design pattern terminology, the use of a static method to construct an object is referred to as a factory pattern or factory method. This is the technique used in the JOptionPane class to construct dialog boxes.)

```
try
{
      fileIn = new BufferedReader(new FileReader("CreditCards.txt"));

      inputLine = fileIn.readLine();
      while (inputLine != null)
      {
         newCreditCard = CreditCard.create(inputLine);
         if (newCreditCard != null)
         {
            newCreditCards[count] = newCreditCard;
            count++;
         }
         inputLine = fileIn.readLine();
      }
      fileIn.close();
}
catch (IOException ioe)
{
      ioe.printStackTrace();
}
```

```
public class CreditCard
{
     String accountNumber;
     String accountName;
     double currentBalance;
     double creditLimit;

     private CreditCard(String number, String name, double balance, double limit)
     {
        accountNumber = number;
        accountName = name;
        currentBalance = balance;
        creditLimit = limit;
     }

     public static CreditCard create(String inputLine)
     {
        CreditCard newCreditCard;
        String[] strings;

        strings = inputLine.split("\\s+");
        try
        {
           newCreditCard = new CreditCard(strings[0],strings[1],
                   Double.parseDouble(strings[2]),Double.parseDouble(strings[3]));
        }
        catch (NumberFormatException ex)
        {  // catch parseDouble errors
           System.out.println("Invalid input value: " +inputLine);
           newCreditCard = null;
        }
        return newCreditCard;
     }
. . .
```

8.7 Complete Credit-Card Example

Now, with the exception of the transactions which are still hard-coded in the main class, the program is complete.

```
public class CreditCardExample
{
     public static void main(String[] parms)
     {
        CreditCardProcessing creditCards;

        creditCards = new CreditCardProcessing();
        creditCards.print();
        processTransactions(creditCards);
        creditCards.print();
     }

     public static void processTransactions(CreditCardProcessing creditCards)
     {
        creditCards.processPurchase("100", 100.00);
        creditCards.processPayment("200", 1000.00);
        creditCards.processPurchase("300", 6000.00);
     }
}
```

```
class CreditCardProcessing
{
      CreditCard[] creditCards;

      public CreditCardProcessing()
      {
          createCreditCards();
      }

      public void createCreditCards()
      {
          BufferedReader fileIn;
          CreditCard[] newCreditCards;
          CreditCard newCreditCard;
          String[] strings;
          String inputLine;
          int count;

          count = 0;
          inputLine = null;
          newCreditCards = new CreditCard[100];
          try
          {
              fileIn = new BufferedReader(new FileReader("CreditCards2.txt"));

              inputLine = fileIn.readLine();
              while (inputLine != null)
              {
                  newCreditCard = CreditCard.create(inputLine);
                  if (newCreditCard != null)
                  {
                      newCreditCards[count] = newCreditCard;
                      count++;
                  }
                  inputLine = fileIn.readLine();
              }
              fileIn.close();
          }
          catch (IOException ioe)
          {
              ioe.printStackTrace();
          }

          if (count > 0)
          {
              creditCards = new CreditCard[count];
              System.arraycopy(newCreditCards,0,creditCards,0,count);
          }
          else
          {
              creditCards = new CreditCard[0];
          }
      }

      public void processPurchase(String accountNumber, double amount)
      {
          CreditCard currentCreditCard;
          double balance;
          double limit;
          int found;

          currentCreditCard = locateCreditCard(accountNumber);
          if (currentCreditCard != null)
          {
              currentCreditCard.makePurchase(amount);
          }
          else
          {
              System.out.println("CreditCard " +accountNumber
                                              +" could not be found.");
          }
      }

      public void processPayment(String accountNumber, double amount)
      {
          CreditCard currentCreditCard;
          double balance;
          int found;

          currentCreditCard = locateCreditCard(accountNumber);
          if (currentCreditCard != null)
          {
              currentCreditCard.makePayment(amount);
          }
          else
          {
              System.out.println("CreditCard " +accountNumber
                                                 +" could not be found.");
          }
      }

      private CreditCard locateCreditCard(String accountNumber)
      {
          CreditCard currentCreditCard;
          CreditCard foundCreditCard;
          int currentElement;

          foundCreditCard = null;
          for (currentElement=0; currentElement<creditCards.length
                                               && foundCreditCard==null; currentElement++)
          {
              currentCreditCard = creditCards[currentElement];
              if (currentCreditCard.compareAccountNumbers(accountNumber))
              {
                  foundCreditCard = currentCreditCard;
              }
          }
          return foundCreditCard;
      }

      public void print()
      {
          int currentElement;

          for (currentElement=0; currentElement<creditCards.length; currentElement++)
          {
              System.out.println(creditCards[currentElement].toString());
          }
          System.out.println();
      }
}

class CreditCard
{
      String accountNumber;
      String accountName;
      double currentBalance;
      double creditLimit;

      private CreditCard(String number, String name, double balance, double limit)
      {
          accountNumber = number;
          accountName = name;
          currentBalance = balance;
          creditLimit = limit;
      }

      public static CreditCard create(String inputLine)
      {
          CreditCard newCreditCard;
          String[] strings;
```

```
         strings = inputLine.split("\\s+");
         try
         {
            newCreditCard = new CreditCard(strings[0],strings[1],
                       Double.parseDouble(strings[2]),Double.parseDouble(strings[3]));
         }
         catch (NumberFormatException ex)
         {   // catch parseDouble errors
            System.out.println("Invalid input value: " +inputLine);
            newCreditCard = null;
         }
         return newCreditCard;
      }

      public boolean compareAccountNumbers(String number)
      {
         return (accountNumber.equals(number));
      }

      public void makePayment(double amount)
      {
         currentBalance -= amount;
      }

      public void makePurchase(double amount)
      {
         if ((currentBalance+amount) <= creditLimit)
         {
            currentBalance += amount;
         }
         else
         {
            System.out.println("Transaction rejected -- customer "
                    +accountNumber +" would be over credit limit of " +creditLimit);
         }
      }

      public String toString()
      {
         return accountNumber +" " +accountName
                              +" " +currentBalance +" " +creditLimit;
      }
}
```

8.8 Summary

By creating the `CreditCardProcessing` class, we have isolated all methods that create and modify the credit cards in the one class: this process is referred to as **encapsulation**. Also, we have hidden the implementation details about how the credit card objects are stored inside the CreditCardProcessing class: this is referred to as **information hiding**. If we decide at a later time that using an array to store the credit card objects is not the best technique, only the `CreditCardProcessing` class must be modified; the programs that require access to the collection of credit cards need not be aware of the change to how credit cards are stored. Encapsulation and information hiding are two of the most significant advantages of using an object-oriented approach to programming. (Encapsulation and information hiding are often used synonymously because they are closely related but there is a subtle difference between them. For example, all of the credit-card collection information may be encapsulated in the `CreditCardProcessing` class but if the variables in `CreditCardProcessing` are public, the information is not hidden.)

By following practices such as "try to avoid using accessors and mutators" and "maintain collections of objects in a separate class", the result is code that is simpler to write and easier to maintain since processing is isolated in the appropriate class.

9 OBJECT COLLECTIONS

9.1 Introduction

The title of these notes is “Growing Algorithms and Data Structures”. We have spent the first half of the notes growing algorithms in an object-oriented context; now we begin to examine data structures.

A **data structure** is a programming construct used to store a collection of elements (or data types). An array is one of the simplest data structures – the elements are stored in the array in a linear manner (the first element is in the first position, the second element is physically adjacent to the first element, the third element is physically adjacent to the second element, etc.).

In the preceding chapters, we used arrays to store collections of elements. For example, the following array is a data structure capable of storing 10 Flight objects:

```
Flight[] myFlights = new Flight[10];
```

There are however, several difficulties that are encounted when using arrays if the contents of the collection change on a frequent basis:

- new elements can not easily be inserted between existing elements in an array;
- existing elements can not easily be deleted from an array;
- the position of an element in an array can not easily be changed;
- the size of an array is fixed and if the array contains less than the maximum number of elements, the programmer must keep track of the number of elements that are actually being used in the array;
- the size of an array is fixed and if it must be increased, a new array must be created and the contents of the old array must be copied into the new array.

These problems were encounted in the programs in Chapter 2 – Growing Algorithms. While it is possible to write additional methods that solve these problems with arrays, this is not a convenient solution since the methods are specific to each array that is being used and also because each program that requires these methods must include the additional methods.

9.2 ArrayLists

Java provides a more sophisticated mechanism for organizing a collection of elements, the **ArrayList**. An ArrayList is a class that maintains a collection of elements that may be of different types. For example, both Integers and Strings may be stored in the same ArrayList. The elements of an ArrayList may also be inserted, deleted, and moved without requiring any additional programming. ArrayLists are also automatically resized as required. One significant difference between arrays and ArrayLists is that only objects can be stored in an ArrayList (which means that primitive data types can not be stored directly in an ArrayList).

ArrayLists are not part of the basic Java framework so any program that uses ArrayLists must import the definition of the ArrayList class:

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

The following statements declare and then instantiate a new ArrayList.

```
ArrayList myArrayList;
myArrayList = new ArrayList();
```

At this time, myArrayList does not contain any elements. Subsequently, if the following statement is executed

```
myArrayList.add("string1");
```

"`string1`" is stored at location 0 of myArrayList. If the following statement is then executed:

```
myArrayList.add("string2");
```

"`string2`" is added to myArrayList in location 1.

If we now add "`string3`" in position 0,

```
myArrayList.add(0,"string3");
```

the 3 elements in myArrayList are:

```
"string3", "string1", "string2"
```

Note that the add instruction caused the elements `string1` and `string2` to be moved one position to the right.

If we subsequently delete the object at position 1

```
myArrayList.remove(1);
```

myArrayList contains the elements:

```
"string3", "string2"
```

So the remove instruction caused the element `string2` to be moved one position to the left.

If we need to locate a specific object in the collection, we can use the indexOf method.

```
myArrayList.indexOf("string2");
```

The statement above returns the location of `"string2"`.

```
1
```

If no identical object is located, indexOf returns a value of -1.

When performing the comparison of objects, the ArrayList class uses the `equals` method. Normally, `equals` should be one of the methods that is included with any class for which comparisons will be performed. For example, the String class contains an `equals` method. If an `equals` method is not included in a class, the comparison process compares object references, not object values. (More on this in Chapter 10 – Object Hierarchies.)

Unfortunately, the ArrayList class does not provide a method for searching for an object beginning at a location other the first location (recall that the String class does provide such a method: `string.indexOf(string, startPos)`). If this facility is required, the programmer would have to write a method that would be added to the application program, not to the ArrayList class.

Unlike arrays, as each element is added to an ArrayList object, the size of the object changes (or appears to change). The `size()` **method** can be used to determine the number of objects currently in the ArrayList object.

```
System.out.println(myArrayList.size());

2
```

There is no limitation on the size of an ArrayList – any number of elements may be added to an ArrayList.

9.3 Compiling ArrayLists

When a program that uses an ArrayList is compiled, Java generates various warning messages:

```
ArrayLists.java:23: warning: [unchecked] unchecked call to add(int,E) as a member of the raw
type java.util.ArrayList
          list.add(3, "String3");
                   ^
```

These warnings can be ignored – in TextPad, just click on the source code window and use the Run command. However, we will examine a technique ("generics") at the end of this chapter that can be used to remove the warning messages.

If you are desperate to remove the warning messages, adding the following statement at the beginning of each method that generates a warning suppress the warnings.

```
@SuppressWarnings("unchecked")
public static void process(ArrayList list)
...
```

9.4 ArrayList Methods

ArrayLists are manipulated using a collection of methods that are provided with the ArrayList class. The following are the basic ArrayList methods:

`new ArrayList()`	create a new ArrayList object; initialize the object so that it contains zero elements
`get(int)`	retrieve the object in position int (relative to zero)
`add(object)`	add the object to the end of the ArrayList
`add(int,object)`	insert the object at position int; the element at position int and all subsequent elements are moved one position to the right
`indexOf(object)`	search for the first occurrence of object and return the position of the object in the ArrayList; uses the object's `equals()` method to test for equality; returns -1 if not found
`lastIndexOf(object)`	search for the last occurrence of object and return the position of the object in the ArrayList; uses the object's `equals()` method to test for equality; returns -1 if not found
`contains(object)`	determine whether or not `object` exists in the collection; returns true or false; uses the object's `equals()` method to test for equality
`remove(int)`	remove the object at position int
`remove(object)`	remove the first occurrence of the object parameter
`set(int,object)`	replace the object that is currently in position int with the object parameter
`size()`	return the number of objects currently in the ArrayList
`clear()`	remove all of the objects that are currently in the ArrayList
`toString()`	return a String that contains the contents of the ArrayList

Other ArrayList methods are defined in the Java documentation:

http://download.oracle.com/javase/6/docs/api/java/util/ArrayList.html

9.5 Casting

The method `get(int)` can be used to extract a value from the specified location (int) in an ArrayList. Since the elements stored in an ArrayList do not have to be of the same type, it is the responsibility of the programmer to "**cast**" any element extracted from an ArrayList to the correct type. For example, to extract a String that is stored in location 0 of myArrayList, the following statement is used:

```
String string = (String) myArrayList.get(0);
```

As was mentioned earlier, the values stored in an ArrayList must be objects. As a result, the primitive data types (int, boolean, etc.) can not be stored directly in an ArrayList. Instead, the primitive value must first be converted to the corresponding object using a wrapper class. For example, to store the int value 10 at the beginning of an ArrayList, the following statement is used:

```
myArrayList.add(0, new Integer(10));
```

To extract this value for processing as an int, the following statement is used:

```
int myInt = ((Integer) myArrayList.get(0)).intValue();
```

Note the position of the parentheses – the element must be cast to an Integer before the intValue method can be applied. (Java's boxing and unboxing mechanisms reduce some of the clutter in this statement; we will examine boxing and unboxing later in this chapter.)

Having to cast the value in an ArrayList to the correct type is more complex than the equivalent array processing but it is necessary since ArrayLists can contain any type of object.

The ArrayList class does not contain a method that moves an element from one location to another location but this operation can be carried out using three of the basic ArrayList methods. The following instructions illustrate the processing required to move an element from location 3 to location 0: first, the string at location 3 is extracted; then, the string at location 3 is deleted; finally, the string is inserted into location 0.

```
String string = (String) myArrayList.get(3);
myArrayList.remove(3);
myArrayList.add(0,string);
```

Will the same instructions work correctly if we want to move an element from location 0 to location 3?

```
String string = (String) myArrayList.get(0);     ??
myArrayList.remove(0);                           ??
myArrayList.add(3,string);                       ??
```

9.6 Reversing the Elements in an ArrayList

The following method reverses the elements in an ArrayList (the last element becomes the first element, the first element becomes the last element, etc.). This method is included to illustrate an algorithm (even though the algorithm does not have a lot of practical application).

```
public static void reverseList(ArrayList list)
{
     ArrayList newList;
     int count;

     newList = new ArrayList();
     for (count=list.size()-1; count>=0; count--)
     {
        newList.add(list.get(count));
     }
     for (count=0; count<list.size(); count++)
     {
        list.set(count,newList.get(count));
     }
}
```

The method above makes copies the elements into a temporary ArrayList and then copies them back into the original ArrayList.

The following method is an improvement of the previous method. The method below makes the changes to the ArrayList without using a temporary ArrayList.

```
public static void reverseList(ArrayList list)
{
     int count;
     for (count=1; count<list.size(); count++)
     {
        list.add(0,list.get(count));
        list.remove(count+1);
     }
}
```

Finally, the method below is the simplest of the 3 versions and also the easiest to understand.

```
public static void reverseList(ArrayList list)
{
     int count;

     for (count=list.size()-2; count>=0; count--)
     {
        list.add(list.get(count));
        list.remove(count);
     }
}
```

9.7 Searching an ArrayList

In Chapter 2 – Growing Algorithms, we examined various search strategies that searched an array containing a collection of integer values. The following program performs the same tasks using an ArrayList of Integer's instead of using an array.

```
public static void main(String[] parms)
{
     ArrayList list;
     int searchValue = 10;
     int result;

     list = createList();
     System.out.println(list);
     result = searchList(list, searchValue);
     System.out.println("Result is: " +result);
}

public static ArrayList createList()
{
     ArrayList list;
     int count;
     final int NUM_ELEMENTS = 10;

     list = new ArrayList();
     for (count=0; count<NUM_ELEMENTS; count++)
     {
        list.add(new Integer(count));
     }
     return list;
}

public static int searchList(ArrayList list, int searchValue)
{
     int count;
     int result;

     result = -1;
     for (count=0; count<list.size() && result == -1; count++)
     {
        if (((Integer) list.get(count)).intValue() == searchValue)
        {
           result = count;
        }
     }
     return result;
}
Result is: 9
```

The following program illustrates the same processing with an ArrayList of Strings.

```
public static void main(String[] parms)
{
     ArrayList list;
     String searchValue = "string3";
     int result;

     list = createList();
     System.out.println(list);
     result = searchList(list, searchValue);
     System.out.println("Result is: " +result);
}

public static ArrayList createList()
{
     ArrayList list;
     int count;
     int numElements = 5;
     String value;
     list = new ArrayList();
     for (count=0; count<numElements; count++)
     {
        value = "string" +count;
        list.add(value);
     }
     return list;
}

public static int searchList(ArrayList list, String searchValue)
{
     int count;
     int result;
     result = -1;
     for (count=0; count<list.size() && result == -1; count++)
     {
        if (((String) list.get(count)).equals(searchValue))
        {
           result = count;
        }
     }
     return result;
}
Result is: 2
```

Remember that if you are searching an ArrayList of programmer-defined objects, an `equals` method must be defined in the corresponding class and the search parameter must be an object of the corresponding type.

```
public static int searchList(ArrayList list, Flight searchFlight)
{
     int count;
     int result;
     result = -1;
     for (count=0; count<list.size() && result == -1; count++)
     {
        if (((Flight) list.get(count)).equals(searchFlight))
        {
           result = count;
        }
     }
     return result;
}
```

This method can be invoked with a statement similar to the following (assuming that flights are considered equal if they have the same flight number):

```
result = searchList(list, new Flight(100));
```

The ArrayList class also includes a built-in search method (indexOf). The parameter used with indexOf must be an object of the same type as the object being searched for in the ArrayList. Thus, if an ArrayList contains a collection of String objects, the following statement could be used to determine whether or not the String `"string3"` occurs in the ArrayList.

```
result = list.indexOf("string3");
```

If an ArrayList contains programmer-defined objects, then a valid object must be used as the search parameter. For example, the following statement searches a list of Flight objects for a Flight with a flight number of `100`.

```
result = list.indexOf(new Flight(100));
```

9.8 Printing the Contents of an ArrayList

A simple loop can be used to print the contents of an ArrayList.

```
public static void printList(ArrayList list)
{
     int count;
     for (count=0; count<list.size(); count++)
     {
        System.out.print(list.get(count) +" ");
     }
     System.out.println();
}
string1 string2 string3 string4 string5
```

However, unlike arrays, an ArrayList object contains a toString method that provides a very convenient mechanism for printing the contents of any ArrayList on one line. Each object in the ArrayList is printed (using the object's own toString method); the information generated for objects is separated by a commas and the entire list is surrounded by square brackets.

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

[string1, string2, string3, string4, string5]
```

9.9 Rotating the Contents of an ArrayList

We saw in Chapter 2 – Growing Algorithms how the contents of an array could be rotated to the left. In this section, we examine how the same processing can be implemented with an

ArrayList.

```
public static void rotate(ArrayList list, int n)
{
     int count;

     for (count=0; (count<n) && (count<list.size()); count++)
     {
         list.add(list.get(0));  // copy first element to end of the collection
         list.remove(0);         // delete the first element
     }
}
```

9.10 Inserting into an ArrayList in Ascending Order

With the ability to insert an element into an ArrayList in any location, we can insert elements into an ArrayList so that the elements are kept sorted in ascending order. The following method takes each element in the int array intList and then searches the existing elements in the ArrayList until an element that is greater than the element to be inserted is found. The new element is then inserted at that position, causing the larger elements to be moved one position to the right in the ArrayList object.

```
public static ArrayList createList3(int[] intList)
{
     ArrayList list;
     int count;
     int insert;
     int value;

     list = new ArrayList();
     for (count=0; count<intList.length; count++)
     {
         value = intList[count];
         for (insert=0; insert<list.size()
                     && value>((Integer)list.get(insert)).intValue(); insert++)
         {
         }
         list.add(insert, new Integer(value));
     }
     return list;
}
```

9.11 Searching an ArrayList of Integers with a Binary Search

The method in the previous section inserts values into an ArrayList so that the values are kept in ascending order. If we are guaranteed that the elements in an ArrayList are sorted, we can improve the search for an element by using a binary search. This method is almost identical to the binary search of an array that was defined in Chapter 2 – Growing Algorithms.

```
public static int searchList(ArrayList list, int searchValue)
{
      int left;
      int right;
      int middle;
      int result;
      int middleElement;

      result = -1;
      left = 0;
      right = list.size()-1;
      while ((left <= right) && (result == -1))
      {
          middle = left + ((right - left) / 2);
          middleElement = ((Integer) list.get(middle)).intValue();
          if (middleElement == searchValue)
          {
              result = middle;
          }
          else if (middleElement < searchValue)
          {
              left = middle + 1;
          }
          else if (middleElement > searchValue)
          {
              right = middle - 1;
          }
      }
      return result;
}
```

9.12 Searching an ArrayList of Strings with a Binary Search

The following method illustrates the equivalent search of an ArrayList of Strings.

```
public static int binarySearchV(ArrayList list, String searchValue)
{
      int left;
      int right;
      int middle;
      int result;
      int comparison;
      String middleElement;
      result = -1;
      left = 0;
      right = list.size()-1;
      while ((left <= right) && (result == -1))
      {
          middle = left + ((right - left) / 2);
          middleElement = ((String) list.get(middle));
          comparison = middleElement.compareTo(searchValue);
          if (comparison == 0)
          {
              result = middle;
          }
          else if (comparison < 0)
          {
              left = middle + 1;
          }
          else if (comparison > 0)
          {
              right = middle - 1;
          }
      }
      return result;
}
```

9.13 Credit-Cards Example

The credit-cards example that was discussed in the previous chapter is now rewritten to use an ArrayList instead of an array. It is important to notice that only the CreditCardProcessing class must be modified – the other classes do not need to be modified. Also, the number of modifications is relatively small – this is an indication that the final design satisfies good object-orientation practices.

```
class CreditCardProcessing
{
      ArrayList creditCards;

      public CreditCardProcessing()
      {
          createCreditCards();
      }

      @SuppressWarnings("unchecked")
      public void createCreditCards()
      {
          BufferedReader fileIn;
          CreditCard newCreditCard;
          String[] strings;
          String inputLine;
          int count;

          count = 0;
          creditCards = new ArrayList();
          try
          {
              fileIn = new BufferedReader(new FileReader("CreditCards.txt"));

              inputLine = fileIn.readLine();
              while (inputLine != null)
              {
                  newCreditCard = CreditCard.create(inputLine);
                  if (newCreditCard != null)
                  {
                      creditCards.add(newCreditCard);
                  }
                  inputLine = fileIn.readLine();
              }
              fileIn.close();
          }
          catch (IOException ioe)
          {
              System.out.println(ioe.getMessage());
              ioe.printStackTrace();
          }
      }

      public void processPurchase(String accountNumber, double amount)
      {
          CreditCard currentCreditCard;
          double balance;
          double limit;
          int found;

          currentCreditCard = locateCreditCard(accountNumber);
          if (currentCreditCard != null)
          {
              currentCreditCard.makePurchase(amount);
          }
          else
          {
              System.out.println("CreditCard " +accountNumber
                                               +" could not be found.");
          }
      }

      public void processPayment(String accountNumber, double amount)
      {
          CreditCard currentCreditCard;
          double balance;
          int found;

          currentCreditCard = locateCreditCard(accountNumber);
          if (currentCreditCard != null)
          {
              currentCreditCard.makePayment(amount);
          }
          else
          {
              System.out.println("CreditCard " +accountNumber +" could not be found.");
          }
      }

      private CreditCard locateCreditCard(String accountNumber)
      {
          CreditCard currentCreditCard;
          CreditCard foundCreditCard;
          int currentElement;

          foundCreditCard = null;
          for (currentElement=0; currentElement<creditCards.size()
                                          && foundCreditCard==null; currentElement++)
          {
              currentCreditCard = (CreditCard) creditCards.get(currentElement);
              if (currentCreditCard.compareAccountNumbers(accountNumber))
              {
                  foundCreditCard = currentCreditCard;
              }
          }
          return foundCreditCard;
      }

      public void print()
      {
          int currentElement;

          System.out.println("\n" +"Credit Cards");

          for (currentElement=0; currentElement<creditCards.size(); currentElement++)
          {
              System.out.println(creditCards.get(currentElement));
          }
          System.out.println();
      }
  }
```

The method `locateCreditCard` can be modified to use the ArrayList method `indexOf`.

```
private CreditCard locateCreditCard(String accountNumber)
{
    CreditCard foundCreditCard;
    int found;

    found = creditCards.indexOf(new CreditCard(accountNumber));
    if (found != -1)
    {
        foundCreditCard = (CreditCard) creditCards.get(found);
    }
    else
```

```
     {
        foundCreditCard = null;
     }
     return foundCreditCard;
}
```

Using `indexOf` requires an appropriate `equals` method in the CreditCard class. This method uses the type Object which is discussed in Chapter 10 – Object Hierarchies. Note that using the type `CreditCard` instead of `Object` will not work.

```
public boolean equals(Object creditCard)
{
     return this.accountNumber.equals(((CreditCard)creditCard).accountNumber);
}
```

9.14 Generics

An ArrayList can contain any type of object. As a result, you must downcast the object when it is extracted from an ArrayList. For example, a Student object that is stored in an ArrayList must be downcast after it is extracted from the ArrayList even though the ArrayList may contain only Student objects.

```
ArrayList students;
students = new ArrayList();
Student student = (Student) students.get(0);
```

With Java 5 (or 1.5 depending on which numbering system you use), Java has included the ability to specify the type of object that is stored in a collection. This feature is referred to as "generics" or generic types. With the introduction of generics, the programmer can specify the type of object that will be stored in a collection. Generics are useful in more than collections but collections provide a simple introduction to generics.

When a generic is used, the type of object that is stored in a collection such as an ArrayList is explicitly specified by the programmer at compile time.

```
ArrayList<Student> students;

students = new ArrayList<Student>;

Student student = students.get(0);
```

Once the generic type has been defined, the programmer can extract an object from the collection without having to downcast the object; plus, Java will ensure that only Student objects are stored in the collection. Ensuring that only the specified type of object is stored in the ArrayList is referred to as "type safety" and is an important advance with Java.

If a collection that has been defined with a type is passed as a parameter to a method, the method header must also include the type specification, as shown below.

```
public void sortStudents(ArrayList<Student> students)
{
}
```

If the type is not specified in the method header, the programmer must downcast all references to the ArrayList to the type of object that is stored in the ArrayList.

One final advantage to including generics is that they eliminate the annoying warning messages that Java generates if generics are not used.

```
ArrayLists.java:23: warning: [unchecked] unchecked call to add(int,E) as a member of the raw
type java.util.ArrayList
         list.add(3, "String3");
                 ^
```

Generics will be examined in more detail in subsequent chapters.

9.15 Boxing and Unboxing

Since Java 5 now performs automatic boxing and unboxing of primitive data types, primitive data values can be added to an ArrayList and extracted from an ArrayList without having to explicitly use the wrapper classes. (Remember though that a primitive data value that is stored in an ArrayList is stored in the corresponding wrapper object.)

In the program segment below, the int value is automatically boxed (converted to a Integer wrapper object) when it is added to the ArrayList. Similarly, the object is automatically unboxed when it is extracted from the ArrayList.

```
ArrayList list;
int count;
int sum;

list = new ArrayList();

sum = 0;
for (count=0; count<5; count++)
{
     list.add(count);
     sum += (Integer) list.get(count);
}
System.out.println(sum);

System.out.println(list);

10
[0, 1, 2, 3, 4]
```

Note though that when we extract a value from the ArrayList, the value must be cast to an Integer because Java does not know what type of object is stored in the ArrayList. (The program also generates a warning message because of the same problem.) If we add a generic type to the ArrayList, these problems disappear.

```
ArrayList<Integer> list;
int count;
int sum;

list = new ArrayList<Integer>();

sum = 0;
for (count=0; count<5; count++)
{
     list.add(count);
     sum += list.get(count);
}
System.out.println(sum);

System.out.println(list);

10
[0, 1, 2, 3, 4]
```

9.16 Our Version of ArrayList

The ArrayList class is quite useful since it eliminates many of the problems that are encounted when using a simple array. However, if an ArrayList class were not available, we could write our own class that provides similar (but somewhat limited) functionality. The following class MyArrayList supports the storage and retrieval of Student objects using an array. The class functions in the same manner as the ArrayList but does not provide all of the functionality that an ArrayList provides. It would be fairly easy to add additional functionality to the class so that it provided more of the features that are provided by ArrayList.

```
class MyArrayList
{
     Student[] students;
     int size;

     public MyArrayList()
     {
        students = new Student[100];
        size = 0;
     }

     public void add(Student newStudent)
     {
        students[size] = newStudent;
        size++;
     }

     public Student get(int position)
     {
        return students[position];
     }

     public int indexOf(Student student)
     {
        int count;
        int found;

        found = -1;
        for (count=0; count<size && found==-1; count++)
        {
           if (students[count].equals(student))
           {
              found = count;
           }
        }
        return found;
     }

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

9.17 ArrayList Algorithms

In this section, we examine a few algorithms that perform various manipulations on ArrayLists.

The first method creates an ArrayList that contains a collection Integer objects and String objects. Note that because the ArrayList contains two different types of objects, it is not possible to add a generic type to the ArrayList (this is not entirely true, as will be seen in Chapter 10 – Object Hierarchies).

```
public static ArrayList createList()
{
     ArrayList list;
     int count;

     list = new ArrayList();
     for (count=0; count<10; count++)
     {
        list.add(count);
        list.add(new String("s"+count));
     }
     return list;
}
```

The second method accepts an ArrayList as a parameter and removes all String objects from the ArrayList. In order to identify a String object, we use the `instanceof` comparison operator that was described earlier.

```
public static void removeList(ArrayList list)
{
     int count;

     for (count=0; count<list.size(); )
     {
         if (list.get(count) instanceof String)
         {
            list.remove(count);
         }
         else
         {
            count++;
         }
      }
}
```

Note that the variable count must be modified very carefully (and is not modified in the `for` statement).

The final method is similar to the preceding method but instead of removing each String object from the ArrayList, the method moves the String objects to the end of the ArrayList.

```
public static void reorderList(ArrayList list)
{
      String string;
      int count, position;

      for (position=0, count=0; count<list.size(); count++)
      {
         if (list.get(position) instanceof String)
         {
            string = (String) list.get(position);
            list.remove(position);
            list.add(string);
         }
         else
         {
            position++;
         }
      }
}
```

Note that in this method, two variables are required to maintain two different positions in the ArrayList. (There are other ways of performing the same processing.)

9.18 Processing Course Marks

The following program reads lines of course and student numbers along with the mark that the student received in the corresponding course. For each line, the corresponding object is created (using a factory method). After all objects with the same course number have been read, the average of the marks for the students in that course is calculated. The average mark for each course is determined and printed.

This program puts together many of the constructrs examined in this chapter. It also demonstrates how groups of related input lines can be read and then processed as a collection. The `process` method is not a trivial method and should be examined closely.

In the `process` method, we could have read all of the lines in the input file and stored all of the objects in the array list. That processing would have worked correctly but would have used more space than is required by the version of `process` shown below.

```
public static void process()
{
     Lines lines;
     ArrayList<CourseStudent> list;
     CourseStudent currentCourseStudent;
     CourseStudent newCourseStudent;

     lines = new Lines("Process.txt");
     list = new ArrayList<CourseStudent>();
     currentCourseStudent = CourseStudent.create(lines.getLine());
     // Not necessary to check for null before adding to list
     list.add(currentCourseStudent);
     while (currentCourseStudent!=null)
     {
        newCourseStudent = CourseStudent.create(lines.getLine());
        while ((newCourseStudent!=null) && (currentCourseStudent.equals(newCourseStudent)))
        {
           list.add(newCourseStudent);
           newCourseStudent = CourseStudent.create(lines.getLine());
        }
        calculateAverage(list);
        list.clear();
        list.add(newCourseStudent);
        currentCourseStudent = newCourseStudent;
     }
}

public static void calculateAverage(ArrayList<CourseStudent> list)
{
     double total;
     int count;

     printList(list);
     total = 0.0;
     for (count=0; count<list.size(); count++)
     {
        total += list.get(count).getMark();
     }
     System.out.format("Average mark:%5.1f%% %n%n%n", total/list.size());
}

public static void printList(ArrayList<CourseStudent> list)
{
     int count;

     for (count=0; count<list.size(); count++)
     {
        System.out.println(list.get(count));
     }
     System.out.println();
}

class CourseStudent
{
     private String courseNumber;
     private String studentNumber;
     private double mark;
     public CourseStudent(String courseNumber, String studentNumber, double mark)
     {
        this.courseNumber = courseNumber;
        this.studentNumber = studentNumber;
        this.mark = mark;
     }

     public static CourseStudent create(String inputLine)
     {
        CourseStudent courseStudent;
        String[] split;

        if (inputLine == null)
        {
           courseStudent = null;
        }
        else
        {
           split = (inputLine.trim()).split("\\s+");
           courseStudent=new CourseStudent(split[0],split[1],Double.parseDouble(split[2]));
        }
        return courseStudent;
     }

     public double getMark()
     {
        return mark;
     }

     public boolean equals(CourseStudent object)
     {
        boolean result;

        result = this.courseNumber.equals(object.courseNumber);
        return result;
     }

     public String toString()
     {
        return String.format("%-8s %-6s %-5.1f", courseNumber, studentNumber, mark);
     }
}

class Lines
{
     private BufferedReader fileIn;
     private String inputLine;

     public Lines(String fileName)
     {
        fileOpen(fileName);
     }

     private void fileOpen(String fileName)
     {
        try
        {
           fileIn = new BufferedReader(new FileReader(fileName));
        }
        catch (IOException ioe)
        {
           System.out.println("Error");
```

```
         }
      }

      public String getLine()
      {
         try
         {
            inputLine = fileIn.readLine();
            if (inputLine == null)
            {
               fileIn.close();
            }
         }
         catch (IOException ioe)
         {
            System.out.println("Error");
         }
         return inputLine;
      }
}

COMP1010 001232 80.0
COMP1010 013451 80.0
COMP1010 123453 70.0

Average mark: 76.7%

COMP1020 000224 70.0
COMP1020 000333 90.0
COMP1020 000452 90.0
COMP1020 202021 70.0

Average mark: 80.0%
```

The class Lines was originally defined at the end of Chapter 6 – Object Examples.

The method `process` uses two loops to read and create the CourseStudent objects and then to calculate the average for each course. The `process` method below shows a modification that uses only one loop and an if statement.

```
public static void process()
{
     Lines lines;
     ArrayList<CourseStudent> list;
     CourseStudent currentCourseStudent;
     CourseStudent newCourseStudent;

     lines = new Lines("Process.txt");
     list = new ArrayList<CourseStudent>();
     currentCourseStudent = null;
     newCourseStudent = CourseStudent.create(lines.getLine());
     while (newCourseStudent!=null || list.size()!=0)
     {
        if ((currentCourseStudent==null) ||
            ((newCourseStudent!=null)&&(currentCourseStudent.equals(newCourseStudent))))
        {
            list.add(newCourseStudent);
            currentCourseStudent = newCourseStudent;
            newCourseStudent = CourseStudent.create(lines.getLine());
        }
        else
        {
            calculateAverage(list);
            list.clear();  // empty the array list
            currentCourseStudent = newCourseStudent;
        }
     }
}
```

Worrying about the amount of space required by a program is normally unnecessary but the `process` method illustrates how a loop can continue reading objects until the objects change in some manner (in this case, the course number changes). This is an important algorithm in Computer Science.

9.19 Summary

The ArrayList class makes the storage and manipulation of a collection of objects significantly easier than if an array is used. The power of the ArrayList does not come without a cost – there is more processing going on inside the ArrayList class than there is inside the array class. However, the ArrayList makes the programmer's life easier and that justifies using an ArrayList.

We will examine additional uses of ArrayLists in subsequent chapters. We will also examine another type of data structure in Chapter 15 – Linked Lists.

10 OBJECT HIERARCHIES

10.1 Introduction

In Chapter 3 – Objects, we saw how useful objects can be. As systems become more complex, the number of classes tends to increase. Often, several classes are related to each other and contain common data (variables) and/or behaviour (methods). In this chapter we examine how classes can be grouped together into an object hierarchy to reduce the amount of duplication that can occur when several similar but not identical classes are created.

10.2 Objects

Objects differ significantly from primitive data types. A primitive data type contains only one data value while an object may contain any number of data values plus the methods required to manipulate the data values. Being able to combine both data (variables) and behaviour (methods) enables us to **encapsulate** data and processing into one entity – the object. For example, if we create a class Student, we can create an instance of the class by providing the appropriate information to the class constructor.

```
Student joeCool = new Student("10011010", "Joe Cool", "Computer Science");
```

Now, all of the information that we need to know about Joe is in the `joeCool` object. This object can easily be passed to methods.

```
someMethod(joeCool);
```

Being able to pass all information about Joe as a unit is significantly simpler than using parallel arrays in a non-object-oriented language.

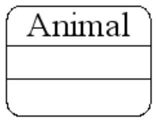
Similarly, any processing required for students is defined inside the object. As a result, programmers can use the Student objects without having to know what information is stored inside the object and how the processing is performed inside the object. Instead, the **interface** (or API – application programming interface) (public methods) to the class is defined and programmers use this interface to interrogate and manipulate the object. Keeping information and processing that are not required by users of an object hidden within the object is referred to as "**information hiding**". In the following example, the method `getGPA()` returns the student's grade-point average but it is not necessary for the user to know how the calculation is carried out. (This is particularly important if the calculation of the student's GPA may vary depending on the student's faculty.)

```
System.out.println(joeCool.getGPA());
```

Similarly, you do not need to know how `System.out` works; all that you need to know is that if you send this object a `println` message, the contents of the parameter are displayed on the system console.

10.3 Animals Example

We will begin our examination of object hierarchies with a simple problem. An animal shelter wants to keep track of animals that currently reside at the shelter. We could simply define an Animal class and include a variable named `type` that contains the type of animal (either dog or cat for now).



However, as we add more information and processing, we will have some processing that is specific to dogs and some that is specific to cats. This leads to the following type of processing:

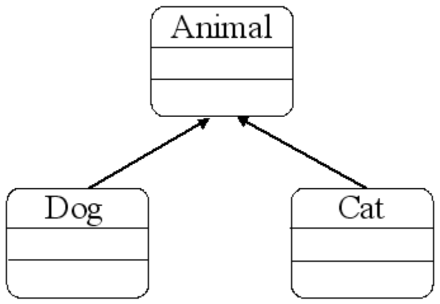
```
if (type.equals("dog")
{
     ... dog processing
}
else
{
     ... cat processing
}
```

If we subsequently need to keep track of other types of animals at the shelter, then the `if` statement becomes even uglier.

Instead of grouping all animals into one class (which is an example of bad object orientation), we could separate the Animal class into individual classes, one per animal.

Now we don't have to check the type of animal when performing the processing but any variables and methods that are common to both classes must be duplicated in each class. Code duplication is not desireable since it often leads to errors when one class is modified but the related classes are not modified. So this is not a good solution either.

Instead, the better solution is to create a simple **hierarchy** that consists of a Dog class, a Cat class, and a more generic Animal class.



With what we have seen of objects so far, there does not appear to be any advantage to this class organization compared with the simple Dog class and Cat class shown earlier. However, the advantage of a class hierarchy is that **all variables and methods defined in the Animal class are also available to the Dog class and to the Cat class.** Information that is common to both dogs and cats (such as `name`) is defined once in the Animal class; information that is specific to dogs is defined once in the Dog class and information that is specific to cats is defined once in the Cat class. We refer to the Animal class as a superclass of both Dog and Cat and we refer to the Dog class and the Cat class as subclasses of the Animal class (more on this in the next section).

The following code defines the 3 classes. To indicate that a class is a subclass of another class, the `extends` parameter is included in the class header. The Cat class and the Dog class headers indicate that Cat and Dog are subclasses of (extend) the Animal class.

```
public class Animal
{
     public String name;
     public String type;

     public String toString()
     {
        return "Animal: Type: " +type +"; Name: " +name;
     }
}
```

```
public class Cat extends Animal
{
     public Cat (String name)
     {
        this.name = name;
        type = "cat";
     }
}
```

```
public class Dog extends Animal
{
     public Dog (String name)
     {
        this.name = name;
        type = "dog";
     }
}
```

In this example, the variable `name` is defined once in the Animal class but may be used in any of the subclasses of Animal (this is referred to as "**inheritance**"). Similarly, the `toString()` method defined in Animal is inherited by all subclasses of Animal. As a result, there is no duplication of variables and methods in these 3 classes.

Cats and dogs are instantiated as though the classes were not related to each other, that is, not organized in a hierarchy. In fact, this is one of the advantages of inheritance – the user does not necessarily know that particular classes are organized in a hierarchy.

```
Cat fluffy;
Dog fido;

fluffy = new Cat("Fluffy");
fido = new Dog("Fido");

System.out.println(fluffy);
System.out.println(fido);
```

If there is information or processing that is specific to one or more of the subclasses, the appropriate processing can be added to that class (or classes). For example, the shelter might want to record whether or not a cat has been declawed. (No cats were actually declawed in the development of this example.) This would be added as a variable only in the Cat class because it has nothing to do with dogs. The following instructions add a declawed variable to the Cat class and by default this variable is set to false. To allow the user to indicate that a cat has been declawed, an appropriate method is also added to the Cat class.

```
public class Cat extends Animal
{
     private boolean declawed;

     public Cat (String name)
     {
        this.name = name;
        type = "cat";
        declawed = false;
     }

     public void setDeclawed()
     {
        declawed = true;
     }
}
```

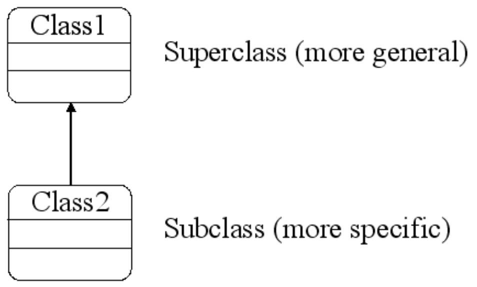
To indicate that `fluffy` has been declawed, we simply send the `setDeclawed()` message to `fluffy`. Since `setDeclawed()` is not defined either in Animal or in Dog, we can not send this message to `fido`.

```
fluffy.setDeclawed();
```

Inheritance provides a mechanism for defining common code in a superclass, code that is inherited by each subclass of the superclass as though the code had been defined directly in each subclass.

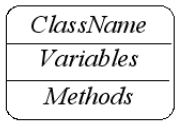
10.4 Class Hierarchies

As indicated in the previous section, classes that are related may be organized into a hierarchy. In the following diagram, Class1 and Class2 are organized in a hierarchy, with Class1 being the superclass and Class2 being a subclass of Class1. The class at the top of the hierarchy is referred to as the "**base class**" and subclasses of the base class are referred to as "**derived classes**".

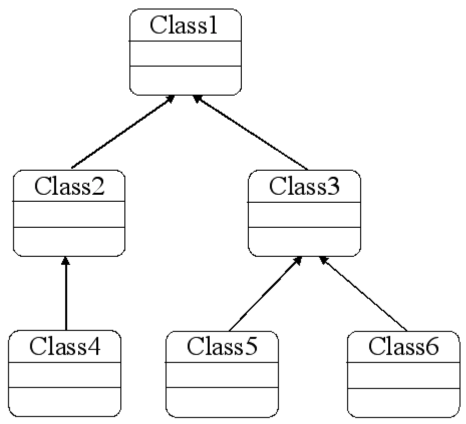


By convention, the superclass is placed at the top of the diagram and subclasses are placed below the superclass and also contain a directed line segment that points upwards to the superclass. The superclass is more general than the subclass and the subclass is more specific than the superclass. **All variables and methods defined in Class1 can be used in Class2 as though the variables and methods had been defined in Class 2** (with one exception, which is discussed in the next section).

The diagrams used above to represent classes are part of the Unified Modelling Language (UML). The standard UML diagram of a class is shown below. The Class name is defined at the top of a rounded rectangle; variables are defined in the middle portion of the rectangle; and methods are defined at the bottom of the rectangle. This class diagram is very convenient when designing classes since it contains the major aspects of a class without including the details. We will use these diagrams occasionally and typically will include only the class name although a variable and/or a method may also be included.

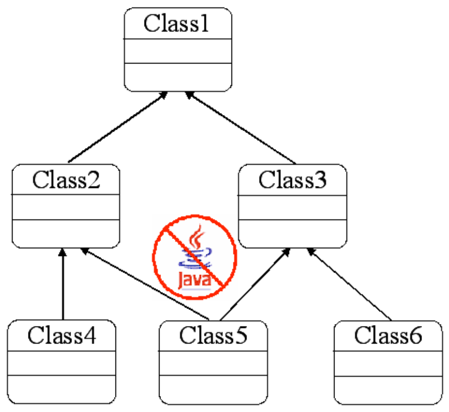


A class hierarchy may consist of any number of classes and the classes may be organized in any manner as long as **each subclass has only one immediate superclass**. The following hierarchy is valid.



As indicated in the diagram, a class may be both a subclass of one class and also a superclass of one or more classes. In this hierarchy, all variables and methods defined in Class1 are inherited by all of its subclasses (Class2, Class3, Class4, Class5, and Class6). Similarly, all variables and methods defined in Class2 are inherited by Class4 and all variables and methods defined in Class3 are inherited by Class5 and Class6.

The following is not a valid hierarchy because Class5 has two immediate superclasses, Class2 and Class3. This is referred to as "**multiple inheritance**" and is not supported in Java although it is supported in some other object-oriented languages.



10.5 Visibility

In the Animals example, you may have noticed that the two variables (`name` and `type`) defined in the Animal class are declared as public variables. This violates our general rule that variables inside an object should be private to prevent users from accidentally or deliberately making modifications to information without necessarily being aware of the consequences. However, if we define the variables to be private, then the Cat and Dog subclasses are not able to access the variables, since private variables are visible only in the class in which they are defined. The solution to this problem in Java was to add a third visibility modifier – **protected**. Any variable or method in a class that is defined with the protected modifier can be accessed and manipulated by all subclasses within the hierarchy but can not be accessed or modified by classes that are outside of the hierarchy. The improved version of the Animal class is:

```
public class Animal
{
     protected String name;
     protected String type;

     public String toString()
     {
        return "Animal: Type: " +type +"; Name: " +name;
     }
}
```

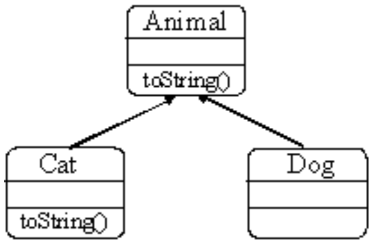
Defining variables in a superclass as protected permits the subclasses to manipulate the variables directly. While this is often desireable, occasionally it is preferable to keep the variables private in the superclass. If the superclass variables are private, the subclasses can not manipulate the variables directly but if appropriate protected accessors and mutators are defined, the subclasses are able to access the variables in a controlled manner.

10.6 Inheriting and Overriding Methods

As we saw in the Animal class, any public method defined in the Animal class is inherited by all subclasses of Animal. Thus, the `toString` method defined in Animal can be used in both the Cat class and the Dog class. Again, inheriting code avoids having to duplicate the code in the subclasses.

At times, though, one subclass may require a different version of the method that is defined in the superclass. One solution is to remove the method from the superclass and define it in each of the subclasses, with all versions identical except for the subclass that requires the different version. This is not a good solution since it leads again to duplicated code. A better solution is to define a different version of the method in the unusual subclass and leave the original version in the superclass. This is referred to as **overriding** a method: a method in a superclass is overridden by a method in a subclass that has the same name, the same signature, and the same return type.

In the following example, we override the `toString` method defined in Animal by defining an alternative `toString` method in Cat that indicates whether or not the cat is declawed. Note that `toString` is not overridden for the Dog class.



```
public class Animal
{
     protected String name;
     protected String type;

     public String toString()
     {
        return "Animal: Type: " +type +"; Name: " +name;
     }
}
```

```
public class Cat extends Animal
{
     private boolean declawed;

     public Cat (String name)
     {
        this.name = name;
        type = "cat";
        declawed = false;
     }

     public void setDeclawed()
     {
        declawed = true;
     }

     public String toString()        // overrides toString in Animal
     {
        String status;
        status = "; is not declawed";
        if (declawed)
        {
           status = "; is declawed";
        }
        return "Animal: Type: " +type +"; Name: " +name +status;
     }
}
```

```
public class Dog extends Animal
{
     public Dog (String name)
     {
        this.name = name;
        type = "dog";
     }
}
```

Now, when the toString message is sent to fluffy, the toString method in Cat generates the following output:

```
Animal: Type: cat; Name: Fluffy; is not declawed
```

When the toString message is sent to fido, the toString method in Animal is used to generate the output because the Dog class does not contain a toString method:

```
Animal: Type: dog; Name: Fido
```

If we add any additional subclasses to our hierarchy, those subclasses will also use the toString method in Animal unless the method is explicitly overridden in one or more of the subclasses.

Notice that when defining the toString method in Cat, we duplicated some of the code that is defined in the `toString` method in Animal. In this particular case, the amount of duplicated code is quite small but in a more complex system, there could be a significant amount of duplicated code. In this case, we would like to be able both to take advantage of the original method in the superclass and also to provide our own extensions in an overriding method. We can do this by calling the method in the superclass and then performing the additional instructions in the overriding method. This is accomplished by adding the prefix **super.** to the name of the overridden method. The following example illustrates this technique.

```
public class Animal
{
     protected String name;
     protected String type;

     public String toString()
     {
        return "Animal: Type: " +type +"; Name: " +name;
     }
}
```

```
public class Cat extends Animal
{
     private boolean declawed;

     . . .

     public String toString()
     {
        String status;
        status = "; is not declawed";
        if (declawed)
        {
           status = "; is declawed";
        }
        return super.toString() +status;
     }
}
```

Now the `toString` method in Cat returns the result of calling the `toString` method in Animal plus the value of `status` as determined by the `toString` method in Cat.

The ability to extend the processing in a higher-level method by calling that method from within a method of the same name in a subclass is very useful when developing object-oriented systems.

10.7 Constructors in Hierarchies

When a subclass in a hierarchy is instantiated, **Java automatically calls the constructor of each superclass** at the beginning of the constructor. Java accomplishes this task by inserting the statement

```
super();
```

at the beginning of the constructor of every subclass **unless there is an explicit reference to the constructor of the superclass** (either with or without parameters). Note that although Java inserts the statement internally, the statement is **not visible** in the program. Thus, the following two versions of the Dog class are identical.

```
public class Dog extends Animal
{
     public Dog (String name)
     {
        this.name = name;
        type = "dog";
     }
}
```

```
public class Dog extends Animal
{
     public Dog (String name)
     {
        super();
        this.name = name;
        type = "dog";
     }
}
```

In this example, Animal does not have an explicit parameter-less constructor so Java creates a default constructor automatically. Calling the constructor of the superclass does not perform any useful function in this example but in some circumstances, this facility can be very useful.

Notice that the processing in both the Cat and Dog constructors is almost identical. If we move the processing in the constructors up into the Animal class, we can reduce the amount of code in the two subclasses.

```
public class Animal
{
     protected String name;
     protected String type;

     public Animal (String name, String type)
     {
        this.name = name;
        this.type = type;
     }

     public String toString()
     {
        return "Animal: Type: " +type +"; Name: " +name;
     }
}
```

```
public class Dog extends Animal
{
     public Dog (String name)
     {
        super(name, "dog");
     }
}
```

```
public class Cat extends Animal
{
     private boolean declawed;

     public Cat (String name)
     {
        super(name, "cat");
        declawed = false;
     }

     public void setDeclawed()
     {
        declawed = true;
     }

     public String toString()
     {
        String status;
        status = "; is not declawed";
        if (declawed)
        {
           status = "; is declawed";
        }
        return super.toString() +status;
     }
}
```

Note that since the constructors in Dog and Cat contain an explicit reference to the superclass constructor, Java **does not add** a parameter-less super reference.

Recall from Chapter 3 – Objects that Java automatically adds a default constructor (no parameters, no statements) to a class if the class does not contain an explicit constructor. This can cause a problem if some classes use a superclass constructor while other classes do not use the superclass constructor.

```
public class Animal
{
     protected String name;
     public String type;

     public Animal (String name, String type)
     {
        this.name = name;
        this.type = type;
     }

     . . .

}
```

```
public class Cat extends Animal
{
     private boolean declawed;

     public Cat (String name)
     {
        super(name, "cat");
        this.declawed = false;
     }

     . . .

}
```

```
public class Dog extends Animal
{
     public Dog (String name)
     {
        this.name = name;
        this.type = "dog";
     }

     . . .

}
```

For example, the classes above generate the compile error shown below.

```
Animals.java:108: cannot find symbol symbol: constructor Animal()
location: class Animal
        {
        ^
1 error

Tool completed with exit code 1
```

Java is complaining that a default constructor does not exist for the Animal class. If you examine the code, you won't find an explicit reference to a default Animal constructor. However, Java inserts `super()` into the beginning of the Dog class (internally) because the

Dog class does not call the superclass constructor explicitly. To fix the problem, either a default constructor must be added to the Animal class or the Dog class constructor must explicitly call the superclass constructor (as was done in the Cat class). The second technique is preferable.

10.8 Casting Objects

When the type of a **variable** is a superclass in a hierarchy, objects that are instances of subclasses may be assigned to such variables. The statements below illustrate this point.

```
Animal animal;
Cat fluffy;
Dog fido;

fluffy = new Cat("Fluffy");
fido = new Dog("Fido");

animal = fluffy;
```

`fluffy` is a Cat object but may be assigned to a variable that is a superclass of Cat. Similarly, the following statement is also valid because `fido` is a dog and Dog is a subclass of the Animal class.

```
animal = fido;
```

This type of assignment is sometimes referred to as "**upcasting**" since we are assigning an object of one class to a variable of a higher-level class in the same hierarchy. An explicit cast is not required because any object that is a Cat (or Dog) is also an Animal according to our hierarchy.

We can also extract an object from a superclass variable and store it in a subclass variable.

```
fluffy = new Cat("Fluffy");
fido = new Dog("Fido");

animal = fluffy;



fluffy = animal;
```

Unfortunately, the statements above do not compile correctly because in the last statement we are performing an assignment from a superclass to a subclass without ensuring that the object in animal is actually a cat.

If we want to extract the object from a superclass variable and store it in a subclass variable, we must explicitly perform a cast, as shown below.

```
fluffy = new Cat("Fluffy");
fido = new Dog("Fido");

animal = fluffy;
fluffy = (Cat) animal;
```

The cast (referred to as a "**downcast**") tells Java that `animal` should contain a valid Cat object and so the assignment should be permitted. Simply including the cast does not ensure that the assignment is **safe** or correct. For example, the following statements compile correctly but generate the run-time error `java.lang.ClassCastException` because the object in the variable animal is not a Cat and so the cast is not successful.

```
fluffy = new Cat("Fluffy");
fido = new Dog("Fido");

animal = fido;
fluffy = (Cat) animal;
```

We will see how to guarantee safe casts in the next section.

10.9 instanceof

As we saw in the previous section, there may be times when we have a subclass object stored in a superclass variable but we may not be certain to which subclass the object belongs. Java provides a comparison operator that allows us to determine whether or not an object is actually a member of a specific class – the `instanceof` operator. Using the instanceof operator appropriately allows us to guarantee safe casts.

```
fluffy = new Cat("Fluffy");
fido = new Dog("Fido");

animal = fido;

if (animal instanceof Cat)
{
     fluffy = (Cat) animal;
     System.out.println(fluffy);
}
else if (animal instanceof Dog)
{
     fido = (Dog) animal;
     System.out.println(fido);
}
else
{
     System.out.println("Unknown object, must be a wildebeest");
}
```

`instanceof` is actually a bit more complicated than is described above – `instanceof` returns true if the object is an instance of the specified class **or if the object is an instance of**

a subclass of the specified class. The following statements determine whether or not an object is an Animal if it is not a Dog or a Cat.

```
fluffy = new Cat("Fluffy");
fido = new Dog("Fido");

animal = fido;

if (animal instanceof Cat)
{
     fluffy = (Cat) animal;
     System.out.println(fluffy);
}
else if (animal instanceof Dog)
{
     fido = (Dog) animal;
     System.out.println(fido);
}
else if (animal instanceof Animal)
{
     System.out.println("The object is an animal.");
}
else
{
     System.out.println("Unknown object");
}
```

If the object is stored in a variable of type Animal, the statement

```
if (animal instanceof Animal)
```

is always true. However, the statements illustrate that an object may be compared to not only the specific class to which the object belongs but also to any superclass in the hierarchy. This creates a problem if statements are not ordered correctly. In our class hierarchy, a Cat object is also an instance of an Animal object. Therefore, **the most specific tests should be performed first**, followed by the more general tests. For example, test for cats and dogs before testing for animals since cats and dogs are subclasses of the Animal class.

10.10 Class Variables and Class Methods

Classes in a hierarchy are no different from individual objects – any data and/or processing that can be added to an individual class can also be added to a class that participates in a hierarchy. Thus, classes in a hierarchy may contain class variables and class methods. Such variables and methods may be defined in any appropriate class in a hierarchy and are then inherited by subclasses of that class.

10.11 Class Object

In Java, all classes are subclasses of the class **`Object`**. Even if a class does not participate in a hierarchy, it is still a subclass of Object. Thus, the definition of any individual class or a class at the top of a hierarchy may explicitly extend the class `Object`:

```
public class Animal extends Object
```

The class `Object` includes several methods but the two most frequently encounted are `equals` and `toString`. The `equals` method in `Object` compares **object references**, **not the contents of objects**. The `toString` method in `Object` returns a string consisting of the name of the class to which the object belongs, an '`@`' character, and a value that represents the internal location at which the object is stored (its "hash code"). For example, if we create a new Flight object but the Flight class does not contain a `toString` method, something similar to the following is displayed when the object is printed.

```
Flight@ad3ba4
```

Normally the `equals` and `toString` methods are overridden in programmer-defined classes.

Variables may be of type `Object`, in which case any object may be assigned to the variable. For example, the following statements assign the cat fluffy to the variable `myObject`. Since we are assigning the object to a superclass (upcasting), it is not necessary to include an explicit cast in the assignment statement.

```
Object myObject;
Cat fluffy;
Dog fido;
fluffy = new Cat("Fluffy");
fido = new Dog("Fido");

myObject = fluffy;
```

However, if we extract the object that is in `myObject`, we must perform an explicit cast since we are assigning the object to a subclass (downcasting).

```
myObject = fluffy;
fluffy = (Cat) myObject;
```

As we saw earlier, if we do not know of which class the object is an instance, we can determine the class using `instanceof`:

```
if (myObject instanceof Cat)
{
     myCat = (Cat) myObject;
}
else if (myObject instanceof Dog)
{
     myDog = (Dog) myObject;
}
else
{
     System.out.println("Unknown object");
}
```

10.12 Arrays are Objects

In Chapter 7 – Object Representation, we stated that although arrays include some non-standard features, arrays are objects. The following program segment illustrates that arrays really are a subclass of Object. An array of int's object may be upcast to an Object. The contents of the object may then be downcast to an array of int's as long as the appropriate downcast (`int[]`) is included.

```
Object object;
int[] myInts = {1, 2, 3, 4, 5};

object = myInts;
myInts = (int[]) object;
```

If we use the printHierarchy method found in Chapter 19 – Miscellanous Topics to print the classes that myInts belongs to, the class definition of an array of int's is the class `[I` .

```
printHierarchy(myInts);

Class: [I
Super Class: java.lang.Object
```

The use of `[I` as the class name for an array of int's is peculiar but every programming language has its quirks. The other arrays of primitive data types also have a corresponding class: the class for double[] is `[D` , the class for char[] is `[C` , and the class for boolean[] is `[Z` This is more than a little weird but is not something that the programmer needs to be aware of.

If an `int[]` object is printed, the result will be something like the following.

```
[I@126b249
```

10.13 Collections of Objects

Objects that participate in a hierarchy may be stored in a collection in the same manner that independent objects may be stored in a collection. For example, we could store the animal objects in an array of type Animal.

```
Animal[] myAnimals = new Animal[3];
Animal myAnimal;
Cat myCat;
Dog myDog;
int count;
myAnimals[0] = new Cat("Fluffy");
myAnimals[1] = new Dog("Fido");
myAnimals[2] = new Cat("Milo");
for (count=0; count<myAnimals.length; count++)
{
      myAnimal = myAnimals[count];
      if (myAnimal instanceof Cat)
      {
          myCat = (Cat) myAnimal;
          System.out.println(myCat);
      }
      else if (myAnimal instanceof Dog)
      {
          myDog = (Dog) myAnimal;
          System.out.println(myDog);
      }
      else
      {
          System.out.println("Unknown object");
      }
}
```

In the example above, the collection is stored in an array of Animal's. This is the preferred method for collecting Animal objects. However, since Object is a superclass of Animal, the following instructions could also be used.

```
Object[] myAnimals = new Object[3];
Object myAnimal;
Cat myCat;
Dog myDog;
int count;

myAnimals[0] = new Cat("Fluffy");
myAnimals[1] = new Dog("Fido");
myAnimals[2] = new Cat("Milo");

for (count=0; count<myAnimals.length; count++)
{
      myAnimal = myAnimals[count];
      if (myAnimal instanceof Cat)
      {
          myCat = (Cat) myAnimal;
          System.out.println(myCat);
      }
      else if (myAnimal instanceof Dog)
      {
          myDog = (Dog) myAnimal;
          System.out.println(myDog);
      }
      else
      {
          System.out.println("Unknown object");
      }
}
```

Although defining an array of Object's works correctly, as a general rule, defining an array of Animal's is the preferred solution.

10.14 ArrayLists

An ArrayList may also be used to store a collection of objects in a hierarchy. Using an ArrayList instead of an array provides the advantages of being able to insert and delete objects anywhere in the collection without having to resize the collection. (Internally, an ArrayList uses an array of type Object to store its contents.)

```
ArrayList myAnimals;
Object myAnimal;
Cat myCat;
Dog myDog;
int count;

myAnimals = new ArrayList();
myAnimals.add(new Cat("Fluffy"));
myAnimals.add(new Dog("Fido"));
myAnimals.add(new Cat("Milo"));

for (count=0; count<myAnimals.size(); count++)
{
     myAnimal = myAnimals.get(count);
     if (myAnimal instanceof Cat)
     {
        myCat = (Cat) myAnimal;
        System.out.println(myCat);
     }
     else if (myAnimal instanceof Dog)
     {
        myDog = (Dog) myAnimal;
        System.out.println(myDog);
     }
     else
     {
        System.out.println("Unknown object");
     }
}
```

In this section, we have assumed that we are storing objects from the same hierarchy in the collection. While this is normally true, Java does not require that all objects in a collection be from the same hierarchy, we may store any objects from multiple hierarchies and/or objects that are not part of an explicit hierarchy in the same collection. This is not a good programming practice but it is valid Java.

As we saw in Chapter 8 – Object Orientation Practices, as programs become more complex, the storing and manipulation of a collection of objects is often moved into its own class in order to separate the data structure used to represent the objects from the manipulations that are made to the collection.

10.15 Generics

We have seen in earlier chapters how Java's generics can be added to an ArrayList to narrow the type of object that can be stored in an ArrayList. For example, if we have an ArrayList that will contain only Cat objects, the ArrayList could be defined as:

```
ArrayList<Cat> list;
```

If an ArrayList contains different types of objects that are in a hierarchy, such as the Dog and Cat classes that are subclasses of Animal, then we can define the ArrayList to contain only those objects by using Animal as the type defined for the ArrayList.

```
ArrayList<Animal> list;
```

Now the ArrayList can contain any combination of Dog and Cat objects but could not contain an object that is not part of the Animal hierarchy.

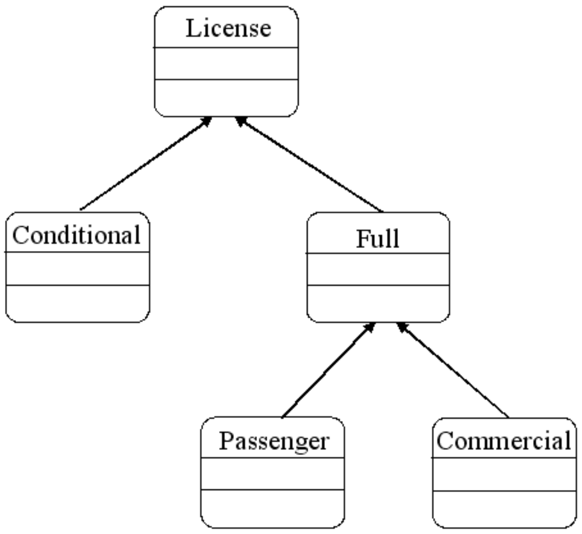
In Chapter 9 – Object Collections, we examined some algorithms that stored different types of objects in an ArrayList. It was noted in the chapter that generics could not be used because the objects stored in the ArrayList were not of the same type. Now that we know that all classes are subclasses of the class Object, we could add the generic type `<Object>` to the ArrayList. Using Object as a generic type does not ensure type-safe assignments but it does remove the warning statements that Java generates if an ArrayList is not given a specific type.

```
public static ArrayList<Object> createList()
{
     ArrayList<Object> list;
     int count;

     list = new ArrayList<Object>();
     for (count=0; count<10; count++)
     {
        list.add(count);
        list.add(new String("s"+count));
     }
     return list;
}

public static removeList(ArrayList<Object> list)
{
     int count;
     for (count=0; count<list.size(); )
     {
        if (list.get(count) instanceof String)
        {
           list.remove(count);
        }
        else
        {
           count++;
        }
     }
}

public static reorderList(ArrayList<Object> list)
{
     Object object;
     int count, position;

     for (position=0, count=0; count<list.size(); count++)
     {
         object = list.get(position);
         if (object instanceof String)
         {
             list.remove(position);
             list.add(object);
         }
         else
         {
             position++;
         }
     }
}
```

10.16 Driver's License Example

The following example develops a set of classes that maintain driver's license information. Every driver's license records the name and address of the driver. Driver's licenses fall into two categories: Conditional or Full. There are also two types of Full drivers' licenses: Passenger and Commercial. All drivers pay a Basic Insurance fee of $40.00. All drivers also pay an additional fee that varies depending on their category. Conditional drivers pay an additional $100 fee. Holders of a Full license are assessed demerit points for driving infractions and are charged a demerit surcharge based on the following formula: the surcharge is $100 for the first demerit, $200 each for the second and third demerits, and $300 each for each demerit over 3. The demerit charges (if any) represent the additional fee for a Passenger driver. The license fee for a Commercial license is an additional 25% of the Basic Insurance fee plus any demerit surcharges as defined above. A `calcFees()` message can be sent to any type of driver; the method returns the total fee for that driver.

The total number of drivers that have ever been given a license (i.e. the total number of driver instances ever created) is also maintained. A method that prints the total number of drivers that have ever been licensed is included. The structure of the classes is shown below. (This example was provided by Dr. John Anderson.)



```
public class License
{
     protected String name;
     protected String address;
     protected double basicFee;
     private static int numberDrivers = 0;

     public License(String name, String address)
     {
        this.name = name;
        this.address = address;
        basicFee = 40.0;
        numberDrivers++;
     }

     protected double calcFees()
     {
        return 0.0;
     }

     public static void printNumberDrivers()
     {
        System.out.println("\nNumber of drivers: " +numberDrivers);
     }

     public String toString()
     {
        return name +" " +address + " " +calcFees();
     }
}
```

```
public class Conditional extends License
{
     protected double conditionalFee;

     public Conditional(String name, String address)
     {
        super(name, address);
        conditionalFee = 100.00;
     }

     protected double calcFees()
     {
        return basicFee + conditionalFee;
     }
}
```

```
public class Full extends License
{
     protected int demerits;

     public Full(String name, String address, int demerits)
     {
        super(name, address);
        this.demerits = demerits;
     }

     protected double processDemerits()
     {
        double fee;

        fee = 0.0;
        if (demerits >= 1)
        {
           fee = fee + 100.0;
        }
        if (demerits >= 2)
        {
           fee = fee + 200.0;
        }
        if (demerits >= 3)
        {
           fee = fee + 200.0;
        }
        if (demerits > 3)
        {
           fee = fee + 300.0 * (demerits-3);
        }
        return fee;
     }
}
```

```
public class Passenger extends Full
{
     public Passenger(String name, String address, int demerits)
     {
        super(name, address, demerits);
     }

     protected double calcFees()
     {
        return basicFee + processDemerits();
     }
}
```

```
public class Commercial extends Full
{
     public Commercial(String name, String address, int demerits)
     {
        super(name, address, demerits);
     }

     protected double calcFees()
```

```
    {
        return basicFee*1.25 + processDemerits();
    }
}
```

The 5 classes are relatively small and easy to understand. If only one large class had been defined, it would have included some very ugly processing. The proper use of inheritance has eliminated code duplication. In fact, the Full and Commercial classes have almost no processing defined in them – the processing is shared by defining it in higher-level classes.

Note that common variables are defined in a superclass but not necessarily in the base class. For example, name and address are defined in License but demerits is used only by some of the classes and so is defined at a lower level in the hierarchy (in Full).

10.17 Recap

In order to use inheritance correctly, it is important to understand how variables and methods are inherited and how methods may be overridden.

Inheritance is a very powerful feature that makes it easier for the programmer to avoid having to duplicate variables and methods.

10.18 Summary

In this chapter we have examined the fundamentals of object hierarchies and object inheritance. If you look at the examples closely, it should be clear that the appropriate use of inheritance increases the amount of code that can be shared by subclasses and reduces the amount duplicate code. There are still additional issues in inheritance that we have not examined but they should not be an impediment to developing object-oriented programs. We will examine some additional issues involving inheritance in the next chapter.

11 MORE OBJECT HIERARCHY TOPICS

11.1 Introduction

In this chapter we examine some additional topics in object inheritance. These topics are not required for a basic understanding of inheritance but they do provide a deeper insight into the nature of object hierarchies.

11.2 Polymorphism

In the previous chapter, when we send a message (such as `toString`) to an object that is stored in a superclass variable, we first extract (downcast) the object to the correct object type and then send the message. Thus, in the statements below, we determine the type of animal and then extract the animal to the appropriate subclass.

```
Animal[] myAnimals = new Animal[3];
Animal myAnimal;
Cat myCat;
Dog myDog;
int count;

myAnimals[0] = new Cat("Fluffy");
myAnimals[1] = new Dog("Fido");
myAnimals[2] = new Cat("Milo");

for (count=0; count<myAnimals.length; count++)
{
     myAnimal = myAnimals[count];
     if (myAnimal instanceof Cat)
     {
        myCat = (Cat) myAnimal;
        System.out.println(myCat);
     }
     else if (myAnimal instanceof Dog)
     {
        myDog = (Dog) myAnimal;
        System.out.println(myDog);
     }
     else
     {
        System.out.println("Unknown object");
     }
}
```

The output of this program is:

```
Animal: Type: cat; Name: Fluffy; is not declawed
Animal: Type: dog; Name: Fido
Animal: Type: cat; Name: Milo; is not declawed
```

It is a nuisance to have to extract an object from a superclass variable and assign it to a variable of the correct type before we can send a message to the object. It would be much more convenient if we could just send the message to the superclass variable (in this example `myAnimal`) instead of to the subclass variable (`myCat` or `myDog`). For example,

```
for (count=0; count<myAnimals.length; count++)
{
     myAnimal = myAnimals[count];
     System.out.println(myAnimal);
}
```

If we did that, we would **expect** to have the following output generated by the `toString` method in **Animal**:

```
Animal: Type: cat; Name: Fluffy
Animal: Type: dog; Name: Fido
Animal: Type: cat; Name: Milo
```

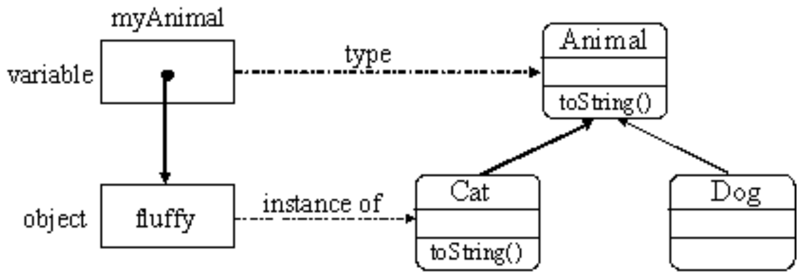
Surprisingly, this is **not** the output that is generated. When the statements are executed the following output is generated.

```
Animal: Type: cat; Name: Fluffy; is not declawed
Animal: Type: dog; Name: Fido
Animal: Type: cat; Name: Milo; is not declawed
```

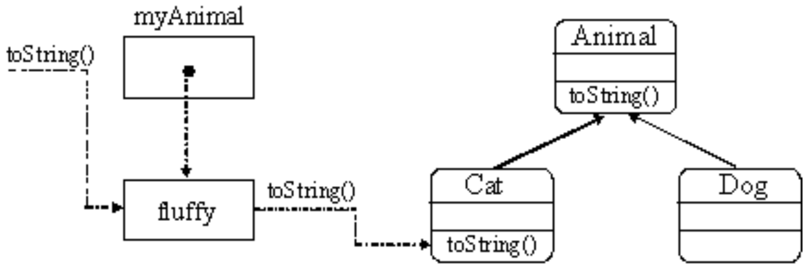
This output is identical to the output generated when the object is extracted to the correct subclass before sending the message. **Why do the statements above work as they do?**

The answer is that when the program is executed (referred to as “run time”), Java determines the class to which an object belongs (which is not necessarily the same as the class of the variable in which the object is currently stored) and sends the message to the class to which the object actually belongs. The process of determining the method to use at run time is referred to as **run-time binding** (or dynamic binding or late binding).

In the example below, the variable myAnimal is of type Animal but the object stored in myAnimal is fluffy which is an instance of Cat.



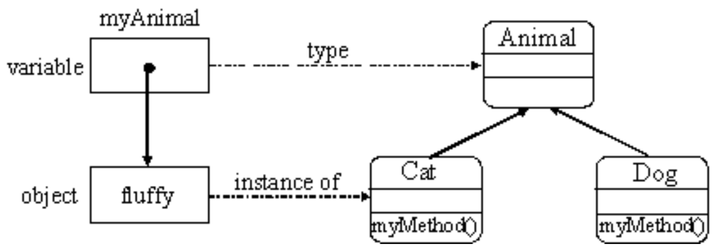
When the `toString` method is sent to myAnimal, Java actually sends the message to the fluffy object which uses the Cat version of `toString`, not the Animal version of `toString`.



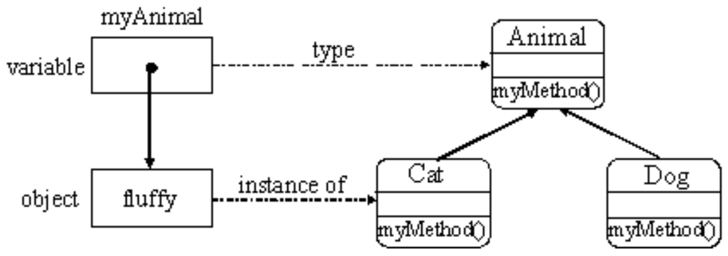
This process is referred to as **polymorphism** and is a very important process in inheritance. As a result of polymorphism, different objects can be stored in the same collection (or data structure) but **each object can respond differently to the same message**. Thus, the following program segment works correctly.

```
Animal[] myAnimals = new Animal[3];
Animal myAnimal;
Cat myCat;
Dog myDog;
int count;

myAnimals[0] = new Cat("Fluffy");
myAnimals[1] = new Dog("Fido");
myAnimals[2] = new Cat("Milo");

for (count=0; count<myAnimals.length; count++)
{
     myAnimal = myAnimals[count];
     System.out.println(myAnimal);
}
```

In order to take advantage of polymorphism, the class of the variable in which an object is stored must also contain (or inherit) a method with the same signature and return type. In the example above, the variable is of type Animal and Animal does contain a `toString` method so everything is fine. However, if the inheritance structure is the one shown below, Animal does not have a `myMethod` method and so the Java compiler generates an error message at compile time if the message `myMethod` is sent to the variable `myAnimal`.



To eliminate this problem, myMethod must be added to the Animal class. The method does not have to do anything since it will not actually be used. However, the method must be defined in the Animal class before Java will compile the program correctly (and the method must have the same signature and return type as the methods in Cat and Dog). Such methods are often referred to as **dummy methods**.



```
public class Animal
{
     protected String name;
     protected String type;

     public Animal (String name, String type)
     {
        this.name = name;
        this.type = type;
     }

     public void myMethod()
     {   // dummy method to support polymorphism
     }

     public String toString()
     {
        return "Animal: Type: " +type +"; Name: " +name;
     }
}
```

In the previous section, we saw that a collection of objects in a hierarchy could be stored in an array of type Object since Object is the ultimate superclass. So, what would happen if we ran the following program?

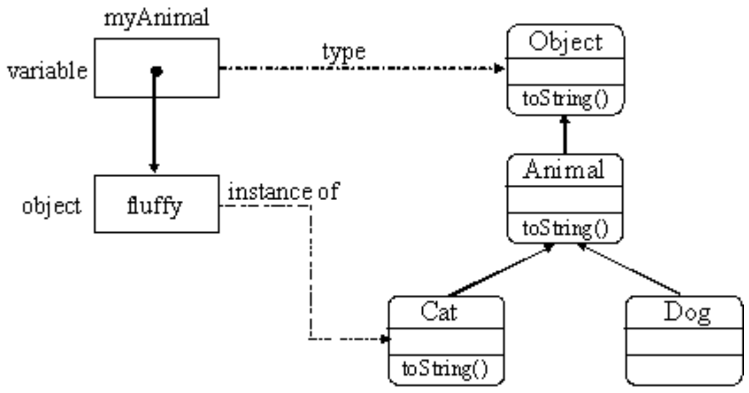
```
Object[] myAnimals = new Object[3];
Object myAnimal;
Cat myCat;
Dog myDog;
int count;

myAnimals[0] = new Cat("Fluffy");
myAnimals[1] = new Dog("Fido");
myAnimals[2] = new Cat("Milo");

for (count=0; count<myAnimals.length; count++)
{
     myAnimal = myAnimals[count];
     System.out.println(myAnimal);
}
```

This program works correctly and generates the desired output.

```
Animal: Type: cat; Name: Fluffy; is not declawed
```

```
Animal: Type: dog; Name: Fido
Animal: Type: cat; Name: Milo; is not declawed
```

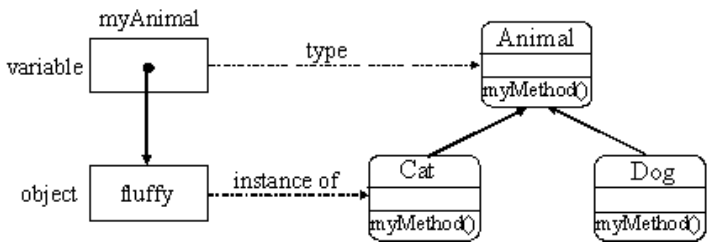
It is important to understand why the program works. The class Object contains a `toString` method so the program compiles correctly. Then, at run time, Java uses polymorphism to locate the `toString` method of the current dog or cat.



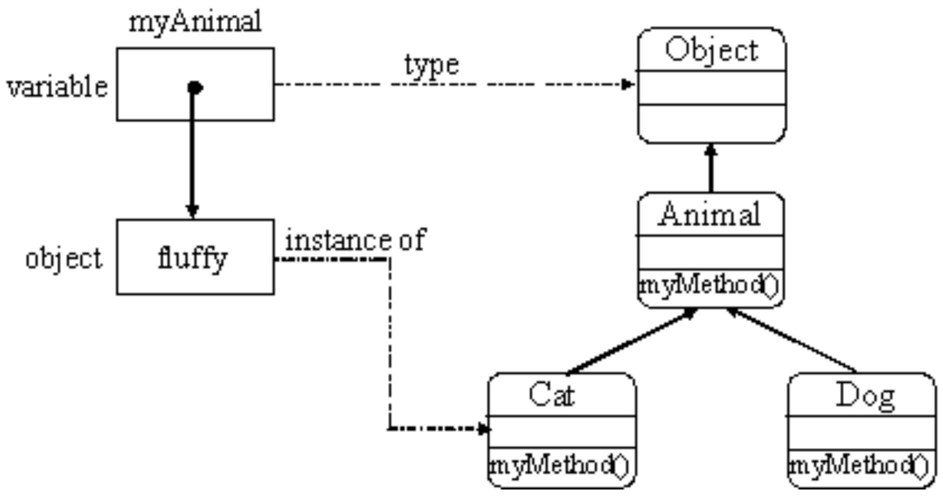
Similarly, if we store our collection of animals in an ArrayList, the program again works correctly, although it is not immediately obvious why it works correctly. If you examine the ArrayList class, you will find that an ArrayList is implemented as an array of type `Object`, so any statements that work with Object's will also work with ArrayList's.

```
ArrayList myAnimals;
Object myAnimal;
int count;

myAnimals = new ArrayList();
myAnimals.add(new Cat("Fluffy"));
myAnimals.add(new Dog("Fido"));
myAnimals.add(new Cat("Milo"));

for (count=0; count<myAnimals.size(); count++)
{
     myAnimal = myAnimals.get(count);
     System.out.println(myAnimal);
}
```

Earlier in this section, we noted that for polymorphism to work correctly, the class of the variable in which an object is stored must contain a method (either dummy or real) that corresponds to each message that can be sent to the object while the object is stored in that variable. Thus, as shown in the earlier example, Animal must contain a myMethod method if the myMethod message is to be sent to the object in the variable myAnimal.



But what will happen if we attempt to send the message myMethod to the variable myAnimal when myAnimal is of type Object? The result is a compile error since Object does not contain a myMethod method. However, unlike the earlier situation in which the problem could be solved simply by adding a dummy version of myMethod to Animal, we can not solve this problem by adding a dummy method to Object since the class Object can not be modified.



There are two alternative ways to solve this problem. The first was mentioned at the end of the Section 10.14 – **do not store a collection in a variable of type `Object`**; instead, store it in the base class of the current hierarchy (Animal in this example).

```
Animal[] myAnimals = new Animal[3];
Animal myAnimal;
```

An alternative solution is to downcast the object to the base class and store it in a variable of the type of the base class prior to sending the message.

```
Object[] myAnimals = new Object[3];
Animal myAnimal;
int count;

myAnimals[0] = new Cat("Fluffy");
myAnimals[1] = new Dog("Fido");
myAnimals[2] = new Cat("Milo");

for (count=0; count<myAnimals.length; count++)
{
     myAnimal = (Animal) myAnimals[count];
     myAnimal.myMethod();
}
```

An equivalent solution is to downcast the object in the statement prior to sending the message.

```
Object[] myAnimals = new Object[3];
Object myAnimal;
int count;

myAnimals[0] = new Cat("Fluffy");
myAnimals[1] = new Dog("Fido");
myAnimals[2] = new Cat("Milo");

for (count=0; count<myAnimals.length; count++)
{
     myAnimal = myAnimals[count];
     ((Animal) myAnimal).myMethod();
}
```

Note the placement of the brackets in the code above: the downcast must be applied to the variable before the message can be sent.

The two downcasting versions shown above are essentially identical and there is no particular reason to prefer one version over the other.

If an ArrayList is used to store the collection, only the 2 downcasting versions can be used.

In this section, we have examined how Java determines at run time which method to use when methods are overridden. It is important to be aware that **polymorphism does not apply to variables**. This is not a problem if variables are not shadowed or hidden. However, if the programmer does shadow variables, strange errors may be introduced into the code.

11.3 Abstract Classes

If a class is never instantiated, then the class should be defined as an **`abstract`** class. The keyword `abstract` in the class header tells Java that the class must not be instantiated. For example, the Animal class used in the examples is used only as a respository for data and processing that are common to the subclasses, instances of Animal are never created.

```
public abstract class Animal
{
     protected String name;
     protected String type;

     public Animal (String name, String type)
     {
        this.name = name;
        this.type = type;
     }

     public String toString()
     {
        return "Animal: Type: " +type +"; Name: " +name;
     }
}
```

The fact that a class is defined as abstract does not mean that it can not contain variables and methods; an abstract class can contain anything that a non-abstract class can contain. Abstract simply ensures that the programmer does not create an instance accidentally. Also, the fact that a class is abstract does not prevent us from creating a variable of the class type and storing instances of subclasses in the variable. The statements used earlier to upcast instances of Cat or Dog to a variable of type Animal still work correctly even if Animal is abstract.

```
Animal animal;
Cat fluffy;
Dog fido;

fluffy = new Cat("Fluffy");
fido = new Dog("Fido");

animal = fluffy;
```

Classes that are abstract may also extend `Object`:

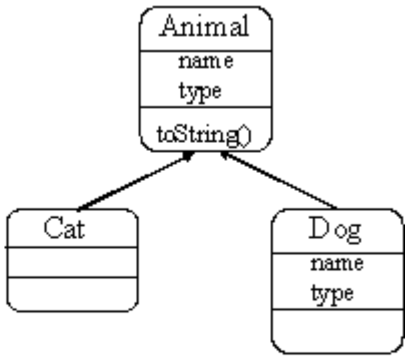
```
public abstract class Animal extends Object
```

11.4 Inheriting and Shadowing Variables

Variables are inherited in the same way that methods are inherited. Any variable defined in a superclass (and is not defined as private) can be accessed and modified in all subclasses.

Variables may also be shadowed or hidden (similar to “overridden”) by declaring another variable in a subclass with the same name as a variable in a superclass. **However, this facility is rarely needed and variables should normally not be shadowed since shadowing variables often leads to programming errors.**

In the following example, the variables name and type are defined in the superclass Animal but are also defined in the subclass Dog.



When the constructor is executed, the assignment statement assigns the values of name and the value “dog” to the local variables, that is, the variables defined in the Dog class. Since these variables shadow the variables defined in Animal, the variables in Animal are not given values.

```
public class Animal
{
     protected String name;
     protected String type;

     public String toString()
     {
        return "Animal: Type: " +type +"; Name: " +name;
     }
}
```

```
public class Dog extends Animal
{
     protected String name;
     protected String type;

     public Dog (String name)
     {
        this.name = name;
        type = "dog";
     }
}
```

If the following statements are executed

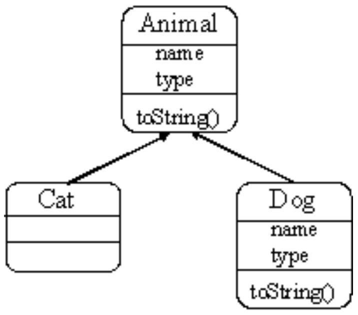
```
fluffy = new Cat("Fluffy");
fido = new Dog("Fido");

System.out.println(fluffy);
System.out.println(fido);
```

the output generated is:

```
Animal: Type: cat; Name: Fluffy; is not declawed
Animal: Type: null; Name: null
```

The output for fluffy is correct because the variables defined in the Animal class were not shadowed in the Cat class. However, the output for fido is incorrect because the variables in the Animal class were shadowed by the additional variables in the Dog class. So, when the toString method in Animal accesses the variables type and name, it accesses the variables in Animal, not in Dog. Since the variables in Animal were never given values, the toString method generates `null` values.

We could override the toString method in Dog and the program would now work correctly **but we are fixing a problem that should not have been created in the first place**.



```
public class Dog extends Animal
{
     protected String name;
     protected String type;

     public Dog (String name)
     {
        this.name = name;
        type = "dog";
     }

     public String toString()
     {
        return "Animal: Type: " +type +"; Name: " +name;
     }
}
```

If for some weird reason it is necessary to shadow variables, the variables that are shadowed can still be accessed by adding the prefix **super.** to the name of the variable. If the following statements are added to the Dog class, the comments illustrate which variable is being referred to.

```
System.out.println(name);           // refers to name in Dog
System.out.println(this.name);      // refers to name in Dog
System.out.println(super.name);     // refers to name in Animal
```

However, to avoid problems, do not shadow/hide variables.

12 OBJECT HIERARCHY EXAMPLES

12.1 Introduction

In this chapter we examine how object inheritance can be used to develop several programs that use inheritance.

12.2 Bank Account Example

The following example illustrates the use of inheritance to define bank accounts.

There are 4 fundamental types of bank accounts: basic chequing, bonus chequing, basic savings, and bonus savings. The characteristics of the bank accounts are as follows:

- Chequing accounts in general:
  flat-fee service charge of $12.00 per month, regardless of the number of transactions; unless otherwise specified, no interest;

- Bonus chequing accounts:
  generate interest (12% annual interest rate) on the portion of the account balance that exceeds $10,000.00 at the end of each month;

- Basic chequing accounts:
  there is no processing that is in addition to the processing defined for chequing accounts in general;

- Savings accounts in general:
  service charge of $1.50 per transaction posted against the account; generate interest (6% annual interest rate) on the account balance at the end of each month;

- Basic savings accounts:
  generate additional interest (9% annual interest rate) on the portion of the account balance that exceeds $5,000.00 at the end of each month (this interest is on top of the basic interest calculated for all savings accounts);

- Bonus Savings accounts:
  generate additional interest (12% annual interest rate) on the portion of the account balance that exceeds $10,000.00 at the end of each month (this interest is on top of the basic interest calculated for all savings accounts).

The month-end processing involves calculating the services charges and the interest that has accrued for an account. After the interest and service charges have been determined, the interest is added to the account balance and the service charges are subtracted from the account balance.

```
public abstract class Account
{
      protected String accountNumber;
      protected double accountBalance;
      protected int transactions;
      protected String accountType;
      protected double serviceCharges;
      protected double interest;
      protected static int totalAccounts = 0;

      public Account(String accountNumber, double accountBalance,
                                          int transactions)
      {
         this.accountNumber = accountNumber;
         this.accountBalance = accountBalance;
         this.transactions = transactions;
         totalAccounts++;
      }

      public void monthEnd()
      {
         calculateInterest();
         calculateServiceCharges();
         accountBalance = accountBalance + interest - serviceCharges;
      }

      protected void calculateInterest()
      {
         interest = 0.0;
      }
      protected void calculateServiceCharges()
      {
         serviceCharges = 0.0;
      }

      public static void printTotalAccounts()
      {
         System.out.println("\nThere are " +totalAccounts +" bank accounts.");
      }

      public static void printHeader()
      {
         System.out.println("\nBank Accounts\n");
         System.out.println(
         "Number\tType\t\tBalance\t\tTransactions\tCharges\t\tInterest");
         System.out.println(
         "------\t----\t\t-------\t\t------------\t-------\t\t--------");
      }

      public String toString()
      {
         return accountNumber +"\t" +accountType +"\t" +accountBalance
             +"\t\t" +transactions  +"\t\t" +serviceCharges
             +"\t\t" +interest;
      }
}
```

```
public abstract class Chequing extends Account
{
      public Chequing(String accountNumber, double accountBalance,
                                                      int transactions)
      {
         super(accountNumber, accountBalance, transactions);
      }

      protected void calculateServiceCharges()
      {
         serviceCharges = 12.00;
      }
}
```

```
public class BasicChequing extends Chequing
{
      public BasicChequing(String accountNumber, double accountBalance,
                                                            int transactions)
      {
          super(accountNumber, accountBalance, transactions);
          accountType = "BasicChequing";
      }
}

public class BonusChequing extends Chequing
{
      private static final double BONUS_CHEQUING_INTEREST = 0.12/12.0;

      public BonusChequing(String accountNumber, double accountBalance,
                                                 int transactions)
      {
          super(accountNumber, accountBalance, transactions);
          accountType = "BonusChequing";
      }

      protected void calculateInterest()
      {
          interest = BONUS_CHEQUING_INTEREST *
                                  Math.max(0.0, (accountBalance - 10000.00));
      }
}

public abstract class Savings extends Account
{
      private static final double SAVINGS_INTEREST = 0.06/12.0;

      public Savings(String accountNumber, double accountBalance,
                                           int transactions)
      {
          super(accountNumber, accountBalance, transactions);
          serviceCharges = 1.50;
      }

      protected void calculateInterest()
      {
          interest = SAVINGS_INTEREST * accountBalance;
      }

      protected void calculateServiceCharges()
      {
          serviceCharges = 1.50 * transactions;
      }
}

public class BasicSavings extends Savings
{
      private static final double BASIC_SAVINGS_INTEREST = 0.09/12.0;

      public BasicSavings(String accountNumber, double accountBalance,
                                                int transactions)
      {
          super(accountNumber, accountBalance, transactions);
          accountType = "BasicSavings";
          serviceCharges = 2.00;
      }

      protected void calculateInterest()
      {
          super.calculateInterest();
          interest += BASIC_SAVINGS_INTEREST *
                                     Math.max(0.0, accountBalance - 5000.00);
      }
}
```

```
public class BonusSavings extends Savings
{
      private static final double BONUS_SAVINGS_INTEREST = 0.12/12.0;

      public BonusSavings(String accountNumber, double accountBalance,
                                           int transactions)
      {
          super(accountNumber, accountBalance, transactions);
          accountType = "BonusSavings";
          serviceCharges = 2.00;
      }

      protected void calculateInterest()
      {
          super.calculateInterest();
          interest += BONUS_SAVINGS_INTEREST *
                                      Math.max(0.0, accountBalance - 10000.00);
      }
}
```

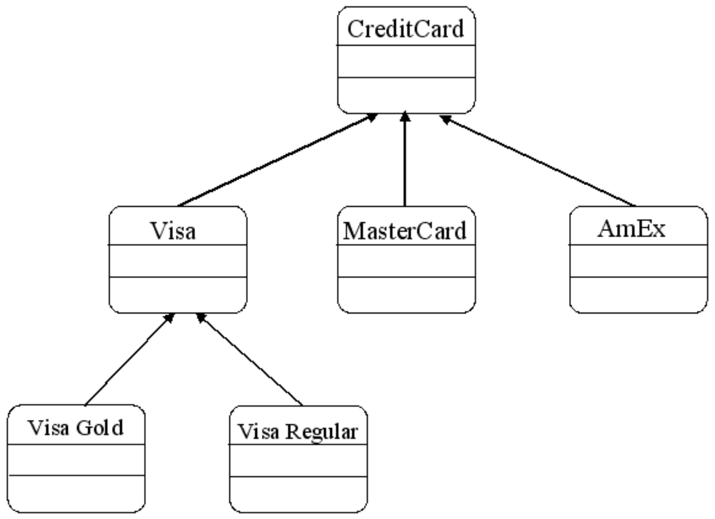
12.2.1 Recap

Once again, the base class contains the majority of the processing and the subclasses contain only the processing that is specific to each subclass.

12.3 Credit Card Example

In this section, we again examine a system that keeps track of credit-card information. In this example, there are 4 types of credit cards: the Visa Regular card, the Visa Gold card, the Master Card, and the American Express card. Each card has different parameters and ways of performing its processing. For each card, the following information is maintained: the type of credit card, the credit card identifier, and the previous month's balance. The credit limit and interest rate (per Month) for each card are: Visa Regular $5,000. 2%; Visa Gold $50,000. *; Master Card $8,000. 3%; American Express $10,000. 4%. (* The interest rate for a Visa Gold card is one-half of the interest rate for a regular Visa card; this value is computed in the program, it is not hard coded.) The account balances are represented so that if a customer owes money, the balance is positive and if the customer has actually overpaid the account, then the balance is negative. The credit card user is not permitted to charge an amount to the card if the amount would put the balance above the user's credit limit. The credit limit is the credit limit for the card (as defined above) minus the current balance owing on the card.

Each class has its own interest rate calculation method. The interest rates are calcuated as follows: Visa Gold: if the previous month's balance is greater than zero, charge interest on the amount of the previous month's balance that exceeds $2,000.00; Visa Regular: if the previous month's balance is greater than zero, charge interest on the previous month's balance; Master Card: if the previous month's balance is greater than zero, charge interest on the first $10,000 of the previous month's balance plus double the rate of interest on any balance the exceeds $10,000; American Express: if the previous month's balance is greater than zero, charge interest on the amount previous month's balance that exceeds $5,000. At the end of each month, the interest is computed and the account balance is recalculated. The Structure of the classes is shown below.



Each credit card is defined in its own class. Visa regular and Visa gold share a superclass that contains data and processing that are common to both. CreditCard is a generic base class that contains data and processing that are common to all subclasses.

```
public abstract class CreditCard
{
      protected String cardType;
      protected String accountNumber;
      protected double currentBalance;
      protected double interestAmount;
      protected double interestRate;
      protected double creditLimit;

      public CreditCard(String accountNumber, double currentBalance,
                        double interestRate, double creditLimit, String cardType)
      {
          this.accountNumber = accountNumber;
          this.currentBalance = currentBalance;
          this.interestRate = interestRate;
          this.creditLimit = creditLimit;
          this.cardType = cardType;
      }

      public void processPayment(double amount)
      {
          currentBalance -= amount;
          System.out.println("Payment of " +amount +" made to account "
                                                              +accountNumber);
      }

      public void processPurchase(double amount)
      {
          if ((amount + currentBalance) > creditLimit)
          {
              System.out.println("Over credit limit on account " +accountNumber
                                 +", purchase denied.");
          }
          else
          {
              currentBalance += amount;
              System.out.println("Purchase of " +amount +" made from account "
                                                                    +accountNumber);
          }
      }

      public void calculateInterest()
      {   // dummy method needed for polymorphism
      }

      public String toString()
      {
          return cardType +" " +accountNumber +" " +currentBalance;
      }
  }
```

```
public abstract class Visa extends CreditCard
{
    protected final static double VISA_REGULAR_INTEREST = 0.02;
    protected final static double VISA_GOLD_INTEREST = VISA_REGULAR_INTEREST/2.0;
    protected final static  double VISA_REGULAR_LIMIT = 5000.0;
    protected final static  double VISA_GOLD_LIMIT = 50000.0;

    public Visa(String accountNumber, double currentBalance, double interestRate,
                                        double creditLimit, String cardType)
    {
        super(accountNumber, currentBalance, interestRate, creditLimit, cardType);
    }
```

```
public class VisaGold extends Visa
{
    public VisaGold(String accountNumber, double currentBalance)
    {
        super(accountNumber, currentBalance, VISA_GOLD_INTEREST, VISA_GOLD_LIMIT,
                                                         "Visa Gold       ");
    }

    public void calculateInterest()
    {
        interestAmount = 0.0;
        if (currentBalance > 2000.0)
        {
            interestAmount = (currentBalance-2000.0) * (interestRate);
            currentBalance += interestAmount;
        }
    }
}
```

```
public class VisaRegular extends Visa
{
    public VisaRegular(String accountNumber, double currentBalance)
    {
        super(accountNumber, currentBalance, VISA_REGULAR_INTEREST,
                                    VISA_REGULAR_LIMIT, "Visa Regular    ");
    }

    public void calculateInterest()
    {
        interestAmount = 0.0;
        if (currentBalance > 0.0)
        {
            interestAmount = currentBalance * interestRate;
            currentBalance += interestAmount;
        }
    }
}
```

```
public class MasterCard extends CreditCard
{
      protected final static double MC_INTEREST = 0.03;
      protected final static  double MC_CREDIT_LIMIT = 8000.0;

      public MasterCard(String accountNumber, double currentBalance)
      {
          super(accountNumber, currentBalance, MC_INTEREST, MC_CREDIT_LIMIT,
                                                             "Master Card     ");
      }

      public void calculateInterest()
      {
          interestAmount = 0.0;
          if (currentBalance > 0.0)
          {
              if (currentBalance <= 10000.0)
              {
                  interestAmount = currentBalance * interestRate;
              }
              else
              {
                  interestAmount = 10000.0 * interestRate
                                  + (currentBalance-10000)*2.0*interestRate;
              }
              currentBalance += interestAmount;
          }
      }
}
```

```
public class AmericanExpress extends CreditCard
{
      protected final static double AMEX_INTEREST = 0.04;
      protected final static  double AMEX_CREDIT_LIMIT = 10000.0;

      public AmericanExpress(String accountNumber, double currentBalance)
      {
          super(accountNumber, currentBalance, AMEX_INTEREST, AMEX_CREDIT_LIMIT,
                                                            "American Express");
      }

      public void calculateInterest()
      {
          interestAmount = 0.0;
          if (currentBalance > 5000.0)
          {
              interestAmount = (currentBalance-5000.0) * (interestRate);
              currentBalance += interestAmount;
          }
      }
}
```

Notice that although there are 5 classes (plus the main class), there is not a lot of processing in any particular class. Through the appropriate use of inheritance, the processing has been distributed over the 5 classes. If changes need to be made to the processing, it should be easy to identify the class or method to change and to make the change.

12.4 Stock Portfolio Example

In this example, we develop a program that manages a portfolio of stocks. We will begin by taking the framework used in Chapter 18 – Growing and Refactoring to maintain a collection of flights. The program will be grown slowly. Again, if you are able to design the program correctly in advance (and you are aware of all of the program requirements in advance), it is not necessary to grow the program so slowly.

12.4.1 Iteration 1

Intially, the framework used in Chapter 18 – Growing and Refactoring is reorganized so that it manipulates a collection (portfolio) of stocks. For each stock, the program maintains the name of the stock, the number of shares currently owned, and the current share price. A particular stock will appear only once in the portfolio, regardless of when the shares were purchased – either at one time or spread out over a period of time. (This is not a realistic assumption but for our purposes, it is acceptable.)

```
public class Portfolio
{
     private ArrayList portfolio;

     public Portfolio()
     {
        createPortfolio();
     }

     private void createPortfolio()
     {
        portfolio = new ArrayList();

        portfolio.add(new Stock("Nortel", 100, 5.00));
        portfolio.add(new Stock("IBM", 100, 500.00));
        portfolio.add(new Stock("Microsoft", 100, 0.05));
     }

     private int locateStock(String accountName)
     {
        Stock currentStock;
        int found;
        int currentElement;

        found = -1;
        for (currentElement=0; currentElement<portfolio.size()
                                    && found==-1; currentElement++)
        {
           currentStock = (Stock) portfolio.get(currentElement);
           if (currentStock.compareAccountNames(accountName))
           {
              found = currentElement;
           }
        }
        return found;
     }

     public void deleteStock(String accountName)
     {
        Stock currentStock;
        int found;

        found = locateStock(accountName);
        if (found != -1)
```

```
         {
            portfolio.remove(found);
            System.out.println("\nStock " +accountName +" was deleted.");
         }
         else
         {
            System.out.println("\nStock " +accountName +" does not exist.");
         }
     }

     public void insertStock(String accountName, int numberShares,
                                                               double sharePrice)
     {
         Stock newStock;
         int found;

         found = locateStock(accountName);
         if (found == -1)
         {
            newStock = new Stock(accountName, numberShares, sharePrice);
            portfolio.add(newStock);
            System.out.println("\nStock " +accountName +" was inserted.");
         }
         else
         {
            System.out.println("\nStock " +accountName +" already exists.");
         }
     }

     public void printPortfolio()
     {
         int currentElement;

         System.out.println();
         for (currentElement=0; currentElement<portfolio.size(); currentElement++)
         {
            System.out.println(portfolio.get(currentElement));
         }
     }
}

public class Stock
{
     protected String accountName;
     protected int numberShares;
     protected double sharePrice;

     public Stock(String accountName, int numberShares, double sharePrice)
     {
         this.accountName = accountName;
         this.numberShares = numberShares;
         this.sharePrice = sharePrice;
     }

     public boolean compareAccountNames(String whichAccount)
     {
         return accountName.equals(whichAccount);
     }
     public String toString()
     {
         return accountName +" " +numberShares +" " +sharePrice;
     }
}
```

12.4.2 Iteration 2

In this iteration, we add commands that permit users to buy and sell stocks. It is assumed that the current price of each stock is always up to date in that stock's object.

```
public class Portfolio
{
     public void sellStock(String accountName, int numberShares)
     {
        Stock currentStock;
        int found;

        found = locateStock(accountName);
        if (found != -1)
        {
           currentStock = (Stock) portfolio.get(found);
           currentStock.sellStock(numberShares);
        }
        else
        {
           System.out.println("Stock " +accountName +" could not be found.");
        }
     }

     public void buyStock(String accountName, int numberShares)
     {
        Stock currentStock;
        int found;

        found = locateStock(accountName);
        if (found != -1)
        {
           currentStock = (Stock) portfolio.get(found);
           currentStock.buyStock(numberShares);
        }
        else
        {
        System.out.println("Stock " +accountName +" could not be found.");
        }
     }
     . . .
```

```
public class Stock
{
     protected String accountName;
     protected int numberShares;
     protected double sharePrice;

     public void sellStock(int numberSharesSold)
     {
        numberShares -= numberSharesSold;
     }

     public void buyStock(int numberSharesPurchased)
     {
        numberShares += numberSharesPurchased;
     }
}
```

12.4.3 Iteration 3

The preceding change was easy due to the good organization of the Portfolio and Stock classes. Now however, your boss decides that since you developed the stock portfolio manager so quickly, you should add the ability to manage Bonds as well as Stocks.

Taking a look at the Stock class, you decide that adding Bonds will be easy so you just copy the Stock class, changing all references from Stock to Bond, and you are done, right?

```
public class Bond
{
     protected String accountName;
     protected int numberShares;
     protected double sharePrice;

     public Bond(String accountName, int numberShares, double sharePrice)
     {
        this.accountName = accountName;
        this.numberShares = numberShares;
        this.sharePrice = sharePrice;
     }

     public void sellBond(int numberSharesSold)
     {
        numberShares -= numberSharesSold;
     }

     public void buyBond(int numberSharesPurchased)
     {
        numberShares += numberSharesPurchased;
     }

     public boolean compareAccountNames(String whichAccount)
     {
        return accountName.equals(whichAccount);
     }

     public String toString()
     {
        return accountName +" " +numberShares +" " +sharePrice;
     }
```

Unfortunately, we aren't done yet. The Portfolio class contains a reference to the Stock class in almost every method in Portfolio. These methods must be either duplicated (one version for Stock and another version for Bond) or extended to handle both a Stock and a Bond. This is not quite as easy as it looked.

```
public class Portfolio
{
     private ArrayList portfolio;

     public Portfolio()
     {
        createPortfolio();
     }
```

```
    private void createPortfolio()
    {
       portfolio = new ArrayList();

       portfolio.add(new Stock("Nortel", 100, 5.00));
       portfolio.add(new Stock("IBM", 100, 500.00));
       portfolio.add(new Bond("CSB", 100, 100.00));
       portfolio.add(new Stock("Microsoft", 100, 0.05));
    }

    private int findElement(String accountName)
    {
       Stock currentStock;
       Bond currentBond;
       int found;
       int currentElement;

       found = -1;
       for (currentElement=0; currentElement<portfolio.size()
                                  && found==-1; currentElement++)
       {
          if (portfolio.get(currentElement) instanceof Stock)
          {
             currentStock = (Stock) portfolio.get(currentElement);
             if (currentStock.compareAccountNames(accountName))
             {
                found = currentElement;
             }
          }
          else
          {
             currentBond = (Bond) portfolio.get(currentElement);
             if (currentBond.compareAccountNames(accountName))
             {
                found = currentElement;
             }
          }
       }
       return found;
    }
```

```
    public void deleteStock(String accountName)
    {
       int found;

       found = findElement(accountName);
       if (found != -1)
       {
          portfolio.remove(found);
          System.out.println("\nStock " +accountName +" was deleted.");
       }
       else
       {
          System.out.println("\nStock " +accountName +" does not exist.");
       }
    }
    public void deleteBond(String accountName)
    {
       int found;

       found = findElement(accountName);
       if (found != -1)
       {
          portfolio.remove(found);
          System.out.println("\nBond " +accountName +" was deleted.");
       }
       else
       {
          System.out.println("\nBond " +accountName +" does not exist.");
       }
    }

    public void insertStock(String accountName, int numberShares,
                                                       double sharePrice)
    {
       Stock newStock;
       int found;

       found = findElement(accountName);
       if (found == -1)
       {
          newStock = new Stock(accountName, numberShares, sharePrice);
          portfolio.add(newStock);
          System.out.println("\nStock " +accountName +" was inserted.");
       }
       else
       {
          System.out.println("\nStock " +accountName +" already exists.");
       }
    }
```

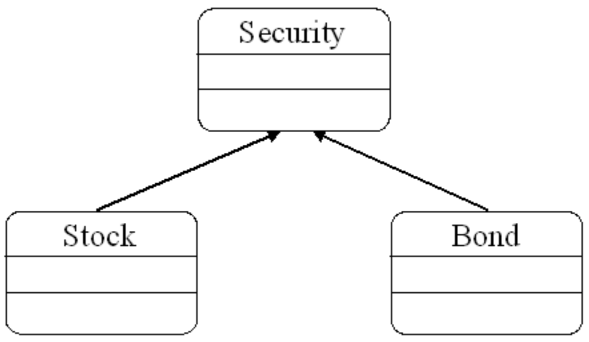
```
    public void insertBond(String accountName, int numberShares,
                                                        double sharePrice)
    {
       Bond newBond;
       int found;

       found = findElement(accountName);
       if (found == -1)
       {
          newBond = new Bond(accountName, numberShares, sharePrice);
          portfolio.add(newBond);
          System.out.println("\nBond " +accountName +" was inserted.");
       }
       else
       {
          System.out.println("\nBond " +accountName +" already exists.");
       }
    }
    public void sellStock(String accountName, int numberShares)
    {
       Stock currentStock;
       int found;

       found = findElement(accountName);
       if (found != -1)
       {
          currentStock = (Stock) portfolio.get(found);
          currentStock.sellStock(numberShares);
       }
       else
       {
          System.out.println("Stock " +accountName +" could not be found.");
       }
    }

    public void sellBond(String accountName, int numberShares)
    {
       Bond currentBond;
       int found;

       found = findElement(accountName);
       if (found != -1)
       {
          currentBond = (Bond) portfolio.get(found);
          currentBond.sellBond(numberShares);
       }
       else
       {
          System.out.println("Stock " +accountName +" could not be found.");
       }
    }

    public void buyStock(String accountName, int numberShares)
    {
       Stock currentStock;
       int found;

       found = findElement(accountName);
       if (found != -1)
       {
          currentStock = (Stock) portfolio.get(found);
          currentStock.buyStock(numberShares);
       }
       else
       {
          System.out.println("Stock " +accountName +" could not be found.");
       }
    }
```

```
     public void buyBond(String accountName, int numberShares)
     {
        Bond currentBond;
        int found;

        found = findElement(accountName);
        if (found != -1)
        {
           currentBond = (Bond) portfolio.get(found);
           currentBond.buyBond(numberShares);
        }
        else
        {
           System.out.println("Stock " +accountName +" could not be found.");
        }
     }

     public void printPortfolio()
     {
        int currentElement;

        System.out.println();
        for (currentElement=0; currentElement<portfolio.size(); currentElement++)
        {
           System.out.println(portfolio.get(currentElement));
        }
     }
}
```

So the program now works but it wasn't as easy as simply copying the Stock class – the Portfolio class essentially doubled in size since most of the methods had to be defined once for Stocks and once for Bonds. Things are not looking good because the boss has been talking about extending the system to handle other types of securities as well.

12.4.4 Iteration 4

So it is time to take a closer look at the organization of the program. Just having taken a one-day course on object inheritance, you decide that adding a common superclass for Stock and Bond would allow you to consolidate the processing in Portfolio.



```
public class Security
{
     . . .
}


public class Stock extends Security
{
     . . .
}

public class Bond extends Security
{
     . . .
}
```

With this class structure, you can now change all references to Stock in the Portfolio class to Security and also remove all methods that refer to Bond. (This of course is the class structure that a good designer would have created at the beginning of Iteration 3, avoiding all of the duplication that was required during that iteration.)

```
public class Portfolio
{
     private ArrayList portfolio;

     public Portfolio()
     {
        createPortfolio();
     }

     private void createPortfolio()
     {
        portfolio = new ArrayList();

        portfolio.add(new Stock("Nortel", 100, 5.00));
        portfolio.add(new Stock("IBM", 100, 500.00));
        portfolio.add(new Bond("CSB", 100, 100.00));
        portfolio.add(new Stock("Microsoft", 100, 0.05));
     }
```

```
     private int findSecurity(String accountName)
     {
        Security currentSecurity;
        int found;
        int currentElement;

        found = -1;
        for (currentElement=0; currentElement<portfolio.size()
                                 && found==-1; currentElement++)
        {
           currentSecurity = (Security) portfolio.get(currentElement);
           if (currentSecurity.compareAccountNames(accountName))
           {
              found = currentElement;
           }
        }
        return found;
     }
```

```
    public void deleteSecurity(String accountName)
    {
       int found;

       found = findSecurity(accountName);
       if (found != -1)
       {
          portfolio.remove(found);
          System.out.println("\nSecurity " +accountName +" was deleted.");
       }
       else
       {
          System.out.println("\nSecurity " +accountName +" not deleted.");
       }
    }

    public void insertSecurity(String securityType, String accountName,
                                        int numberShares, double sharePrice)
    {
       Security newSecurity;
       int found;

       found = findSecurity(accountName);
       if (found == -1)
       {
          if (securityType.equals("Stock"))
          {
             newSecurity = new Stock(accountName, numberShares, sharePrice);
          }
          else
          {
             newSecurity = new Bond(accountName, numberShares, sharePrice);
          }
          portfolio.add(newSecurity);
          System.out.println("\nSecurity " +accountName +" was inserted.");
       }
       else
       {
          System.out.println("\nSecurity " +accountName +" already exists.");
       }
    }
```

```
    public void sellSecurity(String accountName, int numberShares)
    {
       Security currentSecurity;
       int found;

       found = findSecurity(accountName);
       if (found != -1)
       {
          currentSecurity = (Security) portfolio.get(found);
          currentSecurity.sellSecurity(numberShares);
       }
       else
       {
          System.out.println("Security " +accountName +" could not be found.");
       }
    }
```

```
     public void buySecurity(String accountName, int numberShares)
     {
        Security currentSecurity;
        int found;

        found = findSecurity(accountName);
        if (found != -1)
        {
           currentSecurity = (Security) portfolio.get(found);
           currentSecurity.buySecurity(numberShares);
        }
        else
        {
           System.out.println("Security " +accountName +" could not be found.");
        }
     }

     public void printPortfolio()
     {
        int currentElement;

        System.out.println();
        for (currentElement=0; currentElement<portfolio.size(); currentElement++)
        {
           System.out.println(portfolio.get(currentElement));
        }
     }
}
```

Note that there is one place where Stocks and Bonds must be explicitly referenced – in the insertSecurity method it is necessary to know whether a Stock or a Bond is being inserted. Otherwise, the remainder of the class is now generic and can handle any type of security.

Unfortunately, when the program is compiled, the Java compiler generates a series of unexpected error messages:

```
Inherit.java:64: cannot resolve symbol
symbol  : method compareAccountNames (String)
location: class Security
            if (currentSecurity.compareAccountNames(accountName))
                               ^
Inherit.java:125: cannot resolve symbol
symbol  : method sellSecurity (int)
location: class Security
            currentSecurity.sellSecurity(numberShares);
                           ^
Inherit.java:142: cannot resolve symbol
symbol  : method buySecurity (int)
location: class Security
            currentSecurity.buySecurity(numberShares);
                           ^
3 errors
```

What is the problem? The 3 methods that Java has complained about (compareAccountNames, sellSecurity, and buySecurity) all exist in the Stock and Bond subclasses! The problem is polymorphism – you are sending the same message to two different objects (which is perfectly valid) but the objects are stored in a variable of type Security in the Portfolio class

```
Security currentSecurity = (Security) portfolio.get(found);
```

and the Security class does not include any of the 3 methods. So the solution is to add 3 dummy methods to the Security class.

```
public class Security
{
     public boolean compareAccountNames(String whichAccount)
     {   // dummy method for polymorphism
         return false;
     }

     public void sellSecurity(int numberSharesSold)
     {   // dummy method for polymorphism
     }

     public void buySecurity(int numberSharesPurchased)
     {   // dummy method for polymorphism
     }
}
```

The methods added to the Security class will not be executed, they will be overridden at run-time by the corresponding methods in the Stock class or the Bond class. But Java insists that these methods be defined in the superclass in order to guarantee that a method will be available at run-time. This is an example of **polymorphism** – the ability to store different instances of subclasses in a variable of the type of the superclass. But for this to work, the programmer must define dummy methods in the superclass so that Java is able to compile the program.

12.4.5 Iteration 5

Now the program works correctly and could be left as it is but having just returned from another course on object inheritance with some refactoring thrown in, you recognize that the two classes Stock and Bond are identical and the refactorings ***Pull Up Field (Variable)*** and ***Pull Up Method*** can be used in this situation. The *Pull Up Field* refactoring is used when the same variables occur in two or more subclasses of a class hierarchy; in this situation, the variables should be moved up to the superclass so that they are defined only once. The *Pull Up Method* refactoring is used in the same manner for identical methods in two or more subclasses.

```
public class Security
{
     protected String accountName;
     protected int numberShares;
     protected double sharePrice;

     public void sellSecurity(int numberSharesPurchased)
     {
        numberShares -= numberSharesPurchased;
     }

     public void buySecurity(int numberSharesSold)
     {
        numberShares += numberSharesSold;
     }

     public boolean compareAccountNames(String whichAccount)
     {
        return accountName.equals(whichAccount);
     }

     public String toString()
     {
        return accountName +" " +numberShares +" " +sharePrice;
     }
}
```

```
public class Stock extends Security
{
     public Stock(String accountName, int numberShares, double sharePrice)
     {
        this.accountName = accountName;
        this.numberShares = numberShares;
        this.sharePrice = sharePrice;
     }
}
```

```
public class Bond extends Security
{
     public Bond(String accountName, int numberShares, double sharePrice)
     {
        this.accountName = accountName;
        this.numberShares = numberShares;
        this.sharePrice = sharePrice;
     }
}
```

After performing the refactorings, the Stock and Bond subclasses are almost empty, with the processing being defined once in the Security class. Note that this refactoring has also eliminated the need for the dummy methods in the Security class.

12.4.6 Iteration 6

Now the subclasses consist only of the constructors and the constructors are identical. So it is time for another refactoring – ***Pull Up Constructor Body***. This refactoring is used when subclasses have identical (or almost identical processing) in their constructors that could be moved into the constructor of a superclass.

Now, each subclass simply calls the constructor of the superclass, passing the parameters along to the superclass. This is not a huge change to the program structure but it removes some redundancy and ensures that any programmer who must modify the code will make the modification correctly since the statements are defined only once. If there is any processing that is specific to either Stock or Bond, it would be added to the Stock or Bond constructor after the call to the superclass constructor.

```
public class Security
{
     public Security(String accountName, int numberShares, double sharePrice)
     {
        this.accountName = accountName;
        this.numberShares = numberShares;
        this.sharePrice = sharePrice;
     }
```

```
public class Stock extends Security
{
     public Stock(String accountName, int numberShares, double sharePrice)
     {
        super(accountName, numberShares, sharePrice);
     }
}
```

```
public class Bond extends Security
{
     public Bond(String accountName, int numberShares, double sharePrice)
     {
        super(accountName, numberShares, sharePrice);
     }
}
```

12.4.7 Iteration 7

Now that the program is running correctly, it is time to compute the commission on each sale so that the salesmen can finally be paid.

The commission is computed for both purchases and sales but is computed differently for stocks and bonds. For stocks, the commission is $10.00 per share for stocks with a current share price of $100.00 or more and $1.00 per share for stocks with a current price of less than $100.00. For bonds, there is a flat fee of $10.00 per transaction, regardless of the amount of the sale/purchase or the quantity of a particular bond sold/purchased.

```
public class Stock extends Security
{
     public void calculateCommission(int numberShares)
     {
        double commission;

        if (sharePrice < 100.00)
        {
           commission = numberShares * 1.00;
        }
        else
        {
           commission = numberShares * 10.00;
        }
        totalCommission += commission;
     }
}
```

```
public class Bond extends Security
{
     public void calculateCommission(int numberShares)
     {
        double commission;

        commission = 10.00;
        totalCommission += commission;
     }
}
```

Since the calculations are specific to the type of security (Stock or Bond), the calculations are defined in each of the subclasses. Then, in the Security class, a variable that contains the total amount of commission generated for each Stock or Bond is defined. When a security is sold, the calculateCommission method is invoked. Note that a dummy calculateCommission method must be defined in the Security class to permit polymorphism.

```
public class Security
{
     protected String accountName;
     protected int numberShares;
     protected double sharePrice;
     protected double totalCommission;

     public Security(String accountName, int numberShares, double sharePrice)
     {
        this.accountName = accountName;
        this.numberShares = numberShares;
        this.sharePrice = sharePrice;
        totalCommission = 0.0;
     }

     public void sellSecurity(int numberSharesSold)
     {
        numberShares -= numberSharesSold;
        calculateCommission(numberSharesSold);
     }
     public void buySecurity(int numberSharesPurchased)
     {
         numberShares += numberSharesPurchased;
         calculateCommission(numberSharesPurchased);
     }
```

```
     public void calculateCommission(int numberSharesSold)
     {   // dummy method for polymorphism
     }
```

12.4.8 Iteration 8

Now that the commission for each sale as been determined, it is possible to calculate the total value of the portfolio at a given point in time.

The processing to compute the value of a specific security can be defined in the Security class (because it is identical for both stocks and bonds).

```
public double getSecurityValue()
{
     return (numberShares * sharePrice) - totalCommission;
}
```

The processing to determine the total value of the portfolio simply involves a loop over all securities and is performed in the Portfolio class.

```
public double getPortfolioValue()
{
   int currentElement;
   Security currentSecurity;
   double totalValue;

   totalValue = 0;
   for (currentElement=0; currentElement<portfolio.size(); currentElement++)
   {
     currentSecurity = (Security) portfolio.get(currentElement);
     totalValue += currentSecurity.getSecurityValue();
   }
   return totalValue;
}
```

12.4.9 Iteration 9

Since the price of stocks is volatile, a mechanism for changing the current share price of stocks needs to be added. (It is probably not necessary to change the share price of bonds but that facility will be available for free as a result of the other changes.)

```
public void changeSharePrice(String accountName, double sharePriceChange)
{
     Security currentSecurity;
     int found;

     found = findSecurity(accountName);
     if (found != -1)
     {
        currentSecurity = (Security) portfolio.get(found);
        currentSecurity.changeSharePrice(sharePriceChange);
     }
     else
     {
        System.out.println("Security " +accountName +" could not be found.");
     }
}
```

```
public class Security
{
     public void changeSharePrice(double sharePriceChange)
     {
        sharePrice += sharePriceChange;
     }
```

Note that the processing is quite simple given the good organization of the classes. A command to test changing the share price is added to the main class. The method changeSharePrice is added to the Portfolio class and then a simple method to change the instance variable is added to the Security class. So this change was accomplished in a few minutes without any problems and again is due to the fact that each type of processing is located in exactly one class.

12.4.10 Complete Stock Portfolio Example

The final version of the program is shown below. This program is by no means complete, there are many additional features that could be added. For example, maintaining the current value of the portfolio would be a good idea so that the user would know whether or not he/she has the funds available to make a new purchase.

```
import java.util.*;

public class Stocks
{
     public static void main(String[] parms)
     {
        Portfolio portfolio;

        portfolio = new Portfolio();
        processCommands(portfolio);
     }
```

```
      public static void processCommands(Portfolio portfolio)
      {
          portfolio.printPortfolio();
          portfolio.sellSecurity("Nortel", 50);
          portfolio.buySecurity("CSB", 100);
          portfolio.buySecurity("IBM", 100);
          portfolio.deleteSecurity("Microsoft");
          portfolio.changeSharePrice("Nortel", -3.00);
          portfolio.insertSecurity("Stock", "CloneMe", 100, 5.00);
          portfolio.insertSecurity("Bond", "T-Bill", 100, 5.00);
          portfolio.sellSecurity("CSB", 50);
          portfolio.printPortfolio();
      }
}
```

```
public class Portfolio
{
      private ArrayList portfolio;

      public Portfolio()
      {
          createPortfolio();
      }

      private void createPortfolio()
      {
          portfolio = new ArrayList();

          portfolio.add(new Stock("Nortel", 100, 5.00));
          portfolio.add(new Stock("IBM", 100, 500.00));
          portfolio.add(new Bond("CSB", 100, 100.00));
          portfolio.add(new Stock("Microsoft", 100, 0.05));
      }
```

```
      private int findSecurity(String accountName)
      {
          Security currentSecurity;
          int found;
          int currentElement;

          found = -1;
          for (currentElement=0; currentElement<portfolio.size()
                                  && found==-1; currentElement++)
          {
              currentSecurity = (Security) portfolio.get(currentElement);
              if (currentSecurity.compareAccountNames(accountName))
              {
                  found = currentElement;
              }
          }
          return found;
      }

      public void deleteSecurity(String accountName)
      {
          Security currentSecurity;
          int currentElement;
          int found;

          found = findSecurity(accountName);
          if (found != -1)
          {
              portfolio.remove(found);
              System.out.println("\nSecurity " +accountName +" was deleted.");
          }
          else
          {
              System.out.println("\nSecurity " +accountName
                                        +" does not exist.");
```

```
        }
    }
```

```
    public void insertSecurity(String securityType, String accountName,
                                          int numberShares, double sharePrice)
    {
        Security newSecurity;
        int found;

        found = findSecurity(accountName);
        if (found == -1)
        {
            if (securityType.equals("Stock"))
            {
                newSecurity = new Stock(accountName, numberShares, sharePrice);
            }
            else
            {
                newSecurity = new Bond(accountName, numberShares, sharePrice);
            }
            portfolio.add(newSecurity);
            System.out.println("\nSecurity " +accountName +" was inserted.");
        }
        else
        {
            System.out.println("\nSecurity " +accountName
                                          +" already exists.");
        }
    }

    public void sellSecurity(String accountName, int numberShares)
    {
        Security currentSecurity;
        int found;

        found = findSecurity(accountName);
        if (found != -1)
        {
            currentSecurity = (Security) portfolio.get(found);
            currentSecurity.sellSecurity(numberShares);
            System.out.println("\n" +numberShares
                               +" shares of security " +accountName +" were sold.");
        }
        else
        {
            System.out.println("Security " +accountName +" could not be found.");
        }
    }
```

```
    public void buySecurity(String accountName, int numberShares)
    {
        Security currentSecurity;
        int found;

        found = findSecurity(accountName);
        if (found != -1)
        {
            currentSecurity = (Security) portfolio.get(found);
            currentSecurity.buySecurity(numberShares);
            System.out.println("\n" +numberShares +" shares of security "
                                           +accountName +" were purchased.");
        }
        else
        {
            System.out.println("Security " +accountName +" could not be found.");
        }
    }
```

```
      public void changeSharePrice(String accountName, double sharePriceChange)
      {
         Security currentSecurity;
         int found;

         found = findSecurity(accountName);
         if (found != -1)
         {
            currentSecurity = (Security) portfolio.get(found);
            currentSecurity.changeSharePrice(sharePriceChange);
            System.out.println("\nThe price of security " +accountName
                                          +" was changed by " +sharePriceChange);
         }
         else
         {
            System.out.println("Security " +accountName +" could not be found.");
         }
      }

      public double getPortfolioValue()
      {
         int currentElement;
         Security currentSecurity;
         double totalValue;

         totalValue = 0;
         for (currentElement=0; currentElement<portfolio.size(); currentElement++)
         {
            currentSecurity = (Security) portfolio.get(currentElement);
            totalValue += currentSecurity.getSecurityValue();
         }
         return totalValue;
      }
```

```
      public void printPortfolio()
      {
         int currentElement;

         System.out.println();
         for (currentElement=0; currentElement<portfolio.size(); currentElement++)
         {
            System.out.println(portfolio.get(currentElement));
         }

         System.out.println("\nValue of portfolio is " +getPortfolioValue());
      }
}
```

```
public class Security
{
      protected String accountName;
      protected int numberShares;
      protected double sharePrice;
      protected double totalCommission;

      public Security(String accountName, int numberShares, double sharePrice)
      {
         this.accountName = accountName;
         this.numberShares = numberShares;
         this.sharePrice = sharePrice;
         totalCommission = 0.0;
      }

      public void sellSecurity(int numberSharesSold)
      {
         numberShares -= numberSharesSold;
         calculateCommission(numberSharesSold);
      }
      public void buySecurity(int numberSharesBought)
      {
          numberShares += numberSharesBought;
          calculateCommission(numberSharesBought);
      }

      public void changeSharePrice(double sharePriceChange)
      {
          sharePrice += sharePriceChange;
      }

      public void calculateCommission(int numberSharesSold)
      {   // dummy method for polymorphism
      }

      public double getSecurityValue()
      {
          double value;

          value = (numberShares * sharePrice) - totalCommission;
          return value;
      }
```

```
      public boolean compareAccountNames(String whichAccount)
      {
          return accountName.equals(whichAccount);
      }

      public String toString()
      {
          return accountName +" " +numberShares +" " +sharePrice;
      }
}
```

```
public class Stock extends Security
{
      public Stock(String accountName, int numberShares, double sharePrice)
      {
          super(accountName, numberShares, sharePrice);
      }

      public void calculateCommission(int numberShares)
      {
          double commission;

          if (sharePrice < 10.00)
          {
              commission = numberShares * 1.00;
          }
          else
          {
              commission = numberShares * 10.00;
          }
          totalCommission += commission;
      }
}
```

```
public class Bond extends Security
{
      public Bond(String accountName, int numberShares, double sharePrice)
      {
          super(accountName, numberShares, sharePrice);
      }

      public void calculateCommission(int numberShares, double sharePrice)
      {
          double commission;

          commission = 10.00;
```

```
            totalCommission += commission;
        }
}
```

12.4.11 Recap

This example illustrates how a program that utilizes object inheritance can be grown and refactored in the same manner as other programs. The only significant false step that was taken was during iteration 3 when the Portfolio class was expanded to handle stocks and bonds separately instead of recognizing that using two independent classes is not a good idea. Again, this is a situation in which stopping for a few minutes and considering alternative designs would likely have led directly to the object hierarchy used in iteration 4.

The data structure (an ArrayList) used to store the collection of stocks and bonds is separate from the Portfolio class. This is again an example of good object orientation.

The program does not read the initial portfolio information or the commands from files but adding that feature would be straight forward.

The program does not create a stock/bond object if the security is purchased but does not yet exist in the portfolio. This would be easy to fix.

Adding a new type of security is now trivial since most of the processing is performed in the Securities class. Only the calculation of the commission is specific to each subclass.

As a result of the design, the program could maintain any number of client portfolios at the same time. An additional class (Portfolios) could be used to manage the portfolios.

If you go back and take a close look at the program, you should not notice any particularly ugly (smelly) portions of code. The program consists of relatively small methods that do only one thing and this is evidence of a good design.

12.5 Summary

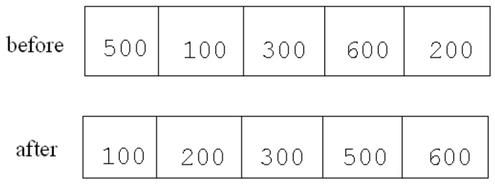
In this chapter we have examined the use of inheritance as programs become larger and more complex. Inheritance should be used judiciously – it works well in applications that have objects that are similar but not identical. It is a common beginner’s mistake to attempt to use inheritance in all programs: before you take advantage of inheritance, ensure that the objects being manipulated actually require inheritance. There is nothing wrong with defining several different types of objects that are not put together into an object hierarchy.

13 SORTING

13.1 Introduction

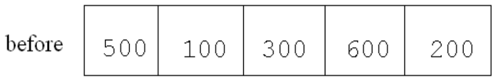
Sorting a collection of elements into ascending or descending order is one of the fundamental algorithms in Computer Science. In this chapter we examine several sorting algorithms.

The process of sorting elements is quite straightforward: we begin with a collection of elements that are not sorted and, after the sorting process is finished, the elements are sorted. We initially assume that we are sorting an array of int's into ascending order. However, we will see that we can sort any type of object into either ascending or descending order. In all of the sorting algorithms that we discuss, we will sort the elements "in place", that is, the array that is passed to the sorting method is modified so that when the method completes, the elements in the original array are correctly sorted.

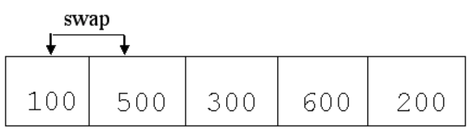


13.2 Bubble Sort

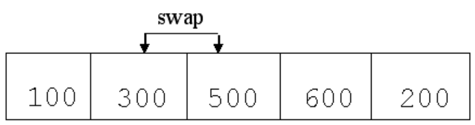
The bubble sort is arguably the simplest type of sorting algorithm. We begin with the unsorted array.



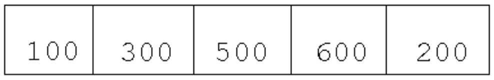
We compare the element in position 0 with the element in position 1. If they are already in ascending order, we do nothing; if they are not in ascending order, we interchange (or swap) the two elements so that the first two elements are now in ascending order.



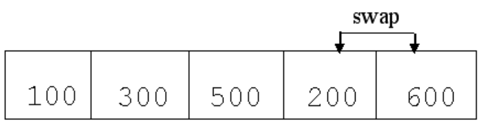
We now continue this process with the elements in position 1 and position 2. These two elements are not in the correct order so we swap the two elements.



Next we compare the elements in position 2 and position 3. These two elements are in the correct order so we leave them as they are.



Finally, we compare the last two elements and swap them since they are not in the correct order.

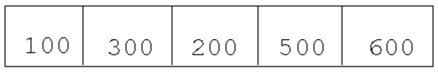


The code to perform this process is shown below.

```
for (count2=0; count2<list.length-1; count2++)
{
     if (list[count2] > list[count2+1])
     {
        temp = list[count2];
        list[count2] = list[count2+1];
        list[count2+1] = temp;
     }
}
```

So, after we have made one pass through the array, what can we say about what we have accomplished? We compared each pair of adjacent elements and interchanged them if they were not in ascending order. The result is that we have pushed the largest element to the end of the list. We can not say anything about the remaining elements, the process may have improved the ordering of the other elements but we can not know that for certain.

Now that we have the largest element in its correct position, what will happen if we repeat the process again? If we repeat the instructions above, the array will have the following organization.



Now the largest two elements are in their correct locations. So, if we repeat the instructions enough times, we will eventually move all of the array elements into their correct locations. The following method sorts the entire array using a bubble sort.

```
public static void bSort(int[] list)
{
     int count1;
     int count2;
     int temp;
     for (count1=0; count1<list.length; count1++)
     {
        for (count2=0; count2<list.length-1; count2++)
        {
           if (list[count2] > list[count2+1])
           {
              temp = list[count2];
              list[count2] = list[count2+1];
              list[count2+1] = temp;
           }
        }
     }
}
```

If we examine the method above, it should be obvious that there are some inefficiencies that can be corrected with very little effort. The inner loop compares adjacent elements for elements 0 to N-1 each time through the loop. This is not necessary since after each completion of the inner loop, one more element is in its correct location and does not need to be included in the comparisons. Therefore, we could improve the inner loop by using the following statement:

```
for (count2=0; count2<list.length-count1-1; count2++)
```

Similarly, the outer loop is executed N times. This is one execution more than is necessary because after N-1 executions of the inner loop, the largest N-1 elements have been moved into their correct locations and so the last element must also be in its correct location. So a slightly more efficient version of the bubble sort is shown below.

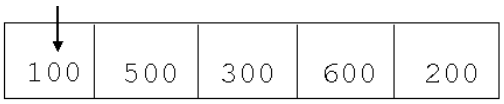
```
public static void bSort(int[] list)
{
     int count1;
     int count2;
     int temp;
     for (count1=0; count1<list.length-1; count1++)
     {
        for (count2=0; count2<list.length-count1-1; count2++)
        {
           if (list[count2] > list[count2+1])
           {
              temp = list[count2];
              list[count2] = list[count2+1];
              list[count2+1] = temp;
           }
        }
     }
}
```

Recall that our general rule is that the efficiency of algorithms is not a major concern as algorithms are developed. However, if minor changes to an algorithm can improve the speed of the algorithm significantly, then it is worth while making those changes as long as the changes do not obscure the clarity of the code.

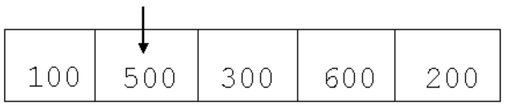
13.3 Selection Sort

The selection sort is very similar to the bubble sort except that instead of interchanging adjacent elements, we simply keep track of the location of the largest element so far.

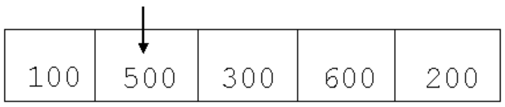
To begin, we point to element 0 as the largest element encounted so far.



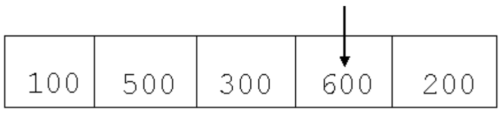
We then compare the largest element so far with element 1. Since the second element is larger, we change the pointer to the largest element to point to the second element.



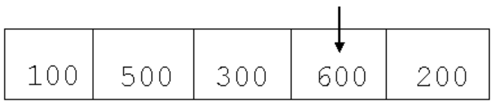
Next, we compare the largest element so far with element 2. Since 300 is less than 500, we don’t need to make any changes.



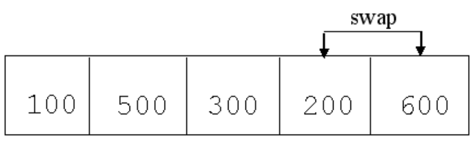
Next, we compare the largest element so far with element 3. Since 500 is less than 600, we must change our pointer to the largest element so that it points at element 3.



Finally, we compare the largest element so far with element 4. Since 600 is greater than 200, there is no need to change the pointer to the largest element.



Now we have identified the largest element but it is not in the correct location. The final step is to swap the largest element with the element in the last position. Note that we are actually swapping the elements this time, not just changing a pointer.



The instructions required to perform this processing are shown below. Note that inside the loop, we simply reset the pointer to the largest element and then swap the elements outside of the inner loop.

```
largest = 0;
for (count2=1; count2<list.length; count2++)
{
     if (list[largest] < list[count2])
     {
        largest = count2;
     }
}
temp = list[list.length-1];
list[list.length-1] = list[largest];
list[largest] = temp;
```

After one pass through the array, we have the same situation as we did with the bubble sort – the largest element has been moved to the end of the array. If we repeat this process, we will eventually have all elements sorted.

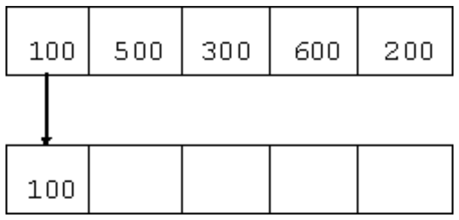
```
public static void sSort(int[] list)
{
     int count1;
     int count2;
     int largest;
     int temp;

     for (count1=0; count1<list.length-1; count1++)
     {
        largest = 0;
        for (count2=1; count2<list.length-count1; count2++)
        {
           if (list[largest] < list[count2])
           {
              largest = count2;
           }
        }
        temp = list[list.length-1-count1];
        list[list.length-1-count1] = list[largest];
        list[largest] = temp;
     }
}
```

Note that as with the bubble sort, the outer loop only needs to be executed N-1 times. Also, the inner loop **must not** continue until the end of the array. With the bubble sort, this was just an **efficiency** concern; however, with the selection sort, not including the elements that have already been sorted is a **correctness** concern. If we continue changing the pointer to the largest element until the end of the entire array, we will find the same element each time and so will not end up sorting the array.

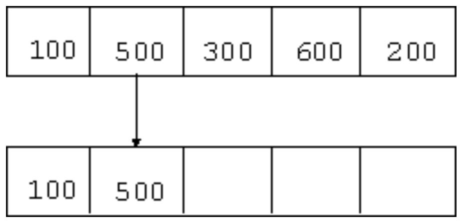
13.4 Insertion Sort

An insertion sort is different from the previous two sorts – it does not attempt to locate the largest element on each pass. Instead, beginning with the first element in the array, each element in the original array is moved into the correct (sorted) location in a new array.

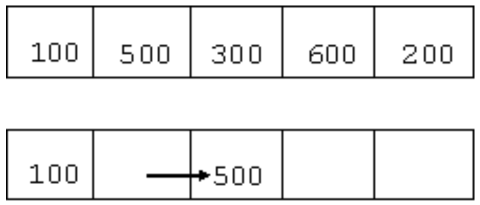
The diagram below shows how the process starts. Element 100 is moved into location 0 in the new, empty array.



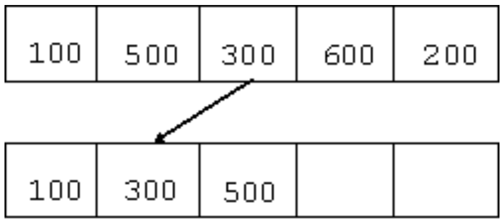
On the next pass, beginning at the end of the new array, each element is moved one position to the right until an element that is smaller than the element being inserted is found. In the diagram below, since 500 is greater than 100, element 500 is moved into location 1 of the new array.



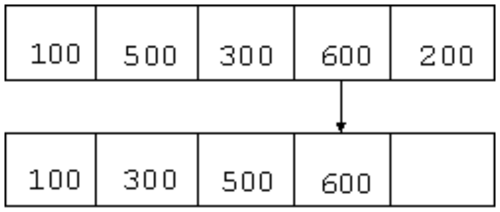
When element 300 is processed, it must be inserted before element 500 in the new array so element 500 is moved one position to the right in the new array.



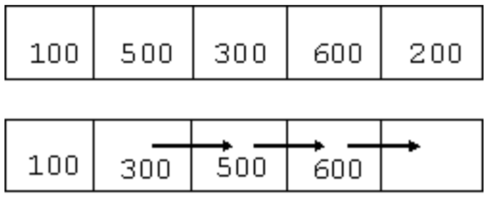
Element 300 is greater than the next element in the new array so now there is room to insert element 300 into the correct location in the new array.



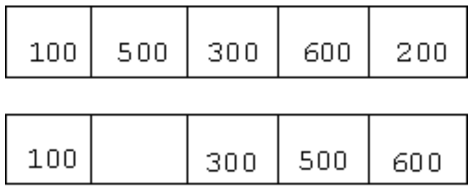
Next, element 600 is processed. Since it is larger than the last element in the new array, it is copied into the next available position in the new array.



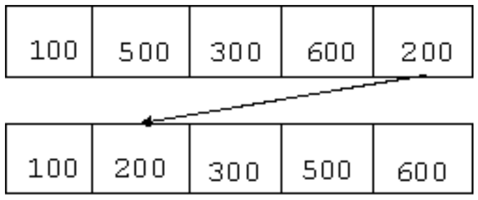
Finally, element 200 is processed. We need to move 3 elements in the new array one position to the right until there is room for element 200 in its correct position. This involves moving 600, then 500, then 300 one position to the right.



So, now the new array looks like the following:



and we can now move element 200 into its correct location in the new array.



The method to perform an insertion sort is shown below. After the new array has been correctly sorted, it's contents are copied back to the original array.

```
public static void iSort(int[] list1)
{
     int count1;
     int count2;
     int element;
     int[] list2;

     if (list1.length >= 1)
     {
        list2 = new int[list1.length];
        list2[0] = list1[0];
     }
     else
     {  // no elements in the array
        list2 = new int[0];
     }
     for (count1=1; count1<list1.length; count1++)
     {
        element = list1[count1];
        for (count2=count1; count2>0&&list2[count2-1]>element; count2--)
        {
           list2[count2] = list2[count2-1];
        }
        list2[count2] = element;
     }
     System.arraycopy(list2, 0, list1, 0, list1.length);
}
```

If you examine the algorithm closely, you should realize that the extra array is not required – since the element to be moved into sorted position is extracted from the array, there is always room at the beginning of the original array for the sorted elements. The following method illustrates this improved insertion sort.

```
public static void iSort(int[] list)
{
     int count1;
     int count2;
     int element;

     for (count1=1; count1<list.length; count1++)
     {
        element = list[count1];
        for (count2=count1; count2>0 && list[count2-1]>element; count2--)
        {
           list[count2] = list[count2-1];
        }
        list[count2] = element;
     }
}
```

13.5 Sorting an Array of Strings

In the 3 sorting methods described above, the arrays being sorted only contained int values. In this section, we modify the selection sort method so that it can sort an array of strings. Once you can sort an array of strings, it is a small step to sort an array of any type of object.

```
public static void sSortStrings(String[] list)
{    //Selection sort
     int count1;
     int count2;
     int largest;
     String temp;

     for (count1=0; count1<list.length-1; count1++)
     {
        largest = 0;
        for (count2=largest+1; count2<list.length-count1; count2++)
        {
           if (list[largest].compareTo(list[count2])<0)
           {
              largest = count2;
           }
        }
        temp = list[list.length-1-count1];
        list[list.length-1-count1] = list[largest];
        list[largest] = temp;
     }
}
```

As can be seen from this example, only 3 statements had to be modified. When sorting objects, the objects being sorted must contain a `compareTo` method that returns the result of comparing two objects.

13.6 Sorting into Descending Order

The sorting algorithms examined in this section have all sorted the elements into ascending order since ascending order is normally required. However, sorting elements into descending order requires only one minor modification to the algorithm.

The operator in the comparison statement must be reversed. The < operator is changed to the > operator or the > operator is changed to the < operator. (The operator that is used depends on the algorithm.)

```
primitive data types:

Selection sort ascending         if (list[largest] < list[count])
Selection sort descending        if (list[largest] > list[count])


object data types:

Selection sort ascending:        if (list[largest].compareTo(list[count])<0)
Selection sort descending:       if (list[largest].compareTo(list[count])>0)
```

13.7 Summary

In this chapter we examined 3 of the simplest (and slowest) sorting algorithms. Two faster algorithms, the merge sort and quicksort, are examined in Chapter 19 – Miscellaneous Topics.

14 RECURSION

14.1 Introduction

Until now, any statements that had to be repeated were controlled by one of two **iterative** control structures, a *for* statement or a *while* statement. In this chapter we examine an alternative to iterative processing, **recursion**.

14.2 Iteration

Iteration is the term used to refer to using a loop control structure that executes a statement or statements until a condition becomes true. For example, n! (n factorial) is defined as:

$$n! = n \times (n-1) \times (n-2) \times \ldots \times 3 \times 2 \times 1 \quad \text{for } n \geq 1$$

N factorial can easily be computed in an iterative manner as follows:

```
public static int fact(int num)
{
     int count;
     int result;
     result = 1;
     for (count=2; count<=num; count++)
     {
        result = result * count;
     }
     return result;
}
```

Similarly, m^n (for $n \geq 0$, where n is an integer and m is an integer) can also be computed in an iterative manner:

```
public static int power(int num, int power)
{
     int count;
     int result;
     result = 1;
     for (count=1; count<=power; count++)
     {
        result = result * num;
     }
     return result;
}
```

The general structure of an iterative method is to have some initialization before the loop, some processing within the loop, and possibly some final processing after the loop.

14.3 Recursion

For many problems, computing a solution using iterative techniques with for and/or while loops works well. After all, you have survived with only iterative structures until now! However, most programming languages also provide an alternative to iterative repetition structures, the recursive method. A recursive method is a method that calls itself (either directly or indirectly) and, by doing so, executes statements repeatedly but without the need for explicit loop control structures.

Our original definition of n! was:

$$n! = n \times (n-1) \times (n-2) \times \ldots \times 3 \times 2 \times 1 \quad \text{for } n \geq 1$$

however, n! can also be defined as:

$$n! = n \times (n-1)! \quad \text{for } n \geq 2 \text{ where } 1! = 1$$

The following method computes n! using the second definition and recursive calls instead of using iteration.

```
public static int fact(int num)
{
     int result;

     if (num < 2)
     {
        result = 1;
     }
     else
     {
        result = num * fact(num - 1);
     }
     return result;
}
```

If the statement fact(1) is executed, since num is less than 2, the value result=1 is returned, and processing is complete. However, if fact(2) is executed, then the statement:

```
        result = num * fact(num - 1);
```

becomes

```
        result = 2 * fact(2 - 1);
```

becomes

```
        result = 2 * fact(1);
```

is executed. This causes the *fact* method to be called **again** with a parameter of 1. fact(1) returns the value 1 so the statement above becomes:

```
         result = 2 * 1;
```

which becomes

```
         result = 2;
```

Thus, a result of 2 is returned to the calling method.

Similarly, if fact(3) is executed,

```
         result = 3 * fact(3 - 1);
```

```
         result = 3 * fact(2);
```

causes fact(2) to be executed, and, as we have seen, fact(2) causes fact(1) to be executed. fact(1) returns 1 so fact(2) returns 2 and the expression becomes:

```
         result = 3 * 2;
```

and a result of 6 is returned for 3!.

The recursive version of factorial is very similar to the iterative version but the processing is handled in quite a different manner. In the iterative version, a loop controls the processing while in the recursive version, the method keeps calling itself until the condition defined at the beginning of the method becomes true and the recursion terminates.

A recursive method normally consists of one or more base (or terminal or stopping) cases and one or more recursive cases. A base case returns a result directly without calling the method; a recursive case calls the method again in order to determine the result.

```
public static int fact(int num)
{
     int result;

     if (num < 2)                           // base case: num < 2
     {
        result = 1;
     }
     else
     {
        result = num * fact(num - 1);   // recursive case: num >= 2
     }
     return result;
}
```

The base case for n! is n! = 1 for n = 1; the recursive case is n! = n × (n-1)! for n ≥ 2.

The recursive version of m^n is:

```
public static int power(int num, int power)
{
     int result;
     result = 1;
     if (power >= 1)
     {
        result = num * power(num, power-1);
     }
     return result;
}
```

As can be seen, this method is almost identical to the recursive calculation of n!, with a base case of $m^n = 1$ for $n = 0$ and a recursive case of $m^n = m \times m^{n-1}$ for $n \geq 1$.

In general, an **iterative** solution consists of a loop that iterates over the values to be processed:

```
initialization;
for (count=0; count < maxValue; count++)
{
     do something;
}
```

while a **recursive** solution typically consists of a decision (base case) that determines whether or not it is time to stop processing and then a recursive case that continues the recursion by calling the method again:

```
if (time to stop)
{
     result = …;
}
else
{
     call method again
}
```

If the base case is not specified correctly, then the recursion will likely create an infinite loop in which the method is called repeatedly until the Java virtual machine runs out of memory and generates a stack overflow error. Thus, the following method attempts to compute n! but never terminates if $n \leq 2$.

```
public static int fact(int num)
{
     int result;
     if (num > 2)
     {
        result = 1;
     }
     else
     {
        result = num * fact(num - 1);
     }
     return result;
}
```

```
java.lang.StackOverflowError
     at Recursion.fact(Recursion.java:115)
     at Recursion.fact(Recursion.java:115)
     at Recursion.fact(Recursion.java:115)
     at Recursion.fact(Recursion.java:115)
     at Recursion.fact(Recursion.java:115)
     at Recursion.fact(Recursion.java:115)
     at Recursion.fact(Recursion.java:115)
     . . .
```

Any time that a **StackOverflowError** is generated, it is almost always the result of an incorrectly specified recursive method.

14.4 Processing Strings

String processing is a natural candidate for recursion since many string processing routines require that the same steps be repeated until there are no more characters remaining to be processed.

14.4.1 String Reverse

The following iterative method accepts a String and returns the characters in the String in the reverse order (i.e. backwards).

```
public static String stringReverse(String string)
{
     String result;
     int count;
     result = "";
     for (count=string.length()-1; count>=0; count--)
     {
        result += string.substring(count, count+1);
     }
     return result;
}
```

The following recursive method performs the same processing.

```
public static String stringReverse(String string)
{
     String result;
     if (string.equals(""))
     {
        result = "";
     }
     else
     {
        result = string.substring(string.length()-1);
        result += stringReverse(string.substring(0, string.length()-1));
     }
     return result;
}
```

This method extracts the last character from the String, and then calls stringReverse again to extract the remaining characters (in reverse order) from the rest of the String. In this example, each call to stringReverse causes a new copy of all local variables (result in this case) be created. This will be discussed in more detail later in this chapter.

The following recursive method generates the same result using a slightly different technique.

```
public static String stringReverse(String string)
{    //Another recursive String reverse
     String result;

     if (string.equals(""))
     {
        result = "";
     }
     else
     {
        result = string.substring(0,1);
        result = stringReverse(string.substring(1)) + result;
     }
     return result;
}
```

14.4.2 Palindromes

A palindrome is a collection of characters that read the same backwards as they do forwards. The following method illustrates how a String can be examined to determine whether or not it is a palindrome (this version is case sensitive). This method has two base cases. The first base case determines whether or not there are zero or one characters remaining in the String: if so, then the String is a palindrome. The next base case compares the first character in the String with the last character; if they are not equal, then the String is not a palindrome. Finally, the recursive case removes the first and last characters and calls the method again.

```
public static boolean palindrome(String string)
{
     boolean result;

     if (string.length() <= 1)
     {
        result = true;
     }
     else if (string.charAt(0) != string.charAt(string.length()-1))
     {
        result = false;
     }
     else
     {
        result = palindrome(string.substring(1,string.length()-1));
     }
     return result;
}
```

14.4.3 String Padding

The following method accepts a String and an int length. The method returns a String that is at least length characters long. If the original String is less than length characters long, the String is padded one character at a time, recursively, on the right with a "+" until the String is exactly length characters long. For example, the parameters "Linux", 7 would cause the String "Linux++" to be returned whereas the parameters "trot", -1 would cause the String "trot" to be returned. (What modification is required to have the method pad on the left instead of on the right?)

```
public static String pad(String string, int length)
{
     String result;
     if (string.length()>=length)
     {
        result = string;
     }
     else
     {
        result = pad(string + "+", length);
     }
     return result;
}
```

14.4.4 Number Conversions

The following method converts numbers from base 10 to base 2. To convert a decimal value to its equivalent binary representation, divide the decimal value by 2. The remainder of this division is either zero or one. This value becomes the right-most digit of the binary value. Continue the process of dividing by 2 and taking the remainder until the value (the dividend) is zero.

```
public static String convert(int value)
{    // base 2 conversion
     if (value < 2)
     {
        return "" +value;
     }
     else
     {
        return convert(value/2) + value%2;
     }
}
```

The following output illustrates the results of the method when applied to the decimal (base 10) values 0, 5, 1024, and 2047.

```
   0  000000000000
   5  000000000101
1024  010000000000
2047  011111111111
```

The method above converts base 10 values to base 2 values. In computer science, it is often necessary to work with numbers in other bases, such as base 8 and base 16. The following method converts a value that is in base 10 to any base from base 2 to base 16.

```
public static String convert(int value, int base)
{    // convert from base 10 to any base from 2 to 16

     String[] values = {"0","1","2","3","4","5","6","7",
                        "8","9","A","B","C","D","E","F"};

     if (value < base)
     {
        return values[value];
     }
     else
     {
        return convert(value/base, base) + values[value % base];
     }
}
```

The following output illustrates the results of the revised method when converting the decimal (base 10) values 0, 5, 1024, and 2047 first to base 2, then to base 8, and finally to base 16.

```
 2     0  000000000000
 2     5  000000000101
 2  1024  010000000000
 2  2047  011111111111

 8     0  000000
 8     5  000005
 8  1024  002000
 8  2047  003777

16     0  0000
16     5  0005
16  1024  0400
16  2047  07FF
```

14.5 Processing Arrays

We can also use recursive methods to perform array processing. The following method performs a linear search of the contents of an array for an array element that contains the value *search*.

```
public static boolean searchList(int[] list, int search)
{
     boolean result;
     int[] list2;

     if (list.length <= 0)
     {
        result = false;
     }
     else if (list[0] == search)
     {
        result = true;
     }
     else
     {
        list2 = new int[list.length - 1];
        System.arraycopy(list, 1, list2, 0, list2.length);
        result = searchList(list2, search);
     }
     return result;
}
```

Once again there are two base cases: the first to determine if the array is empty, the second to compare the first element in the array to the search value. If neither base case succeeds, the recursive portion of the method is executed. In this search method, the first element in the array is removed by creating a new array that is one element smaller than the original array and then copying all of the elements of the first array, except the first element, into the new array.

Up until now, the recursive algorithms looked fairly elegant but this one is quite ugly. There should be a better way to search for an element in an array than having to remove the first element in the array during each recursion. What we need to be able to do is to reference any element in the array during the recursion, not just the first element. We can do this if we add an additional parameter to the searchList method. This parameter specifies which element we are currently examining. Since we don't want the user to have to specify any parameters that the user does not control or understand, we add a simple method in between the user's program and the searchList method that actually does the work. This technique is very common – it is referred to as adding an **interface** between two portions of a program in order to reduce the complexity of the user's program. (In this context, an interface is a design pattern and is not related to the Java interface class.) The interface method has the same name and it simply adds the starting position in the array to the list of parameters.

```
public static boolean searchList(int[] list, int search)
{
     return searchList(list, search, 0);
}
```

```
public static boolean searchList(int[] list, int search, int position)
{
     boolean result;
     if (position >= list.length)
     {
        result = false;
     }
     else if (list[position] == search)
     {
        result = true;
     }
     else
     {
        result = searchList(list, search, position+1);
     }
     return result;
}
```

This version of searchList is much simpler and more satisfying than the previous version that created a new array. Each time through searchList, if the element is not found, the method simply increments the variable that indicates which position is to be examined next.

Be careful when using recursion that involves incrementing a count: using `position+1` in the algorithm above is correct; using `position++` in the algorithm is not correct. (Why?)

The use of an appropriate interface method that adds a parameter to the parameter list is an important programming technique. It ensures that the user does not have to be aware of the extra parameter (and possibly define it incorrectly) and also ensures that if this searchList method is replaced at some time in the future with a different version that requires only two parameters (for example, by the equivalent iterative version), then the programs that access searchList will not have to be modified since they were never aware of the extra parameter.

Our printList method defined earlier prints the contents of a list iteratively.

```
public static void printList(int[] list)
{
     int count;

     System.out.println();
     for (count=0; count<list.length; count++)
     {
        System.out.print(list[count] +"\t");
     }
     System.out.println();
}
```

This method can be rewritten to use recursion as shown below. This method again uses an interface to add a position parameter.

```
public static void printList(int[] list)
{
     printList(list, 0);
}
```

```
public static void printList(int[] list, int position)
{
     if (position < list.length)
     {
        System.out.print(list[position] +"\t");
        printList(list, position+1);
     }
}
```

If we wanted to print the elements in reverse order, all that we need do is reverse the two statements inside the if statement.

```
public static void printListReverse(int[] list, int position)
{
     if (position < list.length)
     {
        printListReverse(list, position+1);
        System.out.print(list[position] +"\t");
     }
}
```

14.6 Binary Search

The recursive version of a binary search also uses an interface method to define the initial search parameters: the search begins with the endpoints being defined as the first and last elements in the list.

```
public static int searchList(int[] list, int search)
{
     return binarySearch(list, search, 0, list.length-1);
}

public static int binarySearch(int[] list, int search, int left, int right)
{
     int middle;
     int result;
     result = -1;
     middle = left + ((right-left)/2);
     if (left > right)
     {
        result = -1;
     }
     else if (search == list[middle])
     {
        result = middle;
     }
     else if (search < list[middle])
     {
        result = binarySearch(list, search, left, middle-1);
     }
     else
     {
        result = binarySearch(list, search, middle+1, right);
     }
     return result;
}
```

Note that this recursive version is just as easy to understand (if not easier) than the iterative version.

14.7 Recursive Merge

The recursive version of a the merge algorithm examined in Chapter 2 – Growing Algorithms is presented below. Again, an interface method is used to add parameters that are required by the merge method.

```
public static int[] mergeLists(int[] list1, int[] list2)
{
      int[] result;

      result = new int[list1.length+list2.length];
      mergeLists(result, list1, list2, 0, 0);
      return result;
}

public static void mergeLists(int result[],int[] list1,int[] list2,int count1,int count2)
{     //Fills the array "result" from the left
      if ((count1<list1.length) &&
                ((count2>=list2.length) || (list1[count1]<list2[count2])))
      {
          result[count1+count2] = list1[count1];
          mergeLists(result, list1, list2, ++count1, count2);
      }
      else if (count2<list2.length)
      {
          result[count1+count2] = list2[count2];
          mergeLists(result, list1, list2, count1, ++count2);
      }
}
```

14.8 Processing ArrayLists

An ArrayList may be processed in the same manner as an array. The following method prints the elements of an ArrayList. As each element is printed, it is removed from the ArrayList and the printList method is called again with the shorter ArrayList.

```
public static void printList(ArrayList arrayList)
{
      if (arrayList.size() > 0)
      {
          System.out.print(arrayList.get(0) +"\t");
          arrayList.remove(0);
          printList(arrayList);
      }
}
```

This method works correctly but has an undesirable side effect: it destroys the contents of the ArrayList. After the elements of the ArrayList have been printed, the ArrayList is empty because each element is deleted after processing. **(It is important to understand why this happens!)** Therefore, this technique is not particularly useful. Instead, we can add a parameter that indicates which element of the ArrayList is to be processed next.

```
public static void printList(ArrayList arrayList)
{
     printList(arrayList, 0);
}
```

```
public static void printList(ArrayList arrayList, int location)
{
     if (location < arrayList.size())
     {
        System.out.print(arrayList.get(location) +"\t");
        printList(arrayList, location+1);
     }
}
```

14.9 Fibonacci Numbers

The generation of Fibonacci numbers is one of the classic examples of recursion. The Fibonacci numbers are defined as:

$$F(n) = F(n-1) + F(n-2) \quad \text{for } n \geq 3; \text{ and } F(1) = 1 \text{ and } F(2) = 1$$

(F(0) is often defined as 0 but we will assume that we only have to compute values for $n \geq 1$.) The iterative version is shown below.

```
public static int fib(int num)
{
     int count;
     int fib1;
     int fib2;
     int temp;

     fib1 = 0;
     fib2 = 1;
     for (count=2; count<=num; count++)
     {
        temp = fib1 + fib2;
        fib1 = fib2;
        fib2 = temp;
     }
     return fib2;
}
```

Programs that illustrate the calculation of Fibonacci numbers often eliminate the temporary variable by changing the loop to:

```
     for (count=2; count<=num; count++)
     {
        fib2 = fib1 + fib2;
        fib1 = fib2 - fib1;
     }
```

This version eliminates the execution of one statement each time through the loop but this is another example of refactoring that does not improve the readability of the method and so is not an improvement over the previous version.

The recursive version is shown below. Note this version is easier to understand than the iterative version. (Unlike the iterative version, this version returns the result F(0)=0.)

```
public static int fib(int num)
{
     int fib;

     fib = 0;
     if ((num == 1) || (num == 2))
     {
        fib = 1;
     }
     else if (num > 2)
     {
        fib = fib(num - 1) + fib(num - 2);
     }
     return fib;
}
```

If you run the recursive Fibonacci program, you will notice that the calculation of the larger Fibonacci numbers takes longer and longer.

The following statistics show the time that it takes to compute the Fibonacci numbers from 40 to 45 using a recursive algorithm. The time is displayed in milliseconds so 5288 represents 5 seconds, 288 milliseconds. Notice that the time required for the recursive algorithm almost doubles as the next number in the sequence is computed. (The use of timers is discussed in Chapter 19 – Miscellanous Topics.)

```
Recursive Fibonacci: F(40) = 102334155, total time in ms: 5288

Recursive Fibonacci: F(41) = 165580141, total time in ms: 8653

Recursive Fibonacci: F(42) = 267914296, total time in ms: 13910

Recursive Fibonacci: F(43) = 433494437, total time in ms: 23083

Recursive Fibonacci: F(44) = 701408733, total time in ms: 39066

Recursive Fibonacci: F(45) = 1134903170, total time in ms: 58845
```

The following statistics show the time required to compute the Fibonacci numbers using the iterative method. Note that the time is so small that it registers as 0 milliseconds (meaning that it took less than 1 millisecond to perform the calculation).

```
Iterative Fibonacci: F(40) = 102334155, total time in ms: 0

Iterative Fibonacci: F(41) = 165580141, total time in ms: 0

Iterative Fibonacci: F(42) = 267914296, total time in ms: 0

Iterative Fibonacci: F(43) = 433494437, total time in ms: 0

Iterative Fibonacci: F(44) = 701408733, total time in ms: 0

Iterative Fibonacci: F(45) = 1134903170, total time in ms: 0
```

If a count is added to the recursive version of the Fibonacci method, 331,160,281 recursive calls are required to compute F(40). Obviously, using recursion to compute the Fibonacci numbers is not a good idea. Do you understand why this is true?

14.10 Processing Objects

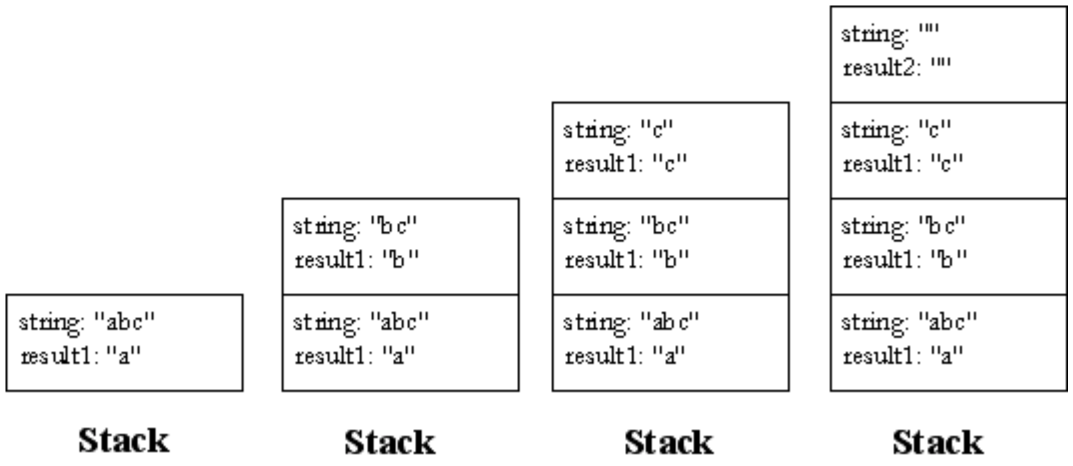
Objects can be processed recursively in the same manner that a primitive data type can be processed. The following example illustrates how an array of Flight objects can be searched recursively for the flight with a specific flight number.

```
public static String searchFlightNumber(Flight[] flights, String flight)
{
     return searchFlightNumber(flights, flight, 0);
}
```

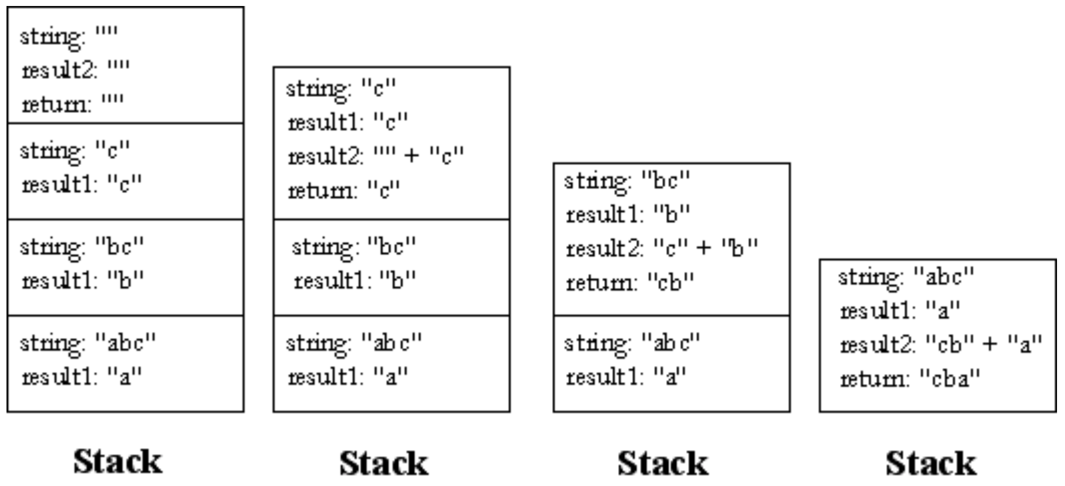
```
public static String searchFlightNumber(Flight[] flights,
                                              String flight,  int location)
{
     String result;
     if (location >= flights.length)
     {
        result = null;
     }
     else if (flights[location].compareFlightNumbers(flight))
     {
        result = flights[location].toString();
     }
     else
     {
        result = searchFlightNumber(flights, flight, location+1);
     }
     return result;
}
```

14.11 The Recursion Stack

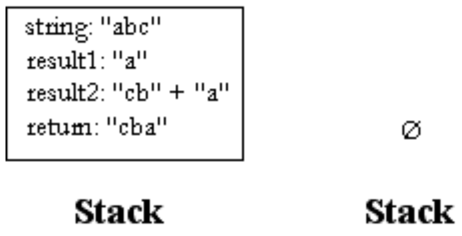
When a method is called, Java requires memory for all variables that are defined within the method. If recursion is used, each time that a method is called recursively, Java uses additional memory for all of the local variables. Thus, each recursive call creates a new version of all local variables. The copies of the versions of the local variables are stored in a data structure referred to as a **stack**. A stack is simply an area of memory which contains the local variables for a method. As a method is called recursively, new memory is allocated on the top of the stack. The diagram below illustrates how the memory required for the stringReverse method is added to the stack. The first time that stringReverse is called, memory is allocated on the stack. If stringReverse is called recursively, more memory is allocated. This continues as long as stringReverse is called recursively.



When the recursion terminates and the results are collected, the memory that was used by the current recursion is removed from the top of the stack.



When recursion ends, the last portion of memory allocated to the recursive method is removed from the stack and the result is returned to the calling method.



The following method has some print statements included that trace the recursive calls.

```
public static String stringReverse(String string)
{
      String result;
      recursionCount++;
      if (string.equals(""))
      {
          result = "";
      }
      else
      {
          result = string.substring(0,1);
          System.out.println("Start level " +recursionCount +", result: <" +result +">");
          result = stringReverse(string.substring(1)) + result;
          recursionCount--;
          System.out.println("End   level " +recursionCount +", result: <" +result +">");
      }
      return result;
}

String reverse of: abc

Start level 1, result: <a>
Start level 2, result: <b>
Start level 3, result: <c>
End   level 3, result: <c>
End   level 2, result: <cb>
End   level 1, result: <cba>

Reversed string: cba
```

As can be seen from the trace, each time a recursive call is made, the last character is extracted from the string and stored in the variable result. After 3 recursions, there are no more characters so the recursion begins working backward, taking the current value of result and adding to it the value returned by the previous recursive call. After all of the recursive calls have been processed, result contains the original string in reverse order.

It is because new memory is allocated on the stack by each recursive call that if a recursion never ends, Java issues a java.lang.StackOverflowError message when there is no more memory left in the stack.

14.12 Iteration versus Recursion

Most students feel more comfortable writing iterative methods than recursive methods because the programmer has more control over the processing in an iterative method. However, it is important to understand how to write recursive methods because some problems can be solved easily with a recursive method while using an iterative method would require a significant amount of work. In fact, some languages, such as Prolog, do not support iterative control structures at all so all loops must be written recursively. But Prolog is unusual and most languages support both iteration and recursion. So how do you decide which technique is better for a particular problem (assuming that you have not been told that the problem must be solved using a specific technique)?

There are two primary criteria that should be examined when determining whether an algorithm should be written recursively or iteratively. The first is the complexity of writing the algorithm; the second is the cost of executing the algorithm.

For the algorithms examined in this chapter, the iterative version and the recursive version are of equal difficulty to write. However, once more complex algorithms are required, recursion may be a better choice. For example, when processing a "tree" structure, a recursive algorithm is normally much easier to write. (See Chapter 19 – Miscellanous Topics for the file directory processing example.)

The second criterium is the cost of executing the resulting algorithm. Normally iterative methods and recursive methods execute in approximately the same amount of time. A recursive method may require a small amount of additional time due to the processing required to add information to and remove information from the stack. However, there are some algorithms, such the Fibonacci numbers, that are very expensive to compute recursively. If this is the situation, then iteration should be used.

If both criteria are equal, then it is really the programmer's choice as to which technique is used (and most programmers favour iteration since they have likely done more programming with iteration). For more complex data structure processing (such as a tree or graph traversal), recursion is almost always the best choice.

14.13 Recursion Examples

The following are some additional examples of using recursion.

14.13.1 Palindrome

The following algorithm is an improved version of the palindrome algorithm described earlier in this chapter.

```
public static boolean palindrome(String string)
{
     return palindrome(string, 0);
}
```

```
public static boolean palindrome(String string, int position)
{
     boolean result;

     if (position > (string.length()/2))
     {
        result = true;
     }
     else if (string.charAt(position) != string.charAt(string.length()-position-1))
     {
        result = false;
     }
     else
     {
        result = palindrome(string, position+1);
     }
     return result;
}
```

14.13.2 Reversing Words

The following methods take a string, split the contents of the string into its component words (using the split method), and then return the words in reverse order in one String.

```
public static String reverseWords(String string)
{
     String[] words;
     words = string.split("\\s+");
     return reverseWords(words, 0);
}
```

```
public static String reverseWords(String[] words, int position)
{
     String result;
     if (position >= words.length)
     {
        result = "";
     }
     else
     {
        result = reverseWords(words, position+1) +words[position] +" ";
     }
     return result;
}
```

14.13.3Filling an Array

The following algorithm fills an array recursively. The elements are assigned values using an algorithm that is similar to a binary search: the middle element is assigned a value; then the middle element of the elements on the left is assigned a value and the middle element of the elements on the right is assigned a value. What can be said about the values that are assigned to the array?

```
public static void fill(int left, int[] list, int right, int value1, int value2)
{
     int mid;

     mid = left + ((right-left)/2);
     if ((left <= mid) && (mid <= right))
     {
        list[mid] = value1 + (int) (((value2-value1)+1) * Math.random());
        fill(left, list, mid-1, value1, list[mid]);
        fill(mid+1, list, right, list[mid], value2);
     }
}
```

14.13.4 Precomputing Values

We noticed earlier in this chapter that computing Fibonacci numbers recursively is a very time-consuming process since the same Fibonacci numbers keep being recalculated. One technique that can be used to reduce the cost of such problems is to **cache** or precompute some of the values. (We ignore the fact that Fibonacci numbers can be computed quite efficiently if an iterative algorithm is used.) The constructor generates Fibonacci numbers and stores every tenth number in an array. Then, when a Fibonacci number is actually required, the recursive method `fib` uses the precomputed values when possible.

```
public class Fibonacci
{
     private static int[] fibs = new int[35];

     public static int fib(int n)
     {
        int result;
        if (n == 0)
        {
           result = 0;
        }
        else if (n == 1 || n == 2)
        {
           result = 1;
        }
        else if ((n < fibs.length) && (fibs[n] != 0))
        {  // cache hit (found the value in the cache)
           result = fibs[n];
        }
        else
        {  // cache miss (did not find the value in the cache)
           result = fib(n-1) + fib(n-2);
           if (n < fibs.length)
           {
              fibs[n] = result;
           }
        }
        return result;
     }
}
```

As noted in the comments in the code, if a value is stored in the cache, a "cache hit" is said to have occurred and the value is extracted from the cache with no additional compution; if a value is not stored in the cache, a "cache miss" is said to have occurred and the value must be computed. However, once a value has been computed, it can be added to the cache where it becomes available for any subsequent requests. The use of caches is an important topic in the development of both computer software and computer hardware.

This technique makes the recursive method run in time that is comparable to the iterative version – precomputing values can make a huge difference to a method that is slow but is frequently used.

14.13.5Frequency Count

The following methods illustrate how a parameter can be passed in a recursive method. The program counts the frequency of each item in an int array named `list`. The maximum value in `list` is 100.

```
public static int[] freq(int[] list)
{
     int[] counts;

     counts = new int[101];
     freq(list, counts, 0);
     return counts;
}


public static void freq(int[] list, int[] counts, int position)
{
     if (position < list.length)
     {
        counts[list[position]]++;
        freq(list, counts, position+1);
     }
}
```

15 LINKED LISTS

15.1 Introduction

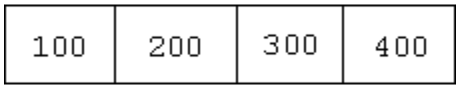
Until now, the array and the ArrayList have been the only data structures used to store a collection of elements. When these data structures are used, the elements are stored in linear order using physical adjacency, that is, each element is beside the next element in sequence. In this chapter we examine an alternative to physical adjacency, the linked list.

15.2 Lists

A list is an abstract structure that maintains a collection of elements. Various operations are permitted on the elements stored in the list such as insert, retrieve, modify and delete. A list is implemented by a specific data structure such as an ArrayList. The ArrayList supports a generic interface or API by implementing the list manipulation commands in such a way that the user does not have to know how the elements in the collection are stored.

15.3 Arrays

Elements are stored in an array using **physical adjacency**: the first element in the list is stored in the first position in the array, the second element is stored next to the first element, etc.



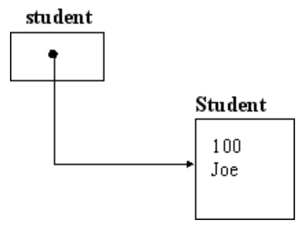
This organization has several advantages: a new element can be added to the end of the list without any effort; elements in the list can be traversed easily – just start at the first element and continue until the last element. However, the organization also has several disadvantages: if the elements must be ordered according to a value, the entire list must be sorted using one of the sorting algorithms; also, if the array becomes full, a new array must be allocated and the elements from the original array must be copied to the new array. (Recall that an ArrayList uses an array of objects internally to store the elements in a collection.)

15.4 A Simple Linked List

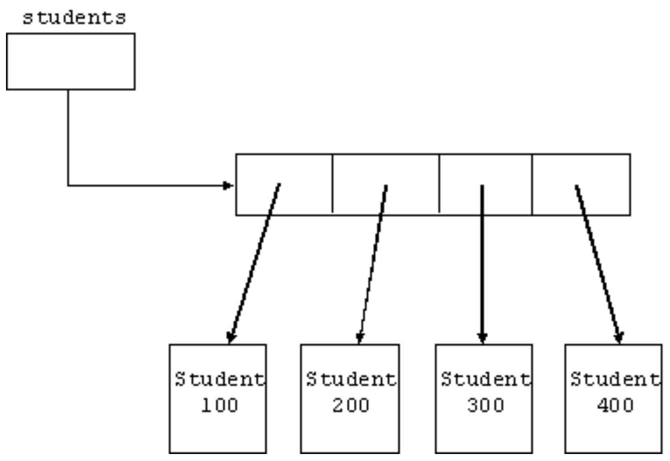
We now begin to examine a data structure, the linked list, that also implements a list but does not use physical adjacency to store the elements in the collection.

Recall that when an object is instantiated, the variable to which the object is assigned does not contain the object itself, instead the variable contains a pointer or reference to the object.

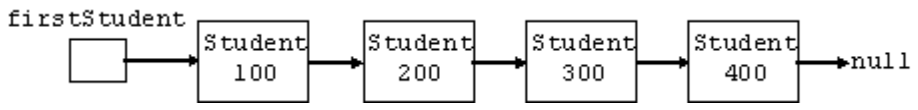
```
Student student;
student = new Student("100", "Joe");
```



If we create an array of Student objects, the array has the following representation.



Since the array just contains pointers to objects, why not use those pointers to link each object to the next object in the list instead of storing the pointers in the array?



We can create a simple linked list using this technique – with pointers (object references) that link the objects together. An object reference of null is normally used to indicate the end of the list.

```
class Student
{
     private static Student firstStudent=null;

     private String studentNumber;
     private String studentName;

     private Student nextStudent;

     public Student(String studentNumber, String studentName, Student nextStudent)
     {
        this.studentNumber = studentNumber;
        this.studentName = studentName;
        this.nextStudent = nextStudent;
     }

     public static void create(String studentNumber, String studentName)
     {
        Student newStudent;

        // add to beginning of the collection
        newStudent = new Student(studentNumber, studentName, firstStudent);
        firstStudent = newStudent;
     }

     public static Student getFirstStudent()
     {  // return the first Student in the collection
        return firstStudent;
     }

     public Student getNextStudent()
     {  // return the next Student in the collection
        return nextStudent;
     }

     public String toString()
     {
        return "Student Number: " +studentNumber +" Student Name: " +studentName;
     }
}
```

Notice that the Student class now contains two additional variables: a static variable that points to the first Student object in the linked list and an instance variable that points to the next Student in the list.

With this data organization, each object contains the reference of the next object in the list. As a result, the array is no longer necessary.

To create the linked list, we don't have to be concerned with the pointers in the list, that processing is handled within the Student class. We use a factory method to create each new Student object and add it to the linked list.

```
public static void createStudents()
{
     Student.create("400", "Sue");
     Student.create("300", "Fred");
     Student.create("200", "Sally");
     Student.create("100", "Joe");
}
```

The following simple loop traverses the linked list and prints each Student object.

```
public static void printStudents()
{
     Student currentStudent;

     currentStudent = Student.getFirstStudent();
     while (currentStudent != null)
     {
        System.out.println(currentStudent);
```

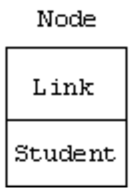
```
         currentStudent = currentStudent.getNextStudent();
      }
}

Student Number: 100 Student Name: Joe
Student Number: 200 Student Name: Sally
Student Number: 300 Student Name: Fred
Student Number: 400 Student Name: Sue
```

Note that when the contents of the list are printed, the students are printed in the reverse order of the order in which they were inserted. The reason for this behaviour is that each new element that is added to the list is added at the beginning of the list, before any existing elements. We will examine other techniques for adding elements to a linked list later in this chapter.

15.5 A Better Linked List

The linked list developed in the previous section is not a bad start but it has the disadvantage that the Student class contains information that is not relevant to the specific student. Mixing the application information with the representation information is not a good programming practice. It would be better if the Student information could be separated from the information used to store the Student objects.

Separating the Student information from the object reference can easily be accomplished by creating a new class that contains a student object and the link/pointer to the next student. Typically, this class is referred to as a **Node** in a linked list.



The following Node class contains a static variable that points to the first Node in the linked list. The class also contains a variable that points to the Student object that is stored in the Node plus a pointer to the next node object in the linked list.

```
class Node
{
     private static Node firstNode=null;

     private Student student;
     private Node nextNode;

     public Node(Student student, Node nextNode)
     {
        this.student = student;
        this.nextNode = nextNode;
     }
     public static Node getFirstNode()
     {
        return firstNode;
     }

     public static void create(String studentNumber, String studentName)
     {
        Student newStudent;
        Node newNode;

        // add to beginning of linked list
        newStudent = new Student(studentNumber, studentName);
        newNode = new Node(newStudent, firstNode);
        firstNode = newNode;
     }

     public Student getElement()
     {
        return student;
     }

     public Node getNextNode()
     {
        return nextNode;
     }
}
```

Now the Student class is back to “normal”, it does not contain any representation information.

```
class Student
{
     private String studentNumber;
     private String studentName;

     public Student(String studentNumber, String studentName)
     {
        this.studentNumber = studentNumber;
        this.studentName = studentName;
     }

     public String toString()
     {
        return "Student Number: " +studentNumber +" Student Name: " +studentName;
     }
}
```

Again, a factory method is used to create each new Student object. However, this time the factory method is in the Node class, not the Student class.

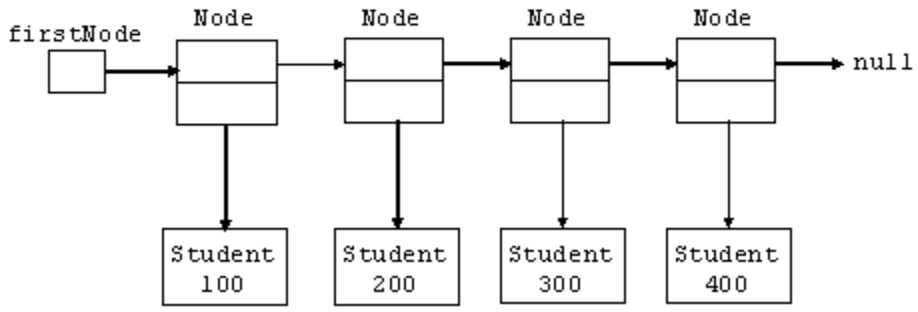
```
public static void createStudents()
{
     Node.create("400", "Sue");
     Node.create("300", "Fred");
     Node.create("200", "Sally");
     Node.create("100", "Joe");
}
```

The method that prints the contents of the linked list is just a minor modification of the previous version.

```
public static void printStudents()
{
     Node currentNode;

     currentNode = Node.getFirstNode();
     while (currentNode != null)
     {
        System.out.println(currentNode.getElement());
        currentNode = currentNode.getNextNode();
     }
}
```

Now there is a clear separation between the application information (student information) and the information used to represent a list of students (the link to the next student).

Note that each Node does not actually contain any Student data, each node contains two object references, a reference to a Student object and a reference to the next Node object in the linked list.



15.6 A Linked List Class

While the linked list data structure developed in the previous section is better than our first attempt at implementing a linked list, it still has the disadvantage that programmers need to be aware of the Node class. The Node class was originally intended to contain only the pointers to a Student object and to the next Node object. Now the class also contains the pointer to the first Node in the linked list and it also contains the factory method that is used to create each Student object. By making this processing visible to the programmer, the application program has become tangled with the data structure representation and this violates our encapsulation principle. When it is time to extend the processing by adding a delete command, the Node class will become even more tightly coupled to the data structure.

If we move the manipulation of Node objects into a separate class, then the programmer can manipulate the linked list without having to be aware of any of the internal details of the processing. The LinkedList class below illustrates how the internal details can be hidden from the users of the class.

```
class LinkedList
{
     private Node firstNode;

     public LinkedList()
     {
        firstNode = null;
     }

     public void add(Student newStudent)
     {  // add to beginning of linked list
        firstNode = new Node(newStudent, firstNode);
     }

     public Student get(int position)
     {
        Node currentNode;
        Student currentStudent;
        int current;

        currentStudent = null;
        current = 0;
        currentNode = firstNode;
        while ((currentNode!=null) && (current<=position))
        {
           currentStudent = currentNode.getElement();
           currentNode = currentNode.getNextNode();
           current++;
        }
        return currentStudent;
     }

     public int size()
     {
        Node currentNode;
        int size;

        size = 0;
        currentNode = firstNode;
        while (currentNode!=null)
        {
           currentNode = currentNode.getNextNode();
           size++;
        }
        return size;
     }
}
```

Note that the `get` method is more complicated than the corresponding get method in an ArrayList because when a linked list is processed, we must traverse the linked list in order to locate a specific element.

Now the linked list can be manipulated in a much more familiar manner.

```
public static LinkedList createList()
{
     LinkedList students;

     students = new LinkedList();

     students.add(new Student("400", "Sue"));
     students.add(new Student("300", "Fred"));
     students.add(new Student("200", "Sally"));
     students.add(new Student("100", "Joe"));
     return students;
}
```

If you examine the LinkedList class closely, you should notice that it implements the some of the same methods that are provided by the ArrayList class. Since the LinkedList class can be manipulated in the same manner as the ArrayList class, it becomes possible to switch from an ArrayList to a LinkedList by modifying only the declaration statement, the instantiation statement, and any statements that pass/return the collection.

As a result of encapsulating the processing in the LinkedList class, only the LinkedList class needs to be aware of the Node class and the Node class does not contain any non-essential processing.

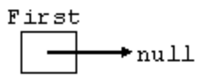
```
class Node
{
     private Student student;
     private Node nextNode;

     public Node(Student student, Node nextNode)
     {
        this.student = student;
        this.nextNode = nextNode;
     }

     public Student getElement()
     {
        return student;
     }

     public Node getNextNode()
     {
        return nextNode;
     }

     public String toString()
     {
        return student.toString();
     }
}
```

One final advantage of this organization is that the static variable that contained a pointer to the first element in the list is now an instance variable in the LinkedList class. As a result, any number of instances of the LinkedList class can be created.

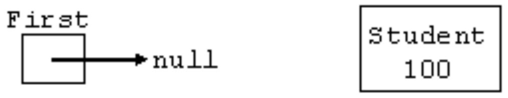
15.7 Inserting at the Beginning of a Linked List

Now that we have selected an appropriate organization for the information in the linked list, we can examine the algorithms that create and modify the linked list.

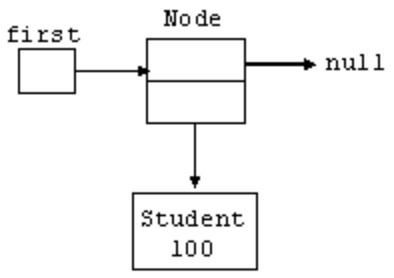
When a linked list is initially instantiated, there are no nodes in the list. So, the constructor for assigns the value `null` to the firstNode variable.



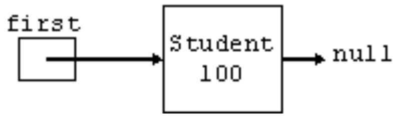
When the first student is added to the collection, a new Student object is created.



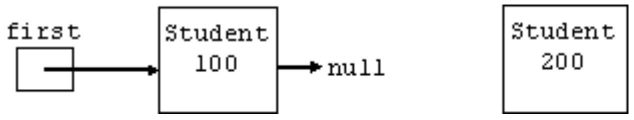
The student is added to the linked list by creating a Node that contains a reference to the Student object and modifying the first variable to point to the new Node. The link to the next Node in the collection is set to null because there are not yet any additional nodes.



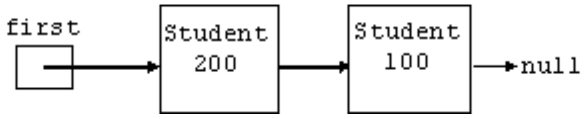
To reduce the clutter, in subsequent diagrams we will show only the Student object (see below). This is just a diagramming convention, the Node object is still required as illustrated above.



When the next student is added to the collection, it is simplest if the Node that references the new student is added to the beginning of the list.



So, in the diagram below, Student 200 is now at the beginning of the list and Student 100 comes after Student 200.



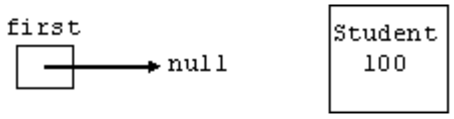
The instructions in the Students class to add each new Student to the beginning of the linked list are quite simple and work correctly both for an empty list and for a non-empty list.

```
public void add(Student newStudent)
{    // add to beginning of linked list
     firstNode = new Node(newStudent, firstNode);
}
```

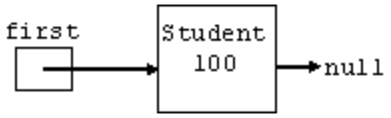
15.8 Inserting at the End of a Linked List

In this section, we examine how we can add a new element to the end of a linked list.

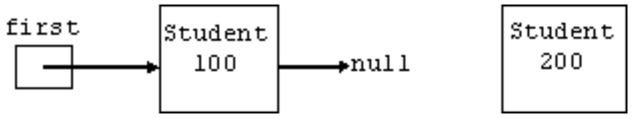
The algorithm to add to the end of linked list is not quite as simple as adding to the beginning. We begin with the simple case, adding to the end of an empty list.



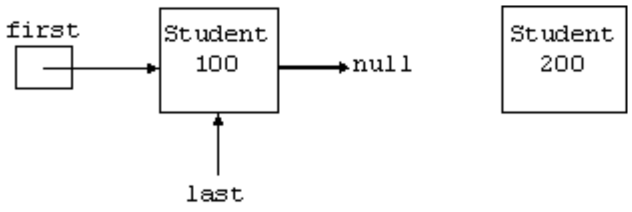
If the list is empty, we simply set the first pointer to point to the new object and set the next pointer to null.



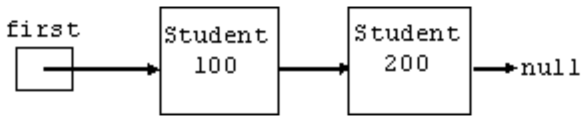
If the list is not empty, we must locate the last node in the list since the new object will be placed after the last node.



Traverse the list until a node with a next pointer equal to null is located.



We then set the next pointer of the last element to point to the new object.



```
public void add(Student newStudent)
{    // add to end of linked list
     Node newNode;
     Node currentNode;

     newNode = new Node(newStudent, null);
     if (firstNode == null)
     {
        firstNode = newNode;
     }
     else
     {
        currentNode = firstNode;
        while (currentNode.getNextNode()!=null)
        {  // find the last node in the linked list
           currentNode = currentNode.getNextNode();
        }
        currentNode.setNextNode(newNode);
     }
}
```

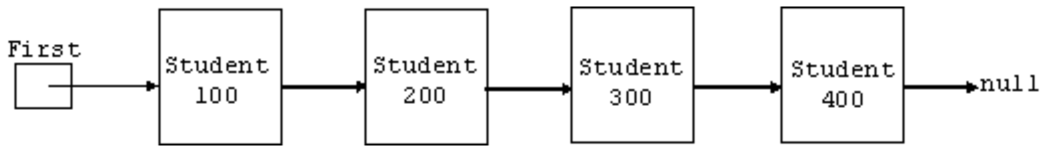
The algorithm is not particularly complex, it involves two cases, adding to an empty list and adding to a non-empty list. In the non-empty list case, it is necessary to traverse the list until the last node in the list is found. The new node is then added to the end of the linked list.

The process of adding to the end of a linked list could be simplified if a pointer to the last node in the linked list is maintained in the LinkedList class.

15.9 Deleting an Element

Now that we have the data structure well defined, we need to add some additional operations that manipulate the linked list.

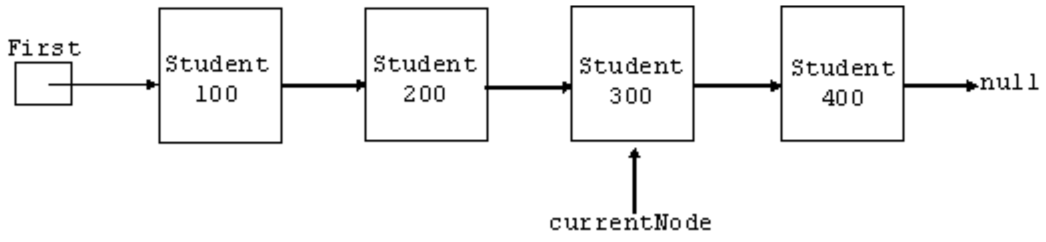
An obvious operation that is missing is the ability to delete an existing element from the collection. For example, for the following linked list, we might want to delete the information about Student 300.



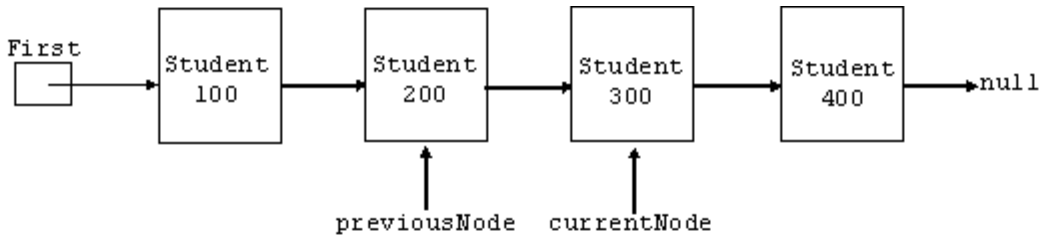
To begin, we need to locate the Node that contains Student 300. The following instructions locate the desired Node.

```
currentNode = firstNode;
while ((currentNode!=null) && !(currentNode.getElement().equals(student)))
{    //find the element to be deleted
     currentNode = currentNode.getNextNode();
}
```

After the loop terminates, currentNode points at the Node that contains the student with the specified value of studentNumber (or, if the student does not exist in the collection, currentNode is null).



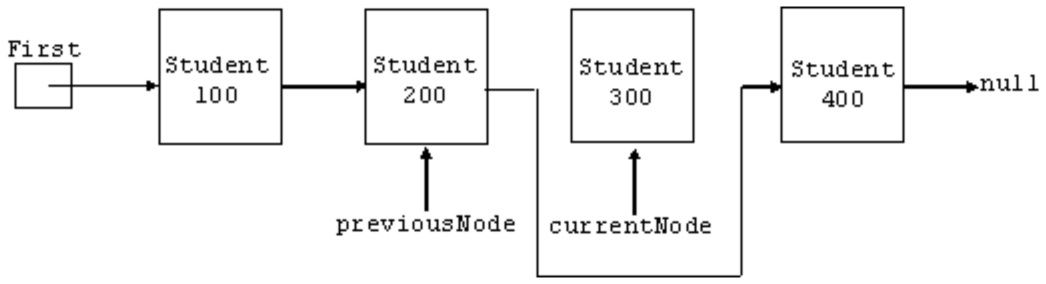
However, knowing which node is to be deleted is not sufficient; we also need to know which node comes before the node to be deleted.



We need to revise our search loop to keep track not only of the current node but also of the node that comes before the current node, that is, the previous node.

```
previousNode = null;
currentNode = firstNode;
while ((currentNode!=null) && !(currentNode.getElement().equals(student)))
{    //find the element to be deleted
     previousNode = currentNode;
     currentNode = currentNode.getNextNode();
}
```

We can now modify the link of the previous node to skip over the deleted node, as is shown in the diagram below.

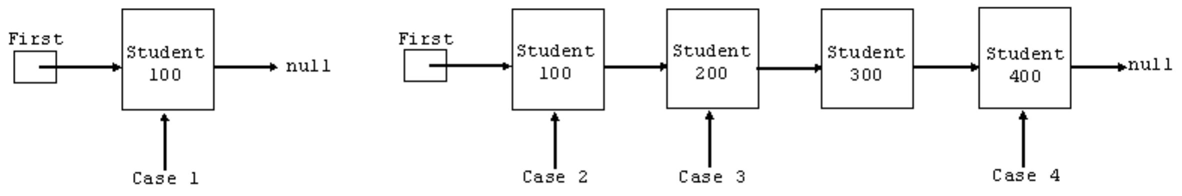


Now that we know both the node to be deleted and the node that immediately precedes the node to be deleted, we can delete the node correctly. First, we take the pointer to the next node from the current node and then we assign that pointer to the link in the previous node.

```
previousNode.setNextNode(currentNode.getNextNode());
```

As a result, the node that contains the information about student 300 is no longer a part of the collection. The node still exists but since it is not pointed to by any variable, it has become an orphan object and the space that it occupies will be reclaimed during the next garbage collection.

Unfortunately, the deletion operation described above handles only one case, that in which the node to be deleted has both a node before it and after it. There are also additional cases as illustrated in the diagram below.



Case 1: the last node in the collection is deleted;
Case 2: the first node in the collection is deleted;
Case 3: a middle node in the collection is deleted;
Case 4: the last node in the collection is deleted.

The instructions that handle these 4 cases are shown below:

```
if ((previousNode == null) && (currentNode.getNextNode()==null))
{    //delete only node in the list
     firstNode = null;
}
else if (previousNode == null)
{    //delete first node in the list
     firstNode = currentNode.getNextNode();
}
else if (currentNode.getNextNode()==null)
{    //delete last node in the list
     previousNode.setNextNode(null);
}
else
{    //delete a node that has at least one node before it and after it
     previousNode.setNextNode(currentNode.getNextNode());
}
```

If you look at the four cases shown above, you should notice that the first two cases are very similar and that the last two cases are also very similar. In fact, the four cases can be reduced to the following two cases:

```
if (previousNode == null)
{    //delete first node in the list
     firstNode = currentNode.getNextNode();
}
else
{    //delete a node that has at least one node before it
     previousNode.setNextNode(currentNode.getNextNode());
}
```

The method now works correctly when deleting any node in the list.

```
public void remove(Student student)
{    //delete a node from the linked list
     Node currentNode;
     Node previousNode;

     previousNode = null;
     currentNode = firstNode;
     while ((currentNode != null) && !(currentNode.getElement().equals(student)))
     {  //find the student to be deleted
        previousNode = currentNode;
        currentNode = currentNode.getNextNode();
     }
     if (previousNode == null)
     {  //delete firstNode node in the list
        firstNode = currentNode.getNextNode();
     }
     else
     {  //delete a node that has at least one node before it
        previousNode.setNextNode(currentNode.getNextNode());
     }
}
```

An additional method must be added to each of the Node class and the Student class. These revised classes are shown below.

```
public class Node
{
     private Student student;
     private Node nextNode;

     public Node(Student student, Node nextNode)
     {
        this.student = student;
        this.nextNode = nextNode;
     }

     public Student getElement()
     {
        return student;
     }

     public Node getNextNode()
     {
        return nextNode;
     }

     public void setNextNode(Node nextNode)
     {
        this.nextNode = nextNode;
     }

     public String toString()
     {
        return student.toString();
     }
}
```

```
public class Student
{
     private String studentNumber;
     private String studentName;

     public Student(String studentNumber, String studentName)
     {
        this.studentNumber = studentNumber;
        this.studentName = studentName;
     }

     public boolean equals(Student student)
     {
        return this.studentNumber.equals(student.studentNumber);
     }

     public String toString()
     {
        return "Student Number: " +studentNumber +" Student Name: " +studentName;
     }
}
```

This linked list class is relatively well designed now. The only significant exception is that the Student object is explicitly referred to in both the LinkedList class and in the Node class. We will examine how this dependency can be removed in Chapter 17 – Generic Data Structures.

15.10 Linked List Examples

This section contains some examples that illustrate various manipulations of a linked list.

The Node class used in the following examples is slightly different from the Node class used above.

```
class Node
{
     private Object object;
     private Node nextNode;
     private Node previousNode;

     public Node(Object object, Node nextNode)
     {
        this.object = object;
        this.nextNode = nextNode;
     }
     public Object getElement()
     {
        return object;
     }
     public void setElement(Object object)
     {
        this.object = object;
     }
     public Node getNextNode()
     {
        return nextNode;
     }
     public Node getPreviousNode()
     {
        return previousNode;
     }
     public void setNextNode(Node newNode)
     {
        nextNode = newNode;
     }
     public void setPreviousNode(Node previousNode)
     {
        this.previousNode = previousNode;
     }
     public String toString()
     {
        return object.toString();
     }
}
```

15.10.1Find Last Node

The method returns a pointer to the last node in a linked list.

```
public static Node findLastNode(Node firstNode)
{
     Node currentNode;

     currentNode = firstNode;
     while ((currentNode!=null) && (currentNode.getNextNode()!=null))
     {
        currentNode = currentNode.getNextNode();
     }

     return currentNode;
}
```

15.10.2Print String Objects

The method below prints each String object in a linked list.

```
public static void printStrings(Node firstNode)
{
     Node currentNode;
     Object currentObject;

     currentNode = firstNode;
     currentObject = null;
     while (currentNode!=null)
     {
        if (currentNode!=null)
        {
           currentObject = currentNode.getElement();
           if (currentObject instanceof String)
           {
              System.out.println(currentObject);
           }
        }
        currentNode = currentNode.getNextNode();
     }
}
```

15.10.3Print Adjacent String Objects

The following method prints String objects that are adjacent in a linked list.

```
public static void printSideBySideStrings(Node firstNode)
{
     Node currentNode;
     Node previousNode;
     Object currentObject;
     Object previousObject;

     previousNode = null;
     currentNode = firstNode;
     currentObject = null;
     previousObject = null;
     while (currentNode!=null)
     {
        if ((previousNode!=null) && (currentNode!=null))
        {
           previousObject = previousNode.getElement();
           currentObject = currentNode.getElement();
           if((previousObject instanceof String)&&(currentObject instanceof String))
           {
              System.out.println(previousObject +" " +currentObject);
           }
        }
        previousNode = currentNode;
        currentNode = currentNode.getNextNode();
     }
}
```

15.10.4Reverse List Elements

The following method reverses the elements in a linked list. (While this processing appears to be complex, it is actually straight forward with a linked list.

```
public static Node reverseList(Node firstNode)
{
     Node currentNode;
     Node previousNode;
     Node nextNode;

     previousNode = null;
     nextNode = null;
     currentNode = firstNode;
     while (currentNode!=null)
     {
        nextNode = currentNode.getNextNode();
        currentNode.setNextNode(previousNode);
        previousNode = currentNode;
        currentNode = nextNode;
     }
     return previousNode;
}
```

15.10.5Add Back Links

The method below adds back links to an existing linked list.

```
public static Node addBackLinks(Node firstNode)
{
     Node currentNode;
     Node previousNode;
     previousNode = null;
     currentNode = firstNode;
     while (currentNode!=null)
     {
        currentNode.setPreviousNode(previousNode);
        previousNode = currentNode;
        currentNode = currentNode.getNextNode();
     }
     return previousNode;
}
```

15.10.6Split a List into Two Lists

The method below takes an existing linked list and splits it into two linked lists, with the first list containing `n` elements. If there are fewer than `n` elements in the original list, the first list will contain all of the elements and the second list will be empty.

```
public static Node splitList(Node firstNode, int n)
{
      Node currentNode;
      Node newList;
      int count;
      currentNode = firstNode;
      count = 1;
      while ((currentNode!=null) && (count<n))
      {
          currentNode = currentNode.getNextNode();
          count++;
      }

      if (currentNode == null)
      {
          newList = null;
      }
      else
      {
          newList = currentNode.getNextNode();
          currentNode.setNextNode(null);
      }
      return newList;
}
```

15.10.7Rotating the Contents of a Linked List

The method below rotates the contents of a linked list `n` elements to the left.

```
public static Node rotate(Node firstNode, int n)
{
      Node currentNode;
      Node previousNode;
      Node newFirstNode;
      int count;

      previousNode = null;
      newFirstNode = firstNode;
      currentNode = firstNode;
      count = 0;
      while ((currentNode!=null) && (count<n))
      {   // skip over the elements to be rotated
          previousNode = currentNode;
          currentNode = currentNode.getNextNode();
          count++;
      }

      if ((count == n) && (currentNode!=null))
      {   // currently pointing at the first element after the elements to be rotated
          newFirstNode = currentNode;
          // set the pointer of the last element being rotated to null
          previousNode.setNextNode(null);

          while (currentNode!=null)
          {   // find the end of the linked list
              previousNode = currentNode;
              currentNode = currentNode.getNextNode();
          }

          // attach the elements being rotated to the end of the list
          previousNode.setNextNode(firstNode);
      }
      return newFirstNode;
}
```

15.10.8Remove String Objects

The method below removes all String objects from an existing linked list.

```
public static Node removeStrings(Node firstNode)
{
      Node currentNode;
      Node previousNode;

      previousNode = null;
      currentNode = firstNode;
      while (currentNode!=null)
      {
          if (currentNode.getElement() instanceof String)
          {
              previousNode.setNextNode(currentNode.getNextNode());
          }
          else
          {
              previousNode = currentNode;
          }
          currentNode = currentNode.getNextNode();
      }
      return firstNode;
}
```

This algorithm is not as simple as it first appears and the method above contains a subtle error. The method works correctly in some cases but not in all cases.

When a program is tested, it should be tested not only with the expected data, but also with unusual cases. For example, a method that manipulates a linked list should be tested with an empty list, a list with only one Node, lists that contain unusual combinations of Nodes, etc. Try testing the method above with a variety of linked lists in order to determine the problem with the method (and then fix the problem and perform some more testing).

15.11 Recap

If you examine the algorithms in the previous section, you should notice that they all have a very similar structure. The following method traverses a linked list and returns the last element. This method contains the processing required by most of the linked list algorithms.

```
public static Node traverse(Node firstNode)
{
      Node currentNode;
      Node previousNode;

      previousNode = null;
      currentNode = firstNode;
      while (currentNode!=null)
      {
          previousNode = currentNode;
          currentNode = currentNode.getNextNode();
      }
      return previousNode;
}
```

The method above continues until it runs off the end of the linked list (currentNode is null and so previousNode points to the last node in the linked list). The method below is almost identical, except that when the loop terminates, currentNode points to the last element in the linked list.

```
public static Node traverse(Node firstNode)
{
     Node currentNode;
     Node previousNode;

     previousNode = null;
     currentNode = firstNode;
     while ((currentNode!=null) && (currentNode.getNextNode()!=null))
     {
        previousNode = currentNode;
        currentNode = currentNode.getNextNode();
     }
     return currentNode;
}
```

Once you are able to traverse a linked list using one of the two algorithms above, you should be able to perform most manipulations on a linked list.

15.12 Summary

A linked list provides an attractive alternative to storing elements using physical adjacency. The linked list makes it significantly easier to insert into the middle of a list and to delete an element from a list because the following elements do not have to be moved (as they do when an array is used). Also, the linked list is not declared to be a specific size so nodes can be added to a linked list as long as there is sufficient memory available.

The linked list does have one disadvantage. Since access to the list must begin at the first element and, since traversal of the list is sequential, fast search algorithms such as the binary search can not be used with a basic linked list. However, there are more sophisticated, non-linear data structures (such as trees) that can be used to provide rapid access to nodes.

The basic physical adjacency list (ArrayList) and the linked list are identical from the user's point of view – both maintain a collection of objects. It is easy to replace either representation with the other if the collection of objects is maintained inside an appropriate collection class.

Linked lists are also the foundation for other important data structures in computer science. For example, when new nodes are inserted at the front of a list, the linked list implements a **stack** – a data structure in which the element most recently added to the structure is the first element removed. If new nodes are always added to the end of the list and nodes are removed from the beginning of the list, the linked list implements a **queue** – a data structure in which the first element added to the structure is the first element that is removed (think of a queue of people waiting for service at a bank).

There are also other versions of the basic linked list that are useful. One of the more common is the "doubly-linked list" which has both forward pointers (to the next element) and backward pointers (to the previous element). This type of linked list makes it easy to traverse the list in either the forward direction or the backward direction.

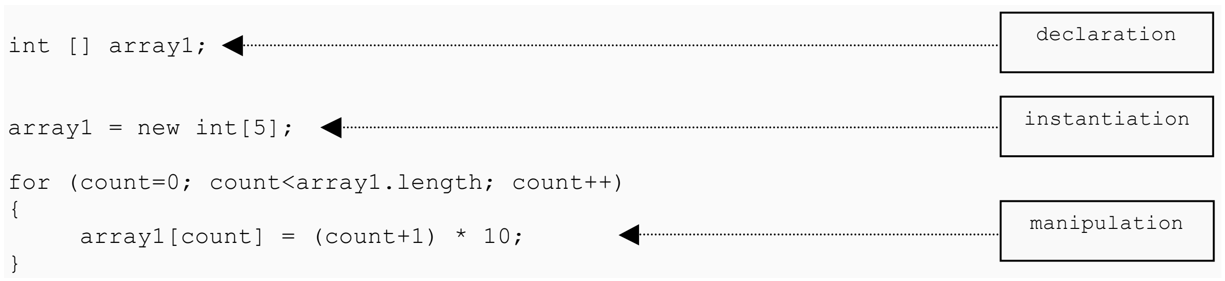
16 MULTI-DIMENSIONAL ARRAYS

16.1 Introduction

We have been using arrays since the beginning of these notes. An array is used to maintain a collection of values that are of a specific data type. In this chapter we examine multi-dimensional arrays – arrays of arrays.

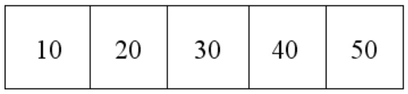
16.2 One-Dimensional Arrays

The creation and manipulation of a one-dimensional array is quite straight forward.



```
int [] array1;

array1 = new int[5];

for (count=0; count<array1.length; count++)
{
     array1[count] = (count+1) * 10;
}
```

The array shown above has one dimension and its elements are accessed by providing one subscript that identifies the element, relative to the beginning of the array. The diagram below illustrates the one-dimensional array that is created by the statements above.



A one-dimensional array is a linear data structure: it can be traversed by beginning at the first element and then moving to the next element in order. If the subscript is known, array elements may also be accessed directly, without having to traverse the preceding elements.

A one-dimensional array may contain any type of data: a primitive data type or an object.

16.3 Multi-Dimensional Arrays

A multidimensional array is simply a collection of one-dimensional arrays. The diagram below illustrates a two-dimensional array that consists of 3 rows and 5 columns.

	Columns				
	10	20	30	40	50
Rows	60	70	80	90	100
	110	120	130	140	150

The statements that create the two-dimensional array above are shown below.

```
int [][] array2;  ◄·······························  declaration

array2 = new int[3][5];  ◄························  instantiation

for (row=0; row<3; row++)
{
     for (column=0; column<5; column++)
     {
        array2[row][column] = (row)*50 + (column+1)*10;  ◄·····  manipulation
     }
}
```

The declaration `int [][] array2` defines array2 to be a two-dimensional array, that is, the array has two dimensions instead of one dimension and so two subscripts are needed to refer to an element in the array.

The array is instantiated in the same manner as a one-dimensional array except that the second dimension is also included: `array2 = new int[3][5]`. The first dimension `[3]` defines the number of rows in the array and the second dimension `[5]` defines the number of columns in the array.

		Columns				
		0	1	2	3	4
	0	10	20	30	40	50
Rows	1	60	70	80	90	100
	2	110	120	130	140	150

An element is referenced by specifying the row number and the column number as the two subscripts. So, `array2[1][3]` refers to the element in row 1, column 3 which contains the value 90 in the array shown above.

In Java, two-dimensional arrays are accessed in what is referred to as **row-major order**. This means that the first subscript refers to the row number and the second subscript refers to the column number.

As with a one-dimensional array, we can determine the dimensions of a two-dimensional array dynamically instead of having to hard-code the dimensions in a program. To determine the number of **rows** in a two-dimensional array, the same instruction is used as for a one-dimensional array:

```
numRows = array2.length;
```

If we want to know the number of **columns** in a two-dimensional array, we simply add a row subscript to the previous statement.

```
numColumns = array2[0].length;
```

It doesn't matter which row subscript is used since all rows have the same number of columns. (We will see later that this does not always have to be true.)

The following method prints the elements in a two-dimensional array. All elements in each row are printed on the same line and then the elements in the next row are printed on the next line.

```
public static void printList(int[][] array)
{
     int row;
     int column;

     System.out.println();

     for (row=0; row<array.length; row++)
     {
         for (column=0; column<array[row].length; column++)
         {
             System.out.print(array[row][column] +" ");
         }
         System.out.println();
     }
}
```

16.4 Matrices

In mathematical terms, a two-dimensional array is referred to as a **matrix**. While the storage and manipulation of matrices is one of the prime uses of two-dimensional arrays, it is not the only use.

In this section we examine some of the common matrix manipulations.

To begin, we will write a method that creates a new matrix of a specified size. The elements in the matrix are initialized to zero.

```
public static int[][] createMatrix(int numRows, int numColumns)
{
     int row;
     int column;
     int[][] matrix1;

     matrix1 = new int[numRows][numColumns];

     for (row=0; row<matrix1.length; row++)
     {
         for (column=0; column<matrix1[row].length; column++)
         {
             matrix1[row][column] = 0;
         }
     }
     return matrix1;
}
```

The following method accepts an existing matrix and sets its elements to the identity matrix (a matrix that consists of zeros everywhere except on the main diagonal where the elements are ones).

```
public static void identityMatrix(int[][] matrix1)
{
     int row;
     int column;

     for (row=0; row<matrix1.length; row++)
     {
         for (column=0; column<matrix1[row].length; column++)
         {
             matrix1[row][column] = 0;
         }
         matrix1[row][row] = 1;
     }
}
```

Recall that arrays are objects in Java so, since the matrix parameter already exists and we do not change the size of the matrix, we do not need to return the result of the processing via a return statement – we have actually updated the matrix object in the calling method.

If we have two matrices that have the same dimensions, we can add the two matrices together to produce a new matrix.

```
public static int[][] matrixAddition(int[][] matrix1, int[][] matrix2)
{
     int row;
     int column;
     int[][] matrix3;

     if ((matrix1.length!=matrix2.length)||(matrix1[0].length!=matrix2[0].length))
     {
        System.out.println("Matrices do not have the same dimensions.");
        matrix3 = new int[0][0];
     }
     else
     {
        matrix3 = new int[matrix1.length][matrix1[0].length];
        for (row=0; row<matrix1.length; row++)
        {
           for (column=0; column<matrix1[row].length; column++)
           {
              matrix3[row][column] = matrix1[row][column] + matrix2[row][column];
           }
        }
     }
     return matrix3;
}
```

16.5 Matrix Object

As the number of our matrix manipulation methods starts to grow, we should move them into their own class. This reduces the complexity of the application class that requires the matrix manipulation operations and also makes it easier to share the matrix manipulation methods with other users since they are isolated in their own class.

The following statements define the beginning of a matrix class. The constructor simply creates a matrix of the appropriate size. If necessary, we could also add another constructor that reads the initial contents of a matrix from a file.

```
public class Matrix
{
     private int[][] matrix;

     public Matrix(int numRows, int numColumns)
     {
        matrix = new int[numRows][ numColumns];
     }
     . . .
```

Our other matrix manipulation operations would also be added to this class. For example, a print method would be a useful addition.

```
public void print()
{
     int row;
     int column;

     for (row=0; row<matrix.length; row++)
     {
        for (column=0; column<matrix[row].length; column++)
        {
           System.out.print(matrix[row][column] +"\t");
        }
        System.out.println();
     }
}
```

Before we can add operations that manipulate two matrices, we must create the appropriate accessors and mutators that permit us to obtain information about a matrix and to retrieve and modify matrix elements.

```
private int getRows()
{
     return matrix.length;
}

private int getCols()
{
     return matrix[0].length;
}

private int getElement(int row, int col)
{
     return this.matrix[row][col];
}

private void setElement(int row, int col, int value)
{
     this.matrix[row][col] = value;
}
```

Once the accessors and mutators are available, matrix manipulation methods can be added to the class. The following method adds a matrix to the current matrix. (The processing in this method is slightly different from the earlier matrix addition method that returned a new matrix; however, this method could easily be modified to return the result as a new Matrix.)

```
public void add(Matrix matrix2)
{
     int row;
     int column;

     if ((this.getRows()!=matrix2.getRows())||(this.getCols()!= matrix2.getCols()))
     {
        System.out.println("Matrices do not have the same dimensions.");
     }
     else
     {
        for (row=0; row<matrix.length; row++)
        {
           for (column=0; column<matrix[row].length; column++)
           {
              matrix[row][column] = this.getElement(row,column)
                     + matrix2.getElement(row,column);
           }
        }
     }
}
```

Now that we have the basic framework for matrix manipulation defined, we could add additional matrix operations without much effort.

16.6 Array Initialization

Two-dimensional arrays may be initialized in the same manner as one-dimensional arrays. For example, the following statement creates a 3 × 5 array of int's and initializes the values as shown. This statement further reinforces the fact that this two-dimensional array is really 3 one-dimensional arrays of 5 elements each. The elements on the first line define the contents of the first row of the array, the elements on the second line define the contents of the second row of the array, and the elements on the third line define the contents of the third row of the array. (It is not necessary to split the contents of the array over multiple lines – this was done to improve readability.)

```
int[][] ints = { {  1,   2,   3,   4,   5},
                 { 10,  20,  30,  40,  50},
                 {100, 200, 300, 400, 500} };
```

16.7 Arrays of Objects

Until now, all of the examples in this chapter have created and manipulated two-dimensional arrays of int's. While the manipulation of mathematical values is often the motivation for using multi-dimensional arrays, an array may contain any type of data. For example, the following method creates and then prints a two-dimensional array of Strings.

```
public static void stringArray()
{
     String[][] strings = { {"The", "quick", "brown"},
                            {"dog", "jumped", "over"},
                            {"the", "lazy", "fox."} };
     int row;
     int column;

     for (row=0; row<strings.length; row++)
     {
        for (column=0; column<strings[row].length; column++)
        {
           System.out.print(strings[row][column] +"\t");
        }
        System.out.println();
     }
}
```

In the statements above, the element at `strings[0][0]` contains the value "`The`" and `strings[1][2]` contains the value "`over`".

16.8 Tic-Tac-Toe

The following class contains the foundation for a game of Tic-Tac-Toe (X's and O's). An array of Strings is used to record the current status of the game. Each location in the array contains either an "X", an "O", or a " " (i.e. a space). Although the array contains Strings, a char array could also have been used.

```
public class TicTacToe
{
     String[][] game;
     static final String EMPTY = " ";

     public TicTacToe (int size)
     {
        game = new String[size][size];
        initialize();
     }

     private void initialize()
     {
        int row;
        int column;

        for (row=0; row<game.length; row++)
        {
           for (column=0; column<game[row].length; column++)
           {
              game[row][column] = EMPTY;
           }
        }
     }
```

```
    public String makeMove(String token, int row, int column)
    {
       String result;

       result = null;
       if (game[row-1][column-1].equals(EMPTY))
       {
          game[row-1][column-1] = token;
       }
       else
       {
          result = "Position already filled.";
       }

       print();

       if (result == null)
       {
          result = checkRows();
       }
       return result;
    }

    private String checkRows()
    {
       String first;
       String result;
       int row, column;
       int length;
       int count;

       length = game[0].length;
       result = null;
       for (row=0; (row<length)&&(result==null); row++)
       {
          first = game[row][0];
          count = 1;
          for (column=1; column<length; column++)
          {
             if (game[row][column].equals(first))
             {
                count++;
             }
          }
          if ((count==length) && (!first.equals(" ")))
          {
             result = first;
          }
       }
       return result;
    }
```

```
     public void print()
     {
        int row;
        int column;

        System.out.println();
        for (row=0; row<game.length; row++)
        {
           for (column=0; column<game[row].length; column++)
           {
              System.out.print(game[row][column] +" ");
           }
           System.out.println();
        }
     }
}
```

There is still work that needs to be done to complete this class. For example, only the rows are examined to determine whether or not one player has won the game.

17 GENERIC DATA STRUCTURES

17.1 Introduction

In Chapter 15 – Linked Lists, we developed a student processing example that used a linked list to maintain a collection of students. The linked list class was designed specifically to maintain student objects, the class could not be used to store any other type of object. In this chapter, we examine how generic data structures that can store any type of object are designed ("generic" is this sense means general and is not related to Java's generics). By doing so, we can use the same data structure in different applications to store different types of data.

17.2 Generic Linked List

In this section, we examine the modifications that are required to make the linked list class capable of storing any type of object.

The following modifications are required:

- the application must downcast all objects extracted from the linked list;
- replace all references to Student with Object in the LinkedList and Node classes;
- the equals method in the Student class must be modified so that it accepts an object.

```
class LinkedList
{
     private Node firstNode;
     private Node lastNode;
     private int size;

     public LinkedList()
     {
        firstNode = null;
        size = 0;
     }

     public void add(Object newObject)
     {  // add to beginning of linked list
        firstNode = new Node(newObject, firstNode);
        size++;
     }

     public Object get(int position)
     {
        Node currentNode;
        Object currentObject;
        int current;

        current = 0;
        currentObject = null;
        currentNode = firstNode;
        while ((currentNode!=null) && (current<=position))
        {
           currentObject = currentNode.getElement();
           currentNode = currentNode.getNextNode();
           current++;
        }
        return currentObject;
     }

     public int size()
     {
        return size;
     }
}

class Node
{
     private Object object;
     private Node nextNode;

     public Node(Object object, Node nextNode)
     {
        this.object = object;
        this.nextNode = nextNode;
     }

     public Object getElement()
     {
        return object;
     }

     public Node getNextNode()
     {
        return nextNode;
     }

     public void setNextNode(Node nextNode)
     {
        this.nextNode = nextNode;
     }

     public String toString()
     {
        return object.toString();
     }
}

class Student
{
     private String studentNumber;
     private String studentName;

     public Student(String studentNumber, String studentName)
     {
        this.studentNumber = studentNumber;
        this.studentName = studentName;
     }

     public Student(String studentNumber)
     {
        this.studentNumber = studentNumber;
        this.studentName = null;
     }
     public boolean equals(Object student)
     {
        return this.studentNumber.equals(((Student) student).studentNumber);
     }

     public String toString()
     {
        return studentNumber +" " +studentName;
     }
}
```

There is one minor problem with the Student class. Now that the collection maintains objects of type Object, the equals method may generate a run-time error if an object that is not of type Student is passed to it. To avoid this situation, the equals method should be modified as follows:

```
public boolean equals(Object student)
{
     boolean result;

     result = false;
     if ((student!=null) && (student instanceof Student))
     {
        result = this.studentNumber.equals(((Student)student).studentNumber);
     }
     return result;
}
```

17.3 Using compareTo

In a generic data structure, the `equals` method can be used without any difficulties because the Object class includes the `equals` method (although the actual class being stored should override the `equals` method). So comparing two objects for equality is not a problem. However, comparing two objects for their relationship to each other (less than, equals, greater than) is a problem because the Object class does not contain a `compareTo` method. (This makes sense because not all objects can be compared for their relationships with each other.) As a result, if objects must be compared for their relationships (for example, the objects are being sorted), some additional programming must be included.

The class that is to be stored in the generic data structure must include the clause `implements Comparable` in the class header. While it is not relevant to us, `Comparable` is a Java interface which is similar to but not identical to a superclass.

```
class Student implements Comparable
```

When two objects are to be compared, the first object must be cast to `Comparable` before `compareTo` can be used.

```
(((Comparable)currentNode.getObject()).compareTo(object))==0
```

Finally, the `compareTo` method must be defined in the class that defines the objects that will be stored in the generic data structure. Note that the type of the parameter being passed to the `compareTo` method is an Object.

```
public int compareTo(Object student)
{
     int result;
     result = this.studentNumber.compareTo(((Student)student).studentNumber);
     return result;
}
```

This organization looks very strange but it is one of the quirks of Java that you just have to get used to.

17.4 Java's Generics

The programs in the previous sections work correctly but do not take advantage of the Java's generics. (This is where the terminology becomes confusing because we are adding Java's generics to a generic data structure.) The main class below illustrates how generics would be added to the LinkedList data types.

```
public class TestLinkedList
{
     public static void main(String[] parms)
     {
        LinkedList<Student> students;
        students = createList();
        printList(students);
        System.out.println("\nProgram completed normally.");
     }

     public static LinkedList<Student> createList()
     {
        LinkedList<Student> students;

        students = new LinkedList<Student>();

        students.add(new Student("400", "Sue"));
        students.add(new Student("300", "Fred"));
        students.add(new Student("200", "Sally"));
        students.add(new Student("100", "Joe"));
        return students;
     }

     public static void printList(LinkedList<Student> students)
     {
        int count;

        System.out.println("\n");
        for (count=0; count<students.size(); count++)
        {
           System.out.println(students.get(count));
        }
     }
}
```

If generics are included in a calling class, they must be defined in the associated collections class(es), in this case, the LinkedList class and the Node class. (Generics are already included in Java's collections classes such as the ArrrayList.)

The following classes show the modifications necessary to support the use of generics in the main class. The symbol **E** represents the generic type that is supplied by the programmer in the calling classes. At compile time, Java substitutes the specified type for the generic type **E**. By convention, the symbol **E** represents an **element** in a collection. Java uses the symbol **T** (**type**) in other classes that are not collections.

```
class LinkedList<E>
{
     private Node<E> firstNode;
     private int size;

     public LinkedList()
     {
        firstNode = null;
        size = 0;
     }

     public void add(E newElement)
     {  // add to beginning of linked list
        firstNode = new Node<E>(newElement, firstNode);
        size++;
     }

     public E get(int position)
     {
        Node<E> currentNode;
        E currentElement;
        int current;

        currentElement = null;
        current = 0;
        if (position < size)
        {
           currentNode = firstNode;
           while ((currentNode!=null) && (current<=position))
           {
              currentElement = currentNode.getElement();
              currentNode = currentNode.getNextNode();
              current++;
           }
        }
        return currentElement;
     }

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

```
class Node<E>
{
     private Node<E> nextNode;
     private E element;

     public Node(E element, Node<E> nextNode)
     {
        this.element = element;
        this.nextNode = nextNode;
     }

     public E getElement()
     {
        return (E) element;
     }

     public Node<E> getNextNode()
     {
        return nextNode;
     }

     public void setNextNode(Node<E> nextNode)
     {
        this.nextNode = nextNode;
     }

     public String toString()
     {
        return element.toString();
     }
}
```

One of the advantages of Java generics is that they do not require any modifications to the Student class.

```
class Student
{
     private String studentNumber;
     private String studentName;

     public Student(String studentNumber, String studentName)
     {
        this.studentNumber = studentNumber;
        this.studentName = studentName;
     }

     public Student(String studentNumber)
     {
        this(studentNumber, null);
     }

     public boolean equals(Student student)
     {
        return this.studentNumber.equals(student.studentNumber);
     }

     public String toString()
     {
        return studentNumber +" " +studentName;
     }
}
```

17.5 Using compareTo with Generics

As was mentioned earlier in this section, in a generic data structure, the `equals` method can be used without any difficulty but comparing two objects for their relationship to each other (less than, equals, greater than) is a problem.

The class that is to be stored in the generic data structure must include the clause `implements Comparable` in the class header. The generic data structure must also specify that any class that is to be stored must also implement Comparable (although the syntax uses `extends Comparable`).

```
class Student implements Comparable<Student>

class LinkedList<E extends Comparable<E>>
```

The `compareTo` method can now be used without having to perform any casts.

```
(currentNode.getElement().compareTo(object))==0
```

Finally, the `compareTo` method must be defined in each class for which objects will be stored in the generic data structure. Note that the type of the parameter being passed to the `compareTo` method no longer has to be an Object.

```
public int compareTo(Student student)
{
     return this.studentNumber.compareTo(student.studentNumber);
}
```

17.6 Using a Java Interface

The following example illustrates how a data structure can implement a Java **Interface** which permits one data structure to be replaced by another data structure that also implements the same interface. In the following program, the `List` interface (provided by Java) is used as the data type for the collections. When this interface is used, only the initial construction of the data structure must be modified if another data structure is to be used. For example, only the statement

```
students = new LinkedList<Student>();
```

must be modified in order to switch to an `ArrayList` (or other data structure). The program below also uses an **inner class** to hide the definition of the `Node` class from all classes except the `LinkedList` class.

```
import java.util.AbstractList;
import java.util.List;
import java.util.ArrayList;

public class TestListInterface
{
     public static void main(String[] parms)
     {
        processStudents();
     }

     public static void processStudents()
     {
        List<Student> students;

        students = createList();
        print(students);
     }

     public static List<Student> createList()
     {
        List<Student> students;

        students = new LinkedList<Student>();

        students.add(new Student("400", "Sue"));
        students.add(new Student("300", "Fred"));
        students.add(new Student("200", "Sally"));
        students.add(new Student("100", "Joe"));
        return students;
     }

     public static void print(List<Student> list)
     {
        int count;

        System.out.println("Students");
        for (count=0; count<list.size(); count++)
        {
           System.out.println(list.get(count));
        }
     }
}

class LinkedList<E> extends AbstractList<E>
{
     private Node<E> firstNode;

     public LinkedList()
     {
        firstNode = null;
     }

     public boolean add(E newObject)
     {  // add to beginning of linked list
        firstNode = new Node<E>(newObject, firstNode);
        return true;
     }

     public E get(int position)
     {
        Node<E> currentNode;
        E currentObject;
        int current;

        currentObject = null;
        current = 0;
        currentNode = firstNode;
        while ((currentNode!=null) && (current<=position))
        {
           currentObject = currentNode.getObject();
           currentNode = currentNode.getNextNode();
           current++;
        }
        return currentObject;
     }

     public int size()
     {
        Node<E> currentNode;
        int size;

        size = 0;
        currentNode = firstNode;
        while (currentNode!=null)
        {
           currentNode = currentNode.getNextNode();
           size++;
        }
        return size;
     }

     class Node<E>  // inner class
     {
        private E object;
        private Node<E> nextNode;

        public Node(E object, Node<E> nextNode)
        {
           this.object = object;
           this.nextNode = nextNode;
        }

        public E getObject()
        {
           return object;
        }

        public Node<E> getNextNode()
        {
           return nextNode;
        }

        public String toString()
        {
           return object.toString();
        }
     }
}

class Student
{
     private String studentNumber;
     private String studentName;

     public Student(String studentNumber, String studentName)
     {
        this.studentNumber = studentNumber;
        this.studentName = studentName;
     }

     public String toString()
     {
        return studentNumber +" " +studentName;
     }
}
```

18 GROWING AND REFACTORING

18.1 Introduction

In Chapter 2 – Growing Algorithms, we introduced the "growing" process for developing programs. With this process, we add data and algorithms in small increments so that we always have a program that works (albeit in a limited manner). As a result, if we make a change that "breaks" the program, that is, causes it to stop working correctly, the portion of the program that caused the problem should be restricted to the changed code or to that portion of the program that interacts with the changed code. Along with growing code, a good programmer will normally make small changes to the program as it is being developed to simplify the program and/or to remove duplicate code. This process is referred to as ***refactoring*** (and this is the third aspect of the design/implement/refactor trilogy). In this chapter we examine the interplay between growing code and refactoring the code.

18.2 Refactoring

The term refactoring was coined by Martin Fowler in his book *Refactoring: Improving the Design of Existing Code*, Addison Wesley, 2000. While a detailed discussion of refactoring belongs in a course on Software Engineering, the general practice of refactoring should be introduced in a first or second course on programming so that it becomes one of the tools that programmers use automatically as they develop programs.

We will briefly describe some of the refactoring that apply to COMP 1020. There are many additional refactorings but the refactorings described below provide a good introduction to the process of refactoring. For additional information on refactoring, see Fowler's book or his web site: http://www.refactoring.com.

It is not important to remember the names of the refactorings; the important thing to understand is that code can be improved in a systematic manner!

18.2.1 Rename Method

The rename method refactoring is exactly what it sounds like – the name of a method is changed to make the intent of the method clearer. For example, the method p(f) is not as descriptive as printFlights(flights).

18.2.2 Extract Method

The extract method refactoring is used for two primary purposes. First, it can be used to take a large method and subdivide it into smaller methods. If each of the smaller methods is named appropriately, the code should be much easier to read. As a general rule, each method should perform only one function; if a method is performing several functions together, then the method is a good candidate for this refactoring. The second use of extract method is to move code that is duplicated in several places into its own method. This has the effect of making the original method smaller and the processing more descriptive.

18.2.3 Convert Procedural Design to Objects

There is nothing wrong with writing procedural code. For example, many utility programs that manipulate files and provide information about the operating system are written in a procedural style. However, as programs grow and more functionality is added, it is more difficult to modify a procedural program than the equivalent object-oriented program. At this point, it is probably a good idea to re-design the program from an object-oriented point of view – moving information and processing that correspond to real-world entities into corresponding objects. While this refactoring will not provide any immediate benefits, it will make subsequent extensions of the program significantly easier.

18.2.4 Comments

If the refactorings described above are followed, the number of comments that are required should be reduced. For example, a method named process() is not very descriptive but if it is refactored into processMonthEnd() which calls a method determineInterestOnBalance() and another method updateCurrentBalance(), the methods become much more self explanatory. You do require comments if the processing is not obvious or if you are not certain about what is required in a method.

Fowler uses the following guidelines (he also identifies many other guidelines) to refactoring:

- "when you find the need to write a comment, first try to refactor the code so that any comment becomes superfluous." (page 88)
- "when you find you have to add a feature to a program, and the program's code is not structured in a convenient way to add the feature, first refactor the program to make it easier to add the feature, then add the feature." (page 7)

18.3 Design

The advocates of eXtreme Programming (XP) have been criticized for avoiding design as they develop systems. This is not a true portrait of XP – XP devotees do not avoid design, instead, they attempt to reduce the amount of design that is carried out before any of the system is implemented because they know that many design decisions will end up being changed as the implementation progresses. The title of Fowler's book *Refactoring: Improving the Design of Existing Code*, emphasizes the fact that design is important in XP. What XP'ers don't like is *BDUF* (*Big Design Up Front* – the engineering approach). Instead, they attempt to design a portion of the system, implement it, refactor it to improve the design or readability, and then proceed to the next iteration. So the process is one of design/implement/refactor, with not a lot of time spent on one specific activity at any particular point in time.

Do the simplest thing that could possibly work (*YAGNI – You Aren't Going to Need it*), that is, don't over-design for circumstances that **might** occur at some point in the future. **Design for the current requirements**. (See the quotation about simplicity on page 1 of these notes.) It is not easy to develop programs so that they contain only the required elements but also so that new functionality can be added to the programs with relative ease. However, the design/implement/refactor process makes this goal more attainable than if the engineering approach is used in which most or all of the design is carried out before implementation begins.

18.4 Flights Example

In the following example, we need to maintain information about a collection of airline flights. We want to be able to read an existing collection of flights between two cities, make changes to the flights by adding and deleting flights, determine which flights travel between two cities, and determine the flight between two cities that has the cheapest cost.

18.4.1 Iteration 1

We begin using the "Code Like Hell" model of program development, that is, we do no initial design at all and just code as fast as we can without any thought for what the consequences may be. (Note that doing no design is just as bad as doing a complete design before beginning to implement a system.) The data structure that is used is parallel arrays – a collection of arrays, each of which contains one value related to a flight.

```
public static final int NUM_FLIGHTS = 6;

public static void main(String[] parms)
{
     int[] flightNumbers = new int [NUM_FLIGHTS];
     String[] flightOrigins = new String[NUM_FLIGHTS];
     String[] flightDestinations = new String[NUM_FLIGHTS];

     createFlights(flightNumbers, flightOrigins, flightDestinations);
     printFlights(flightNumbers, flightOrigins, flightDestinations);
     printTrips(flightNumbers, flightOrigins, flightDestinations,
            "Winnipeg", "Toronto");
}
```

```
public static void createFlights(int[] flightNumbers, String[] flightOrigins,
                                                      String[] flightDestinations)
{
     int currentFlight;
     int[] numbers = {100, 200, 300, 400, 500, 600};
     String[]origins={"Toronto","Winnipeg","Toronto","Ottawa","Ottawa","Winnipeg"};
     String[]destinations={"Winnipeg","Toronto","Montreal","Toronto","Montreal",
                                                                       "Toronto"};

     for (currentFlight=0; currentFlight<NUM_FLIGHTS; currentFlight++)
     {
        flightNumbers[currentFlight] = numbers[currentFlight];
        flightOrigins[currentFlight] = origins[currentFlight];
        flightDestinations[currentFlight] = destinations[currentFlight];
     }
}
```

```
public static void printTrips(int[] flightNumbers, String[] flightOrigins,
         String[] flightDestinations, String origin, String destination)
{    // print all flights that travel directly between origin and destination
     int currentFlight;
     for (currentFlight=0; currentFlight<flightOrigins.length; currentFlight++)
     {
        if ((flightOrigins[currentFlight].equals(origin)) &&
             (flightDestinations[currentFlight].equals(destination)))
        {
           System.out.println("Flight: " +flightNumbers[currentFlight]
                                         +" travels from " +origin
                                         +" to " +destination);
        }
     }
}
```

```
public static void printFlights(int[] flightNumbers, String[] flightOrigins,
                                                      String[] flightDestinations)
{
     int currentFlight;

     for (currentFlight=0; currentFlight<flightOrigins.length; currentFlight++)
     {
        toString(flightNumbers[currentFlight], flightOrigins[currentFlight],
                                        flightDestinations[currentFlight]);
     }
}
```

```
public static void toString(int number, String origin, String destination)
{
     System.out.println(number +" " +origin +" " +destination);
}
```

18.4.2 Iteration 2

After very proudly showing the first iteration to our project manager and receiving a smack on the side of the head instead of lavish praise, we take a look at the code and realize that although it works correctly (so far), it looks ugly and will become even uglier as we extend the functionality. The most significant problem is that by using parallel arrays, we have to pass all of the arrays to each method that is to perform any processing. As we add additional flight variables later and also permit the user to add and delete flights, maintaining the arrays will become a major headache. To simplify the organization of the data, we move the flight information into a Flight class, following the *Convert Procedural Design to Objects* refactoring.

```
public static void main(String[] parms)
{
     Flight[] flights;
     flights = createFlights();
     printFlights(flights);
     printTrips(flights, "Winnipeg", "Toronto");
}
```

```
public static Flight[] createFlights()
{
     Flight[] flights = new Flight[6];
     flights[0] = new Flight(100, "Toronto", "Winnipeg");
     flights[1] = new Flight(200, "Winnipeg", "Toronto");
     flights[2] = new Flight(300, "Toronto", "Montreal");
     flights[3] = new Flight(400, "Ottawa", "Toronto");
     flights[4] = new Flight(500, "Ottawa", "Montreal");
     flights[5] = new Flight(600, "Winnipeg", "Toronto");
     return flights;
}
```

```
public static void printFlights(Flight[] myFlights)
{
     int currentFlight;
     for (currentFlight=0; currentFlight<myFlights.length; currentFlight++)
     {
         System.out.println(myFlights[currentFlight]);
     }
}
```

```
public static void printTrips(Flight[] myFlights, String origin,
                                                            String destination)
{
     int currentFlight;
     Flight flightObject;

     for (currentFlight=0; currentFlight<myFlights.length; currentFlight++)
     {
         flightObject = myFlights[currentFlight];
         if (flightObject.getOrigin().equals(origin) &&
                             flightObject.getDestination().equals(destination))
         {
             System.out.println(flightObject);
         }
     }
}
```

```
public class Flight
{
     private int flightNumber;
     private String flightOrigin;
     private String flightDestination;

     public Flight(int flightNumber, String flightOrigin, String flightDestination)
     {
        this.flightNumber = flightNumber;
        this.flightOrigin = flightOrigin;
        this.flightDestination = flightDestination;
     }

     public String getOrigin()
     {
        return flightOrigin;
     }

     public String getDestination()
     {
        return flightDestination;
     }

     public String toString()
     {
        return flightNumber +" " +flightOrigin +" " +flightDestination;
     }
}
```

18.4.3 Iteration 3

Iteration 2 is much cleaner than Iteration 1 – the data structure (one array of Flight objects) is more manageable (all data can be passed using just the `flights` variable) and can be modified more easily in the future.

The code to compare an origin and a destination requires two accessors in the Flight class:

```
if (flightObject.getOrigin().equals(origin) &&
     flightObject.getDestination().equals(destination))
```

While using two accessors is not a problem, the processing is a bit cumbersome and the condition could easily be moved into a method (checkTrip) in the Flight class that compares both the origin and the destination at the same time. This simplifies the main class logic and also removes the need for the two accessors.

```
public static void printTrips(Flight[] myFlights, String origin,
                                                            String destination)
{
     int currentFlight;
     Flight flightObject;

     for (currentFlight=0; currentFlight<myFlights.length; currentFlight++)
     {
        flightObject = myFlights[currentFlight];
        if (flightObject.checkTrip(origin, destination))
        {
           System.out.println(flightObject);
        }
     }
}
```

```
public class Flight
{
     private int flightNumber;
     private String flightOrigin;
     private String flightDestination;

     public Flight(int flightNumber, String flightOrigin, String flightDestination)
     {
        this.flightNumber = flightNumber;
        this.flightOrigin = flightOrigin;
        this.flightDestination = flightDestination;
     }
```

```
     public boolean checkTrip(String origin, String destination)
     {
        return (origin.equals(flightOrigin)
                  && (destination.equals(flightDestination)));
     }
     . . .
}
```

18.4.4 Iteration 4

The Flight class that maintains flight information works well but there is too much work being performed in the main class to manage the collection of flights. If other programs have to perform similar processing, they would also be responsible for creating the collection of flights and maintaining this collection. Once again, this is a good place to use the *Convert Procedural Design to Objects* refactoring by moving the processing required to maintain the collection of flights into its own class – Flights. We introduced this idea in Chapter 8 – Object Orientation Practices when we created the class CreditCardProcessing that maintained the collection of credit cards and the associated processing.

```
public static void main(String[] parms)
{
     Flights flights;
     flights = new Flights();
     flights.printFlights();
     flights.printTrips("Winnipeg", "Toronto");
}
```

```
public class Flights
{
     private Flight[] flights;

     public Flights()
     {
        createFlights();
     }
```

```
     private void createFlights()
     {
        flights = new Flight[6];
        flights[0] = new Flight(100, "Toronto", "Winnipeg");
        flights[1] = new Flight(200, "Winnipeg", "Toronto");
        flights[2] = new Flight(300, "Toronto", "Montreal");
        flights[3] = new Flight(400, "Ottawa", "Toronto");
        flights[4] = new Flight(500, "Ottawa", "Montreal");
        flights[5] = new Flight(600, "Winnipeg", "Toronto");
     }
```

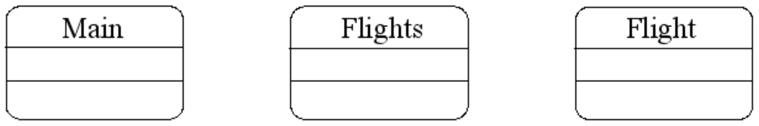
```
     public void printTrips(String origin, String destination)
     {
        Flight flightObject;
        int currentFlight;

        for (currentFlight=0; currentFlight<flights.length; currentFlight++)
        {
           flightObject = flights[currentFlight];
           if (flightObject.checkTrip(origin, destination))
           {
              System.out.println(flightObject);
           }
        }
     }
```

```
     public void printFlights()
     {
        int currentFlight;

        for (currentFlight=0; currentFlight<flights.length; currentFlight++)
        {
           System.out.println(flights[currentFlight]);
        }
     }
}
```

This organization isolates the flight processing in one class that is independent of the programs that require access to the flights. So now the main class simply instantiates the Flights class and then issues commands (such as printFlights) to the Flights class for processing. As a result, the main class has become trivial.

Note that the individual flights are still hard coded in the createFlights method; they will eventually be read from a file but, for now, it does not cause any problems to leave them as they are.

We have now arrived at a decent design for the program consisting of 3 primary classes. The Main class performs the processing specific to one application (for example, add and delete flights). The Flights class maintains the collection of flights and processes any modification requests. Any program that requires access to the flights must use the Flights class. The Flight class defines one specific flight. The following diagram identifies the primary classes used in this application. (This is actually a simple UML diagram; additional information is normally added to a UML diagram but for our purposes, this is sufficient.)



This is the design that a good designer would have created at the beginning. It took us longer to get to this point because we skipped the initial design stage and started to code immediately. However, the refactorings have helped to guide us to a good design. So, while "code like hell" is not a good strategy, the appropriate use of refactorings can improve a bad (or non-existent) initial design. Obviously though, it would have been better to have spent a small amount of time on the design and arrived at the 3 classes shown above prior to beginning programming.

18.4.5 Iteration 5

In the next iteration, we add a cost instance variable to Flight and a printBestCost method to Flights. Since we now have grouped related data and processing together, these changes are quite easy to implement. We add the cost variable to Flight and to the createFlights method that creates the individual flights. (We assume for now that cost is an integer variable; even if this changes at a later time, it will not take much effort to change the type of cost.) Then, we need to add a cost accessor to the Flight class but we can reuse the checkTrip method in Flight to determine whether or not a flight satisfies the origin-destination combination.

```
public class Refactor
{
     public static void processCommands(Flights flights)
     {
        flights.printTrips("Winnipeg", "Toronto");
        flights.printBestCost("Winnipeg", "Toronto");
     }
     . . .
```

```
public class Flights
{
     private void createFlights()
     {
        flights = new Flight[6];
        flights[0] = new Flight(100, "Toronto", "Winnipeg", 400);
        flights[1] = new Flight(200, "Winnipeg", "Toronto", 200);
        flights[2] = new Flight(300, "Toronto", "Montreal", 200);
        flights[3] = new Flight(400, "Ottawa", "Toronto", 100);
        flights[4] = new Flight(500, "Ottawa", "Montreal", 150);
        flights[5] = new Flight(600, "Winnipeg", "Toronto", 600);
     }

     public void printBestCost(String origin, String destination)
     {
        Flight flightObject;
        int currentFlight;
        int currentCost;
        Flight bestFlight;
        int bestCost;

        bestCost = flights[0];  // this will not work if there are no flights
        bestFlight = null;
        for (currentFlight=0; currentFlight<flights.length; currentFlight++)
        {
           flightObject = flights[currentFlight];
           currentCost = flightObject.getFlightCost();
           if (flightObject.checkTrip(origin, destination)
                                                && (currentCost<bestCost))
           {
              bestCost = currentCost;
              bestFlight = flightObject;
           }
        }
        if (bestFlight != null)
        {
           System.out.println("\nCheapest flight between " +origin
                                +" and " +destination +" is: \n" +bestFlight);
        }
     }
     . . .
```

```
public class Flight
{
     private int flightNumber;
     private String flightOrigin;
     private String flightDestination;
     private int flightCost;

     public Flight(int flightNumber, String flightOrigin, String flightDestination,
             int flightCost)
     {
        this.flightNumber = flightNumber;
        this.flightOrigin = flightOrigin;
        this.flightDestination = flightDestination;
        this.flightCost = flightCost;
     }
     public int getFlightCost()
     {
        return flightCost;
     }
     . . .
```

In Chapter 8 – Object Orientation Practices, we stated that accessors and mutators should be avoided whenever possible. We could compare the cost of two flights inside the Flight class but we still require the cost of the cheapest flight and the associated flight information so this is one of the circumstances when using an accessor is appropriate.

18.4.6 Iteration 6

Now that we have the basic functionality defined, we begin to add the processing that permits the program to make changes to the current collection of flights. First, we add a deleteFlight method to Flights. This method will have an impact only on the Flights class.

```
public void deleteFlight(int flightNumber)
{
     Flight[] newFlightsCollection;
     Flight flightObject;
     int currentFlight;
     int found;

     found = -1;
     for (currentFlight=0; (currentFlight<flights.length) && (found==-1);
                                                          currentFlight++)
     {
        flightObject = flights[currentFlight];
        if (flightObject.getFlightNumber() == flightNumber)
        {
           found = currentFlight;
        }
     }
     if (found != -1)
     {
        System.arraycopy(flights,found+1,flights,found,
                                   flights.length-found-1);

        newFlightsCollection = new Flight[flights.length-1];

        System.arraycopy(flights,0,newFlightsCollection,0,
                                 newFlightsCollection.length);

        flights = newFlightsCollection;
     }
}
```

Also, since the number of commands that is being processed by the main class is continuing to increase, the commands are moved into their own method, processCommands.

```
public static void processCommands(Flights flights)
{
     flights.printFlights();
     flights.printTrips("Winnipeg", "Toronto");
     flights.printBestCost("Winnipeg", "Toronto");
     flights.deleteFlight(100);
     flights.printFlights();
}
```

18.4.7 Iteration 7

In the previous iteration, the accessor getFlightNumber managed to sneak into the Flight class so that we could compare two flight numbers when searching the collection of flights. Again, such accessors are not necessary if we examine the real purpose of the information – to compare two flight numbers. A method that performs this task should be added to the Flight class instead of adding an accessor to the Flight class and then having the Flights class perform the comparison.

```
public void deleteFlight(int flightNumber)
{
     Flight[] newFlightsCollection;
     Flight flightObject;
     int currentFlight;
     int found;

     found = -1;
     for (currentFlight=0; (currentFlight<flights.length) && (found==-1);
                                                             currentFlight++)
     {
        flightObject = flights[currentFlight];
        if (flightObject.compareFlightNumbers(flightNumber))
        {
           found = currentFlight;
        }
     }
     if (found != -1)
     {
        System.arraycopy(flights,found+1,flights,found,
                                   flights.length-found-1);

        newFlightsCollection = new Flight[flights.length-1];

        System.arraycopy(flights,0,newFlightsCollection,0,
                                 newFlightsCollection.length);

        flights = newFlightsCollection;
     }
}
```

Then the following method would be added to the Flight class.

```
public boolean compareFlightNumbers(int whichFlight)
{
     return (flightNumber == whichFlight);
}
```

18.4.8 Iteration 8

In the previous iteration, a Flight was deleted by removing it from the array and copying the Flight objects that came after it in the array one position to the left so that there are no unused locations in the array. The array must then be copied to an array of the correct size (1 element smaller than the previous array). While this processing is isolated in the Flights class so it will not be duplicated in application programs that use the Flights class, having to write the instructions to delete elements in an array is somewhat ugly when there is an existing Java class that will do this work for us – the ArrayList class. In this iteration, we change the data structure currently used in Flights (an array) to an ArrayList. Making this change is relatively simple since we have isolated all instructions that manipulate the collection of Flight objects in the one class, Flights.

```
public class Flights
{
     private ArrayList flights;

     public Flights()
     {
        createFlights();
     }
```

```
private void createFlights()
{
     flights = new ArrayList();
     flights.add(new Flight(100, "Toronto", "Winnipeg", 400));
     flights.add(new Flight(200, "Winnipeg", "Toronto", 200));
     flights.add(new Flight(300, "Toronto", "Montreal", 200));
     flights.add(new Flight(400, "Ottawa", "Toronto", 100));
     flights.add(new Flight(500, "Ottawa", "Montreal", 150));
     flights.add(new Flight(600, "Winnipeg", "Toronto", 600));
}
```

```
public void printFlights()
{
     int currentFlight;
     for (currentFlight=0; currentFlight<flights.size(); currentFlight++)
     {
        System.out.println(flights.get(currentFlight));
     }
}
```

Note that whenever an object is extracted from the ArrayList, the ArrayList element must be cast back to a Flight object.

```
public void printTrips(String origin, String destination)
{
     Flight flightObject;
     int currentFlight;

     for (currentFlight=0; currentFlight<flights.size(); currentFlight++)
     {
        flightObject = (Flight) flights.get(currentFlight);
        if (flightObject.checkTrip(origin, destination))
        {
           System.out.println(flightObject);
        }
     }
}
```

```
public void printBestCost(String origin, String destination)
{
     Flight flightObject;
     int currentFlight;
     int currentCost;
     Flight bestFlight;
     int bestCost;

     bestCost = Integer.MAX_VALUE;
     bestFlight = null;
     for (currentFlight=0; currentFlight<flights.size(); currentFlight++)
     {
        flightObject = (Flight) flights.get(currentFlight);
        currentCost = flightObject.getFlightCost();
        if (flightObject.checkTrip(origin, destination)&&(currentCost < bestCost))
        {
           bestCost = currentCost;
           bestFlight = flightObject;
        }
     }
     if (bestFlight != null)
     {
        System.out.println("\nCheapest flight is: \n" +bestFlight);
     }
}
```

```
public void deleteFlight(int flightNumber)
{
     Flight flightObject;
     int currentFlight;
     int found;

     found = -1;
     for (currentFlight=0; (currentFlight<flights.size()) && (found==-1);
                                                        currentFlight++)
     {
        flightObject = (Flight) flights.get(currentFlight);
        if (flightObject.compareFlightNumbers(flightNumber))
        {
           found = currentFlight;
        }
     }
     if (found != -1)
     {
        flights.remove(found);
     }
}
```

18.4.9 Iteration 9

Now that we have improved the way in which a collection of flights is stored, we can add an addFlight method to the Flights class with almost no effort.

```
public static void processCommands(Flights flights)
{
     flights.printFlights();
     flights.printTrips("Winnipeg", "Toronto");
     flights.printBestCost("Winnipeg", "Toronto");
     flights.deleteFlight(100);
     flights.addFlight(700, "Winnipeg", "Gimli", 50);
     flights.printFlights();
}
```

```
public void addFlight(int flightNumber, String flightOrigin,
                                String flightDestination, int flightCost)
{
     flights.add(new Flight(flightNumber, flightOrigin,
                                          flightDestination, flightCost));
}
```

18.4.10Iteration 10

When we created the addFlight method, we did not check for duplicate flights so now is a good time to add that processing. (We also did not print a message in deleteFlight if the flight did not exist; this can be added now too.) In the addFlight method, we must first search for a duplicate flight before inserting the new flight. The instructions to search for a flight have already been defined but they are inside the method deleteFlight. So we first refactor deleteFlight using *Extract to Method.* The original version of deleteFlight was:

```
public void deleteFlight(int flightNumber)
{
     Flight flightObject;
     int currentFlight;
     int found;

     found = -1;
     for (currentFlight=0; (currentFlight<flights.size()) && (found==-1);
                                                          currentFlight++)
     {
        flightObject = (Flight) flights.get(currentFlight);
        if (flightObject.compareFlightNumbers(flightNumber))
        {
           found = currentFlight;
        }
     }
     if (found != -1)
     {
        flights.remove(found);
     }
}
```

Extracting the instructions that locate a flight and moving them to a method named findFlight is straightforward:

```
private int findFlight(int flightNumber)
{
     Flight flightObject;
     int currentFlight;
     int found;

     found = -1;
     for (currentFlight=0; (currentFlight<flights.size()) && (found==-1);
                                                                currentFlight++)
     {
        flightObject = (Flight) flights.get(currentFlight);
        if (flightObject.compareFlightNumbers(flightNumber))
        {
           found = currentFlight;
        }
     }
     return found;
}
```

After extracting the processing that searches for a flight, deleteFlight becomes slightly simpler and adding a message when a flight is not found is trivial.

```
public void deleteFlight(int flightNumber)
{
     int found;

     found = findFlight(flightNumber);
     if (found != -1)
     {
        flights.remove(found);
        System.out.println("\n\nFlight " +flightNumber +" was deleted.\n");
     }
     else
     {
        System.out.println("\n\nFlight " +flightNumber +" does not exist.\n");
     }
}
```

We can now ensure that a new flight does not have the same flight number as an existing flight:

```
public void addFlight(int flightNumber, String flightOrigin,
                                    String flightDestination, int flightCost)
{
     int found;

     found = findFlight(flightNumber);
     if (found == -1)
     {
        flights.add(new Flight(flightNumber,
                                 flightOrigin, flightDestination, flightCost));
        System.out.println("\n\nFlight " +flightNumber +" was added.\n");
     }
     else
     {
        System.out.println("\n\nFlight " +flightNumber +" already exists.\n");
     }
}
```

Notice that a very simple refactoring has made the check for duplicates trivial. This is a good example of how code can be improved iteratively without having to design all of the required methods in advance.

18.4.11Iteration 11

Now that the basic structure of the program is relatively stable, we can read the flight information from a file instead of having it hard-coded in the program.

```
public class Flights
{
     private ArrayList flights;

     public Flights()
     {
        createFlights("flights.txt");
     }
```

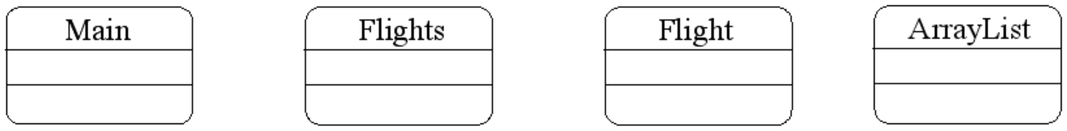
```
private void createFlights(String filename)
{
     Flight new Flight;
     BufferedReader fileIn;
     String inputLine;

     flights = new ArrayList();
     try
     {
        fileIn = new BufferedReader(FileReader(fileName));

        inputLine = fileIn.readLine();
        while (inputLine != null)
        {
           newFlight = createFlight(inputLine);
           flights.add(newFlight);
           inputLine = fileIn.readLine();
        }
        fileIn.close();
     }
     catch (IOException ioe)
     {
        ioe.printStackTrace();
     }
}

private Flight createFlight(String inputLine)
{
     Flight newFlight;
     String[] result;

     result = inputLine.split("\\s+");
     newFlight = new Flight((int) Integer.parseInt(result[0]), result[1],
                              result[2], (int) Integer.parseInt(result[3]));
     return newFlight;
}
```

18.5 Complete Flights Example

The complete program is shown below. Note that the program has grown from a small program that showed what was necessary to a complete program without a lot of development pain.

As a general rule, it is much easier to make changes within a class than to make changes that involve multiple classes. While it takes practice to become familiar with good class organization, much of the effort requires only putting some thought into where data and processing should be placed. The most important lesson is that if we **design** the classes correctly initially, we will likely save the iterations that move us to the correct class design. In the example examined in this chapter, it wasn’t until iteration 4 that a good class design was identified.

This design was improved by moving the collection of Flight objects into its own class (an ArrayList). This is an important modification because it permits us to replace one data structure with another (such as a linked list – see Chapter 15 – Linked Lists) with little effort.



If the classes are not designed well in the beginning, we can still iterate (or refactor) towards a good design but this process does require some additional effort.

```
import java.util.ArrayList;
import java.io.*;

public class Refactor
{
     public static void main(String[] parms)
     {
        Flights flights;

        flights = new Flights();
        flights.printFlights();
        processCommands(flights);
        flights.printFlights();
     }

     public static void processCommands(Flights flights)
     {
        flights.printTrips("Winnipeg", "Toronto");
        flights.printBestCost("Winnipeg", "Toronto");
        flights.deleteFlight(100);
        flights.addFlight(700, "Winnipeg", "Gimli", 50);
     }
}
```

```
public class Flights
{
     private ArrayList flights;

     public Flights()
     {
        createFlights("flights.txt");
     }

     private void createFlights(String fileName)
     {
        Flight newFlight;
        BufferedReader fileIn;
        String inputLine;

        flights = new ArrayList();
        try
        {
           fileIn = new BufferedReader(new FileReader(fileName));

           inputLine = fileIn.readLine();
           while (inputLine != null)
           {
              newFlight = createFlight(inputLine);
              flights.add(newFlight);
              inputLine = fileIn.readLine();
           }
           fileIn.close();
        }
        catch (IOException ioe)
        {
           ioe.printStackTrace();
        }
     }

     private Flight createFlight(String inputLine)
     {
        Flight newFlight;
        String[] result;

        result = inputLine.split("\\s+");
        newFlight = new Flight((int) Integer.parseInt(result[0]), result[1],
                                 result[2], (int) Integer.parseInt(result[3]));
        return newFlight;
     }

     public void printFlights()
     {
        int currentFlight;

        System.out.println("\nFlights:\n");
        for (currentFlight=0; currentFlight<flights.size(); currentFlight++)
        {
           System.out.println(flights.get(currentFlight));
        }
        System.out.println();
     }
```

```
    public void printTrips(String origin, String destination)
    {
       Flight flightObject;
       int currentFlight;
       int found;

       found = 0;

       System.out.println("\nList of Flights between "
                                  +origin +" and " +destination +"\n");
       for (currentFlight=0; currentFlight<flights.size(); currentFlight++)
       {
          flightObject = (Flight) flights.get(currentFlight);
          if (flightObject.checkTrip(origin, destination))
          {
             System.out.println(flightObject);
             found++;
          }
       }
       if (found == 0)
       {
          System.out.println("\nThere are no flights between "
                                     +origin +" and " +destination +"\n");
       }
    }

    public void printBestCost(String origin, String destination)
    {
       Flight flightObject;
       int currentFlight;
       int currentCost;
       Flight bestFlight;
       int bestCost;

       bestCost = Integer.MAX_VALUE;
       bestFlight = null;
       for (currentFlight=0; currentFlight<flights.size(); currentFlight++)
       {
          flightObject = (Flight) flights.get(currentFlight);
          currentCost = flightObject.getFlightCost();
          if (flightObject.checkTrip(origin, destination)
                                              &&(currentCost < bestCost))
          {
             bestCost = currentCost;
             bestFlight = flightObject;
          }
       }
       if (bestFlight != null)
       {
          System.out.println("\nCheapest flight between "
                      +origin +" and " +destination +" is: \n" +bestFlight);
       }
       else
       {
          System.out.println("\nNo flights between "
                      +origin +" and " +destination +" were found.");
       }
    }
    private int findFlight(int flightNumber)
    {
       Flight flightObject;
       int currentFlight;
       int found;

       found = -1;
       for (currentFlight=0; (currentFlight<flights.size()) && (found==-1);
                                                            currentFlight++)
       {
          flightObject = (Flight) flights.get(currentFlight);
          if (flightObject.compareFlightNumbers(flightNumber))
          {
             found = currentFlight;
          }
       }
       return found;
    }
```

```
    public void deleteFlight(int flightNumber)
    {
       int found;

       found = findFlight(flightNumber);
       if (found != -1)
       {
          flights.remove(found);
          System.out.println("\n\nFlight " +flightNumber
                                        +" was successfully deleted.\n");
       }
       else
       {
          System.out.println("\n\nFlight " +flightNumber +" does not exist.\n");
       }
    }
```

```
    public void addFlight(int flightNumber, String flightOrigin,
                                    String flightDestination, int flightCost)
    {
       int found;

       found = findFlight(flightNumber);
       if (found == -1)
       {
          flights.add(new Flight(flightNumber, flightOrigin,
                                          flightDestination, flightCost));
          System.out.println("\n\nFlight " +flightNumber
                                        +" was successfully added.\n");
       }
       else
       {
          System.out.println("\n\nFlight " +flightNumber +" already exists.\n");
       }
    }
}
```

```
public class Flight
{
     private int flightNumber;
     private String flightOrigin;
     private String flightDestination;
     private int flightCost;

     public Flight(int flightNumber, String flightOrigin, String flightDestination,
                                                                      int flightCost)
     {
        this.flightNumber = flightNumber;
        this.flightOrigin = flightOrigin;
        this.flightDestination = flightDestination;
        this.flightCost = flightCost;
     }

     public boolean checkTrip(String origin, String destination)
     {
        return (flightOrigin.equals(origin)
                                   && flightDestination.equals(destination));
     }

     public boolean compareFlightNumbers(int whichFlight)
     {
        return (flightNumber == whichFlight);
     }

     public int getFlightCost()
     {
        return flightCost;
     }

     public String toString()
     {
        return "Flight " +flightNumber +" travels from "
                         +flightOrigin +" to " +flightDestination
                         +" and flight costs $" +flightCost;
     }
}
```

18.6 Summary

The example above illustrates how a program can be developed in small iterations, with refactoring used to improve code after it has been written, rather than attempting to write perfect code from the beginning or using an engineering approach in which the entire system is designed before any code is written. The iterations shown above are not the only way in which the code could have been developed, there are other ways in which the code could have been developed but most of them would end up with essentially the same result.

One of the blind spots that some programmers have is that they may think of an elegant way of solving a more complex problem than the one that they are supposed to be solving. This leads to over-engineering and is wasteful of resources (it takes longer to develop more complex code than it does to develop simple code that solves the problem). Over-engineering also leads to "code bloat" in which a system contains many features that were added by programmers who thought that the features would be nice to have, instead of by the users who will actually use the system. For anyone who disagrees, take a look at the features in Microsoft's Office suite of programs.

To make refactoring easier, many of the IDE's (Interactive Development Environments), such as Eclipse, used to develop large systems include many of the refactorings described in Fowler's book. So, for example, if the name of a method is modified, the IDE ensures that all programs in the system are modified appropriately.

The most important aspect of refactoring is not remembering the names of all of the refactorings and attempting to use the refactorings as often as possible. Instead, you should learn to improve your code as the code is developed. A good programmer will improve his/her code whether or not he/she knows the name of the corresponding refactoring. Kent Beck, the "father" of eXtreme Programming, refers to ugly code as code that "smells". By continually reviewing your code for bad smells (and then removing them), you ensure that the code that you produce will be of high quality.

19 MISCELLANEOUS TOPICS

19.1 Introduction

In this chapter we examine a few miscellaneous topics and also some infrequently used programming techniques (the Reflection package). This chapter is provided for information only.

19.2 System Streams

Until now, output has been directed to the system console using `System.out`. `System.out` is a "stream object" that is automatically created whenever a program runs (`out` is a class variable in the System class). A stream is simply a continuous collection of characters. The statement

```
System.out.print("abc");
```

causes the characters "`abc`" to be added to the stream of characters sent to the system console (i.e. the user's computer monitor). The statement

```
System.out.println("abc");
```

causes the characters "`abc`" to be added to the stream of characters sent to the system console and then adds the characters that cause the system console to begin a new line (more on this later).

There is also an input stream that can be used to read information entered by the user at the system console – `System.in`. `System.in` is also a stream that is automatically created whenever a program runs. The following code segment illustrates the use of `System.in`.

```
int inValue;

inValue = System.in.read();
while(inValue != -1)
{
     System.out.println(inValue);
     inValue = System.in.read();
}
```

Unfortunately, `System.in` is more complicated than `System.out`. First, notice that the input value returned by `System.in` is an integer. If the characters "`abc`" are entered, the following output is generated by the program above:

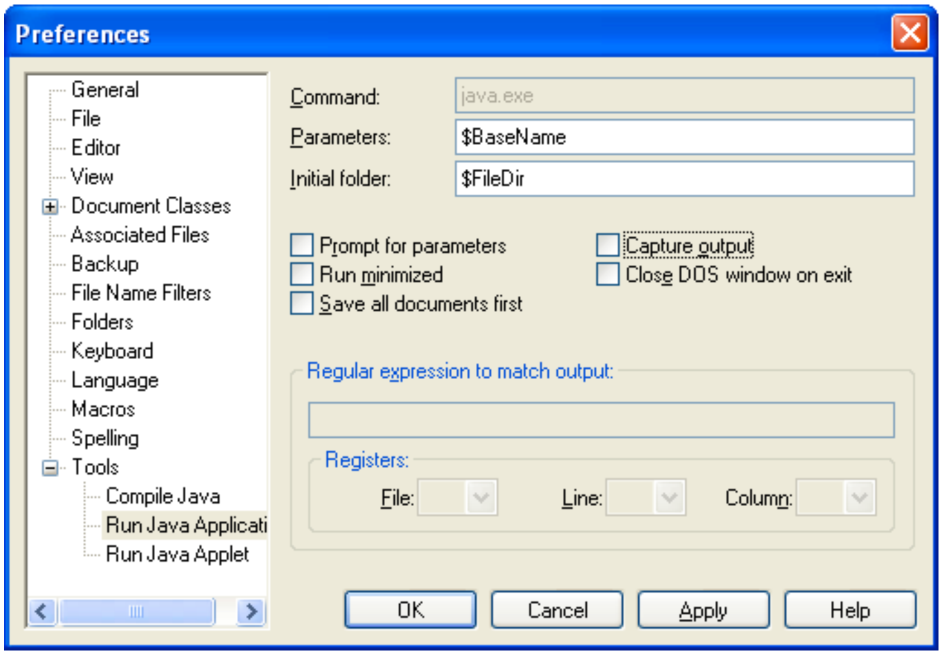
```
97
98
99
13
10
```

The integer 97 represents the character 'a', 98 represents 'b', and 99 represents 'c'. The int 13 represents the **carriage return** character ('\r') and the int 10 represents the **newline** (or **linefeed**) character ('\n'). The carriage return and newline characters are generated when the user types the "enter" key at the end of the input line. (The value used to represent each character is the "Unicode" value which is discussed in Chapter 5 – Strings.)

If you are using TextPad, you must ensure that TextPad has been configured properly to permit values to be entered via the system console. If you receive the error message:

```
The handle is invalid
java.io.IOException: The handle is invalid
at java.io.FileInputStream.readBytes(Native Method)
at java.io.FileInputStream.read(FileInputStream.java:194)
. . .
```

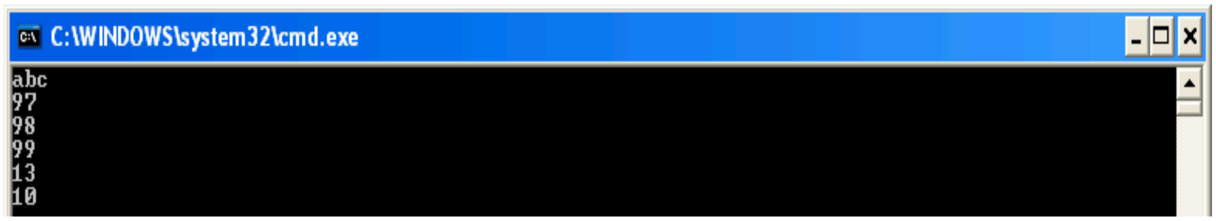
then TextPad has not been configured to permit input from the system console. To correct this problem, select the Configure menu and the Preferences menu item. Expand Tools by clicking on the + beside Tools and then click on Run Java Application. Ensure that the Capture output checkbox is **not** checked.



Command: java.exe
Parameters: $BaseName
Initial folder: $FileDir

- [ ] Prompt for parameters
- [ ] Run minimized
- [ ] Save all documents first
- [ ] Capture output
- [ ] Close DOS window on exit

When the program is executed, the following window is displayed:



```
abc
97
98
99
13
10
```

The program segment can be improved by converting each integer to the corresponding char (character).

```
int inValue;
char inChar;

inValue = System.in.read();
while(inValue != -1)
{
     inChar = (char) inValue;
     System.out.println(inChar);
     inValue = System.in.read();
}
```

This program generates the following output (note that there are extra lines generated by the carriage return and newline characters):

```
a
b
c


```

This is an improvement but still does not make processing the input data in the program particularly easy. The next step is to collect the characters into a String. (We will examine String processing in detail in Chapter 5 – Strings.)

```
int inValue;
char inChar;
String inString;

inString = "";
inValue = System.in.read();
while(inValue != -1)
{
     inChar = (char) inValue;
     inString = inString + inChar;
     inValue = System.in.read();
}
System.out.println(inString);
```

The program generates the following output:

```
abc
```

There are still several points to note about this program segment. First, the string inString contains not only the characters “abc” but also the carriage return and newline characters at the end. This may cause problems if the string is manipulated by the program. Secondly, we have not yet addressed the problem of how the user indicates that the input has been terminated. Although the program above seems to indicate that characters are provided to the program as soon as they are typed, this is not true. Characters are provided to the program only after the user types the “enter” key at the end of each line. When the user has completed entering all characters, the user must indicate to the operating system that the end of the data (**end-of-file**) has been reached. This is accomplished on the Windows platform by typing the characters **ctrl-z** and then the **enter** key. If you are using a **Macintosh or Unix/Linux** operating system, there are two differences from the information described above. First, only the newline character ('\n') is used to mark the end of each line. Secondly, the character **ctrl-d** is used to indicate end-of-file.

When the end-of-file character is typed by the user, Java indicates this fact to the program by returning the integer value -1.

Finally, if the program segment is compiled as it is, a compile error is generated because the Java language requires that all input be enclosed in what is referred to as a “**try-catch block**”, as shown below.

```
int inValue;
char inChar;
String inString;

try
{
     inString = "";
     inValue = System.in.read();
     while(inValue != -1)
     {
        inChar = (char) inValue;
        inString = inString + inChar;
        inValue = System.in.read();
     }
     System.out.println(inString);
}
catch (IOException ioe)
{
     ioe.printStackTrace();
}
```

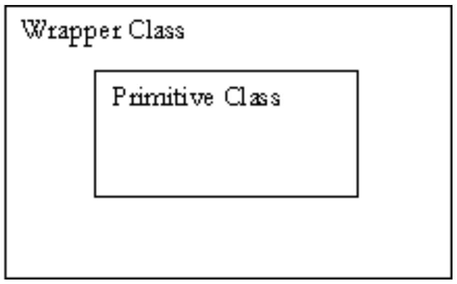
This program segment now compiles and executes correctly. (We will ignore the format of the try-catch block for now.) However, it is important to note that the loop continues until all of the input has been read, not just until one line of input has been read. **If you need to**

process the input one line at a time, you must modify the loop so that it stops when a newline character is encounted.

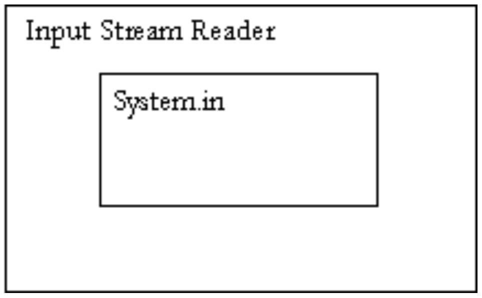
19.3 File Processing Wrapper Classes

While the program discussed above permits the programmer to read input data from the console, the program is quite awkward and requires additional work before input data can be processed line by line. To make programming easier, Java provides some wrapper classes that can be wrapped around primitive input/output objects in order to simplify the programming.

The wrapper classes discussed in Chapter 3 – Objects were used to convert a primitive data value to an object. An input/output wrapper class is a class that can be used to simplify the processing of another object. The wrapper object performs the processing required to simplify the use of the object that it contains. The use of wrapper classes is common in object-oriented languages such as Java since it permits the programmer to use either the primitive object directly or to use the wrapper object.



An InputStreamReader object can be wrapped around System.in in order to read multiple characters per line. The programmer provides an array and the wrapper class object reads characters from System.in until the array is full or until the end of the input line is encounted.



```
InputStreamReader inStream;
int inCount;
char[] inChars = new char[10];

try
{
     inStream = new InputStreamReader(System.in);
     inCount = inStream.read(inChars, 0, 10);
     while(inCount != -1)
     {
         System.out.println(inChars);
         inCount = inStream.read(inChars, 0, 10);
     }
     inStream.close();
}
catch (IOException ioe)
{
     ioe.printStackTrace();
}
```

Since an additional Java Input/Output class is now being used, this class must be included by adding the following statement to the beginning of the program.

```
import java.io.*;
```

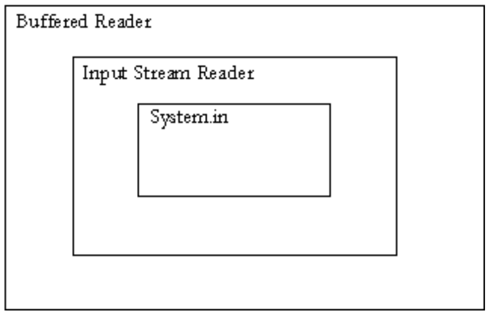
Unfortunately, this technique still is not perfect. If there are more than 10 characters on a particular line, the first 10 characters are returned and then the next read statement returns the next set of characters (up to the maximum of 10). Also, the carriage return and newline characters are returned as part of the string and are included in the count of the number of characters returned (inCount). Once reading from a file is complete, the file should be "closed" by calling the object's close() method.

By making the input array (inChars) very large, the programmer can read a complete line each time through the loop but the programmer still must remove the carriage return and newline characters from the end of each line (or just subtract 2 from inCount before processing inChars).

While this wrapper class simplifies the processing of `System.in` somewhat, it would be even more convenient if one class could read an entire line at a time and also remove the carriage return and newline characters automatically.

19.4 Buffered Console Input

Fortunately, Java contains another wrapper class that can be used to support line-by-line reading from System.in. The BufferedReader class can be wrapped around the InputStreamReader wrapper class which is wrapped around System.in. The BufferedReader class performs the functions that we have identified: it reads an entire line at a time and also removes the control characters from the end of each line.



The following program segment illustrates how an InputStreamReader object can be wrapped around System.in and then how a BufferedReader object can be wrapped around the InputStreamReader object. The result returned by the BufferedReader object is a string instead of an array of characters.

```
InputStreamReader inStream;
BufferedReader console;
String inputLine;

try
{
     inStream = new InputStreamReader(System.in);
     console = new BufferedReader(inStream);

     inputLine = console.readLine();
     while (inputLine != null)
     {
        System.out.println(inputLine);
        inputLine = console.readLine();
     }
     console.close();
}
catch (IOException ioe)
{
     ioe.printStackTrace();
}
```

This program is significantly easier than reading from System.in one integer at a time. Again, as with the preceding programs, the input statements must be enclosed in a try-catch block.

This program eliminates most of the problems that we encounted earlier when reading from the system console. The program reads a line at a time and automatically removes the carriage return and newline characters from the input. Also, there is no predefined length for the input string – the buffered reader object sets the length of the string to the exact number of characters (not including newline and carriage return characters) that are read from a particular line. There may be any number of characters on the line, including blanks and special characters (such as control characters).

A few notes are still required to explain the program structure. First, end-of-file is now indicated by Java by a null string (instead of the value -1 which was used earlier). It is important to note that a null string is not the same as an empty string (""). This will be discussed further Chapter 5 – Strings. End-of-file is still indicated by the user by typing ctrl-z and enter (or ctrl-d on Unix-based systems). Once reading from a file is complete, the file is closed by calling the object's close() method. The additional classes also require that java.io.* be imported at the beginning of the program.

19.5 Files and the End-of-Line Character(s)

Windows systems use \r\n to mark the end of each line while Linux/Max systems just use \n. If you use \n when writing to a file on a Windows system, some Windows applications (such as Notepad) do not recognize the end-of-line correctly.

TextPad is not as brain-dead as Notepad and TextPad will recognize the end of line correctly on a Windows machine whether both \r\n are written or just \n is written. The following program segment illustrates 4 ways of copying the contents of one file to another file. To ensure that the correct end-of-line sequence is generated when writing to a file, it is preferable to use the .println() method instead of using \n.

```
fileIn = new BufferedReader(new FileReader("in.txt"));
fileOut = new PrintWriter(new FileWriter("out.txt"));

String newLine = System.getProperty("line.separator");

inputLine = fileIn.readLine();
while (inputLine != null)
{
     //fileOut.print(inputLine +"\n");     //doesn't work with Notepad
     //fileOut.print(inputLine +"\r\n");   //works with all programs on Windows
     //fileOut.print(inputLine +newLine);  //works with all programs on all systems
     fileOut.println(inputLine);           //works with all programs on all systems
     inputLine = fileIn.readLine();
}
fileIn.close();
fileOut.close();
```

19.6 Non-Rectangular Arrays

Most two-dimensional arrays are rectangular, that is, the number of columns in each row is the same for all rows. However, this need not be true – it is perfectly valid (although somewhat unusual) to create an array that contains differing numbers of columns in each row. For example, the following array has a triangular shape.

```
public static void triangularArray()
{
     int[][] array;
     int row;
     int column;

     array = new int[4][];
     array[0] = new int[1];
     array[1] = new int[2];
     array[2] = new int[3];
     array[3] = new int[4];

     for (row=0; row<array.length; row++)
     {
        for (column=0; column<array[row].length; column++)
        {
           array[row][column] = (row)*50 + (column+1)*10;
           System.out.print(array[row][column] +"\t");
        }
        System.out.println();
     }
}
```

When the contents of this array are printed, the following output is generated:

```
10
60   70
110  120    130
160  170    180    190
```

Non-rectangular arrays are unusual but Java makes it easy to define such arrays. The non-rectangular array emphasizes the fact that a multidimensional array is really just an array in which each element of the array is another array.

19.7 Parallel Arrays

In Chapter 2 – Growing Algorithms, we examined how parallel arrays can be used to maintain multiple pieces of information about a collection of entities. This style of programming is required if object orientation is not a feature of the programming language being used. Multi-dimensional arrays can also be used to maintain multiple pieces of information about entities if an array can be defined so that it can contain different types of data. (Obviously, this does take advantage of object orientation.) The program below illustrates how this can be accomplished.

```
public static void parallelArray()
{
     Object[][] array;
     int row;
     int column;

     System.out.println("\nGenerate parallel arrays\n");

     array = new Object[5][3];
     for (row=0; row<array.length; row++)
     {
        array[row][0] = new Integer(1000+row);
        array[row][1] = new String("Name" +(row));
        array[row][2] = new MyClass("UserID" +(row));
     }
     printArray(array);
}
```

The following example is essentially the same as the previous example except that the data entries are stored down the columns instead of across the rows.

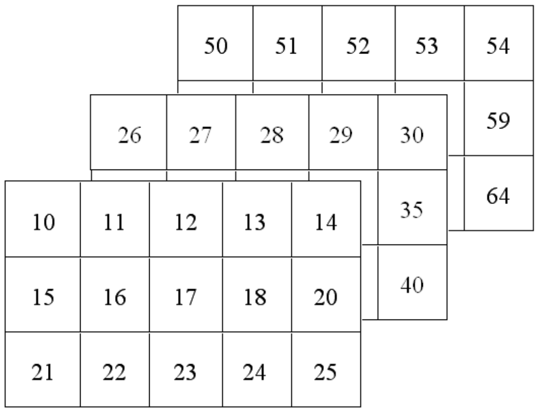
```
public static void parallelArray1()
{
     Object[][] array;
     Integer myInteger;
     int column;

     array = new Object[3][];
     array[0] = new Integer[5];
     array[1] = new String[5];
     array[2] = new MyClass[5];

     for (column=0; column<array[0].length; column++)
     {
        array[0][column] = new Integer(1000+column);
        array[1][column] = new String("Name" +(column));
        array[2][column] = new MyClass("UserID" +(column));
     }
     printArray(array);
}
```

19.8 Higher Dimensional Arrays

While most multi-dimensional arrays involve only two dimensions, arrays may have more than two dimensions. For example, the statement:

```
int[][][] ints = new int[3] [5] [3];
```

creates a 3-dimensional array. Such an array is best viewed as a 3 × 5 two-dimensional array that is repeated 3 times.



The first subscript refers to the row number, the second subscript refers to the column number, and the third subscript refers to the depth. So `ints[1][4][2]` refers to the element 59.

19.9 Merge Sort

The merge sort is a recursive sorting algorithm that splits the list to be sorted into two equal sublists, sorts each sublist, and then merges the two sublists to create the final sorted list. The recursive process continues splitting each list into two halves until only one element remains. At this point, since there is only one element in a list, that list is sorted and the recursion begins merging the sublists together until only one list remains. Note that the following algorithm returns the result of sorting the original list as a new list; the original list is not modified.

```
public static int[] mergeSort(int[] list)
{    // The original version of this algorithm was written by Dr. John Anderson
     int[] leftList, rightList;
     int[] result;
     int length;
     int count;

     if (list.length <= 1)
     {
        result = list;
     }
     else
     {  // split list into two equal-length sublists
        length = list.length;
        leftList=new int[length/2];
        for (count=0; count<length/2; count++)
        {
           leftList[count] = list[count];
        }

        rightList=new int[length-leftList.length];
        for (count=length/2; count<length; count++)
        {
           rightList[count-(length/2)] = list[count];
        }
        // recursively sort and then merge
        result = merge(mergeSort(leftList), mergeSort(rightList));
     }
     return result;
}

public static int[] merge(int[] leftList, int[] rightList)
{    // Merge two sorted lists together (using iteration)
     int[] result;
     int leftPosition;
     int rightPosition;
     int resultPosition;

     result = new int[leftList.length+rightList.length];
     leftPosition = 0;
     rightPosition = 0;
     resultPosition = 0;
     while (leftPosition<leftList.length || rightPosition<rightList.length)
     {
        if (leftPosition<leftList.length && (rightPosition>=rightList.length ||
                                 leftList[leftPosition]<=rightList[rightPosition]))
        {  //copy from left list if that value is smaller, or if there are
           //no items left in the right list
           result[resultPosition]=leftList[leftPosition];
           leftPosition++;
        }
        else
        {  //copy from the right
           result[resultPosition]=rightList[rightPosition];
           rightPosition++;
        }
        resultPosition++;
     }
     return result;
}
```

19.10 QuickSort

The quicksort algorithm is another recursive algorithm and is an improvement on the merge sort. Instead of simply splitting a list in half, the quicksort algorithm partitions the list into two portions such that all of the elements in the left portion of the list are smaller than the elements in the right portion of the list. By performing this partitioning, it is not necessary to merge the resulting sublists since the elements in the left sublist all come before the elements in the right sublist.

```
public static void quickSort(int[] list)
{
     quickSort(list, 0, list.length-1);
}

public static void quickSort(int[] list, int from, int to)
{    // partition the array into two sub-arrays and recursively sort each sub-array
     int pivot;
     if (from < to)
     {
        pivot = partition(list, from, to);
        quickSort(list, from, pivot-1);
```

```
         quickSort(list, pivot+1, to);
      }
}

public static int partition(int[] list, int from, int to)
{    // determine the pivot position
      int pivot;
      int count;
      int smaller;
      pivot = from;
      smaller = from;
      for (count=from+1; count<=to; count++)
      {
         if (list[count] < list[pivot])
         {
            smaller++;
            swap(list, smaller, count);
         }
      }
      // move the pivot element in between the two portions of the list
      swap(list, pivot, smaller);
      return smaller;
}

public static void swap(int[] list, int i, int j)
{    // swap two entries in the array
      int temp = list[i];
      list[i] = list[j];
      list[j] = temp;
}
```

Unlike the merge sort, quicksort manipulates the original list in place.

The quicksort algorithm first locates a “pivot” element, an element in the list to be sorted such that all elements to the left of the pivot are smaller than the pivot element and all elements to the right of the pivot are greater than the pivot element. This process is referred to as “partitioning” the list. The partitioning algorithm used in this example is one of the simplest; there are other partitioning algorithms that are faster but this version is one of the easier to understand.

19.11 Sorting a Linked List

In this section, we examine how a linked list can be sorted. Sorting a linked list requires more care than sorting an array because the elements are no longer physically adjacent. None of the algorithms uses the size of the linked list in its processing but doing so might make the processing a bit easier. Each sorting algorithm involves interchanging the contents of Nodes, not the Nodes themselves. The algorithms are shown in increasing order of complexity.

The method below performs a basic bubble sort of a linked list. The method stops when no interchanges were made during a specific pass of the linked list.

```
public static Node bubbleSort(Node firstNode)
```

```
{
     Node currentNode;
     Node previousNode;
     Object currentObject;
     Object previousObject;
     int interchanges;

     interchanges = 1;
     while (interchanges != 0)
     {
        interchanges = 0;
        previousNode = null;
        currentNode = firstNode;
        while (currentNode!=null)
        {
           if (previousNode != null)
           {
              previousObject = previousNode.getElement();
              currentObject = currentNode.getElement();
              if ((((String)previousObject).compareTo((String)currentObject))>0)
              {
                 previousNode.setElement(currentObject);
                 currentNode.setElement(previousObject);
                 interchanges++;
              }
           }
           previousNode = currentNode;
           currentNode = currentNode.getNextNode();
        }
     }
     return firstNode;
}
```

The method below performs an improved bubble sort of a linked list. The inner loop stops when it reaches the portion of the list that has already been sorted.

```
public static Node bubbleSort(Node firstNode)
{
     Node currentNode;
     Node previousNode;
     Node lastNode;
     Object currentObject;
     Object previousObject;
     int interchanges;
     interchanges = 1;
     lastNode = null;
     while (interchanges != 0)
     {
        interchanges = 0;
        previousNode = null;
        currentNode = firstNode;
        while ((currentNode!=null) && (currentNode!=lastNode))
        {
           if (previousNode != null)
           {
              previousObject = previousNode.getElement();
              currentObject = currentNode.getElement();
              if ((((String)previousObject).compareTo((String)currentObject))>0)
              {
                 previousNode.setElement(currentObject);
                 currentNode.setElement(previousObject);
```

```
                  interchanges++;
               }
            }
            previousNode = currentNode;
            currentNode = currentNode.getNextNode();
            lastNode = previousNode;
         }
      }
      return firstNode;
}
```

The method below performs a selection sort of a linked list.

```
public static Node selectionSort(Node firstNode)
{
      Node currentNode;
      Node previousNode;
      Node lastNode;
      Node maxNode;
      Object currentObject;
      Object maxObject;
      lastNode = firstNode;
      while (lastNode != null)
      {
         currentNode = firstNode;
         currentObject = null;
         maxObject = null;
         maxNode = null;
         previousNode = null;
         if (currentNode!=null)
         {  // Select the first node as the largest so far
            maxNode = currentNode;
            maxObject = maxNode.getElement();
            currentNode = currentNode.getNextNode();
         }

         // lastNode points to the beginning of the elements that are sorted
         // (except for the first time through the outer loop)
         while ((currentNode!=null) && (currentNode!=lastNode))
         {
            currentObject = currentNode.getElement();
            if ((((String)maxObject).compareTo((String)currentObject))<0)
            {  // Reset the largest node’s value
               maxNode = currentNode;
               maxObject = currentObject;
            }
            previousNode = currentNode;
            currentNode = currentNode.getNextNode();
         }

         // At this point, previousNode is the last node processed in the list
         if ((maxNode != previousNode) && (previousNode != null))
         {  // Interchange the largest node's contents with the last node’s contents
            maxNode.setElement(currentObject);
            previousNode.setElement(maxObject);
         }
         lastNode = previousNode;
      }
      return firstNode;
}
```

The method below performs a merge sort of a linked list.

```
public static Node mergeSort(Node list1)
{
     // Destroys list1
     Node result;
     Node list2;
     if(list1 == null || list1.getNextNode() == null)
     {
        result = list1;
     }
     else
     {
        list2 = split(list1);
        list1 = mergeSort(list1);
        list2 = mergeSort(list2);
        result = merge(list1, list2);
     }
     return result;
}

public static Node split(Node list1)
{
     Node result;
     Node list2;
     Node temp;
     int count;
     int middle;

     if(list1 == null || list1.getNextNode() == null)
     {
        result = null;
     }
     else
     {
        count=1;
        temp = list1;
        while (temp.getNextNode() != null)
        {
           count++;
           temp = temp.getNextNode();
        }
        middle = count/2;
        //System.out.println("Count: " +count +" middle: " +middle);

        list2 = list1;
        temp = null;
        for (count=0; count<middle; count++)
        {
           count++;
           temp = list2;
           list2 = list2.getNextNode();
        }
        temp.setNextNode(null);
        result = list2;
     }
     return result;
}
```

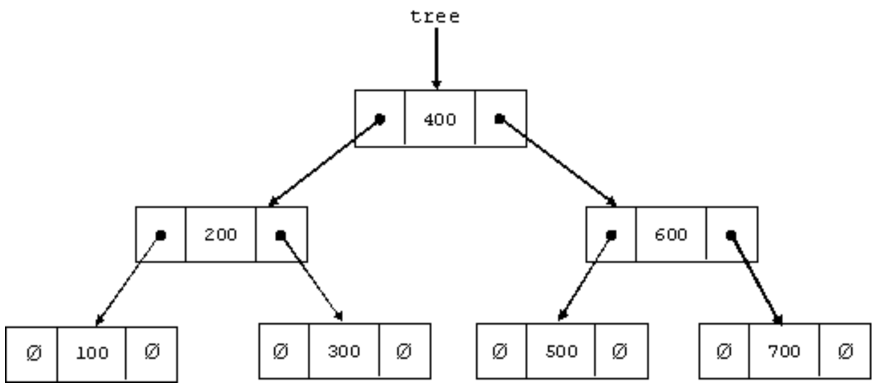
The split method is used to split the linked list into two more or less equal sublists. Since we can not determine the middle of a linked list using physical adjacency, it is necessary first to traverse the list to determine the number of elements in the list (this can be avoided if the size of the list is maintained); then, a second traversal determines the mid-point in the list.

The split method below determines the middle of the list as the list is being traversed to determine the size of the list. The method is an improvement over the first split method but it is not a significant improvement so either method could be used.

```
public static Node split(Node list1)
{
     Node result;
     Node middle;
     Node temp;
     int count;

     if(list1 == null || list1.getNextNode() == null)
     {
        result = null;
     }
     else
     {
        count=1;
        temp = list1;
        middle = list1;
        while (temp.getNextNode() != null)
        {
           if ((count>1) && (count%2)==1)
           {
              middle = middle.getNextNode();
           }
           count++;
           temp = temp.getNextNode();
        }
        result = middle.getNextNode();
        middle.setNextNode(null);
     }
     return result;
}
```

19.12 Binary Trees

A binary tree is similar to a linked list except that the pointer to the first element in the tree points to the middle of the tree (and all of the elements in the tree are sorted). Each node in a tree contains a value (object) plus a left pointer and a right pointer. All values to the left of the current Node are less than the current value and all values to the right of the current value are greater than the current value. The following diagram illustrates a simple binary tree.



The following method hard codes the creation of a binary tree that consists of the 7 values shown above.

```
public static Node newTree()
{
     Node tree, tree1, tree2, tree3,tree4;

     tree1 = new Node(null, "100", null);
     tree2 = new Node(null, "300", null);
     tree3 = new Node(tree1, "200", tree2);  // Create the left-hand tree

     tree1 = new Node(null, "500", null);
     tree2 = new Node(null, "700", null);
     tree4 = new Node(tree1, "600", tree2);  // Create the right-hand tree

     tree = new Node(tree3, "400", tree4); // Join the left tree and the right tree

     return tree;
}
```

To keep the processing as simple as possible, the variables in the Node class are `public`. The method that prints the tree is a simple recursive method.

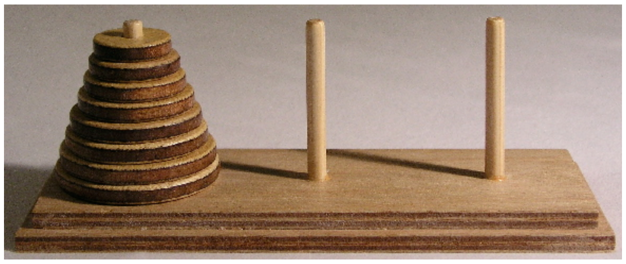
```
public static void printTree(Node node)
{
     if (node != null)
     {
        printTree(node.getLeftNode());
        System.out.print(node +" ");
        printTree(node.getRightNode());
     }
}

100 200 300 400 500 600 700
```

The Node class has been modified to return the Node to the left and the Node to the right of each Node. (You can think of these pointers as previous and next pointers.)

A method that creates a tree is more difficult than the printTree method but if the data are already sorted and presented in an array, then a recursive createTree method is just a collection of 4 special cases for inserting into a tree.

19.13 Towers of Hanoi

The Towers of Hanoi is a children's puzzle that involves moving one disk at a time from one pole to another pole that is either empty or has a larger disk on it. The goal is to end with all disks on the right-hand pole (in the same order as at the beginning). The following program illustrates how the Towers of Hanoi puzzle can be solved using recursion.



http://en.wikipedia.org/wiki/File:Tower_of_Hanoi.jpeg

```
class Hanoi
{
      private StringBuffer fromPole, middlePole, toPole;
      private int moves = 0;
      private int towerHeight = 4;
      private String disks ="KJIHGFEDCBA";

      public Hanoi(int towerHeight)
      {
          this.towerHeight = towerHeight;
      }

      public void run()
      {
          fromPole = new StringBuffer("1=");
          middlePole = new StringBuffer("2=");
          toPole = new StringBuffer("3=");
          fromPole.append(disks.substring(disks.length()-towerHeight));

          printState();
          solveTowers(towerHeight, fromPole, middlePole, toPole);

          System.out.println("The processing required " +moves +" moves.");
      }

      public void solveTowers(int height, StringBuffer fromPole,
                                          StringBuffer middlePole, StringBuffer toPole)
      {
          if (height >= 1)
          {
              // Move all disks except the bottom one from fromPole to middlePole
              solveTowers(height-1, fromPole, toPole, middlePole);
              // Move top disk from fromPole to toPole
              moveDisk(fromPole, toPole);
              // Move all disks from middlePole to toPole
              solveTowers(height-1, middlePole, fromPole, toPole);
          }
      }

      public void moveDisk(StringBuffer fromPole, StringBuffer toPole)
      {
          // Move top disk from fromPole to toPole
          toPole.append(fromPole.substring(fromPole.length()-1));
          fromPole.delete(fromPole.length()-1, fromPole.length());
          moves++;
          printState();
      }

      public void printState()
      {
          String pole1, pole2, pole3;
          final String blanks = "                   ";
          int count;
          int pole;

          pole1 = fromPole +blanks;
          pole2 = middlePole +blanks;
          pole3 = toPole +blanks;

          for (count=towerHeight-1+2; count>=0; count--)
          {
              System.out.print(pole1.substring(count,count+1) +" ");
              System.out.print(pole2.substring(count,count+1) +" ");
              System.out.println(pole3.substring(count,count+1));
          }
          System.out.println();
      }
}
```

The interesting thing to note about the program is that it is easy to solve the problem once an appropriate representation has been selected.

```
A
B
C
= = =
1 2 3


B
C   A
= = =
1 2 3



C B A
= = =
1 2 3


  A
C B
= = =
1 2 3


  A
  B C
= = =
1 2 3



A B C
= = =
1 2 3


    B
A   C
```

```
= = =
1 2 3

    A
    B
    C
= = =
1 2 3

The processing required 7 moves.
```

19.14 System Properties

There are many types of system information that can be obtained in a Java program. For example, the following statements display the version of Java that is currently being used:

```
String version = System.getProperty("java.version");
System.out.println("The current version of Java is: " +version );
```

```
The current version of Java is: 1.5.0_05
```

Java also provides a mechanism for displaying many of the properties that may be of interest to the programmer.

```
System.getProperties().list(System.out);
```

The following is an **abbreviated** list of properties generated by this command:

```
-- listing properties --
java.runtime.name=Java(TM) 2 Runtime Environment, Stand...
sun.boot.library.path=C:\Program Files\Java\jdk1.6.0_23\jre...
java.vm.version=1.5.0_05-b05
user.dir=C:\Users\Fred\Java\Properties
java.runtime.version=1.5.0_05-b05
java.endorsed.dirs=C:\Program Files\Java\jdk1.6.0_23\jre...
os.arch=x86

java.vm.specification.vendor=Sun Microsystems Inc.
user.variant=
os.name=Windows XP
os.version=5.1
user.home=C:\Documents and Settings\Fred
java.specification.version=1.5
user.name=Fred
java.home=C:\Program Files\Java\jdk1.6.0_23\jre
user.language=en
java.version=1.5.0_05
sun.desktop=windows
```

19.15 Information About Objects

Java provides the programmer with a wealth of information about the objects that have been created. Most of this information is useful only in complex programs but some of the information is useful in relatively simple programs. For example, the getClass() method returns an object which represents the class to which an object belongs. The getName() method can then be used on the returned object to obtain the name of that class. For example, ife created the object flight1 as an instance of the class Flight; we can ask Java for the name of the class that flight1 belongs to using the following instruction:

```
System.out.println(flight1.getClass().getName());
```

Java generates the following output:

```
Flight
```

This of course we already knew but there are times when we don't know the associated class for a particular object. For example, the object `System.out` belongs to a class and the name of the associated class can be obtained using the statement

```
System.out.println(System.out.getClass().getName());
```

Java then generates the following information:

```
java.io.PrintStream
```

19.16 The System Timer

When executing a method that takes a significant amount of time to complete, it would be useful to know exactly how long the processing in the method takes. Java provides a variety of timer methods but the easiest to use is the `System.currentTimeMillis()` method. The following program illustrates how this method is typically used. The current time is determined (and saved) before the processing method is called and then after the processing method returns, the current time is determined again. The difference between the two times is approximately the amount of time required to execute the method.

```
public static void main(String[] args)
{
     long startTime;
     long endTime;

     startTime = System.currentTimeMillis();
     myMethod();
     endTime = System.currentTimeMillis();
     System.out.println("myMethod took " +(endTime - startTime) +" milliseconds.");
}

public static void myMethod()
{
     String string;
     int count;

     string = "";
     for (count=0; count<50000; count++)
```

```
      {
          string = string + "  ";
      }
}
```

The output generated by this program is:

```
myMethod took 101563 milliseconds.
```

(The value 101563 milliseconds is 101.563 seconds.)

Timers are not always good indicators of a program's performance because there may be other processes active on the machine while the Java program is running. However, timers do provide an indication of the cost of executing a particular method and if the time required to execute method1 is significantly longer than the time required to execute a similar method, method2, then presumably method2 is more efficient than method1.

Java now includes a more precise timer.

19.17 The Stack Trace

When an error occurs, Java prints the error message and a list of the methods that were active at the time of the error (this is referred to as a stack trace). The stack trace shows each method that was called, beginning with the most recent and proceeding to the original "main" method. The line number within a method that contained the call to the next method is also included. A sample stack trace is shown below.

```
at java.io.FileInputStream.readBytes(Native Method)
at java.io.FileInputStream.read(FileInputStream.java:194)
at java.io.BufferedInputStream.fill(BufferedInputStream.java:218)
at java.io.BufferedInputStream.read(BufferedInputStream.java:235)
at IOExamples.consoleStream(IOExamples.java:26)
at IOExamples.main(IOExamples.java:7)
```

Prior to the release of Java 1.5, it was difficult to generate a programmer controlled stack trace. (It could be done but the code was quite messy.) Now, with Java 1.5, a simple method can be used to generate a stack trace at any time. This facility can be useful when debugging a complex program.

```
public static void dumpStack()
{     // requires Java 1.5
      Thread myThread;

      myThread = Thread.currentThread();
      myThread.dumpStack();
}
```

19.18 Garbage Collection

As was mentioned earlier in the notes, objects that are no longer referenced by any variables are orphan objects – they take up space but can never be used. Java reclaims the space allocated to these objects by performing **garbage collection** when the system is running low on memory.

The user can force the garbage collection method to run using the command:

```
System.gc();
```

However, as a general rule, it should not be necessary to force garbage collection; let Java determine when it is required. Garbage collection can also be turned off but you run the risk that Java may run out of memory and have to terminate the program.

19.19 The Heap

Java uses a data structure referred to as a heap to allocate new objects. (A **heap** is a more complex data structure than we have examined in these notes. Additional information on heaps can be found in a more advanced text on data structures.) As the heap becomes full, Java runs the garbage collection method that frees up any space occupied by objects that are no longer used (they are orphan objects). The following example prints the size of the heap and the amount of free space in the heap. It then allocates a large number of objects that (except for the last one) becomes orphans. Finally, the system garbage collection method is run to free up unused space.

```
public static void main(String[] args)
{
      int count;
      String string;
      checkHeap();
      for (count=0; count<1000; count++)
      {
          string = new String("ABCABCABCABCABCABCABCABCABCABC" +count);
      }
      checkHeap();
      System.gc();
      checkHeap();
}

public static void checkHeap()
{
      long heapSize;
      long heapFreeSize;
      heapSize = Runtime.getRuntime().totalMemory();
      heapFreeSize = Runtime.getRuntime().freeMemory();

      System.out.println("heap size: " +heapSize +" free memory: " +heapFreeSize);
}
```

The following output was generated by running this program. As can be seen, creating 1000 strings which are immediately orphaned (actually 999 are orphans, one is still pointed to by the variable string) reduces the amount of free memory. Running the garbage collection method frees the space allocated to these unused objects.

```
heap size: 2,031,616 free memory: 1,840,368
heap size: 2,031,616 free memory: 1,534,088
heap size: 2,031,616 free memory: 1,899,736
```

The purpose of this section is to illustrate how information about the heap can be obtained. However, it is a good reminder that making repeated modifications to a String variable is an expensive process. If some characters must be modified frequently, it would be better to store the characters in a StringBuffer object than in a String object.

19.20 The Object Hierarchy

Java provides methods that can be used to interrogate a particular object in order to determine the class to which the object belongs plus other useful information.

The following method displays the hierarchy of classes to which an object belongs.

```
public static void printHierarchy(Object obj)
{
      System.out.println();
      Class class1 = obj.getClass();
      String level = "Class: ";
      while (class1 != null)
      {
          System.out.println(level +class1.getName());
          Class class2 = class1.getSuperclass();
          class1 = class2;
          level = "Super " +level;
      }
}
```

For example, the statement

```
printHierarchy("abc");
```

causes the following output to be generated.

```
Class: java.lang.String
Super Class: java.lang.Object
```

and the statement

```
printHierarchy(System.out);
```

causes the following output to be generated.

```
Class: java.io.PrintStream
Super Class: java.io.FilterOutputStream
Super Super Class: java.io.OutputStream
Super Super Super Class: java.lang.Object
```

19.21 Changing the Hash Code

In several of the examples in these notes, we have added an equals method to an object so that objects can be compared for equality. Any time that an equals method is defined, the hash code method should also be defined in the class. For example, if the (String) studentID in the Student class is a unique identifier for each student, the hashCode could be defined as follows:

```
public int hashCode()
{
     int result;
     result = 31 * studentID.hashCode();
     return result;
}
```

19.22 File Directory Processing

Java provides many facilities that permit the programmer to access operating system information and to manipulate that information. For example, information about the files and directories/folders on a hard drive can be accessed quite easily in Java. The following method prints a list of all files in the **current** directory (the current directory is represented by "."). The method requires the programmer to include the command `import java.io.*;` at the beginning of the program.

```
public static void processDirectory()
{
      File dir;
      String[] children;
      int count;

      dir = new File(".");

      children = dir.list();

      if (children == null)
      {   // Either dir does not exist or is not a directory
          System.out.println("Not a directory.");
      }
      else
      {
          for (count=0; count<children.length; count++)
          {
              String filename = children[count]; // Get filename
              System.out.println(filename);
          }
      }
}
```

If you want to print a list of the files in another directory, the full path to the directory is specified. For example:

```
dir = new File("C:\\Program Files");
```

Note that if there are directories with the directory that is being printed, this method does **not** print the contents of those nested directories. In order to traverse nested directories, a more complex algorithm is required. We will examine two such algorithms in the next two sections.

19.23 Recursive File Directory Processing

The following method uses recursion to print the complete directory structure of a file system beginning at a particular directory. The method increases the indentation as each new directory is traversed. Note that the method mixes iteration and recursion: iteration is used to print the files in a specific directory and recursion is used to traverse the directory structure. The iterative portion of the method could also have been written in a recursive manner without any difficulty but there is no reason why the two techniques (recursion and iteration) should not be combined.

```
import java.io.*;
public static void main(String[] parms)
{
      processDirectories("C:\\Users\\Fred\\Java");
}

public static void processDirectories(String whichDirectory)
{
      System.out.println("\n" +whichDirectory);
      processDirectories("  ", whichDirectory);
}

public static void processDirectories(String indent, String whichDirectory)
{
      File dir;
      File fileName;
      String file;
      String[] children;
      int count;
      dir = new File(whichDirectory);
      children = dir.list();
      if (children == null)
      {   // Either dir does not exist or is not a directory
          System.out.println("Not a directory.");
      }
      else
      {
          for (count=0; count<children.length; count++)
          {
              file = children[count]; // Get filename of file or directory
              System.out.println(indent +file);
              fileName = new File(whichDirectory +"\\" +file);
              if (fileName.isDirectory())
              {
                  processDirectories(indent+"  ", whichDirectory +"\\" +file);
              }
          }
      }
}
```

Some sample output generated by this method is shown below. The files within each directory are indented two spaces.

```
C:\Users\Fred\Java
  Fib.java
  JavaTest
    src
      Binary.java
      Main.java
    . . .
```

19.24 Stack-Based File Directory Processing

In the previous section, a recursive method was used to process file directories. The corresponding iterative method is shown below. Note that this method must maintain its own stack of directory states. This is a complex program and is shown only to illustrate that while iterative processing is possible, recursive processing is significantly easier.

```
public static void processDirectories(String indent, String whichDirectory)
{
      File dir;
      File fileName;
      String file;
      String[] children;
      int count;
      Stack stack;
      DirectoryState dirState;

      stack = new Stack();
      dir = new File(whichDirectory);
      children = dir.list();
      if (children == null)
      {   // Either dir does not exist or is not a directory
          System.out.println("Not a directory.");
      }
      else
      {
          count=0;
          while ((count<children.length) || (stack.size()>0))
          {
              if (count<children.length)
              {
                  file = children[count];
                  System.out.println(indent +file);
                  fileName = new File(whichDirectory +"\\" +file);
                  if (fileName.isDirectory())
                  { //add the current directory to the stack
                    //  and process the new directory
                      dirState = new DirectoryState(children, count, whichDirectory);
                      stack.addStack(dirState);
                      whichDirectory += "\\" +file;
                      indent += "  ";
                      dir = new File(whichDirectory);
                      children = dir.list();
                      count = -1;
                  }
              }
              else if (stack.size()>0)
              {  //unstack a directory entry and resume processing
                  dirState = stack.removeStack();
                  children = dirState.getChildren();
                  count = dirState.getCount();
                  whichDirectory = dirState.getPath();
```

```
                indent = indent.substring(2);
            }
            count++;
        }
    }
}
```

```
public class DirectoryState
{
      private int count;
      private String[] children;
      private String path;

      public DirectoryState(String[] children, int count, String path)
      {
          this.children = children;
          this.count = count;
          this.path = path;
      }

      public String[] getChildren()
      {
          return children;
      }

      public String getPath()
      {
          return path;
      }

      public int getCount()
      {
          return count;
      }
}
```

The following Stack class uses an array to maintain the collection of directory states. While this technique works (as long as directories are not nested to a depth greater than 100), the use of an ArrayList or a linked list would be more appropriate. However, starting with an array and later refactoring the method to use a more powerful list structure satisfies the dictum "do the simplest thing that works".

```
public class Stack
{
      private DirectoryState[] stack;
      private int size;

      public Stack(int initialSize)
      {
          createStack(initialSize);
      }

      public Stack()
      {
          createStack(100);
      }

      private void createStack(int initialSize)
      {
          stack = new DirectoryState[initialSize];
          size = 0;
      }
```

```
     public void addStack(DirectoryState object)
     {
        stack[size] = object;
        size++;
     }

     public DirectoryState removeStack()
     {
        DirectoryState object;

        if (size > 0 )
        {
           size--;
           object = stack[size];
        }
        else
        {
           object = null;
        }
        return object;
     }

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

19.25 Creating Objects Dynamically

At times, it is necessary to create an object without knowing the specific type of object in advance. For example, in the following statements, a subclass of the class CreditCard is created. The parameter `cardType` identifies the type of credit card object to be created.

```
CreditCard newCreditCard;

newCreditCard = null;
if (cardType.equals("VisaGold"))
{
     newCreditCard = new VisaGold(accountNumber, currentBalance);
}
else if (cardType.equals("VisaRegular"))
{
     newCreditCard = new VisaRegular(accountNumber, currentBalance);
}
else if (cardType.equals("MasterCard"))
{
     newCreditCard = new MasterCard(accountNumber, currentBalance);
}
else if (cardType.equals("AmericanExpress"))
{
     newCreditCard = new AmericanExpress(accountNumber, currentBalance);
}
else
{
     System.out.println("Invalid card: " +cardType);
}
return newCreditCard;
```

While this type of processing involves some typing, it is not at all complex. However, if the number of types of objects that can be created is likely to change on a frequent basis, the programmer must remember to modify these statements every time that a new class is added or an existing class is deleted.

A more sophisticated technique takes advantage of Java's **Reflection** package. This package provides methods that can be used to perform actions such as creating an object at run-time without the need for a series of `if` statements. The following statements illustrate the use of reflection to create an object of type `cardType`.

```
import java.lang.reflect.*;

CreditCard newCreditCard;
Class newClass;
Constructor constr;

newCreditCard = null;
try
{
     newClass = Class.forName(cardType);
     constr = newClass.getConstructor(new Class[] {String.class, double.class});
     newCreditCard = (CreditCard) constr.newInstance(new Object[]
                                     {accountNumber, new Double(currentBalance)});
}
catch (Exception e)
{
     System.out.println("Invalid card: " +cardType +" " +accountNumber);
}
return newCreditCard;
```

Note that in this example all objects have the same parameters (a String and a double). If this is not always true, some additional work must be performed to create the correct parameters for each object but this is fairly easy with the reflection package.

19.26 Calling Methods Dynamically

As described in the previous section, objects can be created dynamically using Java's reflection package. Creating objects is not the only use of reflection, the following example illustrates how a method may be called dynamically using reflection.

```
import java.lang.reflect.*;

public class MethodTest
{
     public static void main(String[] args)
     {
         Method whichMethod;
         Class whichClass;

         myMethod("Direct call");

         try
         {
             whichClass = MethodTest.class;
             whichMethod=whichClass.getMethod("myMethod", new Class[]{String.class});
             whichMethod.invoke(whichClass, new Object[] {"Dynamic call"});
         }
         catch (Exception e)
         {
             System.out.println("Method not found.");
         }
     }

     public static void myMethod(String string)
     {
         System.out.println(string);
     }
}
```

The output generated by this program is shown below.

```
Direct call
Dynamic call
```

In this particular example, the method that is invoked is a static method but a method associated with an object may also be invoked as long as the object instance is available.

19.27 Polymorphism without Hierarchies

The following example also illustrates the use of reflection. It shows how reflection can be used to call a method dynamically without having to gather all of the classes that contain the method into a hierarchy. (There is an alternative Java technique that uses Java interfaces that is a better way of achieving the same result.)

```
import java.util.ArrayList;
import java.lang.reflect.*;

public class Polymorphism
{
    public static void main(String[] parms)
    {
        ObjectCollection collection;
        Object object;
        Robot4 robot4;
        int count;

        collection = new ObjectCollection();
        collection.printObjects();

        robot4 = (Robot4) collection.get(3);
        robot4.testMethod("direct execution");

        for (count=0; count<collection.size(); count++)
        {
            object = collection.get(count);
            collection.testMethod(object, "polymorphic execution");
        }
    }
}

public class ObjectCollection
{
    ArrayList objects;

    public ObjectCollection()
    {
        objects = new ArrayList();
        createObjects();
    }

    private void createObjects()
    {
        objects.add(new Robot1());
        objects.add(new Robot2());
        objects.add(new Robot3());
        objects.add(new Robot4());
        objects.add(new Robot5());
        objects.add(new Robot6());
        objects.add(null);
    }

    public static void testMethod(Object myObject, String myString)
    {
        Method myMethod;
        Class myClass;

        myMethod = null;
        if (myObject == null)
        {
            System.out.println("\nObject is null.");
        }
        else
        {
            try
            {
                myClass = myObject.getClass();
                myMethod = getMethod(myClass, "testMethod");
                myMethod.invoke(myObject, new Object[] {myString});
            }
            catch (Exception ex)
            {
                System.out.println("\nMethod could not be executed " +myMethod);
            }
        }
    }

    public static Method getMethod(Class whichClass, String whichMethod)
    {   // requires recursion to locate the method
        Method myMethod = null;

        try
        {   // look for the method in the current class
            myMethod =
            whichClass.getDeclaredMethod(whichMethod, new Class[] {String.class});
            myMethod.setAccessible(true);
        }
        catch (Exception ex)
        {   // need to look in the superclass if the method was not found
            myMethod = getMethod(whichClass.getSuperclass(), whichMethod);
        }
        return myMethod;
    }

    public void add(Object newObject)
    {
        objects.add(newObject);
    }

    public Object get(int position)
    {
        return (Object) objects.get(position);
    }
```

```
      public int size()
      {
         return objects.size();
      }

      public void printObjects()
      {
         int currentObject;

         System.out.println("\nList of Objects:");
         System.out.println("-----------------");
         currentObject = 0;
         for (currentObject=0; currentObject<objects.size(); currentObject++)
         {
            System.out.println(objects.get(currentObject));
         }
      }
}

public class Robot1
{
      String string;

      public void testMethod(String string)
      {
         this.string = string;
         System.out.println("\nRobot1 " +string);
      }

      public String toString()
      {
         return "My name is robot1";
      }
}

public class Robot2
{
      String string;

      public void testMethod(String string)
      {
         this.string = string;
         System.out.println("\nRobot2 " +string);
      }

      public String toString()
      {
         return "My name is robot2";
      }
}

public class Robot3
{
      String string;

      public void testMethod(String string)
      {
         this.string = string;
         System.out.println("\nRobot3 " +string);
      }

      public String toString()
      {
         return "My name is robot3";
      }
}

public class Robot4
{
      String string;
      protected void testMethod(String string)
      {
          this.string = string;
          System.out.println("\nRobot4 " +string);
      }
      public String toString()
      {
          return "My name is robot4";
      }
}

public class Robot5 extends Robot4
{
      public void testMethod(String string)
      {   // over-rides method from Robot4
          this.string = string;
          System.out.println("\nRobot5 " +string);
      }
      public String toString()
      {
          return "My name is robot5";
      }
}

public class Robot6 extends Robot5
{     // inherits testMethod from Robot5

      public String toString()
      {
          return "My name is robot6";
      }
}
```

19.28 Compiling and Executing a Java Project

Normally Java programs are compiled and run from within a simple editor such as TextPad (or a more powerful IDE such as Eclipse). However, occasionally it is necessary to compile and run a Java program from the command prompt (or in a batch file or script). The javac command is used to compile one or more .java files and create the corresponding .class files.

```
javac *.java
```

The javac command above assumes that the Java SDK is on the system path. If this is not true, you can either add the appropriate directory to the path variable or just include it with the javac command, as shown below. (The path to the Java SDK will likely be different on your computer.)

```
"C:\Program Files\Java\jdk1.6.0_23\bin\javac" *.java
```

To redirect the output of any command to a file instead of to the console, add `> filename.txt` to the end of the command:

```
javac *.java > filename.txt
```

To append the output of any command to the end of an existing file, add `>> filename.txt` to the end of the command:

```
javac *.java >> filename.txt
```

To execute a Java class file, use the command:

```
java ClassName
```

where `ClassName` is the name of the class file to be executed – ensure that the **case** of the name of the class file is an exact match to the parameter typed in the command window. Also, do not include the `.class` extension with the java command. If the Java SDK or JRE directory is not on the system path, the path to the bin directory must be included with the command.

If command-line parameters are used by the program, just add them after the class name. For example,

```
java ClassName parameter1 parameter2 …
```

Again, to redirect the output of the command to a file instead of to the console, add the following to the end of the command:

```
java ClassName > filename.txt
```

Java does not create native executables but it does permit class files to be grouped together into a Java Archive (JAR) file that can be executed in a manner that is similar to a conventional exe file. A Jar file contains a set of class files compressed into one file. To create a Jar file, use the following command to bundle all class files into the Jar file:

```
jar cfm Java1.jar manifest.txt *.class
```

or

```
"C:\Program Files\Java\jdk1.6.0_23\bin\jar" cfm Java1.jar manifest.txt *.class
```

The manifest file is a text file that identifies the class that contains the main method that is to be executed.

```
Main-Class: Java1

```

The manifest file **must** contain a blank line after the last line of text.

Once the Jar file has been created, it can be executed either by double-clicking on it (if this is supported by your operating system) or by executing one of the following commands:

```
java -jar Java1.jar

"C:\Program Files\Java\jdk1.6.0_23\bin\java" -jar Java1.jar
```

To execute either a class files or a Jar file, the Java run-time environment (JRE) must be installed.

19.29 Summary

The basic system methods introduced in this chapter can be very useful. The Java API mentioned can be examined for additional ways of accessing system information:

```
http://download.oracle.com/javase/6/docs/api/
```

The use of reflection in Java programs should be reserved for sophisticated systems; reflection is a very powerful feature but should not be used unless it is absolutely necessary.

INDEX