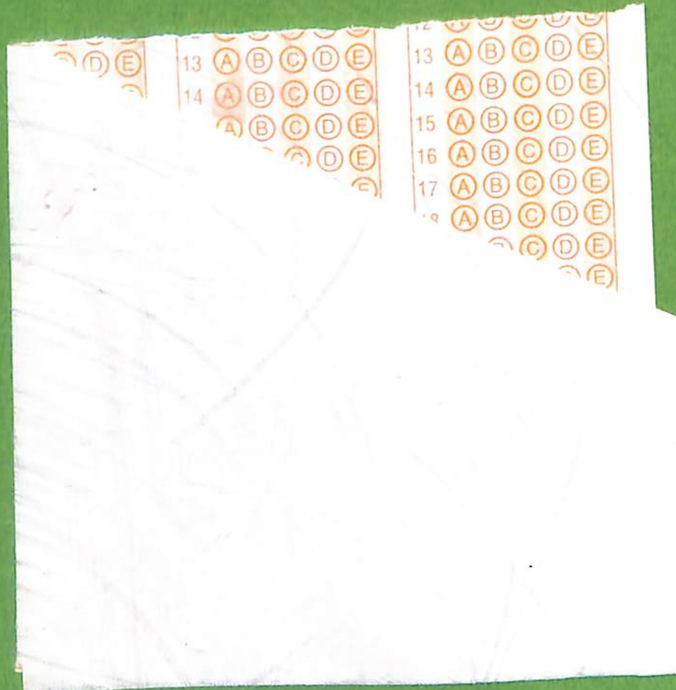
32.973
F13
K17

Süleyman Demirel University
Faculty of Engineering
Department of Natural Sciences, Mathematics and Informatics

Basic Java
Selected topics

(Part 1)

Almaty – 2000



Süleyman Demirel University
Faculty of Engineering
Department of Natural Sciences, Mathematics and Informatics

Basic Java
Selected topics

(Part 1)

Editor
Eldar K. Hajilarov

The Introductory Course was approved by the Faculty of Engineering and Administration of the Suleyman Demirel University. Recommended to the learners of the course "Algorithms and Programming".

Almaty – 2000

The university of Suleyman Demirel
Library
№ 30231

ББК 32.973-01я73

Г13

Eldar Hajilarov
Г13 Basic Java. Selected Topics, Almaty – 2000, 137 pages
ISBN 9965-9150-1-6

ББК 32.973-01я73

Г $\frac{16602120000}{00(15)-01}$
ISBN 9965-9150-1-6

© Eldar Hajilarov, 2000

CONTENTS

Preface 4
Chapter 1. Language Basics 5
1.1. Variables 5
1.2. Operators 15
1.3. Expressions, Statements, and Blocks 35
1.4. Control Flow Statements 41
Chapter 2. Object Basics and Simple Data Objects 61
2.1. The Life Cycle of an Object 61
2.2. Characters and Strings 70
2.3. Arrays 82
Chapter 3. Classes and Inheritance 92
3.1. Creating Classes 92
3.2. Controlling Access to Members of a Class 117
3.3. Understanding Instance and Class Members 125
3.4. Managing Inheritance 133
3.5. Implementing Nested Classes 134

Preface

This book gives a brief but concise introduction to Java programming language and Object-Oriented Programming. Basic notions and constructions of the language are discussed, and illustrated by simple programs. Can be used as additional workbook by students studying Java.

!

Chapter1.Language Basics

The BasicsDemo program that follows adds the numbers from 1 to 10 and displays the result.

```
public class BasicsDemo {
   public static void main(String[] args) {
      int sum = 0;
      for (int current = 1; current <= 10; current++) {
         sum += current;
      }
      System.out.println("Sum = " + sum);
   }
}
```

The output from this program is:

```
Sum = 55
```

Even a small program such as this uses many of the traditional features of the Java programming language, including variables, operators, and control flow statements. The code might look a little mysterious now. But this trail teaches what you need to know about the Java programming language to understand this program.

1.1.Variables

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

Definition: A variable is an item of data named by an identifier.

You must explicitly provide a name and a type for each variable you want to use in your program. The variable's name must be a legal identifier --an unlimited series of Unicode characters that begins with a letter. You use the variable name to refer to the data that the variable contains. The variable's type determines what values it can hold and what operations can be performed on it. To give a variable a type and a name, you write a variable declaration, which generally looks like this:

type name

In addition to the name and type that you explicitly give a variable, a variable has scope. The section of code where the variable's simple name can be used is the variable's scope. The variable's scope is determined implicitly by the location of the variable declaration, that is, where the declaration appears in relation to other code elements.
The MaxVariablesDemo program, shown below, declares eight variables of different types within its main method.

```
public class MaxVariablesDemo {
   public static void main(String args[]) {

      // integers
      byte largestByte = Byte.MAX_VALUE;
      short largestShort = Short.MAX_VALUE;
      int largestInteger = Integer.MAX_VALUE;
      long largestLong = Long.MAX_VALUE;

      // real numbers
      float largestFloat = Float.MAX_VALUE;
      double largestDouble = Double.MAX_VALUE;

      // other primitive types
```



```
      char aChar = 'S';
      boolean aBoolean = true;

      // display them all
      System.out.println("The largest byte value is " +
largestByte);
      System.out.println("The largest short value is " +
largestShort);
      System.out.println("The largest integer value is " +
largestInteger);
      System.out.println("The largest long value is " +
largestLong);

      System.out.println("The largest float value is " +
largestFloat);
      System.out.println("The largest double value is " +
largestDouble);

      if (Character.isUpperCase(aChar)) {
         System.out.println("The character " + aChar + "
is upper case.");
      } else {
         System.out.println("The character " + aChar + "
is lower case.");
      }
      System.out.println("The value of aBoolean is " +
aBoolean);
   }
}
```

The output from this program is:

```
The largest byte value is 127
The largest short value is 32767
The largest integer value is 2147483647
```

The largest long value is 9223372036854775807
The largest float value is 3.40282e+38
The largest double value is 1.79769e+308
The character S is upper case.
The value of aBoolean is true

The following sections elaborate on the various aspects of variables, including data types, names, scope, initialization, and final variables. The MaxVariablesDemo program uses two items with which you might not yet be familiar and are not covered in this section: several constants named MAX_VALUE and an if-else statement. Each MAX_VALUE constant is defined in one of the number classes provided by the Java platform and is the largest value that can be assigned to a variable of that numeric type.

Data Types

Every variable must have a data type. A variable's data type determines the values that the variable can contain and the operations that can be performed on it. For example, in the MaxVariablesDemo program, the declaration int largestInteger declares that largestInteger has an integer data type (int). Integers can contain only integral values (both positive and negative). You can perform arithmetic operations, such as addition, on integer variables.
The Java programming language has two categories of data types: primitive and reference. A variable of primitive type contains a single value of the appropriate size and format for its type: a number, a character, or a boolean value. For example, an integer value is 32 bits of data in a format known as two's complement, the value of a char is 16 bits of data formatted as a Unicode character, and so on.

variableName [value]

The following table lists, by keyword, all of the primitive data types supported by Java, their sizes and formats, and a brief

description of each. The MaxVariablesDemo program declares one variable of each primitive type.

Primitive Data Types

Keyword	Description	Size/Format
	(integers)	
byte	Byte-length integer	8-bit two's complement
short	Short integer	16-bit two's complement
int	Integer	32-bit two's complement
long	Long integer	64-bit two's complement
	(real numbers)	
float	Single-precision floating point	32-bit IEEE 754
double	Double-precision floating point	64-bit IEEE 754
	(other types)	
char	A single character	16-bit Unicode character
boolean	A boolean value (true or false)	true or false

Purity Tip: In other languages, the format and size of primitive data types may depend on the platform on which a program is running. In contrast, the Java programming language specifies the size and format of its primitive data types. Hence, you don't have to worry about system-dependencies.

You can put a literal primitive value directly in your code. For example, if you need to assign the value 4 to an integer variable you can write this:

int anInt = 4;

The digit 4 is a literal integer value. Here are some examples of literal values of various primitive types:

Examples of Literal Values and Their Data Types

Literal	Data Type
178	int
8864L	long
37.266	double
37.266D	double
87.363F	float
26.77e3	double
' c '	char
true	boolean
false	boolean

Generally speaking, a series of digits with no decimal point is typed as an integer. You can specify a long integer by putting an 'L' or 'l' after the number. 'L' is preferred as it cannot be confused with the digit '1'. A series of digits with a decimal point is of type double. You can specify a float by putting an 'f' or 'F' after the number. A literal character value is any single Unicode character between single quote marks. The two boolean literals are simply true and false.

Arrays, classes, and interfaces are reference types. The value of a reference type variable, in contrast to that of a primitive type, is a reference to (an address of) the value or set of values represented by the variable.

A reference is called a pointer, or a memory address in other languages. The Java programming language does not support the explicit use of addresses like other languages do. You use the variable's name instead.

Variable Names

A program refers to a variable's value by the variable's name. For example, when it displays the value of the largestByte variable, the MaxVariablesDemo program uses the name largestByte. A name, such as largestByte, that's composed of a single identifier, is called a simple name. Simple names are in contrast to qualified names, which a class uses to refer to a member variable that's in another object or class.

In the Java programming language, the following must hold true for a simple name:

1.It must be a legal identifier. An identifier is an unlimited series of Unicode characters that begins with a letter.

2.It must not be a keyword, a boolean literal (true or false), or the reserved word null.

3.It must be unique within its scope. A variable may have the same name as a variable whose declaration appears in a different scope. In some situations, a variable may share the same name as another variable if it is declared within a nested block of code. (We will cover this in the next section, Scope.)

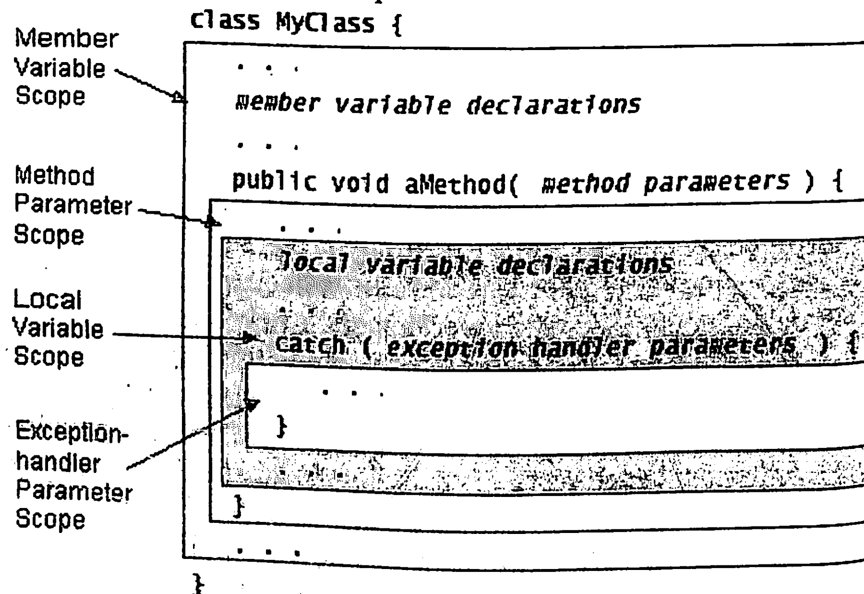
By Convention : Variable names begin with a lowercase letter, and class names begin with an uppercase letter. If a variable name consists of more than one word, the words are joined together, and each word after the first begins with an uppercase letter, like this: isVisible. The underscore character (_) is acceptable anywhere in a name, but by convention is used only to separate words in constants (because constants are all caps by convention and thus cannot be case-delimited).

Scope

A variable's scope is the region of a program within which the variable can be referred to by its simple name. Secondarily, scope also determines when the system creates and destroys memory for the variable. Scope is distinct from visibility, which applies only to member variables and determines whether the variable can be used from outside of the class

within which it is declared. Visibility is set with an access modifier.
The location of the variable declaration within your program establishes its scope and places it into one of these four categories:

- member variable
- local variable
- method parameter
- exception-handler parameter



A member variable is a member of a class or an object. It is declared within a class but outside of any method or constructor. A member variable's scope is the entire declaration of the class. However, the declaration of a member needs to appear before it is used when the use is in a member initialization expression. For information about declaring member variables, refer to Declaring Member Variables in the next lesson.

You declare local variables within a block of code. In general, the scope of a local variable extends from its declaration to the end of the code block in which it was declared. In MaxVariablesDemo, all of the variables declared within the main method are local variables. The scope of each variable in that program extends from the declaration of the variable to the end of the main method --indicated by the first right curly bracket } in the program code.
Parameters are formal arguments to methods or constructors and are used to pass values into methods and constructors. The scope of a parameter is the entire method or constructor for which it is a parameter.
Exception-handler parameters are similar to parameters but are arguments to an exception handler rather than to a method or a constructor. The scope of an exception-handler parameter is the code block between { and } that follow a catch statement. Handling Errors with Exceptions talks about using exceptions to handle errors and shows you how to write an exception handler that has a parameter.

Consider the following code sample:

```
if (...) {
   int i = 17;
   ...
}
System.out.println("The value of i = " + i);    // error
```

The final line won't compile because the local variable i is out of scope. The scope of i is the block of code between the { and }. The i variable does not exist anymore after the closing }. Either the variable declaration needs to be moved outside of the if statement block, or the println method call needs to be moved into the if statement block.

Variable Initialization

Local variables and member variables can be initialized with an assignment statement when they're declared. The data type of the variable must match the data type of the value assigned to it. The MaxVariablesDemo program provides initial values for all its local variables when they are declared. The local variable declarations from that program are as follows.

```
// integers
byte largestByte = Byte.MAX_VALUE;
short largestShort = Short.MAX_VALUE;
int largestInteger = Integer.MAX_VALUE;
long largestLong = Long.MAX_VALUE;

// real numbers
float largestFloat = Float.MAX_VALUE;
double largestDouble = Double.MAX_VALUE;

// other primitive types
char aChar = 'S';
boolean aBoolean = true;
```

Parameters and exception-handler parameters cannot be initialized in this way. The value for a parameter is set by the caller.

Final Variables

You can declare a variable in any scope to be final. The value of a final variable cannot change after it has been initialized. Such variables are similar to constants in other programming languages.

To declare a final variable, use the final keyword in the variable declaration before the type:

```
final int aFinalVar = 0;
```

The previous statement declares a final variable and initializes it, all at once. Subsequent attempts to assign a value to aFinalVar result in a compiler error. You may, if necessary, defer initialization of a final local variable. Simply declare the local variable and initialize it later, like this:

```
final int blankfinal;
. . .
blankfinal = 0;
```

A final local variable that has been declared but not yet initialized is called a blank final. Again, once a final local variable has been initialized, it cannot be set, and any later attempts to assign a value to blankfinal result in a compile-time error.

Summary of Variables

When you declare a variable, you explicitly set the variable's name and data type. The Java programming language has two categories of data types: primitive and reference. A variable of primitive type contains a value.

The location of a variable declaration implicitly sets the variable's scope, which determines what section of code may refer to the variable by its simple name. There are four categories of scope: member variable scope, local variable scope, parameter scope, and exception-handler parameter scope.

You can provide an initial value for a variable within its declaration by using the assignment operator (=).

You can declare a variable as final. The value of a final variable cannot change after it's been initialized.

1.2.Operators

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

An operator performs a function on one, two, or three operands. An operator that requires one operand is called a unary operator. For example, ++ is a unary operator that increments the value of its operand by 1. An operator that requires two operands is a binary operator . For example, = is a binary operator that assigns the value from its right-hand operand to its left-hand operand. And finally, a ternary operator is one that requires three operands. The Java programming language has one ternary operator, ?:, which is a short-hand if-else statement.
The unary operators support either prefix or postfix notation. Prefix notation means that the operator appears before its operand:

```
operator op          //prefix notation
```

Postfix notation means that the operator appears after its operand:

```
op operator          //postfix notation
```

All of the binary operators use infix notation, which means that the operator appears between its operands:

```
op1 operator op2     //infix notation
```

The ternary operator is also infix; each component of the operator appears between operands:

```
op1 ? op2 : op3      //infix notation
```

In addition to performing the operation, an operator returns a value. The return value and its type depend on the operator and the type of its operands. For example, the arithmetic operators, which perform basic arithmetic operations such as addition and

subtraction, return numbers-the result of the arithmetic operation. The data type returned by an arithmetic operator depends on the type of its operands: If you add two integers, you get an integer back. An operation is said to evaluate to its result.
We divide the operators into the categories described below.

Arithmetic Operators

The Java programming language supports various arithmetic operators for all floating-point and integer numbers.
These operators are + (addition), - (subtraction), * (multiplication), / (division), and % (modulo).
The following table summarizes the binary arithmetic operations in the Java programming language.

Operator	Use	Description
+	op1 + op2	Adds op1 and op2
-	op1 - op2	Subtracts op2 from op1
*	op1 * op2	Multiplies op1 by op2
/	op1 / op2	Divides op1 by op2
%	op1 % op2	Computes the remainder of dividing op1 by op2

Here's an example program, ArithmeticDemo, that defines two integers and two double-precision floating-point numbers and uses the five arithmetic operators to perform different arithmetic operations. This program also uses + to concatenate strings. The arithmetic operations are shown in red:

```
public class ArithmeticDemo {
   public static void main(String[] args) {

      //a few numbers
      int i = 37;
      int j = 42;
```

The university of
Suleyman Demirel
Library
№ 30231

```
double x = 27.475;
double y = 7.22;
System.out.println("Variable values...");
System.out.println("    i = " + i);
System.out.println("    j = " + j);
System.out.println("    x = " + x);
System.out.println("    y = " + y);

//adding numbers
System.out.println("Adding...");
System.out.println("    i + j = " + (i + j));
System.out.println("    x + y = " + (x + y));

//subtracting numbers
System.out.println("Subtracting...");
System.out.println("    i - j = " + (i - j));
System.out.println("    x - y = " + (x - y));

//multiplying numbers
System.out.println("Multiplying...");
System.out.println("    i * j = " + (i * j));
System.out.println("    x * y = " + (x * y));

//dividing numbers
System.out.println("Dividing...");
System.out.println("    i / j = " + (i / j));
System.out.println("    x / y = " + (x / y));

//computing the remainder resulting from dividing
numbers
System.out.println("Computing the remainder...");
System.out.println("    i % j = " + (i % j));
System.out.println("    x % y = " + (x % y));

//mixing types
System.out.println("Mixing types...");
System.out.println("    j + y = " + (j + y));
System.out.println("    i * x = " + (i * x));
    }
}
```

The output from this program is:

```
Variable values...
    i = 37
    j = 42
    x = 27.475
    y = 7.22
Adding...
    i + j = 79
    x + y = 34.695
Subtracting...
    i - j = -5
    x - y = 20.255
Multiplying...
    i * j = 1554
    x * y = 198.37
Dividing...
    i / j = 0
    x / y = 3.8054
Computing the remainder...
    i % j = 37
    x % y = 5.815
Mixing types...
    j + y = 49.22
    i * x = 1016.58
```

Note that when an integer and a floating-point number are used as operands to a single arithmetic operation, the result is floating point. The integer is implicitly converted to a floating-point number before the operation takes place.

The following table summarizes the data type returned by the arithmetic operators, based on the data type of the operands. The necessary conversions take place before the operation is performed.

Data Type of Result	Data Type of Operands
long	Neither operand is a float or a double (integer arithmetic); at least one operand is a long.
int	Neither operand is a float or a double (integer arithmetic); neither operand is a long.
double	At least one operand is a double.
float	At least one operand is a float; neither operand is a double.

In addition to the binary forms of + and -, each of these operators has unary versions that perform the following operations:

Operator	Use	Description
+	+op	Promotes op to int if it's a byte, short, or char
-	-op	Arithmetically negates op

Two shortcut arithmetic operators are ++, which increments its operand by 1, and --, which decrements its operand by 1. Either ++ or -- can appear before (prefix) or after (postfix) its operand. The prefix version, ++op/--op, evaluates to the value of the operand after the increment/decrement operation. The postfix version, op++/op--, evaluates the value of the operand before the increment/decrement operation.
The following program, called SortDemo , uses ++ twice and -- once.

```
public class SortDemo {
   public static void main(String[] args) {
      int[] arrayOfInts = { 32, 87, 3, 589, 12, 1076,
                           2000, 8, 622, 127 };

      for (int i = arrayOfInts.length; --i >= 0; ) {
         for (int j = 0; j < i; j++) {
            if (arrayOfInts[j] > arrayOfInts[j+1]) {
               int temp = arrayOfInts[j];
               arrayOfInts[j] = arrayOfInts[j+1];
               arrayOfInts[j+1] = temp;
            }
         }
      }

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

This program puts 10 integer values into an array-a fixed length structure that can hold multiple values of the same type-then sorts them. The line of code in red declares an array referred to by arrayOfInts, creates the array, and puts 10 integer values into it. The program uses arrayOfInts.length to get the number of elements in the array.
Individual elements are accessed with this notation: arrayOfInts[index], where index is an integer indicating the position of the element within the array. Note that indices begin at 0.
The output from this program is a list of numbers sorted from highest to lowest:

3 8 12 32 87 127 589 622 1076 2000

Let's look at how the SortDemo program uses -- to control the outer of its two nested sorting loops. Here's the statement that controls the outer loop:

```
for (int i = arrayOfInts.length; --i >= 0; ) {
    ...
}
```

The for statement is a looping construct, which you'll meet later in this trail. What's important here is the code in red, which continues the for loop as long as the value returned by --i is greater than or equal to 0. Using the prefix version of -- means that the last iteration of this loop occurs when i is equal to 0. If we change the code to use the postfix version of --, the last iteration of this loop occurs when i is equal to -1, which is incorrect for this program because i is used as an array index and -1 is not a valid array index.
The other two loops in the program use the postfix version of ++. In both cases, the version used doesn't really matter, because the value returned by the operator isn't used for anything. When the return value of one of these shortcut operations isn't used for anything, convention prefers the postfix version.
The shortcut increment/decrement operators are summarized in the following table.

Operator	Use	Description
++	op++	Increments op by 1; evaluates to the value of op before it was incremented
++	++op	Increments op by 1; evaluates to the value of op after it was incremented
--	op--	Decrements op by 1; evaluates to the value of op before it was decremented
--	--op	Decrements op by 1; evaluates to the value of op after it was decremented

Relational and Conditional Operators
A relational operator compares two values and determines the relationship between them. For example, != returns true if the two operands are unequal. This table summarizes the relational operators:

Operator	Use	Returns true if
>	op1 > op2	op1 is greater than op2
>=	op1 >= op2	op1 is greater than or equal to op2
<	op1 < op2	op1 is less than op2
<=	op1 <= op2	op1 is less than or equal to op2
==	op1 == op2	op1 and op2 are equal
!=	op1 != op2	op1 and op2 are not equal

Following is an example, RelationalDemo, that defines three integer numbers and uses the relational operators to compare them.

```
public class RelationalDemo {
    public static void main(String[] args) {

        //a few numbers
        int i = 37;
        int j = 42;
        int k = 42;
        System.out.println("Variable values...");
        System.out.println("    i = " + i);
```

```
3 8 12 32 87 127 589 622 1076 2000
```

Let's look at how the SortDemo program uses -- to control the outer of its two nested sorting loops. Here's the statement that controls the outer loop:

```
for (int i = arrayOfInts.length; --i >= 0; ) {
    ...
}
```

The for statement is a looping construct, which you'll meet later in this trail. What's important here is the code in red, which continues the for loop as long as the value returned by --i is greater than or equal to 0. Using the prefix version of -- means that the last iteration of this loop occurs when i is equal to 0. If we change the code to use the postfix version of --, the last iteration of this loop occurs when i is equal to -1, which is incorrect for this program because i is used as an array index and -1 is not a valid array index.

The other two loops in the program use the postfix version of ++. In both cases, the version used doesn't really matter, because the value returned by the operator isn't used for anything. When the return value of one of these shortcut operations isn't used for anything, convention prefers the postfix version.

The shortcut increment/decrement operators are summarized in the following table.

Operator	Use	Description
++	op++	Increments op by 1; evaluates to the value of op before it was incremented
++	++op	Increments op by 1; evaluates to the value of op after it was incremented
--	op--	Decrements op by 1; evaluates to the value of op before it was decremented
--	--op	Decrements op by 1; evaluates to the value of op after it was decremented

Relational and Conditional Operators

A relational operator compares two values and determines the relationship between them. For example, != returns true if the two operands are unequal. This table summarizes the relational operators:

Operator	Use	Returns true if
>	op1 > op2	op1 is greater than op2
>=	op1 >= op2	op1 is greater than or equal to op2
<	op1 < op2	op1 is less than op2
<=	op1 <= op2	op1 is less than or equal to op2
==	op1 == op2	op1 and op2 are equal
!=	op1 != op2	op1 and op2 are not equal

Following is an example, RelationalDemo, that defines three integer numbers and uses the relational operators to compare them.

```
public class RelationalDemo {
    public static void main(String[] args) {

        //a few numbers
        int i = 37;
        int j = 42;
        int k = 42;
        System.out.println("Variable values...");
        System.out.println("    i = " + i);
```

```
        System.out.println("    j = " + j);
        System.out.println("    k = " + k);

        //greater than
        System.out.println("Greater than...");
        System.out.println("    i > j = " + (i > j));     //false
        System.out.println("    j > i = " + (j > i));     //true
        System.out.println("    k > j = " + (k > j));     //false,
they are equal

        //greater than or equal to
        System.out.println("Greater than or equal to...");
        System.out.println("    i >= j = " + (i >= j));   //false
        System.out.println("    j >= i = " + (j >= i));   //true
        System.out.println("    k >= j = " + (k >= j));   //true

        //less than
        System.out.println("Less than...");
        System.out.println("    i < j = " + (i < j));     //true
        System.out.println("    j < i = " + (j < i));     //false
        System.out.println("    k < j = " + (k < j));     //false

        //less than or equal to
        System.out.println("Less than or equal to...");
        System.out.println("    i <= j = " + (i <= j));   //true
        System.out.println("    j <= i = " + (j <= i));   //false
        System.out.println("    k <= j = " + (k <= j));   //true

        //equal to
        System.out.println("Equal to...");
        System.out.println("    i == j = " + (i == j));   //false
        System.out.println("    k == j = " + (k == j));   //true

        //not equal to
        System.out.println("Not equal to...");
```



```
        System.out.println("    i != j = " + (i != j));   //true
        System.out.println("    k != j = " + (k != j));   //false

    }
}
```

Here's the output from this program:

```
Variable values...
    i = 37
    j = 42
    k = 42
Greater than...
    i > j = false
    j > i = true
    k > j = false
Greater than or equal to...
    i >= j = false
    j >= i = true
    k >= j = true
Less than...
    i < j = true
    j < i = false
    k < j = false
Less than or equal to...
    i <= j = true
    j <= i = false
    k <= j = true
Equal to...
    i == j = false
    k == j = true
Not equal to...
    i != j = true
    k != j = false
```

Relational operators often are used with conditional operators to construct more complex decision-making expressions. The Java programming language supports six conditional operators-five binary and one unary--as shown in the following table.

<table>
<tr><th>Operator</th><th>Use</th><th>Returns true if</th></tr>
<tr><td>&&</td><td>op1 && op2</td><td>op1 and op2 are both true, conditionally evaluates op2</td></tr>
<tr><td>||</td><td>op1 || op2</td><td>either op1 or op2 is true, conditionally evaluates op2</td></tr>
<tr><td>!</td><td>! op</td><td>op is false</td></tr>
<tr><td>&</td><td>op1 & op2</td><td>op1 and op2 are both true, always evaluates op1 and op2</td></tr>
<tr><td>|</td><td>op1 | op2</td><td>either op1 or op2 is true, always evaluates op1 and op2</td></tr>
<tr><td>^</td><td>op1 ^ op2</td><td>if op1 and op2 are different--that is if one or the other of the operands is true but not both</td></tr>
</table>

One such operator is &&, which performs the conditional AND operation. You can use two different relational operators along with && to determine whether both relationships are true. The following line of code uses this technique to determine whether an array index is between two boundaries. It determines whether the index is both greater than or equal to 0 and less than NUM_ENTRIES, which is a previously defined constant value.

```
0 <= index && index < NUM_ENTRIES
```

Note that in some instances, the second operand to a conditional operator may not be evaluated. Consider this code segment:

```
(numChars < LIMIT) && (...)
```


The && operator will return true only if both operands are true. So, if numChars is greater than or equal to LIMIT, the left-hand operand for && is false, and the return value of && can be determined without evaluating the right-hand operand. In such a case, the interpreter will not evaluate the right-hand operand. This has important implications if the right-hand operand has side effects, such as reading from a stream, updating a value, or making a calculation.
When both operands are boolean, the operator & performs the same operation as &&. However, & always evaluates both of its operands and returns true if both are true. Likewise, when the operands are boolean, | performs the same operation as ||. The | operator always evaluates both of its operands and returns true if at least one of its operands is true. When their operands are numbers, & and | perform bitwise manipulations.

Shift and Logical Operators
A shift operator performs bit manipulation on data by shifting the bits of its first operand right or left. This table summarizes the shift operators available in the Java programming language.

Operator	Use	Operation
>>	op1 >> op2	shift bits of op1 right by distance op2
<<	op1 << op2	shift bits of op1 left by distance op2
>>>	op1 >>> op2	shift bits of op1 right by distance op2 (unsigned)

Each operator shifts the bits of the left-hand operand over by the number of positions indicated by the right-hand operand. The shift occurs in the direction indicated by the operator itself. For example, the following statement shifts the bits of the integer 13 to the right by one position:

13 >> 1;

The binary representation of the number 13 is 1101. The result of the shift operation is 1101 shifted to the right by one position-110, or 6 in decimal. The left-hand bits are filled with 0s as needed.

The following table shows the four operators the Java programming language provides to perform bitwise functions on their operands:

Operator	Use	Operation
&	op1 & op2	bitwise and
|	op1 | op2	bitwise or
^	op1 ^ op2	bitwise xor
~	~op2	bitwise complement

When its operands are numbers, the & operation performs the bitwise AND function on each parallel pair of bits in each operand. The AND function sets the resulting bit to 1 if the corresponding bit in both operands is 1, as shown in the following table.

op1	op2	Result
0	0	0
0	1	0
1	0	0
1	1	1

Suppose that you were to AND the values 13 and 12, like this: 13 & 12. The result of this operation is 12 because the binary representation of 12 is 1100, and the binary representation of 13 is 1101.

```
  1101    //13
& 1100    //12
  ------
  1100    //12
```


If both operand bits are 1, the AND function sets the resulting bit to 1; otherwise, the resulting bit is 0. So, when you line up the two operands and perform the AND function, you can see that the two high-order bits (the two bits farthest to the left of each number) of each operand are 1. Thus, the resulting bit in the result is also 1. The low-order bits evaluate to 0 because either one or both bits in the operands are 0. When both of its operands are numbers, the | operator performs the inclusive or operation, and ^ performs the exclusive or (XOR) operation. Inclusive or means that if either of the two bits is 1, the result is 1. The following table shows the results of inclusive or operations:

op1	op2	Result
0	0	0
0	1	1
1	0	1
1	1	1

Exclusive or means that if the two operand bits are different the result is 1, otherwise the result is 0. The following table shows the results of an exclusive or operation.

op1	op2	Result
0	0	0
0	1	1
1	0	1
1	1	0

And finally, the complement operator inverts the value of each bit of the operand: if the operand bit is 1 the result is 0 and if the operand bit is 0 the result is 1.

Among other things, bitwise manipulations are useful for managing sets of boolean flags. Suppose, for example, that your program had several boolean flags that indicated the state of various components in your program: is it visible, is it draggable, and so on. Rather than define a separate boolean variable to hold each flag, you could define a single variable, flags, for all of them. Each bit within flags would represent the current state of one of the flags. You would then use bit manipulations to set and to get each flag. First, set up constants that indicate the various flags for your program. These flags should each be a different power of 2 to ensure that each bit is used by only one flag.
Define a variable, flags, whose bits would be set according to the current state of each flag. The following code sample initializes flags to 0, which means that all flags are false (none of the bits are set).

```
static final int VISIBLE = 1;
static final int DRAGGABLE = 2;
static final int SELECTABLE = 4;
static final int EDITABLE = 8;

int flags = 0;
```

To set the "visible" flag when something became visible you would use this statement:

```
flags = flags | VISIBLE;
```

To test for visibility, you could then write:


```
if ((flags & VISIBLE) == VISIBLE) {
    ...
}
```

Here's the complete program, BitwiseDemo, that includes this code.

```
public class BitwiseDemo {

    static final int VISIBLE = 1;
    static final int DRAGGABLE = 2;
    static final int SELECTABLE = 4;
    static final int EDITABLE = 8;

    public static void main(String[] args)
    {
        int flags = 0;

        flags = flags | VISIBLE;
        flags = flags | DRAGGABLE;

        if ((flags & VISIBLE) == VISIBLE) {
            if ((flags & DRAGGABLE) == DRAGGABLE)
{
                System.out.println("Flags are Visible and
Draggable.");
            }
        }

        flags = flags | EDITABLE;

        if ((flags & EDITABLE) == EDITABLE) {
            System.out.println("Flags are now also
Editable.");
        }
```

```
        }
    }
```

Here's the output from this program:

```
Flags are Visible and Draggable.
Flags are now also Editable.
```

Assignment Operators

You use the basic assignment operator, =, to assign one value to another. The MaxVariablesDemo program uses = to initialize all of its local variables:

```
// integers
byte largestByte = Byte.MAX_VALUE;
short largestShort = Short.MAX_VALUE;
int largestInteger = Integer.MAX_VALUE;
long largestLong = Long.MAX_VALUE;

// real numbers
float largestFloat = Float.MAX_VALUE;
double largestDouble = Double.MAX_VALUE;

// other primitive types
char aChar = 'S';
boolean aBoolean = true;
```

The Java programming language also provides several shortcut assignment operators that allow you to perform an arithmetic, shift, or bitwise operation and an assignment operation all with one operator. Suppose you wanted to add a number to a variable and assign the result back into the variable, like this:

```
i = i + 2;
```


You can shorten this statement using the shortcut operator +=, like this:

```
i += 2;
```

The two previous lines of code are equivalent. The following table lists the shortcut assignment operators and their lengthy equivalents:

Operator	**Use**	**Equivalent to**
+=	op1 += op2	op1 = op1 + op2
-=	op1 -= op2	op1 = op1 - op2
*=	op1 *= op2	op1 = op1 * op2
/=	op1 /= op2	op1 = op1 / op2
%=	op1 %= op2	op1 = op1 % op2
&=	op1 &= op2	op1 = op1 & op2
|=	op1 |= op2	op1 = op1 | op2
^=	op1 ^= op2	op1 = op1 ^ op2
<<=	op1 <<= op2	op1 = op1 << op2
>>=	op1 >>= op2	op1 = op1 >> op2
>>>=	op1 >>>= op2	op1 = op1 >>> op2

Other Operators

The following table lists the other operators that the Java programming language supports.

Operator	**Description**
?:	Shortcut if-else statement
[]	Used to declare arrays, create arrays, and access array elements
.	Used to form qualified names
(params)	Delimits a comma-separated list of parameters
(type)	Casts (converts) a value to the specified type

new	Creates a new object or a new array
instanceof	Determines whether its first operand is an instance of its second operand

Shortcut if-else Statement

The ?: operator is a conditional operator that is short-hand for an if-else statement:

```
op1 ? op2 : op3
```

The ?: operator returns op2 if op1 is true or returns op3 if op1 is false.

The [ ] Operator

You use square brackets to declare arrays, to create arrays, and to access a particular element in an array. Here's an example of an array declaration:

```
float[] arrayOfFloats = new float[10];
```

The previous code declares an array that can hold ten floating point numbers. Here's how you would access the 7th item in that array:

```
arrayOfFloats[6];
```

Note that array indices begin at 0.

The . Operator

The dot (.) operator accesses instance members of an object or class members of a class.

The () Operator

When declaring or calling a method, you list the method's arguments between (and). You can specify an empty argument list by using () with nothing between them.

The (type) Operator

Casts (or "converts") a value to the specified type.

The new Operator

You use the new operator to create a new object or a new array. Here's an example of creating a new Integer object from the Integer class in the java.lang package:

```
Integer anInteger = new Integer(10);
```

The instanceof Operator

The instanceof operator tests whether its first operand is an instance of its second.

```
op1 instanceof op2
```

op1 must be the name of an object and op2 must be the name of a class. An object is considered to be an instance of a class if that object directly or indirectly descends from that class.

1.3.Expressions, Statements, and Blocks

Variables and operators, which you met in the previous two sections, are basic building blocks of programs. You combine literals, variables, and operators to form expressions- segments of code that perform computations and return values. Certain expressions can be made into statements-complete units of execution. By grouping statements together with curly braces { and }, you create blocks of code.

Expressions

Expressions perform the work of a program. Among other things, expressions are used to compute and to assign values to variables and to help control the execution flow of a program. The job of an expression is twofold: to perform the computation indicated by the elements of the expression and to return a value that is the result of the computation.

Definition: An expression is a series of variables, operators, and method calls (constructed according to the syntax of the language) that evaluates to a single value.

As discussed in the previous section, operators return a value, so the use of an operator is an expression. This partial listing of the MaxVariablesDemo program shows some of the program's expressions:

```
...
// other primitive types
char aChar = 'S';
boolean aBoolean = true;

// display them all
System.out.println("The largest byte value is " +
largestByte);
...

if (Character.isUpperCase(aChar)) {
    ...
}
```

Each of these expressions performs an operation and returns a value.

The data type of the value returned by an expression depends on the elements used in the expression. The expression aChar = 'S' returns a character because the assignment operator returns a value of the same data type as its operands and aChar and 'S' are characters. As you see from the other expressions, an expression can return a boolean value, a string, and so on.

The Java programming language allows you to construct compound expressions and statements from various smaller expressions as long as the data types required by one part of the expression matches the data types of the other.

Here's an example of a compound expression:

```
x * y * z
```

In this particular example, the order in which the expression is evaluated is unimportant because the results of multiplication is independent of order--the outcome is always the same no matter what order you apply the multiplications. However, this is not true of all expressions. For example, the following expression gives different results depending on whether you perform the addition or the division operation first:

```
x + y / 100    //ambiguous
```

You can specify exactly how you want an expression to be evaluated by using balanced parentheses (and). For example to make the previous expression unambiguous, you could write:

```
(x + y)/ 100    //unambiguous, recommended
```

If you don't explicitly indicate the order in which you want the operations in a compound expression to be performed, the order is determined by the precedence assigned to the operators in use within the expression. Operators with a higher

precedence get evaluated first. For example, the division operator has a higher precedence than does the addition operator. Thus, the two following statements are equivalent:

```
x + y / 100
x + (y / 100) //unambiguous, recommended
```

When writing compound expressions, you should be explicit and indicate with parentheses which operators should be evaluated first. This will make your code easier to read and to maintain.

The following table shows the precedence assigned to the operators. The operators in this table are listed in precedence order: the higher in the table an operator appears, the higher its precedence. Operators with higher precedence are evaluated before operators with a relatively lower precedence. Operators on the same line have equal precedence.

<table>
<tr><td>postfix operators</td><td>[] . (params) expr++ expr--</td></tr>
<tr><td>unary operators</td><td>++expr --expr +expr -expr ~ !</td></tr>
<tr><td>creation or cast</td><td>new (type)expr</td></tr>
<tr><td>multiplicative</td><td>* / %</td></tr>
<tr><td>additive</td><td>+ -</td></tr>
<tr><td>shift</td><td><< >> >>></td></tr>
<tr><td>relational</td><td>< > <= >= instanceof</td></tr>
<tr><td>equality</td><td>== !=</td></tr>
<tr><td>bitwise AND</td><td>&</td></tr>
<tr><td>bitwise exclusive OR</td><td>^</td></tr>
<tr><td>bitwise inclusive OR</td><td>|</td></tr>
<tr><td>logical AND</td><td>&&</td></tr>
<tr><td>logical OR</td><td>||</td></tr>
<tr><td>conditional</td><td>? :</td></tr>
<tr><td>assignment</td><td>= += -= *= /= %= &= ^= |= <<= >>= >>>=</td></tr>
</table>

When operators of equal precedence appear in the same expression, a rule must govern which is evaluated first. All binary operators except for the assignment operators are evaluated in left-to-right order. Assignment operators are evaluated right to left.

Statements

Statements are roughly equivalent to sentences in natural languages. A statement forms a complete unit of execution. The following types of expressions can be made into a statement by terminating the expression with a semicolon (;):

Assignment expressions
Any use of ++ or --
Method calls
Object creation expressions

These kinds of statements are called expression statements. Here are some examples of expression statements:

```
    aValue = 8933.234;                    //assignment
statement
    aValue++;                        //increment statement
    System.out.println(aValue);           //method call
statement
    Integer integerObject = new Integer(4);  //object creation
statement
```

In addition to these kinds of expression statements, there are two other kinds of statements. A declaration statement declares a variable. You've seen many examples of declaration statements.

```
    double aValue = 8933.234;             // declaration
statement
```

A control flow statement regulates the order in which statements get executed. The for loop and the if statement are both examples of control flow statements.

Blocks

A block is a group of zero or more statements between balanced braces and can be used anywhere a single statement is allowed. The following listing shows two blocks from the MaxVariablesDemo program, each containing a single statement:

```
        if (Character.isUpperCase(aChar)) {
            System.out.println("The character " + aChar + " is
upper case.");
        } else {
            System.out.println("The character " + aChar + " is
lower case.");
        }
```

Summary of Expressions, Statements, and Blocks

An expression is a series of variables, operators, and method calls (constructed according to the syntax of the language) that evaluates to a single value. You can write compound expressions by combining expressions as long as the types required by all of the operators involved in the compound expression are correct. When writing compound expressions, you should be explicit and indicate with parentheses which operators should be evaluated first.
If you choose not to use parentheses, then the Java platform evaluates the compound expression in the order dictated by operator precedence. A statement forms a complete unit of execution and is terminated with a semicolon (;). There are three kinds of statements: expression statements, declaration statements, and control flow statements.

You can group zero or more statements together into a block with curly brackets ({ and }). Even though not required, we recommend using blocks with control flow statements even if there's only one statement in the block.

1.4. Control Flow Statements

When you write a program, you type statements into a file. Without control flow statements, the interpreter executes these statements in the order they appear in the file from left to right, top to bottom. You can use control flow statements in your programs to conditionally execute statements, to repeatedly execute a block of statements, and to otherwise change the normal, sequential flow of control. For example, in the following code snippet, the if statement conditionally executes the System.out.println statement within the braces, based on the return value of Character.isUpperCase(aChar):

```
        char c;
        ...
        if (Character.isUpperCase(aChar)) {
            System.out.println("The character " + aChar + " is
upper case.");
        }
```

The Java programming language provides several control flow statements, which are listed in the following table.

Statement Type	Keyword
looping	while, do-while , for
decision making	if-else, switch-case
exception handling	try-catch-finally, throw
branching	break, continue, label:, return

In the sections that follow, you will see the following notation to describe the general form of a control flow statement:

```
control flow statement details {
   statement(s)
}
```

Technically, the braces, { and }, are not required if the block contains only one statement. However, we recommend that you always use { and }, because the code is easier to read and it helps to prevent errors when modifying code.

The while and do-while Statements

You use a while statement to continually execute a block of statements while a condition remains true. The general syntax of the while statement is:

```
while (expression) {
   statement
}
```

First, the while statement evaluates expression, which must return a boolean value. If the expression returns true, then the while statement executes the statement(s) associated with it. The while statement continues testing the expression and executing its block until the expression returns false.

The example program shown below, called WhileDemo, uses a while statement to step through the characters of a string, appending each character from the string to the end of a string buffer until it encounters the letter g.

```
public class WhileDemo {
   public static void main(String[] args) {

      String copyFromMe = "Copy this string until you "
+
                     "encounter the letter 'g'.";
      StringBuffer copyToMe = new StringBuffer();

      int i = 0;
      char c = copyFromMe.charAt(i);

      while (c != 'g') {
         copyToMe.append(c);
         c = copyFromMe.charAt(++i);
      }
      System.out.println(copyToMe);
   }
}
```

The value printed by the last line is: Copy this strin.

The Java programming language provides another statement that is similar to the while statement--the do-while statement. The general syntax of the do-while is:

```
do {
   statement(s)
} while (expression);
```

Instead of evaluating the expression at the top of the loop, do-while evaluates the expression at the bottom. Thus the statements associated with a do-while are executed at least once.

Here's the previous program rewritten to use do-while and renamed to DoWhileDemo:

```
public class DoWhileDemo {
   public static void main(String[] args) {
```

```
+           String copyFromMe = "Copy this string until you "

                        "encounter the letter 'g'.";
            StringBuffer copyToMe = new StringBuffer();

            int i = 0;
            char c = copyFromMe.charAt(i);

            do {
               copyToMe.append(c);
               c = copyFromMe.charAt(++i);
            } while (c != 'g');
            System.out.println(copyToMe);
        }
    }
```

The value printed by the last line is: Copy this strin.

The for Statement

The for statement provides a compact way to iterate over a range of values. The general form of the for statement can be expressed like this:

```
for (initialization; termination; increment) {
   statement
}
```

The initialization is an expression that initializes the loop-it's executed once at the beginning of the loop. The termination expression determines when to terminate the loop. This expression is evaluated at the top of each iteration of the loop. When the expression evaluates to false, the loop terminates. Finally, increment is an expression that gets invoked after each

iteration through the loop. All these components are optional. In fact, to write an infinite loop, you omit all three expressions:

```
for ( ; ; ) {    // infinite loop
   ...
}
```

Often for loops are used to iterate over the elements in an array, or the characters in a string. The following sample, ForDemo, uses a for statement to iterate over the elements of an array and print them:

```
public class ForDemo {
   public static void main(String[] args) {
      int[] arrayOfInts = { 32, 87, 3, 589, 12, 1076,
                            2000, 8, 622, 127 };

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

The output of the program is: 32 87 3 589 12 1076 2000 8 622 127.

Note that you can declare a local variable within the initialization expression of a for loop. The scope of this variable extends from its declaration to the end of the block governed by the for statement so it can be used in the termination and increment expressions as well. If the variable that controls a for loop is not needed outside of the loop, it's best to declare the variable in the initialization expression. The names i, j, and k are often used to control for loops; declaring

them within the for loop initialization expression limits their life-span and reduces errors.

The if/else Statements

The if statement enables your program to selectively execute other statements, based on some criteria. For example, suppose that your program prints debugging information, based on the value of a boolean variable named DEBUG. If DEBUG is true, your program prints debugging information, such as the value of a variable, such as x. Otherwise, your program proceeds normally. A segment of code to implement this might look like this:

```
if (DEBUG) {
   System.out.println("DEBUG: x = " + x);
}
```

This is the simplest version of the if statement: The block governed by the if is executed if a condition is true. Generally, the simple form of if can be written like this:

```
if (expression) {
   statement(s)
}
```

What if you want to perform a different set of statements if the expression is false? You use the else statement for that. Consider another example. Suppose that your program needs to perform different actions depending on whether the user clicks the OK button or another button in an alert window. Your program could do this by using an if statement along with an else statement:

```
   . . .
   // response is either OK or CANCEL depending
```



```
   // on the button that the user pressed
   . . .
if (response == OK) {
   // code to perform OK action
} else {
   // code to perform Cancel action
}
```

The else block is executed if the if part is false. Another form of the else statement, else if, executes a statement based on another expression. An if statement can have any number of companion else if statements but only one else. Following is a program, IfElseDemo, that assigns a grade based on the value of a test score: an A for a score of 90% or above, a B for a score of 80% or above, and so on:

```
public class IfElseDemo {
   public static void main(String[] args) {

      int testscore = 76;
      char grade;

      if (testscore >= 90) {
         grade = 'A';
      } else if (testscore >= 80) {
         grade = 'B';
      } else if (testscore >= 70) {
         grade = 'C';
      } else if (testscore >= 60) {
         grade = 'D';
      } else {
         grade = 'F';
      }
      System.out.println("Grade = " + grade);
   }
```

```
}
```

The output from this program is:

```
Grade = C
```

You may have noticed that the value of testscore can satisfy more than one of the expressions in the compound if statement: 76 >= 70 and 76 >= 60. However, as the runtime system processes a compound if statement such as this one, once a condition is satisfied, the appropriate statements are executed (grade = 'C';), and control passes out of the if statement without evaluating the remaining conditions.
The Java programming language supports an operator, ?:, that is a compact version of an if statement. Recall this statement from the MaxVariablesDemo program:

```
if (Character.isUpperCase(aChar)) {
   System.out.println("The character " + aChar + " is
upper case.");
} else {
   System.out.println("The character " + aChar + " is
lower case.");
}
```

Here's how you could rewrite that statement using the ?: operator:

```
System.out.println("The character " + aChar + " is " +
            (Character.isUpperCase(aChar) ? "upper" :
"lower") +
            "case.");
```

The ?: operator returns the string "upper" if the isUpperCase method returns true. Otherwise, it returns the string "lower".

The result is concatenated with other parts of a message to be displayed. Using ?: makes sense here because the if statement is secondary to the call to the println method. Once you get used to this construct, it also makes the code easier to read.

The switch Statement

Use the switch statement to conditionally perform statements based on an integer expression. Following is a sample program, SwitchDemo, that declares an integer named month whose value supposedly represents the month in a date. The program displays the name of the month, based on the value of month, using the switch statement:

```
public class SwitchDemo {
   public static void main(String[] args) {

      int month = 8;
      switch (month) {
         case 1:  System.out.println("January"); break;
         case 2:  System.out.println("February"); break;
         case 3:  System.out.println("March"); break;
         case 4:  System.out.println("April"); break;
         case 5:  System.out.println("May"); break;
         case 6:  System.out.println("June"); break;
         case 7:  System.out.println("July"); break;
         case 8:  System.out.println("August"); break;
         case 9:  System.out.println("September"); break;
         case 10: System.out.println("October"); break;
         case 11: System.out.println("November"); break;
         case 12: System.out.println("December"); break;
      }
   }
}
```

The switch statement evaluates its expression, in this case the value of month, and executes the appropriate case statement. Thus, the output of the program is: August. Of course, you could implement this by using an if statement:

```
int month = 8;
if (month == 1) {
   System.out.println("January");
} else if (month == 2) {
   System.out.println("February");
}
. . . // and so on
```

Deciding whether to use an if statement or a switch statement is a judgment call. You can decide which to use, based on readability and other factors. An if statement can be used to make decisions based on ranges of values or conditions, whereas a switch statement can make decisions based only on a single integer value. Also, the value provided to each case statement must be unique.

Another point of interest in the switch statement is the break statement after each case. Each break statement terminates the enclosing switch statement, and the flow of control continues with the first statement following the switch block. The break statements are necessary because without them, the case statements fall through. That is, without an explicit break, control will flow sequentially through subsequent case statements. Following is an example, SwitchDemo2, which illustrates why it might be useful to have case statements fall through:

```
public class SwitchDemo2 {
   public static void main(String[] args) {

      int month = 2;
```



```
      int year = 2000;
      int numDays = 0;

      switch (month) {
         case 1:
         case 3:
         case 5:
         case 7:
         case 8:
         case 10:
         case 12:
            numDays = 31;
            break;
         case 4:
         case 6:
         case 9:
         case 11:
            numDays = 30;
            break;
         case 2:
            if ( ((year % 4 == 0) && !(year % 100 == 0))
                || (year % 400 == 0) )
               numDays = 29;
            else
               numDays = 28;
            break;
      }
      System.out.println("Number of Days = " +
numDays);
   }
}
```

The output from this program is:

```
Number of Days = 29
```

Technically, the final break is not required because flow would fall out of the switch statement anyway. However, we recommend using a break for the last case statement just in case you need to add more case statements at a later date. This makes modifying the code easier and less error-prone.
You will see break used to terminate loops in Branching Statements.
Finally, you can use the default statement at the end of the switch to handle all values that aren't explicitly handled by one of the case statements.

```
int month = 8;
. . .
switch (month) {
   case 1:  System.out.println("January"); break;
   case 2:  System.out.println("February"); break;
   case 3:  System.out.println("March"); break;
   case 4:  System.out.println("April"); break;
   case 5:  System.out.println("May"); break;
   case 6:  System.out.println("June"); break;
   case 7:  System.out.println("July"); break;
   case 8:  System.out.println("August"); break;
   case 9:  System.out.println("September"); break;
   case 10: System.out.println("October"); break;
   case 11: System.out.println("November"); break;
   case 12: System.out.println("December"); break;
   default: System.out.println("Hey, that's not a valid
month!"); break;
}
```

Exception Handling Statements
The Java programming language provides a mechanism known as exceptions to help programs report and handle errors. When an error occurs, the program throws an exception. What does

this mean? It means that the normal flow of the program is interrupted and that the runtime environment attempts to find an exception handler--a block of code that can handle a particular type of error. The exception handler can attempt to recover from the error or, if it determines that the error is unrecoverable, provide a gentle exit from the program.
Three statements play a part in handling exceptions:
The try statement identifies a block of statements within which an exception might be thrown.
The catch statement must be associated with a try statement and identifies a block of statements that can handle a particular type of exception. The statements are executed if an exception of a particular type occurs within the try block.
The finally statement must be associated with a try statement and identifies a block of statements that are executed regardless of whether or not an error occurs within the try block.
Here's the general form of these statements:

```
try {
   statement(s)
} catch (exceptiontype name) {
   statement(s)
} finally {
   statement(s)
}
```

This has been a brief overview of the statements provided by the Java programming language used in reporting and handling errors. However, other factors and considerations, such as the difference between runtime and checked exceptions and the hierarchy of exceptions classes, which represent various types of exceptions, play a role in using the exception mechanism. The Handling Errors with Exceptions lesson provides a complete discussion on this subject.

Branching Statements

The Java programming language supports three branching statements:

- The break statement
- The continue statement
- The return statement

The break statement and the continue statement, which are covered next, can be used with or without a label. A label is an identifier placed before a statement. The label is followed by a colon (:):

```
statementName: someJavaStatement;
```

You'll see an example of a label within the context of a program in the next section.

The break Statement

The break statement has two forms: unlabeled and labeled. You saw the unlabeled form of the break statement used with switch earlier. As noted there, an unlabeled break terminates the enclosing switch statement, and flow of control transfers to the statement immediately following the switch. You can also use the unlabeled form of the break statement to terminate a for, while, or do-while loop. The following sample program, BreakDemo, contains a for loop that searches for a particular value within an array:

```
public class BreakDemo {
    public static void main(String[] args) {

        int[] arrayOfInts = { 32, 87, 3, 589, 12, 1076,
                              2000, 8, 622, 127 };
        int searchfor = 12;
```



```
        int i = 0;
        boolean foundIt = false;

        for ( ; i < arrayOfInts.length; i++) {
            if (arrayOfInts[i] == searchfor) {
                foundIt = true;
                break;
            }
        }

        if (foundIt) {
            System.out.println("Found " + searchfor + " at
index " + i);
        } else {
            System.out.println(searchfor + "not in the array");
        }
    }
}
```

The break statement terminates the for loop when the value is found. The flow of control transfers to the statement following the enclosing for, which is the print statement at the end of the program.

The output of this program is:

```
Found 12 at index 4
```

The unlabeled form of the break statement is used to terminate the innermost switch, for, while, or do-while; the labeled form terminates an outer statement, which is identified by the label specified in the break statement. The following program, BreakWithLabelDemo, is similar to the previous one, but it searches for a value in a two-dimensional array. Two nested for loops traverse the array. When the value is found, a labeled

break terminates the statement labeled search, which is the outer for loop:

```
public class BreakWithLabelDemo {
   public static void main(String[] args) {

      int[][] arrayOfInts = { { 32, 87, 3, 589 },
                              { 12, 1076, 2000, 8 },
                              { 622, 127, 77, 955 }
                            };
      int searchfor = 12;

      int i = 0;
      int j = 0;
      boolean foundIt = false;

   search:
      for ( ; i < arrayOfInts.length; i++) {
         for (j = 0; j < arrayOfInts[i].length; j++) {
            if (arrayOfInts[i][j] == searchfor) {
               foundIt = true;
               break search;
            }
         }
      }

      if (foundIt) {
         System.out.println("Found " + searchfor + " at "
+ i + ", " + j);
      } else {
         System.out.println(searchfor + "not in the array");
      }

   }
}
```


The output of this program is:

```
Found 12 at 1, 0
```

This syntax can be a little confusing. The break statement terminates the labeled statement; it does not transfer the flow of control to the label. The flow of control transfers to the statement immediately following the labeled (terminated) statement.

The continue Statement

You use the continue statement to skip the current iteration of a for, while , or do-while loop. The unlabeled form skips to the end of the innermost loop's body and evaluates the boolean expression that controls the loop, basically skipping the remainder of this iteration of the loop. The following program, ContinueDemo, steps through a string buffer checking each letter. If the current character is not a p, the continue statement skips the rest of the loop and proceeds to the next character. If it is a p, the program increments a counter, and converts the p to an uppercase letter.

```
public class ContinueDemo {
   public static void main(String[] args) {

      StringBuffer searchMe = new StringBuffer(
            "peter piper picked a peck of pickled
peppers");
      int max = searchMe.length();
      int numPs = 0;

      for (int i = 0; i < max; i++) {
         //interested only in p's
         if (searchMe.charAt(i) != 'p')
```

```
            continue;

          //process p's
          numPs++;
          searchMe.setCharAt(i, 'P');
        }
        System.out.println("Found " + numPs + " p's in the
string.");
        System.out.println(searchMe);
    }
}
```

Here is the output of this program:

```
Found 9 p's in the string.
Peter PiPer Picked a Peck of Pickled PePPers
```

The labeled form of the continue statement skips the current iteration of an outer loop marked with the given label. The following example program, ContinueWithLabelDemo, uses nested loops to search for a substring within another string. Two nested loops are required: one to iterate over the substring and one to iterate over the string being searched. This program uses the labeled form of continue to skip an iteration in the outer loop:

```
public class ContinueWithLabelDemo {
   public static void main(String[] args) {

      String searchMe = "Look for a substring in me";
      String substring = "sub";
      boolean foundIt = false;

      int max = searchMe.length() - substring.length();

      test:
         for (int i = 0; i <= max; i++) {
            int n = substring.length();
            int j = i;
            int k = 0;
            while (n-- != 0) {
               if (searchMe.charAt(j++) !=
substring.charAt(k++)) {
                   continue test;
               }
            }
            foundIt = true;
                   break test;
         }
         System.out.println(foundIt ? "Found it" : "Didn't
find it");
      }
   }
```

Here is the output from this program:

```
Found it
```

The return Statement

The last of Java's branching statements is the return statement. You use return to exit from the current method. The flow of control returns to the statement that follows the original method call. The return statement has two forms: one that returns a value and one that doesn't. To return a value, simply put the value (or an expression that calculates the value) after the return keyword:

```
return ++count;
```

The data type of the value returned by return must match the type of the method's declared return value. When a method is declared void, use the form of return that doesn't return a value:

```
return;
```

Note : Although goto is a reserved word, currently the Java programming language does not support the goto statement.

Chapter2.Object Basics and Simple Data Objects

This trail begins with a general discussion about the life cycle of objects. The information presented applies to objects of all types and includes how to create an object, how to use it, and, finally, how the system cleans up the object when it's no longer being used.

2.1.The Life Cycle of an Object

A typical Java program creates many objects, which interact with one another by sending each other messages.

Through these object interactions, a Java program can implement a GUI, run an animation, or send and receive information over a network. Once an object has completed the work for which it was created, it is garbage-collected and its resources are recycled for use by other objects.

Here's a small program, called CreateObjectDemo, that creates three objects: one Point object and two Rectangle objects. You will need all three source files to compile this program.

```
public class CreateObjectDemo {
    public static void main(String[] args) {

        // create a point object and two rectangle objects
        Point origin_one = new Point(23, 94);
        Rectangle rect_one = new Rectangle(origin_one,
100, 200);
        Rectangle rect_two = new Rectangle(50, 100);

        // display rect_one's width, height, and area
```

```
            System.out.println("Width of rect_one: " +
rect_one.width);
            System.out.println("Height of rect_one: " +
rect_one.height);
            System.out.println("Area of rect_one: " +
rect_one.area());

            // set rect_two's position
            rect_two.origin = origin_one;

            // display rect_two's position
            System.out.println("X Position of rect_two: " +
rect_two.origin.x);
            System.out.println("Y Position of rect_two: " +
rect_two.origin.y);

            // move rect_two and display its new position
            rect_two.move(40, 72);
            System.out.println("X Position of rect_two: " +
rect_two.origin.x);
            System.out.println("Y Position of rect_two: " +
rect_two.origin.y);
        }
    }
```

After creating the objects, the program manipulates the objects and displays some information about them. Here's the output from the program:

```
Width of rect_one: 100
Height of rect_one: 200
Area of rect_one: 20000
X Position of rect_two: 23
Y Position of rect_two: 94
```



```
X Position of rect_two: 40
Y Position of rect_two: 72
```

This section uses this example to describe the life cycle of an object within a program. From this, you can learn how to write code that creates and uses an object and how the system cleans it up.

Creating Objects

As you know, a class provides the description for objects; you create an object from a class. Each of the following statements taken from the CreateObjectDemo program creates an object:

```
        Point origin_one = new Point(23, 94);
        Rectangle rect_one = new Rectangle(origin_one, 100,
200);
        Rectangle rect_two = new Rectangle(50, 100);
```

The first line creates an object from the Point class and the second and third lines each create an object from the Rectangle class.

Each statement has three parts:

1.Declaration: The code in the previous listing uses a number of variable declarations that associate a name with a type. When you create an object, you do not have to declare a variable to refer to it. However, a variable declaration often appears on the same line as the code to create an object.

2.Instantiation: new is a Java operator that creates the new object (allocates space for it).

3.Initialization: The new operator is followed by a call to a constructor. For example, Point(23, 94) is a call to Point's only constructor. The constructor initializes the new object.

The next three subsections discuss each of these actions in detail:

Declaring a Variable to Refer to an Object

From the Variables section in the previous lesson, you learned that to declare a variable, you write:

```
type name
```

This notifies the compiler that you will use name to refer to data whose type is type.

In addition to the primitive types, such as int and boolean, provided directly by the Java platform, classes and interfaces are also types. So to declare a variable to refer to an object, you can use the name of a class or an interface, as the variable's type. The sample program uses both the Point and the Rectangle class names as types to declare variables.

```
Point origin_one = new Point(23, 94);
Rectangle rect_one = new Rectangle(origin_one, 100,
200);
Rectangle rect_two = new Rectangle(50, 100);
```

Declarations do not create new objects. The code Point origin_one does not create a new Point object; it just declares a variable, named origin_one, that will be used to refer to a Point object. The reference is empty until assigned. An empty reference is known as a null reference.

To create an object you must instantiate it with the new operator.

Instantiating an Object

The new operator instantiates a class by allocating memory for a new object. The new operator requires a single, postfix

argument: a call to a constructor. The name of the constructor provides the name of the class to instantiate. The constructor initializes the new object.

The new operator returns a reference to the object it created. Often, this reference is assigned to a variable of the appropriate type. If the reference is not assigned to a variable, the object is unreachable after the statement in which the new operator appears finishes executing.

Initializing an Object

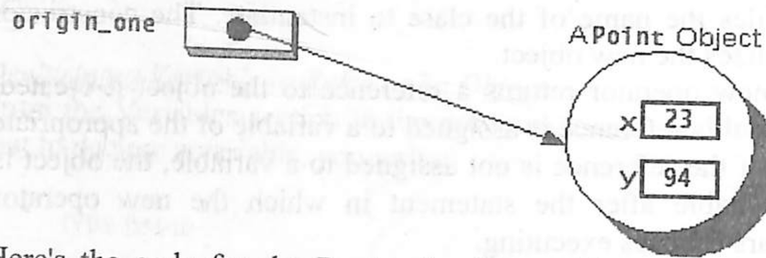
Here's the code for the Point class:

```
public class Point {
   public int x = 0;
   public int y = 0;
   //A constructor!
   public Point(int x, int y) {
      this.x = x;
      this.y = y;
   }
}
```

This class contains a single constructor. You can recognize a constructor because it has the same name as the class and has no return type. The constructor in the Point class takes two integer arguments, as declared by the code (int x, int y). The following statement provides 23 and 94 as values for those arguments:

```
Point origin_one = new Point(23, 94);
```

The effect of the previous line of code can be illustrated in the next figure:



Here's the code for the Rectangle class, which contains four constructors:

```
public class Rectangle {
    public int width = 0;
    public int height = 0;
    public Point origin;

    //Four constructors
    public Rectangle() {
        origin = new Point(0, 0);
    }

    public Rectangle(Point p) {
        origin = p;
    }

    public Rectangle(int w, int h) {
        this(new Point(0, 0), w, h);
    }

    public Rectangle(Point p, int w, int h) {
        origin = p;
        width = w;
        height = h;
    }
```



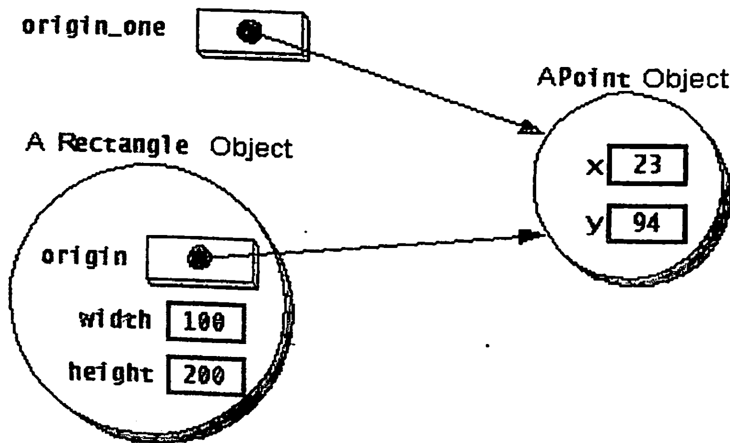
```
    //A method for moving the rectangle
    public void move(int x, int y) {
        origin.x = x;
        origin.y = y;
    }

    //A method for computing the area of the rectangle
    public int area() {
        return width * height;
    }
}
```

Each constructor lets you provide initial values for different aspects of the rectangle: the origin; the width, and the height; all three; or none. If a class has multiple constructors, they all have the same name but a different number of arguments or different typed arguments. The Java platform differentiates the constructors, based on the number and the type of the arguments. When the Java platform encounters the following code, it knows to call the constructor in the Rectangle class that requires a Point argument followed by two integer arguments:

```
Rectangle rect_one = new Rectangle(origin_one, 100,
200);
```

This call initializes the rectangle's origin variable to the Point object referred to by origin_one. The code also sets width to 100 and height to 200. Now there are two references to the same Point object; an object can have multiple references to it, as shown in the next figure:



Multiple references can refer to the same object. The following line of code calls the constructor that requires two integer arguments, which provide the initial values for width and height. If you inspect the code within the constructor, you will see that it creates a new Point object whose x and y values are initialized to 0:

```
Rectangle rect_two = new Rectangle(50; 100);
```

The Rectangle constructor used in the following statement doesn't take any arguments, so it's called a no-argument constructor:

```
Rectangle rect = new Rectangle();
```

If a class does not explicitly declare any constructors, the Java platform automatically provides a no-argument constructor, called the default constructor, that does nothing. Thus, all classes have at least one constructor.

Cleaning Up Unused Objects

Some object-oriented languages require that you keep track of all the objects you create and that you explicitly destroy them when they are no longer needed. Managing memory explicitly is tedious and error prone. The Java platform allows you to create as many objects as you want (limited, of course, by what your system can handle), and you don't have to worry about destroying them. The Java runtime environment deletes objects when it determines that they are no longer being used. This process is called garbage collection.

An object is eligible for garbage collection when there are no more references to that object. References that are held in a variable are usually dropped when the variable goes out of scope. Or, you can explicitly drop an object reference by setting the variable to the special value null. Remember that a program can have multiple references to the same object; all references to an object must be dropped before the object is eligible for garbage collection.

The Garbage Collector

The Java runtime environment has a garbage collector that periodically frees the memory used by objects that are no longer referenced. The garbage collector does its job automatically, although, in some situations, you may want to run the garbage collection explicitly by calling the gc method in the System class. For instance, you might want to run the garbage collector after a section of code that creates a large amount of garbage or before a section of code that needs a lot of memory.

Finalization

Before an object gets garbage-collected, the garbage collector gives the object an opportunity to clean up after itself through a

call to the object's finalize method. This process is known as finalization.

Most programmers don't have to worry about implementing the finalize method. In rare cases, however, a programmer might have to implement a finalize method to release resources, such as native peers, that aren't under the control of the garbage collector.

The finalize method is a member of the Object class, which is the top of the Java platform's class hierarchy and a superclass of all classes. A class can override the finalize method to perform any finalization necessary for objects of that type. If you override finalize, your implementation of the method should call super.finalize as the last thing it does.

2.2. Characters and Strings

The Java platform contains three classes that you can use when working with character data:

Character--A class whose instances can hold a single character value. This class also defines handy methods that can manipulate or inspect single-character data.

String-- A class for working with immutable (unchanging) data composed of multiple characters.

StringBuffer--A class for storing and manipulating mutable data composed of multiple characters.

Characters

An object of Character type contains a single character value. You use a Character object instead of a primitive char variable when an object is required-for example, when passing a character value into a method that changes the value or when placing a character value into a data structure, such as a vector, that requires objects. The following sample program, CharacterDemo, creates a few character objects and displays

some information about them. The code that is related to the Character class is shown below:

```
public class CharacterDemo {
   public static void main(String args[]) {
      Character a = new Character('a');
      Character a2 = new Character('a');
      Character b = new Character('b');

      int difference = a.compareTo(b);

      if (difference == 0) {
         System.out.println("a is equal to b.");
      } else if (difference < 0) {
         System.out.println("a is less than b.");
      } else if (difference > 0) {
         System.out.println("a is greater than b.");
      }
      System.out.println("a is "
                  + ((a.equals(a2)) ? "equal" : "not equal")
                  + " to a2.");
      System.out.println("The character " + a.toString() +
" is "
         + (Character.isUpperCase(a.charValue()) ?
"upper" : "lower")
         + "case.");
   }
}
```

The following is the output from this program:

```
a is less than b.
a is equal to a2.
The character a is lowercase.
```

The CharacterDemo program calls the following constructors and methods provided by the Character class:
Character(char)
The Character class's only constructor, which creates a Character object containing the value provided by the argument. Once a Character object has been created, the value it contains cannot be changed.

compareTo(Character)
An instance method that compares the values held by two character objects: the object on which the method is called (a in the example) and the argument to the method (b in the example). This method returns an integer indicating whether the value in the current object is greater than, equal to, or less than the value held by the argument. A letter is greater than another letter if its numeric value is greater.

equals(Object)
An instance method that compares the value held by the current object with the value held by another. This method returns true if the values held by both objects are equal.

toString()
An instance method that converts the object to a string. The resulting string is one character in length and contains the value held by the character object.

charValue()
An instance method that returns the value held by the character object as a primitive char value.

isUpperCase(char)

A class method that determines whether a primitive char value is uppercase. This is one of many Character class methods that inspect or manipulate character data.

Why Two String Classes?
The Java platform provides two classes, String and StringBuffer, that store and manipulate strings-character data consisting of more than one character. The String class provides for strings whose value will not change. For example, if you write a method that requires string data and the method is not going to modify the string in any way, pass a String object into the method. The StringBuffer class provides for strings that will be modified; you use string buffers when you know that the value of the character data will change. You typically use string buffers for constructing character data dynamically: for example, when reading text data from a file. Because strings are constants, they are more efficient to use than are string buffers and can be shared. So it's important to use strings when you can.

Following is a sample program called StringsDemo, which reverses the characters of a string. This program uses both a string and a string buffer.

```
public class StringsDemo {
   public static void main(String[] args) {
      String palindrome = "Dot saw I was Tod";
      int len = palindrome.length();
      StringBuffer dest = new StringBuffer(len);

      for (int i = (len - 1); i >= 0; i--) {
         dest.append(palindrome.charAt(i));
      }
      System.out.println(dest.toString());
```

```
    }
}
```

The output from this program is:

```
doT saw I was toD
```

In addition to highlighting the differences between strings and string buffers, this section discusses several features of the String and StringBuffer classes: creating strings and string buffers, using accessor methods to get information about a string or string buffer, and modifying a string buffer.

Creating Strings and StringBuffers

A string is often created from a string literal--a series of characters enclosed in double quotes. For example, when it encounters the following string literal, the Java platform creates a String object whose value is Gobbledygook.

```
"Gobbledygook"
```

The StringsDemo program uses this technique to create the string referred to by the palindrome variable:

```
String palindrome = "Dot saw I was Tod";
```

You can also create String objects as you would any other Java object: using the new keyword and a constructor. The String class provides several constructors that allow you to provide the initial value of the string, using different sources, such as an array of characters, an array of bytes, or a string buffer.

Here's an example of creating a string from a character array:


```
char[] helloArray = { 'h', 'e', 'l', 'l', 'o' };
helloString = new String(helloArray);
System.out.println(helloString);
```

The last line of this code snippet displays: hello.

You must always use new to create a string buffer. The StringsDemo program creates the string buffer referred to by dest, using the constructor that sets the buffer's capacity:

```
String palindrome = "Dot saw I was Tod";
int len = palindrome.length();
StringBuffer dest = new StringBuffer(len);
```

This code creates the string buffer with an initial capacity equal to the length of the string referred to by the name palindrome. This ensures only one memory allocation for dest because it's just big enough to contain the characters that will be copied to it. By initializing the string buffer's capacity to a reasonable first guess, you minimize the number of times memory must be allocated for it. This makes your code more efficient because memory allocation is a relatively expensive operation.

Accessor Methods

Getting the Length of a String or a String Buffer

Methods used to obtain information about an object are known as accessor methods. One accessor method that you can use with both strings and string buffers is the length method, which returns the number of characters contained in the string or the string buffer. After the following two lines of code have been executed, len equals 17:

```
String palindrome = "Dot saw I was Tod";
```

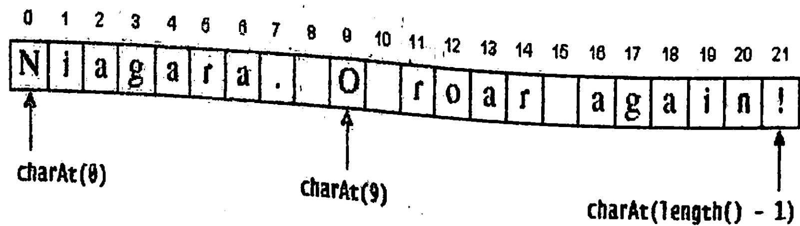
```
int len = palindrome.length();
```

In addition to length, the StringBuffer class has a method called capacity, which returns the amount of space allocated for the string buffer rather than the amount of space used. For example, the capacity of the string buffer referred to by dest in the StringsDemo program never changes, although its length increases by 1 for each iteration of the loop. The following figure shows the capacity and the length of dest after nine characters have been appended to it.
A string buffer's length is the number of characters it contains; a string buffer's capacity is the number of character spaces that have been allocated. The String class doesn't have a capacity method, because a string cannot change.

Getting Characters by Index from a String or a String Buffer
You can get the character at a particular index within a string or a string buffer by using the charAt accessor. The index of the first character is 0; the index of the last is length()-1. For example, the following code gets the character at index 9 in a string:

```
String anotherPalindrome = "Niagara. O roar again!";
char aChar = anotherPalindrome.charAt(9);
```

Indices begin at 0, so the character at index 9 is 'O', as illustrated in the following figure:



Use the charAt method to get a character at a particular index.
The figure also shows that to compute the index of the last character of a string, you have to subtract 1 from the value returned by the length method.
If you want to get more than one character from a string or a string buffer, you can use the substring method. The substring method has two versions, as shown in the following table:
The following code gets from the Niagara palindrome the substring that extends from index 11 to index 15, which is the word "roar":

```
String anotherPalindrome = "Niagara. O roar again!";
String roar = anotherPalindrome.substring(11, 15);
```

Use the substring method to get part of a string or string buffer.
Remember that indices begin at 0.

Modifying StringBuffers
The reverseIt method uses StringBuffer's append method to add a character to the end of the destination string: dest.

```
class ReverseString {
   public static String reverseIt(String source) {
      int i, len = source.length();
      StringBuffer dest = new StringBuffer(len);

      for (i = (len - 1); i >= 0; i--) {
         dest.append(source.charAt(i));
      }
      return dest.toString();
   }
}
```

If the appended character causes the size of the StringBuffer to grow beyond its current capacity, the StringBuffer allocates more memory. Because memory allocation is a relatively expensive operation, you can make your code more efficient by initializing a StringBuffer's capacity to a reasonable first guess, thereby minimizing the number of times memory must be allocated for it. For example, the reverseIt method constructs the StringBuffer with an initial capacity equal to the length of the source string, ensuring only one memory allocation for dest.

The version of the append method used in reverseIt is only one of the StringBuffer methods that appends data to the end of a StringBuffer. There are several append methods that append data of various types, such as float, int, boolean, and even Object, to the end of the StringBuffer. The data is converted to a string before the append operation takes place.

Inserting Characters

At times, you may want to insert data into the middle of a StringBuffer. You do this with one of StringBufffer's insert methods. This example illustrates how you would insert a string into a StringBuffer:

```
StringBuffer sb = new StringBuffer("Drink Java!");
sb.insert(6, "Hot ");
System.out.println(sb.toString());
```

This code snippet prints:

```
Drink Hot Java!
```

With StringBuffer's many insert methods, you specify the index before which you want the data inserted. In the example, "Hot " needed to be inserted before the 'J' in "Java". Indices

begin at 0, so the index for 'J' is 6. To insert data at the beginning of a StringBuffer, use an index of 0. To add data at the end of a StringBuffer, use an index equal to the current length of the StringBuffer or use append.

Setting Characters

Another useful StringBuffer modifier is setCharAt, which replaces the character at a specific location in the StringBuffer with the character specified in the argument list.. setCharAt is useful when you want to reuse a StringBuffer.

Converting Objects to Strings

The toString Method

It's often convenient or necessary to convert an object to a String because you need to pass it to a method that accepts only String values. The reverseIt method used earlier in this lesson uses StringBuffer's toString method to convert the StringBuffer to a String object before returning the String.

```
class ReverseString {
   public static String reverseIt(String source) {
      int i, len = source.length();
      StringBuffer dest = new StringBuffer(len);

      for (i = (len - 1); i >= 0; i--) {
         dest.append(source.charAt(i));
      }
      return dest.toString();
   }
}
```

All classes inherit toString from the Object class and many classes in the java.lang package override this method to provide an implementation that is meaningful to that class. For

example, the "type wrapper" classes--Character, Integer, Boolean, and the others--all override toString to provide a String representation of the object.

The valueOf Method
As a convenience, the String class provides the class method valueOf, You can use valueOf to convert variables of different types to Strings. For example, to print the value of pi:

```
System.out.println(String.valueOf(Math.PI));
```

Converting Strings to Numbers
The String class itself does not provide any methods for converting a String to a floating point, integer, or other numerical type. However, four of the "type wrapper" classes (Integer, Double, Float, and Long) provide a class method named valueOf that converts a String to an object of that type. Here's a small, contrived example of the Float class's valueOf:

```
String piStr = "3.14159";
Float pi = Float.valueOf(piStr);
```

Strings and the Java Compiler
The Java compiler uses the String and StringBuffer classes behind the scenes to handle literal strings and concatenation.
Literal Strings
In Java, you specify literal strings between double quotes:

```
"Hello World!"
```

You can use literal strings anywhere you would use a String object. For example, System.out.println accepts a String argument, so you could use a literal string in place of a String there.


```
System.out.println("Might I add that you look lovely today.");
```

You can also use String methods directly from a literal string.

```
int len = "Goodbye Cruel World".length();
```

Because the compiler automatically creates a new String object for every literal string it encounters, you can use a literal string to initialize a String.

```
String s = "Hola Mundo";
```

The above construct is equivalent to, but more efficient than, this one, which ends up creating two Strings instead of one:

```
String s = new String("Hola Mundo");
```

The compiler creates the first string when it encounters the literal string "Hola Mundo!", and the second one when it encounters new String.

Concatenation and the + Operator
In Java, you can use + to concatenate Strings together:

```
String cat = "cat";
System.out.println("con" + cat + "enation");
```

This is a little deceptive because, as you know, Strings can't be changed. However, behind the scenes the compiler uses StringBuffers to implement concatenation. The above example compiles to:

```
String cat = "cat";
System.out.println(new StringBuffer().append("con").
                    append(cat).append("enation").toString());
```

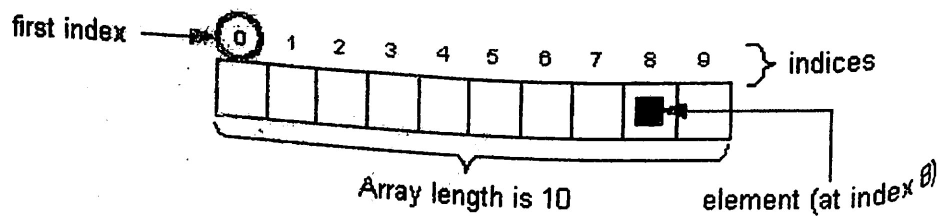
You can also use the + operator to append values to a String that are not themselves Strings:

```
System.out.println("Java's Number " + 1);
```

The compiler converts the non-String value (the integer 1 in the example) to a String object. before performing the concatenation operation.

2.3.Arrays

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



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

Creating and Using Arrays

Here's a simple program, called ArrayDemo, that creates the array, puts some values in it, and displays the values.


```
public class ArrayDemo {
   public static void main(String[] args) {
      int[] anArray;         // declare an array of integers

      anArray = new int[10];  // create an array of integers

      // assign a value to each array element and print
      for (int i = 0; i < anArray.length; i++) {
         anArray[i] = i;
         System.out.print(anArray[i] + " ");
      }
      System.out.println();
   }
}
```

Declaring a Variable to Refer to an Array

This line of code from the sample program declares an array variable:

```
int[] anArray;         // declare an array of integers
```

Like declarations for variables of other types, an array declaration has two components: the array's type and the array's name. An array's type is written type[], where type is the data type of the elements contained within the array, and [] indicates that this is an array. Remember that all of the elements within an array are of the same type.

The sample program uses int[], so the array called anArray will be used to hold integer data. Here are declarations for arrays that hold other types of data:

```
float[] anArrayOfFloats;
boolean[] anArrayOfBooleans;
```

```
Object[] anArrayOfObjects;
String[] anArrayOfStrings;
```

As with declarations for variables of other types, the declaration for an array variable does not allocate any memory to contain the array elements. The sample program must assign a value to anArray before the name refers to an array.

Creating an Array
You create an array explicitly using Java's new operator. The next statement in the sample program allocates an array with enough memory for ten integer elements and assigns the array to the variable anArray declared earlier.

```
anArray = new int[10]; // create an array of integers
```

In general, when creating an array, you use the new operator, plus the data type of the array elements, plus the number of elements desired enclosed within square brackets ('[' and ']').

```
new elementType[arraySize]
```

If the new statement were ommitted from the sample program, the compiler would print an error like the following one and compilation would fail.

```
ArrayDemo.java:4: Variable anArray may not have been
initialized.
```

Accessing an Array Element
Now that some memory has been allocated for the array, the program assigns values to the array elements:

```
for (int i = 0; i < anArray.length; i++) {
```



```
    anArray[i] = i;
    System.out.print(anArray[i] + " ");
}
```

This part of the code shows that to reference an array element, either to assign a value to it, or to access the value, you append square brackets to the array name. The value between the square brackets indicates (either with a variable or some other expression) the index of the element to access. Note that in Java, array indices begin at 0 and end at the array length minus 1.

Getting the Size of an Array
To get the size of an array, you write

```
arrayname.length
```

Be careful: Programmers new to the Java programming language are tempted to follow length with an empty set of parenthesis. This doesn't work because length is not a method. length is a property provided by the Java platform for all arrays.
The for loop in our sample program iterates over each element of anArray, assigning values to its elements. Thefor loop uses anArray.length to determine when to terminate the loop.

Array Initializers
The Java programming language provides a shortcut syntax for creating and initializing an array. Here's an example of this syntax:

```
boolean[] answers = { true, false, true, true, false };
```

The length of the array is determined by the number of values provided between { and }.

Arrays of Objects

Arrays can hold reference types as well as primitive types. You create such an array in much the same way you create an array with primitive types. Here's a small program, ArrayOfStringsDemo that creates an array containing three string objects then prints the strings in all lower case letters.

```
public class ArrayOfStringsDemo {
    public static void main(String[] args) {
        String[] anArray = { "String One", "String Two",
"String Three" };

        for (int i = 0; i < anArray.length; i++) {
            System.out.println(anArray[i].toLowerCase());
        }
    }
}
```

This program creates and populates the array in a single statement. However, you can create an array without putting any elements in it. This brings us to a potential stumbling block, often encountered by new programmers, when using arrays that contain objects. Consider this line of code:

```
String[] anArray = new String[5];
```

After this line of code is executed, the array called anArray exists and has enough room to hold 5 string objects. However, the array doesn't contain any strings yet. It is empty. The program must explicitly create strings and put them in the array. This might seem obvious, however, many beginners assume that the previous line of code creates the array and creates 5 empty strings in it. Thus they end up writing code like the following, which generates a NullPointerException:

```
String[] anArray = new String[5];

for (int i = 0; i < anArray.length; i++) {
    // ERROR: the following line gives a runtime error
    System.out.println(anArray[i].toLowerCase());
}
```

The problem is more likely to occur when the array is created in a constructor or other initializer and then used somewhere else in the program.

Arrays of Arrays

Arrays can contain arrays. ArrayOfArraysDemo creates an array and uses an initializer to populate it with four sub-arrays.

```
public class ArrayOfArraysDemo {
    public static void main(String[] args) {
        String[][] cartoons =
        {
            { "Flintstones", "Fred", "Wilma", "Pebbles",
"Dino" },
            { "Rubbles", "Barney", "Betty", "Bam Bam" },
            { "Jetsons", "George", "Jane", "Elroy", "Judy",
"Rosie", "Astro" },
            { "Scooby Doo Gang", "Scooby Doo", "Shaggy",
"Velma", "Fred", "Daphne" }
        };

        for (int i = 0; i < cartoons.length; i++) {
            System.out.print(cartoons[i][0] + ": ");
```

```
            for (int j = 1; j < cartoons[i].length; j++) {
                System.out.print(cartoons[i][j] + " ");
            }
            System.out.println();
        }
    }
}
```

Notice that the sub-arrays are all of different lengths. The names of the sub-arrays are cartoons[0], cartoons[1], and so on. As with arrays of objects, you must explicitly create the sub-arrays within an array. So if you don't use an initializer, you need to write code like the following, which you can find in: ArrayOfArraysDemo2

```
public class ArrayOfArraysDemo2 {
    public static void main(String[] args) {
        int[][] aMatrix = new int[4][];

        //populate matrix
        for (int i = 0; i < aMatrix.length; i++) {
            aMatrix[i] = new int[5];  //create sub-array
            for (int j = 0; j < aMatrix[i].length; j++) {
                aMatrix[i][j] = i + j;
            }
        }

        //print matrix
        for (int i = 0; i < aMatrix.length; i++) {
            for (int j = 0; j < aMatrix[i].length; j++) {
                System.out.print(aMatrix[i][j] + " ");
            }
            System.out.println();
        }
```



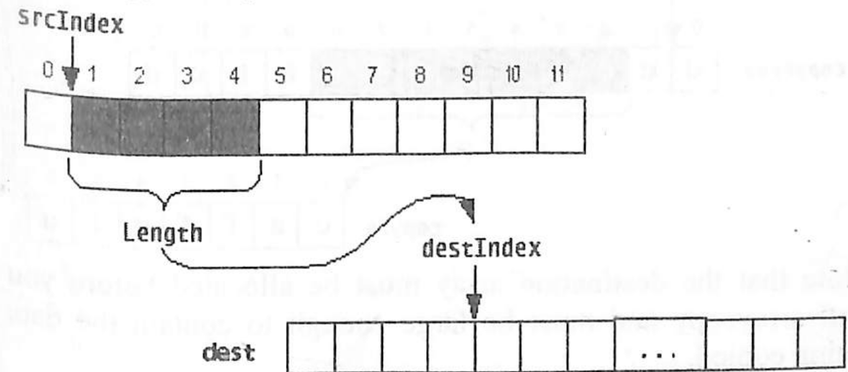
```
    }
}
```

You must specify the length of the primary array when you create the array. You can leave the length of the sub-arrays unspecified until you create them.

Copying Arrays

Use System's arraycopy method to efficiently copy data from one array into another. The arraycopy method requires five arguments: public static void arraycopy(Object source, int srcIndex, Object dest, int destIndex, int length)

The two Object arguments indicate the array to copy from and the array to copy to. The three integer arguments indicate the starting location ineach the source and the destination array, and the number of elements to copy. This diagram illustrates how the copy takes place:



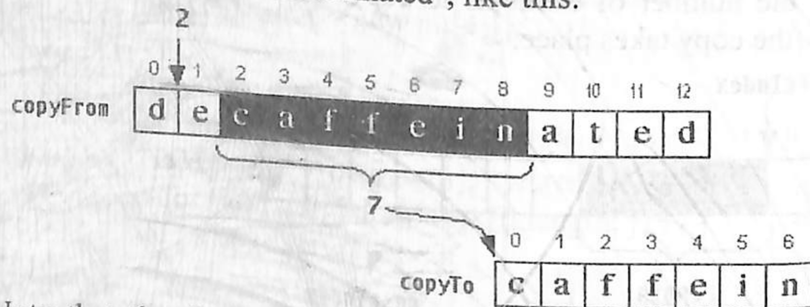
The following program, ArrayCopyDemo, uses arraycopy to copy some elemets from the copyFrom array to the copyTo array.

```
public class ArrayCopyDemo {
    public static void main(String[] args) {
```

```
        char[] copyFrom = { 'd', 'e', 'c', 'a', 'f', 'f', 'e',
                            'i', 'n', 'a', 't', 'e', 'd' };
        char[] copyTo = new char[7];

        System.arraycopy(copyFrom, 2, copyTo, 0, 7);
        System.out.println(new String(copyTo));
    }
}
```

The arraycopy method call in this example program begins the copy at element number2 in the source array. Recall that array indices start at 0, so that the copy begins at the array element 'c'. The arraycopy method call puts the copied elements into the destination array beginning at the first element (element 0) in the destination array copyTo. The copy copies 7 elements: 'c', 'a', 'f', 'f', 'e', 'i', and 'n'. Effectively, the arraycopy method takes the "caffein" out of "decaffeinated", like this:



Note that the destination array must be allocated before you call arraycopy and must be large enough to contain the data being copied.

Summary of Arrays

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

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

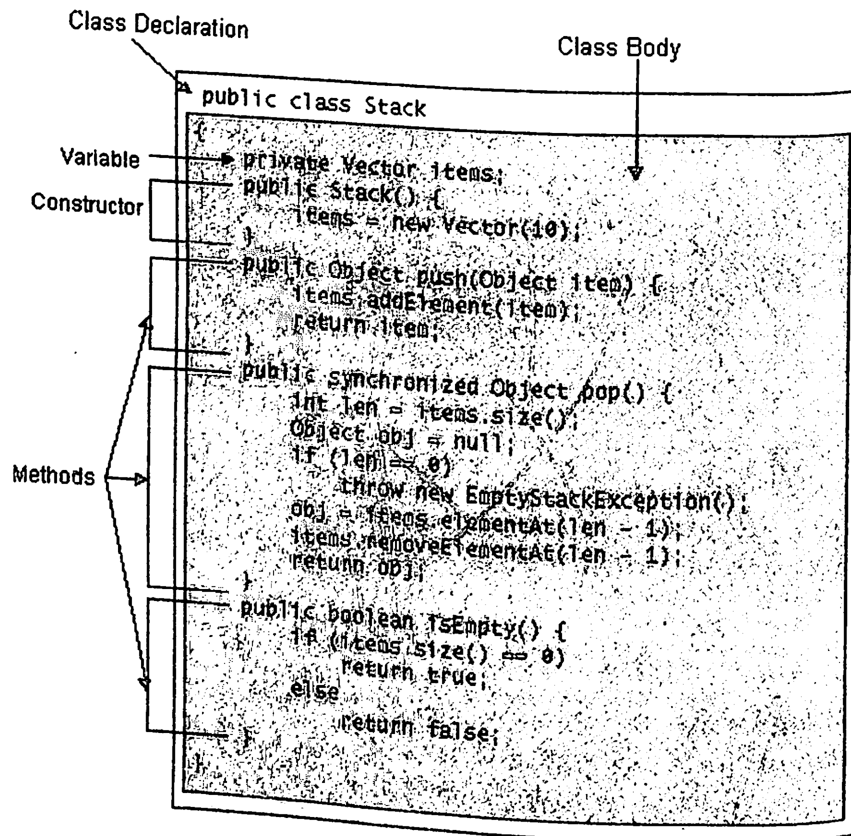
An element within an array can be accessed by its index. Indices begin at 0 and end at the length of the array minus 1.

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

Chapter3.Classes and Inheritance

3.1.Creating Classes

Now that we've covered how to create and use objects, and how objects are cleaned up, it's time to show you how to write the classes from which objects are created. This section shows you the main components of a class through a small example that implements a last-in-first-out (LIFO) stack. The following diagram lists the class and identifies the structure of the code.



This implementation of a stack uses another object, a Vector, to store its elements. Vector is a growable array of objects and does a nice job of allocating space for new objects as space is required. The Stack class makes use of this code by using a Vector to store its elements. However, it imposes LIFO restrictions on the Vector-- that is, you can only add elements to and remove elements from the top of the stack.

The Class Declaration

The left side of the following diagram shows the possible components of a class declaration in the order they should or must appear in your class declaration. The right side describes their purposes. The required components are the class keyword and the class name and are shown in bold. All the other components are optional, and each appears on a line by itself (thus "extends Super" is a single component). Italics indicates an identifier such as the name of a class or interface. If you do not explicitly declare the optional items, the Java compiler assumes certain defaults: a nonpublic, nonabstract, nonfinal subclass of Object that implements no interfaces.

Component	Purpose
public	Class is publicly accessible.
abstract	Class cannot be instantiated.
final	Class cannot be subclassed.
class ***NameOfClass***	***Name of the Class.***
extends *Super*	Superclass of the class.
implements *Interfaces*	Interfaces implemented by the class.
{ *ClassBody* }	

The following list provides a few more details about each class declaration component. It also provides references to sections later in this trail that talk about what each component means,

how to use each, and how it affects your class, other classes, and your Java program.

public

By default, a class can be used only by other classes in the same package. The public modifier declares that the class can be used by any class regardless of its package. Look in Creating and Using Packages for information about how to use modifiers to limit access to your classes and how it affects your access to other classes.

abstract

Declares that the class cannot be instantiated. For a discussion about when abstract classes are appropriate and how to write them, see Writing Abstract Classes and Methods.

final

Declares that the class cannot be subclassed. Writing Final Classes and Methods shows you how to use final and discusses the reasons for using it.

class NameOfClass

The class keyword indicates to the compiler that this is a class declaration and that the name of the class is NameOfClass.

extends Super

The extends clause identifies Super as the superclass of the class, thereby inserting the class within the class hierarchy.

The Class Body

The class body contains all of the code that provides for the life cycle of the objects created from it: constructors for initializing new objects, declarations for the variables that provide the state of the class and its objects, methods to implement the behavior of the class and its objects, and in rare cases, a finalize method to provide for cleaning up an object after it has done its job. Variables and methods collectively are called members.

Note: Constructors are not methods. Nor are they members.

The Stack class defines one member variable in its body to contain its elements--the items Vector. It also defines one constructor--a no-argument constructor--and three methods: push, pop, and isEmpty.

Providing Constructors for Your Classes

All Java classes have constructors that are used to initialize a new object of that type. A constructor has the same name as the class. For example, the name of the Stack class's constructor is Stack, the name of the Rectangle class's constructor is Rectangle, and the name of the Thread class's constructor is Thread. Stack defines a single constructor:

```
public Stack() {
   items = new Vector(10);
}
```

Java supports name overloading for constructors so that a class can have any number of constructors, all of which have the same name. Following is another constructor that could be defined by Stack. This particular constructor sets the initial size of the stack according to its parameter:

```
public Stack(int initialSize) {
   items = new Vector(initialSize);
}
```

Both constructors share the same name, Stack, but they have different parameter lists. The compiler differentiates these constructors based on the number of parameters in the list and their types.

Typically, a constructor uses its arguments to initialize the new object's state. When creating an object, choose the constructor whose arguments best reflect how you want to initialize the new object.

Based on the number and type of the arguments that you pass into the constructor, the compiler can determine which

constructor to use. The compiler knows that when you write the following code, it should use the constructor that requires a single integer argument:

```
new Stack(10);
```

Similarly, when you write the following code, the compiler chooses the no-argument constructor or the default constructor:

```
new Stack();
```

When writing your own class, you don't have to provide constructors for it. The default constructor is automatically provided by the runtime system for any class that contains no constructors. The default provided by the runtime system doesn't do anything. So, if you want to perform some initialization, you will have to write some constructors for your class.

The constructor for the following subclass of Thread performs animation, sets up some default values, such as the frame speed and the number of images, and then loads the images:

```
class AnimationThread extends Thread {
   int framesPerSecond;
   int numImages;
   Image[] images;

   AnimationThread(int fps, int num) {

      super("AnimationThread");
      this.framesPerSecond = fps;
      this.numImages = num;

      this.images = new Image[numImages];
      for (int i = 0; i <= numImages; i++) {
         . . .
         // Load all the images.
         . . .
      }
   }
            . . .
}
```

Note how the body of a constructor is like the body of a method; that is, it contains local variable declarations, loops, and other statements. However, one line in the AnimationThread constructor that you wouldn't see in a method is the second line:

```
super("AnimationThread");
```

This line invokes a constructor provided by the superclass of AnimationThread, namely, Thread. This particular Thread constructor takes a String that sets the name of Thread. Often a constructor wants to take advantage of initialization code written in a class's superclass. Indeed, some classes must call their superclass constructor in order for the object to work properly. If present, the superclass constructor must be the first statement in the subclass's constructor: An object should perform the higher-level initialization first.
You can specify what other objects can create instances of your class by using an access specifier in the constructors' declaration:

private
No other class can instantiate your class. Your class may contain public class methods (sometimes called factory methods), and those methods can construct an object and return it, but no other classes can.

protected
Only subclasses of the class and classes in the same package can create instances of it.
public
Any class can create an instance of your class.
no specifier gives package access
Only classes within the same package as your class can construct an instance of it.
Constructors provide a way to initialize a new object. Initializing Instance and Class Members describes other ways you can provide for the initialization of your class and a new object created from the class. That section also discusses when and why you would use each technique.

Declaring Member Variables
Stack uses the following line of code to define its single member variable:

```
private Vector items;
```

This declares a member variable and not some other type of variable (like a local variable) because the declaration appears within the class body but outside of any methods or constructors. The member variable declared is named items, and its data type is Vector. Also, the private keyword identifies items as a private member. This means that only code within the Stack class can access it.

accessLevel	Indicates the access level for this member.
static	Declares a class member.
final	Indicates that it is constant.
transient	This variable is transient.
volatile	This variable is volatile.
type name	The type and name of the variable.

This is a relatively simple member variable declaration, but declarations can be more complex. You can specify not only type, name, and access level, but also other attributes, including whether the variable is a class or instance variable and whether it's a constant. Each component of a member variable declaration is further defined below:

accessLevel
Lets you control which other classes have access to a member variable by using one of four access levels: public, protected, package, and private. You control access to methods in the same way.
static
Declares this is a class variable rather than an instance variable. You also use static to declare class methods.
final
Indicates that the value of this member cannot change. The following variable declaration defines a constant named AVOGADRO, whose value is Avogadro's number (6.022 * 1023) and cannot be changed:

```
final double AVOGADRO = 6.022e23;
```

It's a compile-time error if your program ever tries to change a final variable. By convention, the name of constant values are spelled in uppercase letters.
transient
The transient marker is not fully specified by The Java Language Specification but is used in object serialization which is covered in Object Serialization to mark member variables that should not be serialized.
volatile
The volatile keyword is used to prevent the compiler from performing certain optimizations on a member. This is an advanced Java feature, used by only a few Java programmers, and is outside the scope of this tutorial.
type

Like other variables, a member variable must have a type. You can use primitive type names such as int, float, or boolean. Or you can use reference types, such as array, object, or interface names.

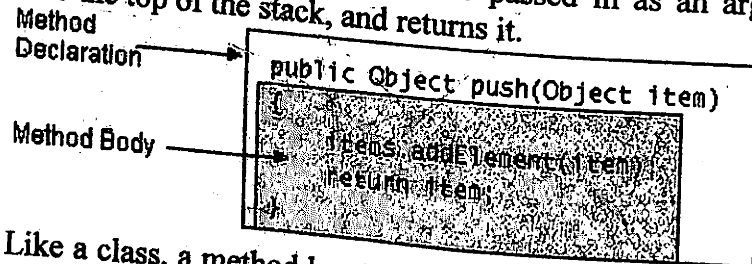
name

A member variable's name can be any legal Java identifier and, by convention, begins with a lowercase letter. You cannot declare more than one member variable with the same name in the same class, but a subclass can hide a member variable of the same name in its superclass. Additionally, a member variable and a method can have the same name. For example, the following code is legal:

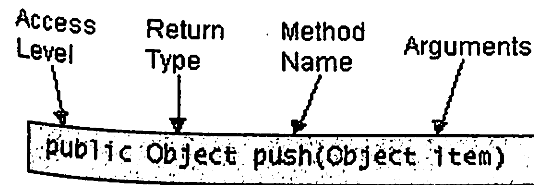
```
public class Stack {
    private Vector items;
    // a method with same name as a member variable
    public Vector items() {
        . . .
    }
}
```

Implementing Methods

This figure shows the code for Stack's push method. This method pushes an object, the one passed in as an argument, onto the top of the stack, and returns it.



Like a class, a method has two major parts: method declaration and method body. The method declaration defines all of the method's attributes, such as access level, return type, name, and arguments, as illustrated here:



The method body is where all the action takes place. It contains the Java instructions that implement the method.

Details of a Method Declaration

A method's declaration provides a lot of information about the method to the compiler, to the runtime system, and to other classes and objects. Included is not only the name of the method, but also such information as the return type of the method, the number and type of the arguments required by the method, and which other classes and objects can call the method.

While this may sound like writing a novel rather than simply declaring a method, most method attributes can be declared implicitly. The only required elements of a method declaration are the method's name, its return type, and a pair of parentheses ( ). This figure shows the elements of a method declaration.

`accessLevel`	Access level for this method.
`static`	This is a class method.
`abstract`	This method is not implemented.
`final`	Method cannot be overridden.
`native`	Method implemented in another language.
`synchronized`	Method requires a monitor to run.
`returnType methodName`	The return type and method name.
`( paramlist )`	The list of arguments.
`throws exceptions`	The exceptions thrown by this method.

Each element of a method declaration is further defined below:

accessLevel

As with member variables, you control which other classes have access to a method using one of four access levels: public, protected, package, and private. Controlling Access to Members of a Class covers access levels in detail.

static

As with member variables, static declares this method as a class method rather than an instance method. Understanding Instance and Class Members talks about declaring instance and class methods.

abstract

An abstract method has no implementation and must be a member of an abstract class. Refer to Writing Abstract Classes and Methods for information about why you might want to write an abstract method and how such methods affect subclasses.

final

A final method cannot be overridden by subclasses. Writing Final Classes and Methods discusses why you might want to write final methods, how they affect subclasses, and whether you might want to write a final class instead.

native

If you have a significant library of functions written in another language such as C, you may wish to preserve that investment and use those functions from Java. Methods implemented in a language other than Java are called native methods and are declared as such using the native keyword. Check out our Java Native Interface trail for information about writing native methods.

synchronized

Concurrently running threads often invoke methods that operate on the same data. These methods may be declared synchronized to ensure that the threads access information in a thread-safe manner. Synchronizing method calls is covered in Threads: Doing Two or More Tasks At Once. Take particular note of the section entitled Synchronizing Threads.

returnType

Java requires that a method declare the data type of the value that it returns. If your method does not return a value, use the keyword void for the return type. Returning a Value from a Method talks about the issues related to returning values from a method.

methodName

A method name can be any legal Java identifier. You need to consider several issues in regards to Java method names. These are covered in Method Names.

(paramlist)

You pass information into a method through its arguments. See the next section, Passing Information into a Method.

[throws exceptions]

If your method throws any checked exceptions, your method declaration must indicate the type of those exceptions. See Handling Errors with Exceptionsfor information. In particular, refer to Specifying the Exceptions Thrown by a Method.

Returning a Value from a Method

You declare a method's return type in its method declaration. Within the body of the method, you use the return operator to return the value. Any method that is not declared void must contain a return statement. The Stack class declares the isEmpty method, which returns a boolean:

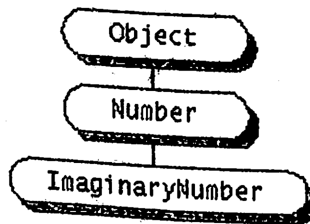
```
public boolean isEmpty() {
   if (items.size() == 0)
      return true;
   else
      return false;
}
```

The data type of the return value must match the method's return type; you can't return an Object type from a method

declared to return an integer. The isEmpty method returns either the boolean value true or false, depending on the outcome of a test. A compiler error results if you try to write a method in which the return value doesn't match the return type. The isEmpty method returns a primitive type. Methods also can return a reference type. For example, Stack declares the pop method that returns the Object reference type:

```
public synchronized Object pop() {
   int len = items.size();
   Object obj = null;
   if (len == 0)
      throw new EmptyStackException();
   obj = items.elementAt(len - 1);
   items.removeElementAt(len - 1);
   return obj;
}
```

When a method returns an object such as pop does, the class of the returned object must be either a subclass of or the exact class of the return type. This can be a source of confusion, so let's look at this more closely. Suppose you have a class hierarchy where ImaginaryNumber is a subclass of java.lang.Number, which is, in turn, a subclass of Object, as illustrated here:



Now suppose you have a method declared to return a Number:

```
public Number returnANumber() {
   . . .
}
```

The returnANumber method can return an ImaginaryNumber but not an Object. ImaginaryNumber "is a" Number because it's a subclass of Number. However, an Object is not necessarily a Number--it could be a String or some other type. You also can use interface names as return types. In this case, the object returned must implement the specified interface.

A Method's Name

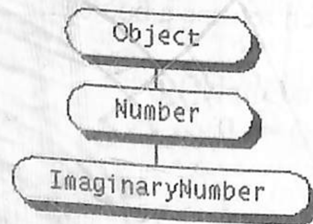
Java supports method name overloading so that multiple methods can share the same name. For example, suppose you are writing a class that can render various types of data (strings, integers, and so on) to its drawing area. You need to write a method that knows how to render each data type. In other languages, you have to think of a new name for each method, for example, drawString, drawInteger, drawFloat, and so on. In Java, you can use the same name for all of the drawing methods but pass a different type of parameter to each method. So, in your data rendering class, you can declare three methods named draw, each of which takes a different type of parameter:

```
class DataRenderer {
   void draw(String s) {
      . . .
   }
   void draw(int i) {
      . . .
   }
   void draw(float f) {
      . . .
   }
}
```

declared to return an integer. The isEmpty method returns either the boolean value true or false, depending on the outcome of a test. A compiler error results if you try to write a method in which the return value doesn't match the return type. The isEmpty method returns a primitive type. Methods also can return a reference type. For example, Stack declares the pop method that returns the Object reference type:

```
public synchronized Object pop() {
    int len = items.size();
    Object obj = null;
    if (len == 0)
        throw new EmptyStackException();
    obj = items.elementAt(len - 1);
    items.removeElementAt(len - 1);
    return obj;
}
```

When a method returns an object such as pop does, the class of the returned object must be either a subclass of or the exact class of the return type. This can be a source of confusion, so let's look at this more closely. Suppose you have a class hierarchy where ImaginaryNumber is a subclass of java.lang.Number, which is, in turn, a subclass of Object, as illustrated here:



Now suppose you have a method declared to return a Number:

```
public Number returnANumber() {
    . . .
}
```


The returnANumber method can return an ImaginaryNumber but not an Object. ImaginaryNumber "is a" Number because it's a subclass of Number. However, an Object is not necessarily a Number--it could be a String or some other type. You also can use interface names as return types. In this case, the object returned must implement the specified interface.

A Method's Name

Java supports method name overloading so that multiple methods can share the same name. For example, suppose you are writing a class that can render various types of data (strings, integers, and so on) to its drawing area. You need to write a method that knows how to render each data type. In other languages, you have to think of a new name for each method, for example, drawString, drawInteger, drawFloat, and so on. In Java, you can use the same name for all of the drawing methods but pass a different type of parameter to each method. So, in your data rendering class, you can declare three methods named draw, each of which takes a different type of parameter:

```
class DataRenderer {
    void draw(String s) {
        . . .
    }
    void draw(int i) {
        . . .
    }
    void draw(float f) {
        . . .
    }
}
```

Overloaded methods are differentiated by the number and type of the arguments passed into the method. In the code sample, draw(String s) and draw(int i) are distinct and unique methods because they require different argument types. You cannot declare more than one method with the same name and the same number and type of arguments because the compiler cannot differentiate them. So, draw(String s) and draw(String t) are identical and result in a compiler error.

A class may override a method in its superclass. The overriding method must have the same name, return type, and parameter list as the method it overrides. Overriding Methods shows you how to override methods.

Passing Information into a Method

When you write your method, you declare the number and type of the arguments required by that method. You declare the type and name for each argument in the method signature. For example, the following is a method that computes the monthly payments for a home loan based on the amount of the loan, the interest rate, the length of the loan (the number of periods), and the future value of the loan (presumably the future value of the loan is zero because at the end of the loan, you've paid it off):

```
double computePayment(double loanAmt, double rate,
                      double futureValue,
                      int numPeriods) {
    double I, partial1, denominator, answer;

    I = rate / 100.0;
    partial1 = Math.pow((1 + I), (0.0 - numPeriods));
    denominator = (1 - partial1) / I;
    answer = ((-1 * loanAmt) / denominator)
            - ((futureValue * partial1) / denominator);
    return answer;
}
```

This method takes four arguments: the loan amount, the interest rate, the future value and the number of periods. The first three are double-precision floating point numbers, and the fourth is an integer.

As with this method, the set of arguments to any method is a comma-separated list of variable declarations where each variable declaration is a type/name pair:

```
type name
```

As you can see from the body of the computePayment method, you simply use the argument name to refer to the argument's value.

Argument Types

In Java, you can pass an argument of any valid Java data type into a method. This includes primitive data types such as doubles, floats and integers as you saw in the computePayment method, and reference data types such as objects and arrays. Here's an example of a factory method that accepts an array as an argument. In this example, the method creates a new Polygon object and initializes it from a list of Points (Point is a class that represents an x, y coordinate):

```
static Polygon polygonFrom(Point[] listOfPoints) {
    . . .
}
```

Unlike some other languages, you cannot pass methods into Java methods. But you can pass an object into a method and then invoke the object's methods.

Argument Names

When you declare an argument to a Java method, you provide a name for that argument. This name is used within the method body to refer to the item.

A method argument can have the same name as one of the class's member variables. If this is the case, then the argument is said to hide the member variable. Arguments that hide member variables are often used in constructors to initialize a class. For example, take the following Circle class and its constructor:

```
class Circle {
   int x, y, radius;
   public Circle(int x, int y, int radius) {
      . . .
   }
}
```

The Circle class has three member variables: x, y and radius. In addition, the constructor for the Circle class accepts three arguments each of which shares its name with the member variable for which the argument provides an initial value.

The argument names hide the member variables. So using x, y or radius within the body of the constructor refers to the argument, not to the member variable. To access the member variable, you must reference it through this--the current object:

```
class Circle {
   int x, y, radius;
   public Circle(int x, int y, int radius) {
      this.x = x;
      this.y = y;
      this.radius = radius;
   }
}
```

Names of method arguments cannot be the same as another argument name for the same method, the name of any variable local to the method, or the name of any parameter to a catch clause within the same method.

Pass by Value

In Java methods, arguments are passed by value. When invoked, the method receives the value of the variable passed in. When the argument is of primitive type, pass-by-value means that the method cannot change its value.

When the argument is of reference type, pass-by-value means that the method cannot change the object reference, but can invoke the object's methods and modify the accessible variables within the object.

This is often the source of confusion--a programmer writes a method that attempts to modify the value of one its arguments and the method doesn't work as expected. Let's look at such method and then investigate how to change it so that it does what the programmer originally intended.

Consider this series of Java statements which attempts to retrieve the current color of a Pen object in a graphics application:

```
. . .
int r = -1, g = -1, b = -1;
pen.getRGBColor(r, g, b);
System.out.println("red = " + r +
                   ", green = " + g +
                   ", blue = " + b);
. . .
```

At the time when the getRGBColor method is called, the variables r, g, and b all have the value -1. The caller is

expecting the getRGBColor method to pass back the red, green and blue values of the current color in the r, g, and b variables. However, the Java runtime passes the variables' values (-1) into the getRGBColor method; not a reference to the r, g, and b variables. So you could visualize the call to getRGBColor like this: getRGBColor(-1, -1, -1).
When control passes into the getRGBColor method, the arguments come into scope (get allocated) and are initialized to the value passed into the method:

```
class Pen {
   int redValue, greenValue, blueValue;
   void getRGBColor(int red, int green, int blue) {
      // red, green, and blue have been created
      // and their values are -1
      . . .
   }
}
```

So getRGBColor gets access to the values of r, g, and b in the caller through its arguments red, green, and blue, respectively. The method gets its own copy of the values to use within the scope of the method. Any changes made to those local copies are not reflected in the original variables from the caller.
Now, let's look at the implementation of getRGBColor within the Pen class that the method signature above implies:

```
class Pen {
   int redValue, greenValue, blueValue;
   . . .
      // this method does not work as intended
   void getRGBColor(int red, int green, int blue) {
      red = redValue;
      green = greenValue;
      blue = blueValue;
   }
}
```

This method will not work as intended. When control gets to the println statement in the following code, which was shown previously, getRGBColor's arguments, red, green, and blue, no longer exist. Therefore the assignments made to them within the method had no effect; r, g, and b are all still equal to -1.

```
. . .
int r = -1, g = -1, b = -1;
pen.getRGBColor(r, g, b);
System.out.println("red = " + r +
                   ", green = " + g +
                   ", blue = " + b);
. . .
```

Passing variables by value affords the programmer some safety: Methods cannot unintentionally modify a variable that is outside of its scope. However, you often want a method to be able to modify one or more of its arguments.
The getRGBColor method is a case in point. The caller wants the method to return three values through its arguments. However, the method cannot modify its arguments, and, furthermore, a method can only return one value through its return value. So, how can a method return more than one value, or have an effect (modify some value) outside of its scope?
For a method to modify an argument, it must be of a reference type such as an object or array. Objects and arrays are also passed by value, but the value of an object is a reference. So the effect is that arguments of reference types are passed in by reference. Hence the name. A reference to an object is the address of the object in memory. Now, the argument in the method is referring to the same memory location as the caller.

Let's rewrite the getRGBColor method so that it actually does what you want. First, you must introduce a new type of object, RGBColor, that can hold the red, green and blue values of a color in RGB space:

```
class RGBColor {
   public int red, green, blue;
}
```

Now, we can rewrite getRGBColor so that it accepts an RGBColor object as an argument. The getRGBColor method returns the current color of the pen by setting the red, green and blue member variables of its RGBColor argument:

```
class Pen {
   int redValue, greenValue, blueValue;
   void getRGBColor(RGBColor aColor) {
      aColor.red = redValue;
      aColor.green = greenValue;
      aColor.blue = blueValue;
   }
}
```

And finally, let's rewrite the calling sequence:

```
...
RGBColor penColor = new RGBColor();
pen.getRGBColor(penColor);
System.out.println("red = " + penColor.red +
                 ", green = " + penColor.green +
                 ", blue = " + penColor.blue);
...
```

The modifications made to the RGBColor object within the getRGBColor method affect the object created in the calling sequence because the names penColor (in the calling sequence) and aColor (in the getRGBColor method) refer to the same object.

The Method Body
In the code sample that follows, the method bodies for the isEmpty and pop methods are shown in bold:

```
class Stack {
   static final int STACK_EMPTY = -1;
   Object[] stackelements;
   int topelement = STACK_EMPTY;
   ...
   boolean isEmpty() {
      if (topelement == STACK_EMPTY)
         return true;
      else
         return false;
   }
   Object pop() {
      if (topelement == STACK_EMPTY)
         return null;
      else {
         return stackelements[topelement--];
      }
   }
}
```

Besides regular Java language elements, you can use this in the method body to refer to members in the current object. The current object is the object whose method is being called. You can also use super to refer to members in the superclass that the current object has hidden or overridden. Also, a method body may contain declarations for variables that are local to that method.

this

Typically, within an object's method body you can just refer directly to the object's member variables. However, sometimes you need to disambiguate the member variable name if one of the arguments to the method has the same name.
For example, the following constructor for the HSBColor class initializes some of an object's member variables according to the arguments passed into the constructor. Each argument to the constructor has the same name as the object's member variable whose initial value the argument contains.

```
class HSBColor {
   int hue, saturation, brightness;
   HSBColor (int hue, int saturation, int brightness) {
      this.hue = hue;
      this.saturation = saturation;
      this.brightness = brightness;
   }
```

You must use the this keyword in this constructor because you have to disambiguate the argument hue from the member variable hue (and so on with the other arguments). Writing hue = hue; makes no sense. Argument names take precedence and hide member variables with the same name. So to refer to the member variable you must do so through the current object--using the this keyword to refer to the current object--explicitly.
Some programmers prefer to always use the this keywrod when referring to a member variable of the object whose method the reference appears. Doing so makes the intent of the code explicit and reduces errors based on name sharing.
You can also use the this keyword to call one of the current object's methods. Again this is only necessary if there is some ambiguity in the method name and is often used to make the intent of the code clearer.

super

If your method hides one of its superclass's member variables, your method can refer to the hidden variable through the use of the super keyword. Similarly, if your method overrides one of its superclass's methods, your method can invoke the overridden method through the use of the super keyword.
Consider this class:

```
class ASillyClass {
   boolean aVariable;
   void aMethod() {
      aVariable = true;
   }
}
```

and its subclass which hides aVariable and overrides aMethod:

```
class ASillierClass extends ASillyClass {
   boolean aVariable;
   void aMethod() {
      aVariable = false;
      super.aMethod();
      System.out.println(aVariable);
      System.out.println(super.aVariable);
   }
}
```

First aMethod sets aVariable (the one declared in ASillierClass that hides the one declared in ASillyClass) to false. Next aMethod invoked its overridden method with this statement:

```
super.aMethod();
```

This sets the hidden version of the aVariable (the one declared in ASillyClass) to true. Then aMethod displays both versions of aVariable which have different values:

```
false
true
```

Local Variables

Within the body of the method you can declare more variables for use within that method. These variables are local variables and live only while control remains within the method. This method declares a local variable i that it uses to iterate over the elements of its array argument.

```
    Object findObjectInArray(Object o, Object[]
arrayOfObjects) {
        int i;      // local variable
        for (i = 0; i < arrayOfObjects.length; i++) {
            if (arrayOfObjects[i] == o)
                return o;
        }
        return null;
    }
```

After this method returns, i no longer exists.

3.2.Controlling Access to Members of a Class

One of the benefits of classes is that classes can protect their member variables and methods from access by other objects. Why is this important? Well, consider this. You're writing a class that represents a query on a database that contains all kinds of secret information, say employee records or income statements for your startup company.

Certain information and queries contained in the class, the ones supported by the publicly accessible methods and variables in your query object, are OK for the consumption of any other object in the system. Other queries contained in the class are there simply for the personal use of the class. They support the operation of the class but should not be used by objects of another type--you've got secret information to protect. You'd like to be able to protect these personal variables and methods at the language level and disallow access by objects of another type.

In Java, you can use access specifiers to protect both a class's variables and its methods when you declare them. The Java language supports four distinct access levels for member variables and methods: private, protected, public, and, if left unspecified, package.

The following chart shows the access level permitted by each specifier.

Specifier	**class**	**subclass**	**package**	**world**
private	X			
protected	X	X*	X	
public	X	X	X	X
package	X		X	

The first column indicates whether the class itself has access to the member defined by the access specifier. As you can see, a class always has access to its own members. The second column indicates whether subclasses of the class (regardless of which package they are in) have access to the member. The third column indicates whether classes in the same package as the class (regardless of their parentage) have access to the member. The fourth column indicates whether all classes have access to the member.

Let's look at each access level in more detail.

Private

The most restrictive access level is private. A private member is accessible only to the class in which it is defined.

Use this access to declare members that should only be used by the class. This includes variables that contain information that if accessed by an outsider could put the object in an inconsistent state, or methods that, if invoked by an outsider, could jeopardize the state of the object or the program in which it's running. Private members are like secrets you never tell anybody.

To declare a private member, use the private keyword in its declaration. The following class contains one private member variable and one private method:

```
class Alpha {
   private int iamprivate;
   private void privateMethod() {
      System.out.println("privateMethod");
   }
}
```

Objects of type Alpha can inspect or modify the iamprivate variable and can invoke privateMethod, but objects of other types cannot. For example, the Beta class defined here:

```
class Beta {
   void accessMethod() {
      Alpha a = new Alpha();
      a.iamprivate = 10;     // illegal
      a.privateMethod();     // illegal
   }
}
```

cannot access the iamprivate variable or invoke privateMethod on an object of type Alpha because Beta is not of type Alpha.

When one of your classes is attempting to access a member varible to which it does not have access, the compiler prints an error message similar to the following and refuses to compile your program:

```
Beta.java:9: Variable iamprivate in class Alpha not
accessible from class Beta.
      a.iamprivate = 10;     // illegal
      ^
1 error
```

Also, if your program is attempting to access a method to which it does not have access, you will see a compiler error like this:

```
Beta.java:12: No method matching privateMethod()
found in class Alpha.
      a.privateMethod();       // illegal
1 error
```

New Java programmers might ask if one Alpha object can access the private members of another Alpha object. This is illustrated by the following example. Suppose the Alpha class contained an instance method that compared the current Alpha object (this) to another object based on their iamprivate variables:

```
class Alpha {
   private int iamprivate;
   boolean isEqualTo(Alpha anotherAlpha) {
      if (this.iamprivate == anotherAlpha.iamprivate)
         return true;
      else
         return false;
   }
```

```
}
```

This is perfectly legal. Objects of the same type have access to one another's private members. This is because access restrictions apply at the class or type level (all instances of a class) rather than at the object level (this particular instance of a class).

Protected

The next access level specifier is protected, which allows the class itself, subclasses (with the caveat that we referred to earlier), and all classes in the same package to access the members. Use the protected access level when it's appropriate for a class's subclasses to have access to the member, but not unrelated classes. Protected members are like family secrets--you don't mind if the whole family knows, and even a few trusted friends but you wouldn't want any outsiders to know. To declare a protected member, use the keyword protected. First, let's look at how the protected specifier affects access for classes in the same package. Consider this version of the Alpha class which is now declared to be within a package named Greek and which has one protected member variable and one protected method declared in it:

```
package Greek;

public class Alpha {
   protected int iamprotected;
   protected void protectedMethod() {
      System.out.println("protectedMethod");
   }
}
```

Now, suppose that the class Gamma was also declared to be a member of the Greek package (and is not a subclass of Alpha). The Gamma class can legally access an Alpha object's iamprotected member variable and can legally invoke its protectedMethod:

```
package Greek;

class Gamma {
   void accessMethod() {
      Alpha a = new Alpha();
      a.iamprotected = 10;   // legal
      a.protectedMethod();   // legal
   }
}
```

That's pretty straightforward. Now, let's investigate how the protected specifier affects access for subclasses of Alpha. Let's introduce a new class, Delta, that derives from Alpha but lives in a different package--Latin. The Delta class can access both iamprotected and protectedMethod, but only on objects of type Delta or its subclasses. The Delta class cannot access iamprotected or protectedMethod on objects of type Alpha. accessMethod in the following code sample attempts to access the iamprotected member variable on an object of type Alpha, which is illegal, and on an object of type Delta, which is legal. Similarly, accessMethod attempts to invoke an Alpha object's protectedMethod which is also illegal:

```
package Latin;

import Greek.*;

class Delta extends Alpha {
   void accessMethod(Alpha a, Delta d) {
      a.iamprotected = 10;   // illegal
      d.iamprotected = 10;   // legal
```

```
        a.protectedMethod();   // illegal
        d.protectedMethod();   // legal
    }
}
```

If a class is both a subclass of and in the same package as the class with the protected member, then the class has access to the protected member.

Public

The easiest access specifier is public. Any class, in any package, has access to a class's public members. Declare public members only if such access cannot produce undesirable results if an outsider uses them. There are no personal or family secrets here; this is for stuff you don't mind anybody else knowing.

To declare a public member, use the keyword public. For example,

```
package Greek;

public class Alpha {
    public int iampublic;
    public void publicMethod() {
        System.out.println("publicMethod");
    }
}
```

Let's rewrite our Beta class one more time and put it in a different package than Alpha and make sure that it is completely unrelated to (not a subclass of) Alpha:

```
package Roman;

import Greek.*;
```

```
class Beta {
    void accessMethod() {
        Alpha a = new Alpha();
        a.iampublic = 10;       // legal
        a.publicMethod();       // legal
    }
}
```

As you can see from the above code snippet, Beta can legally inspect and modify the iampublic variable in the Alpha class and can legally invoke publicMethod.

Package

The package access level is what you get if you don't explicitly set a member's access to one of the other levels.

This access level allows classes in the same package as your class to access the members. This level of access assumes that classes in the same package are trusted friends. This level of trust is like that which you extend to your closest friends but wouldn't trust even to your family.

For example, this version of the Alpha class declares a single package-access member variable and a single package-access method. Alpha lives in the Greek package:

```
package Greek;

class Alpha {
    int iampackage;
    void packageMethod() {
        System.out.println("packageMethod");
    }
}
```

The Alpha class has access both to iampackage and packageMethod. In addition, all the classes declared within the

same package as Alpha also have access to iampackage and packageMethod. Suppose that both Alpha and Beta were declared as part of the Greek package:

```
package Greek;

class Beta {
    void accessMethod() {
        Alpha a = new Alpha();
        a.iampackage = 10;     // legal
        a.packageMethod();     // legal
    }
}
```

Beta can legally access iampackage and packageMethod as shown.

3.3.Understanding Instance and Class Members

When you declare a member variable such as aFloat in MyClass:

```
class MyClass {
    float aFloat;
}
```

you declare an instance variable. Every time you create an instance of a class, the runtime system creates one copy of each the class's instance variables for the instance.
Instance variables are in contrast to class variables (which you declare using the static modifier). The runtime system allocates class variables once per class regardless of the number of instances created of that class. The system allocates memory for class variables the first time it encounters the class. All instances share the same copy of the class's class variables.

You can access class variables through an instance or through the class itself.
Methods are similar: Your classes can have instance methods and class methods. Instance methods operate on the current object's instance variables but also have access to the class variables. Class methods, on the other hand, cannot access the instance variables declared within the class (unless they create a new object and access them through the object). Also, class methods can be invoked on the class, you don't need an instance to call a class method.
By default, unless otherwise specified, a member declared within a class is an instance member. The class defined below has one instance variable--an integer named x--and two instance methods--x and setX--that let other objects set and query the value of x:

```
class AnIntegerNamedX {
    int x;
    public int x() {
        return x;
    }
    public void setX(int newX) {
        x = newX;
    }
}
```

Every time you instantiate a new object from a class, you get a new copy of each of the class's instance variables. These copies are associated with the new object. So, every time you instantiate a new AnIntegerNamedX object from the class, you get a new copy of x that is associated with the new AnIntegerNamedX object.
All instances of a class share the same implementation of an instance method; all instances of AnIntegerNamedX share the same implementation of x and setX. Note that both methods, x

and setX, refer to the object's instance variable x by name. "But", you ask, "if all instances of AnIntegerNamedX share the same implementation of x and setX isn't this ambiguous?" The answer is "no." Within an instance method, the name of an instance variable refers to the current object's instance variable, assuming that the instance variable isn't hidden by a method parameter. So, within x and setX, x is equivalent to this.x.
Objects outside of AnIntegerNamedX that wish to access x must do so through a particular instance of AnIntegerNamedX. Suppose that this code snippet was in another object's method. It creates two different objects of type AnIntegerNamedX, sets their x values to different values, then displays them:

```
...
AnIntegerNamedX myX = new AnIntegerNamedX();
AnIntegerNamedX anotherX = new
AnIntegerNamedX();
myX.setX(1);
anotherX.x = 2;
System.out.println("myX.x = " + myX.x());
System.out.println("anotherX.x = " + anotherX.x());
...
```

Notice that the code used setX to set the x value for myX but just assigned a value to anotherX.x directly. Either way, the code is manipulating two different copies of x: the one contained in the myX object and the one contained in the anotherX object. The output produced by this code snippet is:

```
myX.x = 1
anotherX.x = 2
```

showing that each instance of the class AnIntegerNamedX has its own copy of the instance variable x and each x has a different value.
You can, when declaring a member variable, specify that the variable is a class variable rather than an instance variable.

Similarly, you can specify that a method is a class method rather than an instance method. The system creates a single copy of a class variable the first time it encounters the class in which the variable is defined. All instances of that class share the same copy of the class variable. Class methods can only operate on class variables--they cannot access the instance variables defined in the class.
To specify that a member variable is a class variable, use the static keyword. For example, let's change the AnIntegerNamedX class such that its x variable is now a class variable:

```
class AnIntegerNamedX {
    static int x;
    public int x() {
        return x;
    }
    public void setX(int newX) {
        x = newX;
    }
}
```

Now the exact same code snippet from before that creates two instances of AnIntegerNamedX, sets their x values, and then displays them produces this, different, output.

```
myX.x = 2
anotherX.x = 2
```

The output is different because x is now a class variable so there is only one copy of the variable and it is shared by all instances of AnIntegerNamedX, including myX and anotherX. When you invoke setX on either instance, you change the value of x for all instances of AnIntegerNamedX.

You use class variables for items that you need only one copy of and which must be accessible by all objects inheriting from the class in which the variable is declared. For example, class variables are often used with final to define constants; this is more memory efficient than final instance variables because constants can't change, so you really only need one copy).
Similarly, when declaring a method, you can specify that method to be a class method rather than an instance method. Class methods can only operate on class variables and cannot access the instance variables defined in the class.
To specify that a method is a class method, use the static keyword in the method declaration. Let's change the AnIntegerNamedX class such that its member variable x is once again an instance variable, and its two methods are now class methods:

```
class AnIntegerNamedX {
   int x;
   static public int x() {
      return x;
   }
   static public void setX(int newX) {
      x = newX;
   }
}
```

When you try to compile this version of AnIntegerNamedX, the compiler displays an error like this one:

```
AnIntegerNamedX.java:4: Can't make a static reference
to
nonstatic variable x in class AnIntegerNamedX.
      return x;
         ^
```


This is because class methods cannot access instance variables unless the method created an instance of AnIntegerNamedX first and accessed the variable through it.
Let's fix AnIntegerNamedX by making its x variable a class variable:

```
class AnIntegerNamedX {
   static int x;
   static public int x() {
      return x;
   }
   static public void setX(int newX) {
      x = newX;
   }
}
```

Now the class will compile and the same code snippet from before that creates two instances of
AnIntegerNamedX, sets their x values, and then prints the x values produces this output:

```
myX.x = 2
anotherX.x = 2
```

Again, changing x through myX also changes it for other instances of AnIntegerNamedX.
Another difference between instance members and class members is that class members are accessible from the class itself. You don't need to instantiate a class to access its class members. Let's rewrite the code snippet from before to access x and setX directly from the AnIntegerNamedX class:

```
. . .
AnIntegerNamedX.setX(1);
```

```
    System.out.println("AnIntegerNamedX.x = " +
AnIntegerNamedX.x());
    . . .
```

Notice that you no longer have to create myX and anotherX. You can set x and retrieve x directly from the AnIntegerNamedX class. You cannot do this with instance members, you can only invoke instance methods from an object and can only access instance variables from an object. You can access class variables and methods either from an instance of the class or from the class itself.

Initializing Instance and Class Members

You can use static initializers and instance initializers to provide initial values for class and instance members when you declare them in a class:

```
    class BedAndBreakfast {
        static final int MAX_CAPACITY = 10;
        boolean full = false;
    }
```

This works well for members of primitive data type. Sometimes, it even works when creating arrays and objects. But this form of initialization has limitations, as follows:

1.Initializers can perform only initializations that can be expressed in an assignment statement.

2.Initializers cannot call any method that can throw a checked exception.

3.If the initializer calls a method that throws a runtime exception, then it cannot do error recovery.

If you have some initialization to perform that cannot be done in an initializer because of one of these limitations, you have to put the initialization code elsewhere. To initialize class members, put the initialization code in a static initialization

block. To initialize instance members, put the initialization code in a constructor.

Using Static Initialization Blocks

Here's an example of a static initialization block:

```
import java.util.ResourceBundle;
class Errors {
    static ResourceBundle errorStrings;
    static {
        try {
            errorStrings = ResourceBundle.
                        getBundle("ErrorStrings");
        } catch (java.util.MissingResourceException e) {
            // error recovery code here
        }
    }
}
```

The errorStrings resource bundle must be initialized in a static initialization block. This is because error recovery must be performed if the bundle cannot be found. Also, errorStrings is a class member and it doesn't make sense for it to be initialized in a constructor. As the previous example shows, a static initialization block begins with the static keyword and is a normal block of Java code enclosed in curly braces {}. A class can have any number of static initialization blocks that appear anywhere in the class body. The runtime system guarantees that static initialization blocks and static initializers are called in the order (left-to-right, top-to-bottom) that they appear in the source code.

Initializing Instance Members

If you want to initialize an instance variable and cannot do it in the variable declaration for the reasons cited previously, then put the initialization in the constructor(s) for the class. Suppose the errorStrings bundle in the previous example is an instance variable rather than a class variable. Then you'd use the following code to initialize it:

```
import java.util.ResourceBundle;
class Errors {
   ResourceBundle errorStrings;
   Errors() {
      try {
         errorStrings = ResourceBundle.
                        getBundle("ErrorStrings");
      } catch (java.util.MissingResourceException e) {
         // error recovery code here
      }
   }
}
```

The code that initializes errorStrings is now in a constructor for the class.

Sometimes a class contains many constructors and each constructor allows the caller to provide initial values for different instance variables of the new object. For example, java.awt.Rectangle has these three constructors:

```
Rectangle();
Rectangle(int width, int height);
Rectangle(int x, int y, int width, int height);
```

The no-argument constructor doesn't let the caller provide initial values for anything, and the other two constructors let the caller set initial values either for the size or for the origin and size. Yet, all of the instance variables, the origin and the size, for Rectangle must be initialized. In this case, classes often have one constructor that does all of the work. The other constructors call this constructor and provide it either with the values from their parameters or with default values. For example, here are the possible implementations of the three Rectangle constructors shown previously (assume x, y, width,

and height are the names of the instance variables to be initialized):

```
Rectangle() {
   this(0,0,0,0);
}
Rectangle(int width, int height) {
   this(0,0,width,height);
}
Rectangle(int x, int y, int width, int height) {
   this.x = x;
   this.y = y;
   this.width = width;
   this.height = height;
}
```

The Java language supports instance initialization blocks, which you could use instead. However, these are intended to be used with anonymous classes, which cannot declare constructors.

The approach described here that uses constructors is better for these reasons:

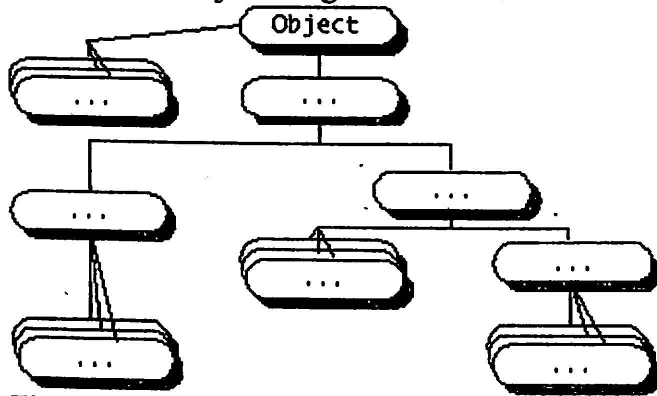
All of the initialization code is in one place, thus making the code easier to maintain and read.

Defaults are handled explicitly. Constructors are widely understood by the Java community, including relatively new Java programmers, while instance initializers are not and may cause confusion to others reading your code.

3.4.Managing Inheritance

Recall from the previous lesson that the extends clause declares that your class is a subclass of another. You can specify only one superclass for your class (Java does not support multiple class inheritance), and even though you can omit the extends clause from your class declaration, your class has a superclass.

So, every class in Java has one and only one immediate superclass. This statement leads to the question, "Where does it all begin?"
As depicted in the following figure, the top-most class, the class from which all other classes are derived, is the Object class defined in java.lang.



The Object class defines and implements behavior that every class in the Java system needs. It is the most general of all classes. Its immediate subclasses, and other classes near top of the hierarchy, implement general behavior; classes near the bottom of the hierarchy provide for more specialized behavior.
Definition: A subclass is a class that extends another class. A subclass inherits state and behavior from all of its ancestors. The term "superclass" refers to a class's direct ancestor as well as to all of its ascendant classes.

3.5.Implementing Nested Classes

Java lets you define a class as a member of another class. Such a class is called a nested class and is illustrated here:

```
class EnclosingClass{
   ...
   class ANestedClass {
      ...
   }
}
```

Definition: A nested class is a class that is a member of another class.

You use nested classes to reflect and enforce the relationship between two classes. You should define a class within another class when the nested class makes sense only in the context of its enclosing class or when it relies on the enclosing class for its function. For example, a text cursor makes sense only in the context of a particular text component.
As a member of its enclosing class, a nested class has a special privilege: It has unlimited access to its enclosing class's members, even if they are declared private. However, this special privilege isn't really special at all. It is fully consistent with the meaning of private and the other access specifiers. The access specifiers restrict access to members for classes outside of the enclosing class. The nested class is inside of its enclosing class so that it has access to its enclosing class's members.
Like other members, a nested class can be declared static (or not). A static nested class is called just that: a static nested class. A nonstatic nested class is called an inner class. These are illustrated in the following code:

```
class EnclosingClass{
   ...
   static class AStaticNestedClass {
      ...
   }
   class InnerClass {
      ...
   }
}
```

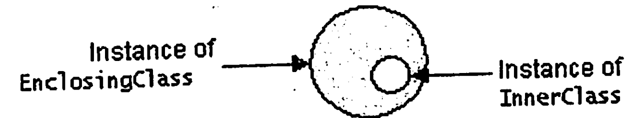
As with static methods and variables (normally called class methods and variables), a static nested class is associated with its enclosing class. And like class methods, a static nested class cannot refer directly to instance variables or methods defined in its enclosing class-it can use them only through an object reference.

As with instance methods and variables, an inner class is associated with an instance of its enclosing class and has direct access to that object's instance variables and methods. Also, because an inner class is associated with an instance, it cannot define any static members itself.

To help differentiate the terms nested class and inner class further, we suggest you think about them in the following way. The term "nested class" reflects the syntactic relationship between two classes; that is, syntactically, the code for one class appears within the code of another. In contrast, the term "inner class" reflects the relationship between instances of the two classes. Consider the following classes:

```
class EnclosingClass {
   . . .
   class InnerClass {
      . . .
   }
}
```

The interesting feature about the relationship between these two classes is not that InnerClass is syntactically defined within EnclosingClass. Rather, it's that an instance of InnerClass can exist only within an instance of EnclosingClass and that it has direct access to instance variables and methods of its enclosing instance. The following diagram illustrates this idea.



You may encounter nested classes of both kinds in the Java API and be required to use them. However, most nested classes that you write will be inner classes.

Definition: An inner class is a nested class whose instance exists within an instance of its enclosing class and has direct access to the instance members of its enclosing instance.

Other Facts about Nested Classes

Like other classes, nested classes can be declared abstract or final. The meaning of these two modifiers for nested classes is the same as for other classes. Also, the access specifiers-- private, public, protected, and package--- may be used to restrict access to nested classes just as they do to other class members.

Any nested class, not just anonymous ones, can be declared in any block of code. A nested class declared within a method or other smaller block of code has access to any final, local variables in scope.

Basic Java
Selected topics

(Part 1)

Тираж 100 экз. Объём 140 стр.
Формат 60x84/16. Бумага офсетная №1
Плотность 80 г/м^2. Печать RISO.
Подписано в печать 06.03.2001 г.

г. Алматы, ул. Байтурсынова, 22, оф. 9
Тел.: 8 (3272) 39-32-69, факс 32-38-43;
E-mail: evero@nursat.kz

Basic Java: Selected topics

30231

5091 K
30231