DECLARE A VARIABLE

A variable is a name that represents a value. For example, you could have the variable myAge represent the value 29. Variables can be used to perform many types of calculations. Before a variable can be used in a Java program, you must declare the variable. Declaring a variable tells the computer to set aside an area of memory to store information.

A variable can hold only a specific type of data, such as a text character or a number. When you declare a variable, you specify the type of data the variable can store. For example, to specify that a variable will hold only a whole number that is not a fraction, you would use an integer data type. To declare a variable that will hold an integer, place the keyword `int` before the variable name. For more information about variable data types, see page 30.

A variable name can consist of multiple words. You can use a lowercase first letter and then capitalize the first letter of each of the following words to make the name easy to read. The underscore character (_) can also be used to separate the words in the name, such as `my_age`.

When you declare a variable, you can assign an initial value to the variable. To assign a value to a variable, you use the *assignment operator* (=). For information about operators, see page 34.

If you have multiple variables of the same type, you can declare all the variables on the same line by separating each variable name with a comma.

Once a variable has been declared, it can be used within the method in which it was created. If the variable was created outside of a method, it can be used by any code within the class. Variables declared outside of a method should be declared at the top of the class body. For information about declaring variables in the class body, see page 56.

Extra

If you have not yet determined the value for a variable, you can create a variable without assigning a value. Java will assign a default value to the variable, but you can later assign a value in a separate statement.

Example:

```
int firstValue;
System.out.println("Welcome to my program.");
firstValue = 10;
```

When selecting a variable name, choose a name that describes the value of the variable. For example, `employeeNumber` is more descriptive than `variable1`. Also keep in mind that variable names are case sensitive. This means the variable `AGE` will be different than the variable `age` or `Age`.

Any method you use to determine variable names is acceptable if it makes sense to you and is easy for other people to interpret. You should consistently use the style you choose to make your script easier to understand.

Typing mistakes are a common source of errors in Java code. If the Java compiler displays error messages that refer to undeclared or missing variables, you should first check to make sure you typed each variable name the same way throughout your code.

If the name of a variable is not self-explanatory, you may want to add a comment to the variable declaration to explain the purpose of the variable.

Example:

```
int minutes;    //Minutes to display welcome message
```

DECLARE A VARIABLE

```
class DisplayVariables
{
     public static void main(String[ ] args)
     {
          int oneValue = 50;
     }
}
```

1 To declare a variable, type the keyword for the data type you want to use.

2 Type a name for the variable you want to create.

3 Type = followed by the value you want to assign to the variable.

DECLARE MULTIPLE VARIABLES

```
class DisplayVariables
{
     public static void main(String[ ] args)
     {
          int oneValue = 50;
          int firstValue = 10, secondValue = 20;
     }
}
```

4 To declare multiple variables, type the keyword for the data type of the variables.

5 Type the name and value of each variable you want to create, separated by a comma.

```
class DisplayVariables
{
     public static void main(String[ ] args)
     {
          int oneValue = 50;
          int firstValue = 10, secondValue = 20;

          System.out.println(oneValue);
          System.out.println(firstValue);
          System.out.println(secondValue);
     }
}
```

6 Type the code that uses the variables.

```
C:\WINDOWS>cd\java

C:\java>javac DisplayVariables.java

C:\java>java DisplayVariables
50
10
20

C:\java>
```

7 Compile the Java code and then execute the program.

■ The result of using the variables is displayed.

SPECIFY THE DATA TYPE FOR A VARIABLE

Java is a 'strongly typed language', which means that you must specify a data type for each variable you use in a Java program. This distinguishes Java from many other programming languages, such as Perl, which do not require variables to have assigned data types.

There are eight basic data types, called primitive types, that variables can use. The data type you specify for a variable determines the range of values that the variable can store and the amount of memory, measured in bits, that the variable requires. For example, a variable with the `byte` data type can store a number between -128 and 127 and requires 8 bits of memory.

Each primitive data type has a default value. If you declare a variable without assigning a value, the default value for the variable's data type will be assigned to the variable.

The specifications for data types in Java, such as memory requirements and default values, are not affected by the operating system or compiler that is used. This ensures that a data type will have the same meaning when a program is executed on different computers.

Specifying the data type for a variable requires that you know in advance the types of values that will be stored in the variable throughout your program. Once you declare a variable, you cannot change the data type for the variable. If you want to convert the value stored in a variable to a different data type, you must assign the value to a new variable that uses the desired data type. This process is called casting. When converting a value to a new data type, make sure that the conversion will not result in an unintended loss of data. For example, converting the number 13.56 to an integer value will result in a new value of 13.

Extra

Primitive Data Types

TYPE:	SIZE IN BITS:	DEFAULT VALUE:	POSSIBLE VALUES:
`boolean`	8	false	'true' or 'false'
`char`	16	\u0000	Unicode character, '\u0000' to 'uFFFF'
`byte`	8	0	-128 to 127
`short`	16	0	-32,768 to 32,767
`int`	32	0	-2,147,483,648 to 2,147,483,647
`long`	64	0	-9,223,372,036,854,775,808 to 9,223,372,036,854,775,807
`float`	32	0.0	±1.4E-45 to ±3.4028235E+38
`double`	64	0.0	±4.9E-324 to ±1.7976931348623157E+308

SPECIFY THE DATA TYPE FOR A VARIABLE

```
class DisplayVariables
{
     public static void main(String[] args)
          {
          double firstValue = 1.1;
          }
}
```

1 To specify a data type for a variable you want to create, type the name of the data type in the body of the method.

2 Type the code that names the variable and assigns it a value.

Note: If you do not assign a value, the variable will use the default value for its data type.

```
class DisplayVariables
{
     public static void main(String[] args)
          {
          double firstValue = 1.1;
          int secondValue;
          secondValue = (int) firstValue;
          }
}
```

CONVERT A VALUE TO A DIFFERENT DATA TYPE

3 Type the code that declares a variable that will store the converted value.

4 Type the name of the variable you created in step 3, followed by =.

5 Type the data type you want to convert the value to, enclosed in parentheses.

6 Type the name of the variable that stores the value you want to convert.

```
class DisplayVariables
{
     public static void main(String[] args)
          {
          double firstValue = 1.1;
          int secondValue;
          secondValue = (int) firstValue;
          System.out.println("The double value is " + firstValue);
          System.out.println("The integer value is " + secondValue);
          }
}
```

7 Type the code that uses the values of the variables you created.

■ In this example, we display the values of the `firstValue` and `secondValue` variables.

```
C:\WINDOWS>cd\java

C:\java>javac DisplayVariables.java

C:\java>java DisplayVariables
The double value is 1.1
The integer value is 1

C:\java>
```

8 Compile the Java code and then execute the program.

■ The results of using the variables are displayed.

WORK WITH STRINGS

A string is a collection of characters, which can contain any combination of letters, numbers and special characters, such as $, & or #.

Before a string variable can be used in a Java program, you must declare the string variable. The process of declaring a string variable is similar to that of declaring other types of variables. To declare a string variable, use the keyword `String` followed by the variable name. The capital S at the beginning of the keyword `String` indicates that a string variable is an object of the `String` class. The `String` class is part of the `java.lang` package that is available to all Java programs as part of the standard class library. For information about the Java standard class library, see page 8.

After a string variable has been declared, a value can be assigned to the variable. To assign a value to a string variable, you use the assignment operator (=). A string value must be enclosed in double quotation marks (""), which identify the beginning and end of the string and allow Java to work with the string as one piece of information.

You can use the concatenation operator (+) to join multiple strings together. The concatenation operator can also be used to join other types of variables and values together.

If you installed the documentation package available for the Java Software Development Kit, you can find more information about the `String` class under the main JDK directory at \docs\api\java\lang\String.html. You can also find documentation for the Java SDK at the java.sun.com Web site.

Extra

You can determine the number of characters a string contains by using the `length` method of the `String` class.

TYPE THIS:

```
String message = "The temperature is ";
System.out.print("The length of the string is ");
System.out.print(message.length());
```

RESULT:

```
The length of the string is 19
```

You can use the `equals` method of the `String` class to compare two strings and determine if the strings are the same.

TYPE THIS:

```
String message = "weather";
System.out.print(message.equals("temperature"));
```

RESULT:

```
false
```

You can insert instructions you want Java to interpret into a string. These instructions begin with the backslash symbol (\) and are called escape sequences. Escape sequences allow you to include special characters, such as tabs, newlines and backspaces in a string. Escape sequences are often used to format text that will be displayed on a screen or stored in a file.

\b	Insert a backspace
\t	Insert a tab
\n	Start a new line
\f	Insert a form feed
\r	Insert a carriage return
\"	Insert a double quotation mark
\'	Insert a single quotation mark
\\	Insert a backslash

WORK WITH STRINGS

```
public class DisplayTemperature
{
     public static void main(String[] args)
     {
          String heading = "";

     }
}
```

1 To declare a string variable, type **String** followed by a name for the variable.

2 Type = followed by "".

```
public class DisplayTemperature
{
     public static void main(String[] args)
     {
          String heading = "Current Temperature";
          System.out.println(heading);
     }
}
```

3 Between the quotation marks, type the text you want the string to contain.

4 Type the code that uses the string variable.

```
public class DisplayTemperature
{
     public static void main(String[] args)
     {
          String heading = "Current Temperature";
          System.out.println(heading);

          String message = "The temperature is ";
          int tempNumber = 34;

          System.out.println(message + tempNumber + " degrees.");
     }
}
```

CONCATENATE VARIABLES

5 To join the string with other variables or values, type the concatenation operator (+) between each variable or value you want to join.

6 Type the code that uses the concatenated variables.

```
C:\WINDOWS>cd\java

C:\java>javac DisplayTemperature.java

C:\java>java DisplayTemperature
Current Temperature
The temperature is 34 degrees.

C:\java>
```

7 Compile the Java code and then execute the program.

■ The result of using strings is displayed.

WORKING WITH OPERATORS

Java provides numerous operators that can be used to assign values to variables, perform calculations and create complex expressions. There are several general categories of operators, including assignment, relational, arithmetic, logical, conditional and shift.

TYPES OF OPERATORS

A Java operator can be classified by the number of operands it accepts. An operand is an argument used by an operator. An expression is a sequence of operands separated by one or more operators that produces a result.

Unary

A unary operator accepts a single operand. All unary operators support prefix notation, which means the operator appears before the operand. A commonly used unary operator is `!`, which indicates 'not.' For example, !0 would be used to indicate a value that is not zero.

The increment (`++`) and decrement (`--`) operators also support the postfix notation, which means the operator can be placed after the operand. For example, both ++hitCounter and hitCounter++ increment the operand by one.

Binary

The most common type of operator is the binary operator. A binary operator performs calculations based on two operands, with the operator placed between the operands. For example, the expression 2 + 3 contains the operands 2 and 3, separated by the operator, `+`.

Ternary

The ternary operator `?:` accepts three operands. The `?:` operator tests the first operand and then returns the value of the second or third operand, depending on the result. If the result of the first operand is true, the expression returns the value of the second operand. If the result of the first operand is false, the expression returns the value of the third operand.

PRECEDENCE AND ASSOCIATIVITY

Order of Precedence

When an expression contains several operators, such as 4 - 5 + 2 * 2, Java processes the operators in a specific order, known as the order of precedence. The order of precedence ranks operators from highest to lowest precedence. Operators with higher precedence are evaluated before operators with lower precedence.

Parentheses

Regardless of the precedence and associativity of operators, you can use parentheses to dictate the order in which Java should process operators. In an expression, Java processes operators and operands enclosed in parentheses first.

Associativity

When an expression contains multiple operators that have the same precedence, the associativity of the operators determines which part of the expression will be evaluated first. Operators can have left associativity or right associativity.

If operators have left associativity, then the leftmost operator is processed first. For example, the result of the expression 5 - 3 + 2 is 4 rather than 0. The opposite holds true for operators that have right associativity.

The following table shows the order of precedence from the highest to the lowest, type, category and associativity of operators.

Operator	Type	Category	Associativity
`()`	parentheses	miscellaneous	left
`[]`	array subscript	miscellaneous	left
`.`	member selection	miscellaneous	left
`++`	unary postfix	arithmetic	right
`--`	unary postfix	arithmetic	right
`++`	unary prefix	arithmetic	right
`--`	unary prefix	arithmetic	right
`+`	unary plus	arithmetic	right
`-`	unary minus	arithmetic	right
`!`	unary negation	conditional	right
`~`	bitwise complement	logical	right
`new`	creation	miscellaneous	right
(*type*)	unary cast	miscellaneous	right
`*`	multiplication	arithmetic	left
`/`	division	arithmetic	left
`%`	modulus	arithmetic	left
`+`	addition	arithmetic	left
`-`	subtraction	arithmetic	left
`<<`	bitwise left	shift	left
`>>`	bitwise right with sign extension	shift	left
`>>>`	bitwise right with zero extension	shift	left
`<`	less than	relational	left
`<=`	less than or equal to	relational	left
`>`	greater than	relational	left
`>=`	greater than or equal to	relational	left
`instanceof`	type comparison	miscellaneous	left
`==`	is equal to	relational	left
`!=`	is not equal to	relational	left
`&`	bitwise AND	logical	left
`^`	bitwise XOR	logical	left
`|`	bitwise OR	logical	left
`&&`	logical AND	conditional	left
`||`	logical OR	conditional	left
`?:`	ternary conditional	miscellaneous	right
`=`	assignment	assignment	right
`+=`	addition	assignment	right
`-=`	subtraction	assignment	right
`*=`	multiplication	assignment	right
`/=`	division	assignment	right
`%=`	modulus	assignment	right
`&=`	bitwise AND	assignment	right
`^=`	bitwise XOR	assignment	right
`|=`	bitwise OR	assignment	right
`<<=`	bitwise left shift	assignment	right
`>>=`	bitwise right shift with sign extension	assignment	right
`>>>=`	bitwise right shift with zero extension	assignment	right

CALL A METHOD

Once you have created a method, you need to call the method to tell Java to access and execute the code in the method. The code included in a method will not be executed until the method is called.

To call a method in the same class it was declared in, you type the name of the method followed by a set of parentheses where you want to execute the code specified in the method. You must be sure to type the method name exactly as it was typed in the code that declares the method. Some methods require you to include *arguments* within the parentheses that follow the method name. For information about passing arguments to methods, see page 38.

When a method is called, the code included in the method is executed as if the code was typed in the location where you called the method. Once Java has finished processing the code in the method, Java continues execution from the line following the method call.

In some programs, you may need to call a method that is declared in a different class. The access modifiers used in method declaration determine the locations from which you can call the method. For more information about access modifiers, see page 16.

Classes that contain methods can also be grouped into a *package*. You may need to specify the package that contains the method you want to call. For more information about packages, see page 50.

In addition to calling methods you have created, you can also call methods provided in the Java class library. For example, `System.out.println()` calls a Java class library method which is used to display data. For more information about the Java class library, see page 8.

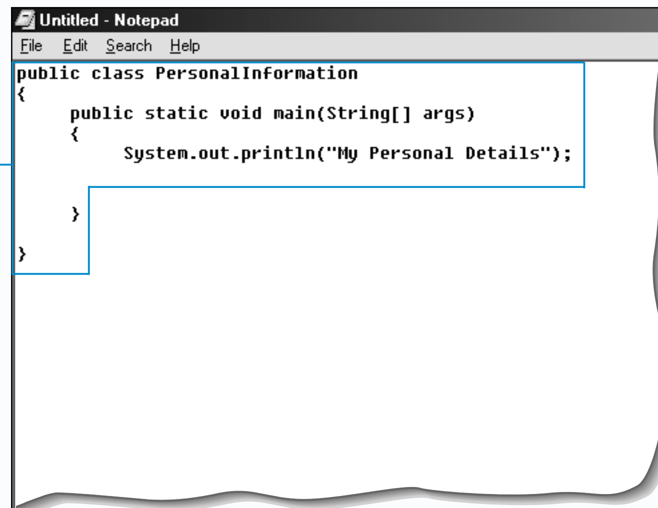
Extra

If a method you want to call is declared in a different class, you must specify the class that contains the method you want to call. You use the dot operator (.) to link the class name and the method name. Any methods called from another class should be created with the `public` access modifier.

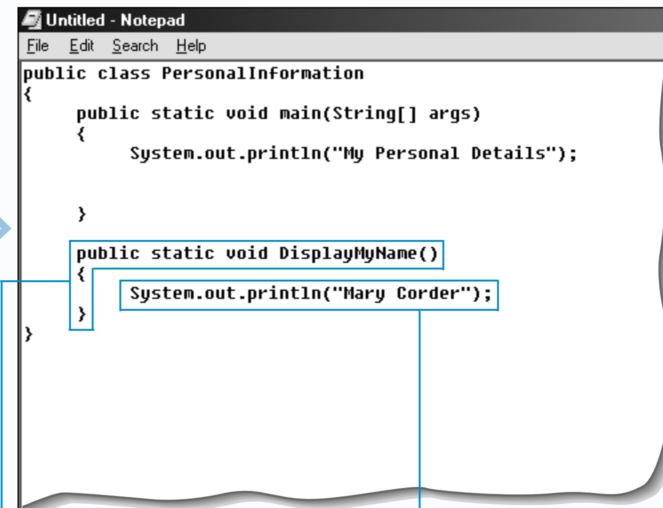
Example:

```
public class PersonalInformation
{
      public static void DisplayMyName()
      {
            System.out.println("David Gregory");
      }
}

public class CallingClassMethods
{
      public static void main(String[] args)
      {
            System.out.println("My Personal Details");

            PersonalInformation.DisplayMyName();
      }
}
```

CALL A METHOD

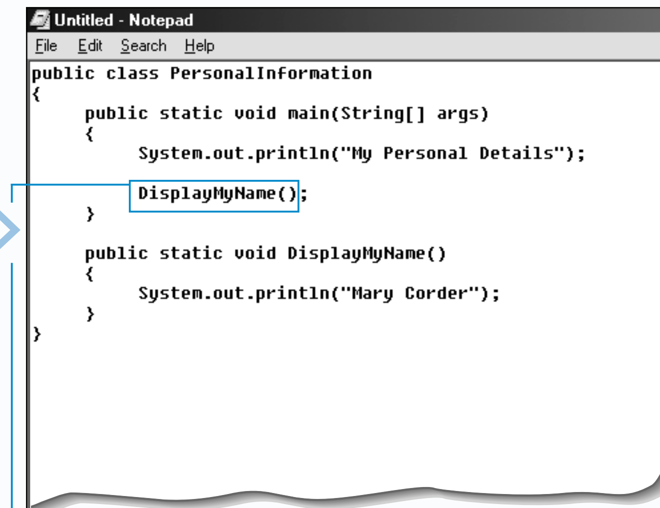


1 Create a class file with a `main` method.

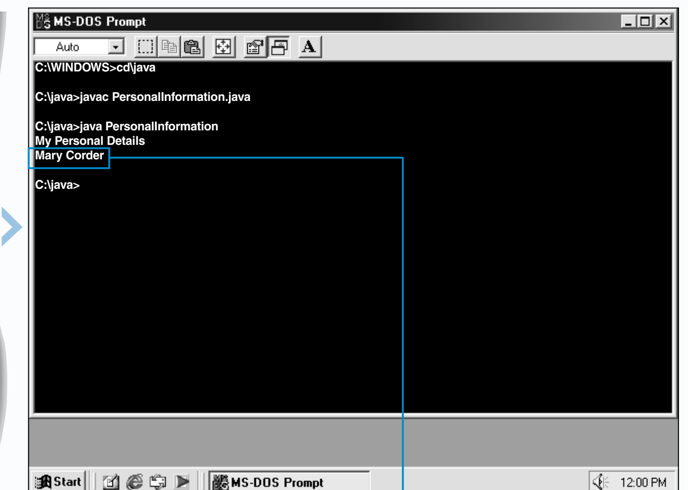


2 Declare the method you want to call.

3 Create the body of the method you want to call.



4 In the body of the `main` method, type the name of the method you want to call, followed by a set of parentheses.



5 Compile the Java code and then execute the program.

■ The result of executing the code in the method is displayed.

USING RETURN VALUES AND ARGUMENTS IN METHODS

You can have a method return a value to the code. A return value may be the result of a calculation or procedure or may indicate whether a process was successfully completed.

The data type of a return value for a method must be specified when the method is declared. Return values can be any valid data type in Java, such as `String`, `byte` or `boolean`. An error may occur if the data type of the value that is returned does not match the return type specified in the method declaration.

Information is returned from a method using the keyword `return`. Once the `return` statement is executed in a method, the processing of the method ends and the value specified in the `return` statement is passed back to the calling statement.

A method with a return value can be used as if it were a variable. For example, you could display the value returned by a method using the `System.out.print` command. You could also assign the value returned by a method to a variable.

You can also pass one or more values, called arguments, to a method you have created. Passing arguments to a method allows you to use one method throughout a program to process different data.

To pass an argument to a method, you include a data type and variable name in the parentheses at the end of the method name in a method declaration. When you call the method, you include the data you want to pass in the parentheses following the method name.

You can pass any type of data to a method, but the type of data must match the data type specified in the method declaration. For example, if a method expects an integer value to be passed for calculation, passing a string value to the method would cause an error to occur.

Apply It

A method can have more than one `return` statement. This is commonly found in methods that use conditional statements. Although a method can have more than one `return` statement, only one `return` statement will be executed. When a `return` statement is encountered, the execution of the method is terminated.

TYPE THIS:

```
class MakeList
{
      public static void main(String[] args)
      {
            System.out.println(CheckAge(29));
      }
      static String CheckAge(int age)
      {
            if (age > 21)
            {
                  return "You may take the survey";
            }
            else
            {
                  return "You are too young to take the survey";
            }
      }
}
```

RESULT:

```
You may take the survey
```

USING RETURN VALUES AND ARGUMENTS IN METHODS

PersonalInformation.java - Notepad

```
class PersonalInformation
{
      public static void main(String[] args)
      {
            System.out.println("My Personal Information");
            System.out.println();
      }
      public static String DisplayMyName()
      {
            String myInfo;
            myInfo = "My name is: " + name;
            return myInfo;
      }
}
```

CREATE A RETURN STATEMENT

1 Type the code that declares the method you want to use.

- The data type of the value the method will return must be specified in this code.

2 Type the code for the body of the method.

3 In the body of the method, type **return** followed by the information you want the method to return.

PersonalInformation.java - Notepad

```
class PersonalInformation
{
      public static void main(String[] args)
      {
            System.out.println("My Personal Information");
            System.out.println();
      }
      public static String DisplayMyName(String name)
      {
            String myInfo;
            myInfo = "My name is: " + name;
            return myInfo;
      }
}
```

PREPARE A METHOD TO ACCEPT ARGUMENTS

4 Between the parentheses following the method name in the method declaration, specify the data type of the argument that the method will accept.

5 Type the name of the variable that will store the value of the argument.

Note: When preparing a method to accept multiple arguments, each data type and variable pair must be separated by a comma.

PersonalInformation.java - Notepad

```
class PersonalInformation
{
      public static void main(String[] args)
      {
            System.out.println("My Personal Information");
            System.out.println(DisplayMyName("Sandy Rodrigues"));
      }
      public static String DisplayMyName(String name)
      {
            String myInfo;
            myInfo = "My name is: " + name;
            return myInfo;
      }
}
```

MS-DOS Prompt

```
C:\WINDOWS>cd\java

C:\java>javac PersonalInformation.java

C:\java>java PersonalInformation
My Personal Information
My name is:  Sandy Rodrigues

C:\java>
```

CALL A METHOD USING ARGUMENTS

6 In the body of the `main` method, type the code that calls the method you want to use.

7 Between the parentheses following the method name, type the arguments you want to pass to the method.

- String arguments must be enclosed in quotation marks.

Note: When passing multiple arguments, the arguments must be separated by a comma.

8 Compile the Java code and then execute the program.

- The result of passing arguments to a method and using a return value is displayed.

USING THE IF STATEMENT

Using an `if` statement allows you to test a condition to determine whether the condition is true or false. The condition can be as complex as necessary, but it must always produce a value that evaluates to either true or false. When the condition is true, the section of code directly following the `if` statement is executed. For example, you can create a program that displays a Good Morning message when a user runs the program between 5:00 AM and 11:59 AM. If the condition is false, no code from the `if` statement will be executed.

A section of code you want to be executed must be enclosed in braces { } and is referred to as a statement block. The condition for an `if` statement must be enclosed in parentheses `()`.

If you want an `if` statement to execute a block when a condition is false, you must include an `else` statement. Using an `if` statement with an `else` statement allows you to execute one of two sections of code, depending on the outcome of testing the condition. If the condition is true, the statement block directly following the `if` statement is executed. If the condition is false, the statement block directly following the `else` statement is executed. Using an `else` statement ensures that a section of code is executed regardless of the outcome of testing the condition. For example, you can have a program display a Good Morning message or a Good Evening message, depending on the time set on the computer that executes the program.

To make your code easier to read and understand, you should always indent the statement block that contains the code to be executed. Many programmers also use spaces within statements to make the statements easier to read. White-space characters, such as tabs and blank lines, are ignored by the Java compiler, so using these characters will not affect the function or performance of your Java program.

Apply It

If you are going to execute only one line of code based on a condition being true, you can place the code to be executed on the same line as the `if` statement.

FOR EXAMPLE:

```
if (currentTemp > hot)
{
    System.out.println("It's hot.");
}
```

CAN BE TYPED AS:

```
if (currentTemp > hot) System.out.println("It's hot.");
```

Nested `if` statements allow you to specify multiple conditions for an `if` statement at the same time. Each `if` statement will be evaluated only if the previous `if` statement is true. If all the `if` statements are true, a section of code is executed. If any of the `if` statements are false, no code from the `if` statements will be executed.

TYPE THIS:

```
int hot = 80, veryHot = 85, currentTemp = 88;
    if (currentTemp > hot)
    {
        System.out.print(currentTemp + " degrees. It's ");
        if (currentTemp > veryHot)
        {
            System.out.print("very, very ");
        }
        System.out.println("hot.");
    }
```

RESULT:

88 degrees. It's very, very hot.

USING THE IF STATEMENT

```
WeatherProgram - Notepad
File Edit Search Help
class WeatherProgram
{
    public static void main(String[] args)
    {
        int hot = 80;
        int currentTemp = 88;

        if (currentTemp > hot)
    }
}
```

1 Type the code that declares the variables and assigns their values.

2 Type **if**.

3 Type the condition you want to test. Enclose the condition in parentheses.

```
WeatherProgram - Notepad
File Edit Search Help
class WeatherProgram
{
    public static void main(String[] args)
    {
        int hot = 80;
        int currentTemp = 88;

        if (currentTemp > hot)
        {
            System.out.println(currentTemp + " degrees. It's hot.");
        }
    }
}
```

4 Type the code you want to execute if the condition you specified is true. Enclose the code in braces.

```
WeatherProgram - Notepad
File Edit Search Help
class WeatherProgram
{
    public static void main(String[] args)
    {
        int hot = 80;
        int currentTemp = 88;

        if (currentTemp > hot)
        {
            System.out.println(currentTemp + " degrees. It's hot.");
        }
        else
        {
            System.out.println(currentTemp + " degrees. It's not that
            hot.");
        }
    }
}
```

5 To use the `else` statement, type **else**.

6 Type the code you want to execute if the condition you specified is false. Enclose the code in braces.

```
MS-DOS Prompt
C:\WINDOWS>cd\java

C:\java>javac WeatherProgram.java

C:\java>java WeatherProgram
88 degrees. It's hot.

C:\java>
```

7 Compile the Java code and then execute the program.

■ The result of testing the condition is displayed on the screen.

USING THE FOR STATEMENT

Programmers often need to execute the same statement or block of statements several times. The `for` statement allows you to create a loop that repeats the execution of code a specific number of times. For example, you may want to create five line breaks on a Web page. Instead of typing the code that creates a line break five times, you can create a loop that executes the code to create a line break and then repeats the loop until the value of a counter reaches 5.

When creating a `for` statement, you usually use a variable, called an iterator, that acts as a counter for the loop. You use an initialization expression to specify a starting value for the iterator.

You must also specify a condition that evaluates the value of the iterator. If the condition is true, the loop is executed and a block of code you specify is processed. If the condition is false, the block of code is not executed and the loop is ended.

The re-initialization expression is used to modify the value of the iterator. For example, if you use the *increment operator* (++) in the re-initialization expression, the value of the iterator will be incremented by one each time the loop is executed. The expression i++ functions the same as i = i + 1.

The block of code you want to execute is placed between braces { } and is known as the body of the loop. You should indent the code in the body of a loop to make the code easier to read and understand. The code in the body of a `for` loop can include any valid Java statements, such as calls to other methods. You may also place another loop within the body of a `for` loop. This is referred to as nesting. You should avoid having too many nested loops because it makes the program difficult to read and troubleshoot.

Extra

A loop can still be executed even if one or more expressions are omitted from the `for` statement. However, any expressions you omit from the `for` statement must be specified elsewhere in the code. For example, if you specify the starting value of the iterator in another part of your code, you do not need to include an initialization expression in the `for` statement. Keep in mind that you must still include all the necessary semicolons in the `for` statement.

Example:

```
int loopCounter = 3;
for (; loopCounter < 5; loopCounter++)
{
      System.out.println(loopCounter);
}
```

If a `for` statement does not include a condition and no condition is specified in the body of the loop, Java assumes that the condition is always true and an *infinite loop* is created. You should be careful not to accidentally create an infinite loop.

Example:

```
int loopCounter;
for (loopCounter = 1; ; loopCounter++)
{
      System.out.println(loopCounter);
}
```

If the body of a `for` loop is composed of a single line of code, you do not have to enclose the line in braces. Although the braces are optional in this situation, most programmers use the braces to keep their code consistent.

Example:

```
for (loopCounter = 0; loopCounter < 10; loopCounter++)
      System.out.println(loopCounter);
```

USING THE FOR STATEMENT

```
class MyJavaProgram
{
      public static void main(String[] args)
      {
            int loopCounter;
            for ()
      }
}
```

1 In the body of the method, declare a variable that will be used as the iterator.

2 Type **for ( )**.

```
class MyJavaProgram
{
      public static void main(String[] args)
      {
            int loopCounter;
            for (loopCounter = 0; loopCounter < 5;)
      }
}
```

3 Type the initialization expression that specifies the starting value of the iterator followed by a semicolon.

4 Type the condition that evaluates the value of the iterator followed by a semicolon.

```
class MyJavaProgram
{
      public static void main(String[] args)
      {
            int loopCounter;
            for (loopCounter = 0; loopCounter < 5; loopCounter++)
            {
                  System.out.println(loopCounter);
            }
      }
}
```

5 Type the re-initialization expression that will modify the value of the iterator each time the loop is executed.

6 Type the code you want to execute as long as the specified condition is true. Enclose the code in braces.

```
C:\WINDOWS>cd\java

C:\java>javac MyJavaProgram.java

C:\java>java MyJavaProgram
0
1
2
3
4

C:\java>
```

7 Compile the Java code and execute the program.

■ The result of using the `for` statement is displayed.

USING THE WHILE STATEMENT

The `while` statement allows you to create a conditional loop that will execute a section of code as long as a specified condition is true. Conditions often test the value of an *iterator*. For example, you may want to process a pay statement for each of the 100 employees in a company. Instead of typing the code that processes a pay statement 100 times, you could create a loop to process the pay statement for each employee. The condition would check how many pay statements have been processed. After the 100th pay statement has been processed, the condition would be evaluated as false and the loop would end.

The body of a `while` loop is enclosed in braces { } and contains the section of code to be executed. If the condition tests the value of an iterator, the loop body will also contain code to alter the value of the iterator. The value of an iterator can be increased or decreased.

When the condition is true, the section of code in the body of the loop is executed. When Java reaches the end of the loop body, the condition is re-evaluated. If the condition is still true, the section of code is executed again. If the condition is false, the section of code in the loop body is not executed and the loop ends.

When creating a loop using the `while` statement, you must ensure that the condition being tested will be evaluated as false at some time. If the condition is always true, the code in the loop body will be executed indefinitely. This kind of never-ending loop is known as an infinite loop. If an infinite loop is created, you will have to forcibly stop the execution of the Java program.

Apply It

A `do-while` statement can be used to test a condition after the code in the loop body has been executed. This is useful if you have a section of code that you want to execute at least once, regardless of how the condition is evaluated.

TYPE THIS:

```
int loopCounter = 0;
do
{
      System.out.println("This is line number "
            + loopCounter);
      loopCounter++;
} while (loopCounter < 0);
```

RESULT:

```
This is line number 0
```

You can place another loop within the body of a `do-while` loop to create a nested loop.

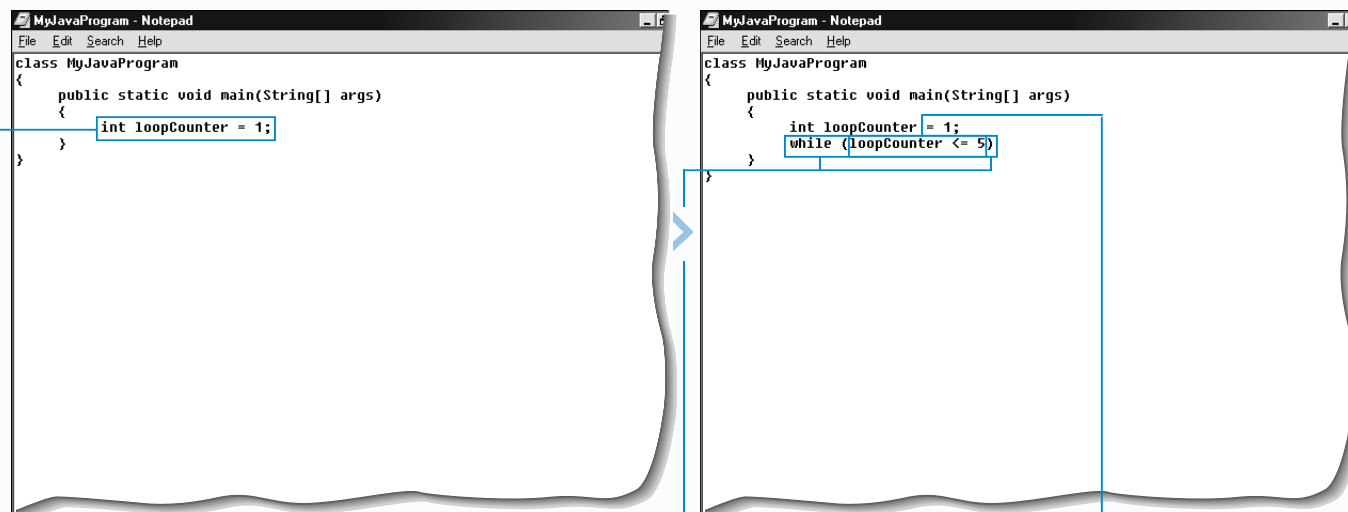
TYPE THIS:

```
int loopCounter = 0, dotCounter;
do
{
      System.out.print("This is line number");
      for (dotCounter = 0; dotCounter < 8; dotCounter++)
      {
            System.out.print(".");
      }
      System.out.println(loopCounter);
      loopCounter++;
} while (loopCounter < 3);
```

RESULT:

```
This is line number........0
This is line number........1
This is line number........2
```

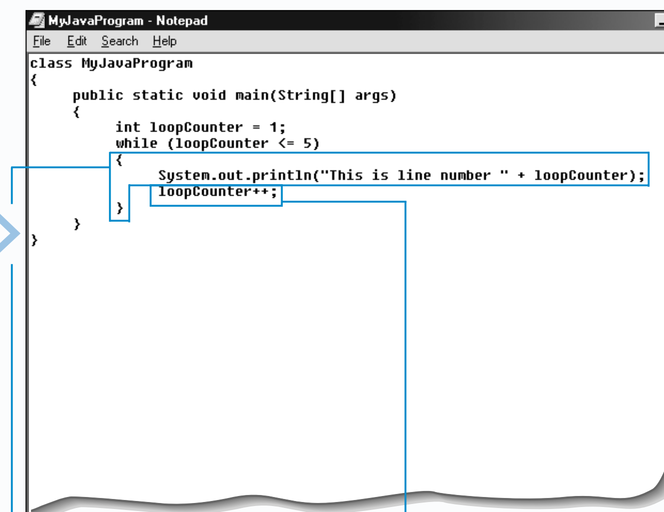
USING THE WHILE STATEMENT



1 In the body of the method, type the code that creates an iterator and assigns it a value.

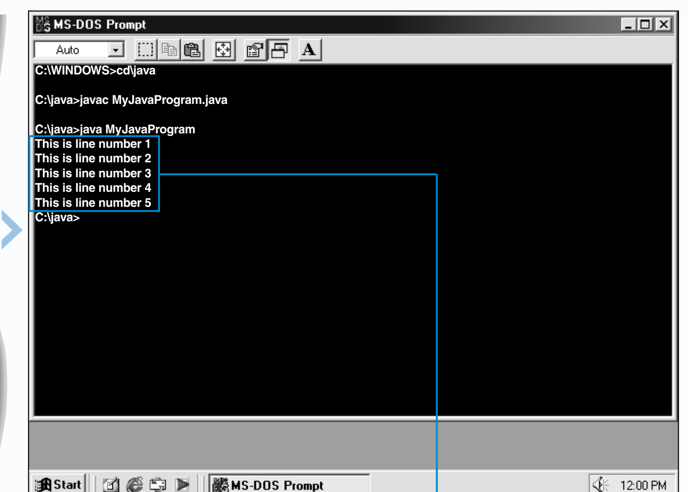
2 Type **while ()**.

3 Type the condition you want to evaluate.



4 Type the code you want to execute as long as the specified condition is true. Enclose the code in braces.

5 In the body of the loop, type the code that will alter the value of the iterator each time the loop is executed.



6 Compile the Java code and then execute the program.

■ The result of using the `while` statement is displayed.

USING THE SWITCH STATEMENT

The switch statement allows you to execute a section of code, depending on the value of an expression you specify. When a switch statement is executed, the value of the expression is compared to a number of possible choices, called case values. If the value of the expression matches a case value, the section of code following the case value is executed. For example, you can create a switch statement that displays a specific message, depending on information entered by a user.

To use the switch statement, you must first specify the expression you want to use. The value of the expression must have a char, byte, short or int data type. After specifying the expression, you must create the case values that the expression will be compared to. The expression must match the case value exactly. You cannot use an indefinite expression, such as x > 10, for a case value.

The switch statement compares the value of the expression to each case value in order, from top to bottom. The case statements can be in any order, but to make your program more efficient, you should place the most commonly used case values first.

To prevent the switch statement from testing the remaining case values after a match has been made, you should use the break statement to skip the remaining case statements and continue processing the code after the closing brace of the switch statement. The break statement should be used as the last statement for each case statement. Although the last case statement does not require a break statement, some programmers include it to be consistent. This can help prevent you from forgetting to include the break statement if you later add another case statement to the switch statement.

Extra

You can execute one section of code for multiple case statements. Each case statement you want to match must be followed by a colon.

Example:

```
switch (gender)
{
	case M: case m:
		System.out.println("Male");
		break;
	case F: case f:
		System.out.println("Female");
		break;
}
```

You can include a default statement in a switch statement if you want to execute specific code when none of the other case values match the specified expression. The default statement is usually placed last in the switch statement structure.

Example:

```
switch (priority)
{
	case 1:
		System.out.println("Urgent");
		break;
	case 2:
		System.out.println("Not Important");
		break;
	default:
		System.out.println("Ignore");
}
```

USING THE SWITCH STATEMENT

```
class MyJavaProgram
{
	public static void main (String[] args)
	{
		int priority = 2;
		switch (priority)
		{
		}
	}
}
```

1 Create the expression you want to use in the switch statement.

2 Type **switch**.

3 Type the name of the expression, enclosed in parentheses.

4 Type a pair of braces to hold the case statements.

```
class MyJavaProgram
{
	public static void main (String[] args)
	{
		int priority = 2;
		switch (priority)
		{
			case 1:
				System.out.println("Urgent");
		}
	}
}
```

5 Type **case** followed by a value the expression may contain.

6 Type **:** to complete the case statement.

7 Type the statements you want to execute if the case value matches the expression you specified in step 1.

```
class MyJavaProgram
{
	public static void main (String[] args)
	{
		int priority = 2;
		switch (priority)
		{
			case 1:
				System.out.println("Urgent");
				break;
			case 2:
				System.out.println("Very Important");
				break;
			case 3:
				System.out.println("Not Important");
		}
	}
}
```

8 Type **break** to prevent the switch statement from testing the remaining case values after a section of code is executed.

9 Repeat steps 5 to 8 for each value the expression may contain.

```
C:\WINDOWS>cd\java

C:\java>javac MyJavaProgram.java

C:\java>java MyJavaProgram
Very Important

C:\java>
```

10 Compile the Java code and then execute the program.

■ The result of using the switch statement is displayed.

CREATE AN ARRAY

An array stores a set of related values, called elements, that are of the same data type. For example, an array could store the name of each day of the week. Using an array allows you to work with multiple values at the same time.

The first step in creating an array is to declare an array variable. To declare an array variable, you specify the data type of the values that the array will store, followed by brackets []. For more information about data types, see page 30. You must also give the array a name. Array names use the same naming conventions as other variables.

Once you have declared the array variable, you can define the array. The `new` operator is used to define an array and indicates that you want to set aside space in memory for the new array. When defining an array, you must also specify the number of elements the array will store.

Each element in an array is uniquely identified by an index number. Index numbers in an array start at 0, not 1. For example, an array defined as items = new int[6] would contain six elements indexed from 0 to 5.

You can specify the values you want each element to store. String values must be enclosed in quotation marks.

To access an individual element in an array, you use the name of the array followed by the index number for the element enclosed in brackets. When brackets are used in this context, they are referred to as the *array access operator*. You can use an array element in a Java program as you would use a variable. Changing the value of an element will not affect the other elements in the array.

Apply It

Unlike most other programming languages, Java treats arrays as objects. The `length` member of the `array` object allows you to determine the number of elements in an array.

TYPE THIS:

```
class ArrayLength
{
  public static void main(String[] args)
  {
    int[] items;
    items = new int[3];

    items[0] = 331;
    items[1] = 324;
    items[2] = 298;

    int total = items.length;
    System.out.print("Number of items = " + total);
  }
}
```

RESULT:

```
Number of items = 3
```

You can use code that creates a loop, such as a `for` statement, to work with all the elements in an array at once.

TYPE THIS:

```
class MyArray
{
  public static void main(String[] args)
  {
    int[] items;
    items = new int[3];

    items[0] = 331;
    items[1] = 324;
    items[2] = 298;
    int total = items.length;

    for (int i = 0; i < total; i++)
    System.out.println(items[i]);
  }
}
```

RESULT:

```
331
324
298
```

CREATE AN ARRAY

MyJavaProgram - Notepad

File Edit Search Help

```
class MyJavaProgram
{
     public static void main(String[] args)
     {
          int[] items;
     }
}
```

1 To declare an array variable, type the data type of the values that will be stored in the array, followed by [ ].

2 Type a name for the array variable.

MyJavaProgram - Notepad

File Edit Search Help

```
class MyJavaProgram
{
     public static void main(String[] args)
     {
          int[] items;
          items = new int[3];
     }
}
```

3 To define the array, type the name of the array variable, followed by =.

4 Type **new** to create the new array, followed by the data type for the array.

5 Type the number of elements the array will contain, enclosed in brackets.

MyJavaProgram - Notepad

File Edit Search Help

```
class MyJavaProgram
{
     public static void main(String[] args)
     {
          int[] items;
          items = new int[3];

          items[0] = 331;
          items[1] = 324;
          items[2] = 298;

          int total = items[0] + items[1] + items[2];
          System.out.println("Items on hand = " + total);
     }
}
```

6 To create an element in the array, type the name of the array, followed by the index number of the element enclosed in brackets.

7 Type = followed by the value for the element.

■ If the value for the element is a string, you must enclose the string in quotation marks.

8 Repeat steps 6 and 7 for each element in the array.

9 Type the code that accesses elements in the array.

MS-DOS Prompt

```
C:\WINDOWS>cd\java

C:\java>javac MyJavaProgram.java

C:\java>java MyjavaProgram
Items on hand = 953

C:\java>
```

10 Compile the Java code and then execute the program.

■ The results of creating an array and accessing elements are displayed.

CREATE A PACKAGE

If your Java program contains a large number of class files, you can organize the files by grouping them into packages. A package stores a collection of related classes. For example, all the shipping-related classes in a program could be grouped into a package called shipping.

Packages allow you to use classes with identical names in the same Java program. Using classes with the same name in one program is normally not permitted in Java. However, when you place classes with the same name in different packages, the classes can be used in a single application without conflict.

When creating a package, you must create a directory to store all the classes for the package. Package directories must always be created in the default class directory that was specified when the Java Software Development Kit was installed on your computer. The lib directory, which is located in the main Java SDK directory, is usually the default class directory.

The name of the directory you create should describe the classes the package will store. All the classes belonging to a package must be saved in the same directory.

You add a `package` statement to a class file to specify the name of the package you want the class to belong to. The `package` statement must be the first line of code in the class file. If the package name consists of multiple words, the words are separated by dots. Each word in the name must represent an actual directory on your computer. For example, classes that are placed in a package called myapps.internet would be stored in a directory called internet, located within the myapps directory.

To use a class stored in a package in an application, you specify the package name and the class name.

Extra

A class always belongs to a package, even when no package is specified. If a package is not specified for a class, the class will belong to the default package, which is the empty string "".

If you are using a Java development tool, such as an Integrated Development Environment (IDE), package directories may already be set up for you within a main class directory. You can usually change the configuration of the program to specify another directory as the main class directory.

The method you use to create directories will depend on the type of operating system installed on your computer. If you are using a UNIX-based operating system, such as Linux, you might use the `mkdir` command to create directories in a terminal window. If you are using an operating system with a Graphical User Interface (GUI), such as Macintosh or Windows, you would use the graphical tools provided to create directories.

When you use a class stored in a package, you must specify the name of the package in addition to the class name. To avoid having to specify the package name each time you want to use the class, you can *import* the package into your program.

CREATE A PACKAGE

```
package myapps;
```

```
package myapps;
public class MyInformation
{
      public static String emailAddress()
      {
            return "tom@abc.com";
      }
}
```

1 Create a directory on your computer that will store classes for the package.

■ In this example, a directory named myapps is created in the lib directory. The lib directory is located in the main Java SDK directory.

2 On the first line of code in a class file, type **package** followed by the name of the package you want to create.

Note: The package name must be the same as the name of the directory you created in step 1.

3 Enter the code that declares a class and a method that you want to use in other Java programs.

4 In the body of the method, type the code for the task you want to perform.

5 Save the code with the .java extension in the directory you created in step 1.

```
public class TestPackage
{
      public static void main(String[] args)
      {
            System.out.println("My email address is " +
            myapps.MyInformation.emailAddress());
      }
}
```

```
C:\WINDOWS>cd\jdk1.3\lib
C:\jdk1.3\lib>javac TestPackage.java
C:\jdk1.3\lib>java TestPackage
My email address is tom@abc.com
C:\jdk1.3\lib>
```

USE A CLASS STORED IN A PACKAGE

1 To use a class stored in a package you created, type the name of the package followed by a dot.

2 Type the name of the class you want to use from the package, followed by a dot.

3 Type the name of the method you want to access.

4 Compile the Java code and then execute the program.

■ The result of using a class stored in a package is displayed.

IMPORT A PACKAGE

You can import a class from a package you have created into a Java program. This is useful if you plan to use the class several times in the program. Once a package and a class have been imported, you do not need to specify the name of the package each time you want to access the class.

The `import` statement is used to import a package and is usually placed at the beginning of your Java program. If your program contains a `package` statement, the `import` statement must be placed after the `package` statement. You can import several packages and classes into one Java program. Each package you want to import must have its own `import` statement. You should not import two classes with the same name into one program.

You must first create the package you want to import. For more information about creating a package, see page 50.

To help ensure an error is not generated when you compile the code for your program, you must ensure that the package directory and the class you want to import are available. In most situations this is not a concern, but it becomes important if you are developing programs on different computers or different platforms.

When importing a class from a package, you must specify the name of the class you want to import. You should only import class files you intend to use. Imported class files increase the size of the bytecode created when you compile Java code.

In addition to packages and classes that you create, you can import packages and classes that are part of the Java class library. You can refer to page 8 for more information about the packages included in the Java class library.

Extra

You can use the wildcard character `*` to have Java import all the classes a package contains. This is useful if you want to access several classes in a package. In the following example, the package is named myapps.webutils.

Example:

```
import myapps.webutils.*
```

When using the wildcard character `*`, it is important to note that only the classes in the named package will be imported. For example, the import myapps.webutils.* statement will only import the classes found in the myapps.webutils package and will not import any classes found in the myapps.webutils.text package. To import classes from the myapps.webutils.text package, you must use the import myapps.webutils.text.* statement.

Java may not import every class a package contains when you use the wildcard character `*`. When you compile your code, Java searches the code and imports only the classes that are used. This prevents your bytecode from becoming too large.

Java can automatically import certain packages when you compile code. The `java.lang` package, which is part of the Java class library, is automatically imported whenever you compile code. If your code contains classes that do not belong to a package, Java imports the default package "" and assigns the classes to that package. If your Java code contains a `package` statement, the named package is also automatically imported.

IMPORT A PACKAGE

```
import myapps.
```

```
import myapps.MyInformation;

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

     }
}
```

```
import myapps.MyInformation;

public class TestPackage
{
     public static void main(String[] args)
     {
          System.out.println("My email address is "
               + MyInformation.emailAddress());
     }
}
```

```
C:\WINDOWS>cd\jdk1.3\lib

C:\jdk1.3\lib>javac TestPackage.java

C:\jdk1.3\lib>java TestPackage
My email address is tom@abc.com

C:\jdk1.3\lib>
```

1 To import a package, type **import** in the first line of code.

2 Type the name of the package you want to import followed by a dot.

3 Type the name of the class you want to import.

4 Enter the code that declares the class and the method you want to use.

5 In the body of the method, type the code for the task you want to perform.

6 To use the imported class, type the name of the class followed by a dot.

7 Type the name of the method you want to access.

8 Compile the Java code and then execute the program.

■ The result of using a class from an imported package is displayed.

EXTEND A CLASS

If a class you are creating is related to a class you have previously created, you can make the new class an extension of the original class. For example, you can make a new class that performs tasks using a database an extension of the class that connects to the database. This allows you to re-use Java code in the original class without having to retype the code in the new class.

When you extend a class, the original class is usually referred to as the super-class, while the new class is called the sub-class.

When declaring a class you want to use as a sub-class, you must use the `extends` keyword to specify the name of the class that will act as the super-class. The class you specify using the `extends` keyword must be a valid class that will be accessible to the sub-class when the sub-class is compiled.

Whether or not a method within a super-class will be accessible to a sub-class depends on the access modifier the method uses. A method that uses the `public` access modifier will be accessible to any sub-class, while a method that uses the `private` access modifier will not be accessible to sub-classes. A method that does not have an access modifier specified will be accessible only to sub-classes that are stored in the same package as the super-class.

Once you have created a sub-class as an extension of a super-class, you can create a new class that accesses the sub-class. For example, a new class can create an object using the sub-class. The class information from both the sub-class and the super-class will be combined to form a single object, with methods from both the sub-class and the super-class available to the object.

Many of the classes included with the Java SDK extend to other classes. For information about the Java SDK classes, refer to the Java SDK documentation.

Extra

As with methods, fields within a super-class will also be available to a sub-class, depending on the access modifier a field uses. A field that uses the `private` access modifier will not be accessible to any sub-classes, while a field that uses the `public` access modifier will be available to all sub-classes.

When creating a sub-class, you can override a method in the super-class that you do not want to be available when the sub-class is accessed. To override a method in the super-class, create a method in the sub-class that has the same name as the method you want to override. The access modifier of the method in the sub-class must be the same or less restrictive than the access modifier of the method in the super-class. When an object is created using the sub-class, the method in the sub-class will be available instead of the method in the super-class.

A class you created as a sub-class can be used as the super-class of another class. This allows you to create a chain of sub-classes and super-classes. A class that extends directly from a super-class is called a direct sub-class of the super-class. A class that is an extension of another sub-class is called a non-direct sub-class of the super-class. There is no limit to the number of sub-classes that can be created from other sub-classes.

EXTEND A CLASS

Untitled - Notepad

```
public class BoldMessage
{
     public void showBolded(String message)
     {
          System.out.print("<b>" + message + "</b>");
     }
}
```

CREATE THE SUPER-CLASS

1 Type the code that defines a class you want to be able to extend to another class.

2 Compile the Java code for the class.

Untitled - Notepad

```
public class ItalicMessage extends BoldMessage
{
     public void showItalic(String message)
     {
          System.out.print("<i>" + message + "</i>");
     }
}
```

CREATE THE SUB-CLASS

1 Type the code that defines a class you want to use as an extension of another class.

2 In the method declaration, type **extends** followed by the name of the class you want to use as the super-class.

3 Compile the Java code for the class.

Untitled - Notepad

```
public class DisplayMessage
{
     public static void main(String[] args)
     {
          ItalicMessage text = new ItalicMessage();
          text.showBolded("Copyright");
          text.showItalic("1999, 2000, 2001.");
     }
}
```

MS-DOS Prompt

```
C:\WINDOWS>cd\java

C:\java>javac DisplayMessage.java

C:\java>java DisplayMessage
<b>Copyright</b><i>1999, 2000, 2001.</i>

C:\java>
```

USING AN EXTENDED CLASS

1 To create a class that will instantiate an object of the sub-class you created, type the code that defines the class and method you want to use.

2 In the body of the method, type the code that creates the object.

3 Type the code that accesses methods from the sub-class and the super-class.

4 Compile the Java code and then execute the program.

■ The results of instantiating the object of a sub-class and accessing methods of the sub-class and super-class are displayed.

UNDERSTANDING VARIABLE SCOPE

The scope of a variable determines the part of a program that can access the variable and use its value. In Java, there are strict guidelines governing variable scope. These guidelines are referred to as scoping rules.

The scope of a variable is determined by the position of the variable declaration within a block of code. An opening brace and a closing brace denote a block of code. The scope of a variable is from the line of code containing the variable declaration to the closing brace of the block.

If you declare a variable in the body of a class, outside of any method, the variable will be accessible to all the methods in the class. A variable declared in a class body is referred to as a member variable.

When using the Java interpreter to execute a class file, methods and variables created in the body of the class file must be declared using the `static` access modifier.

A variable declared within a method is referred to as a local variable. A local variable is accessible only within the method in which it was declared. Other blocks of code created within the method can access the local variable.

You can use the same name to declare a member variable and a local variable in one class. When you use the same name to declare two variables of different scope, Java treats the variables as distinct. Although variables with different scopes can have the same name, using unique variable names will make your code easier to understand. For example, instead of using a variable named counter for all your counting functions, you should use variations of the name, such as loopCounter for counting loop iterations or processCounter for counting the number of times a particular process is executed.

Extra

The scope of a variable is restricted to the block of code that contains the variable declaration. If you declare a variable in a block of code created by an `if` statement or a statement that produces a loop, the variable will be a local variable.

Example:

```
boolean go = true;

if (go)
{
      int x = 3;
}

System.out.print(x);
```

Produces this error message when compiled:

```
Scope.java:12: cannot resolve symbol
symbol  : variable x
location: class Scope
          System.out.print(x);
                           ^
1 error
```

UNDERSTANDING VARIABLE SCOPE

```
public class ShowScope
{
    static int x = 10;

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

}
```

1 To create a member variable, type **static** in the body of the class.

2 Type the code that declares the member variable.

3 Type the code that declares a `main` method.

```
public class ShowScope
{
    static int x = 10;

    public static void main(String[] args)
    {

    }

    private static void firstMethod()
    {
        int x = 55;
        System.out.println("This is the value of the local variable x: " +
        x);
    }

}
```

4 Type the code that declares a method.

5 To create a local variable, type the code that declares a variable in the body of the method.

■ Give the local variable the same name as the member variable, but a different value.

6 Type the code that displays the value of the local variable.

```
public class ShowScope
{
    static int x = 10;

    public static void main(String[] args)
    {
        firstMethod();
        secondMethod();
    }

    private static void firstMethod()
    {
        int x = 55;
        System.out.println("This is the value of the local variable x: " +
        x);
    }

    private static void secondMethod()
    {
        System.out.println("This is the value of the member variable x: " +
        x);
    }
}
```

7 Type the code that declares another method.

8 In the body of the method, type the code that displays the value of the member variable.

9 In the body of the `main` method, type the code that calls each method.

```
C:\WINDOWS>cd\java

C:\java>javac ShowScope.java

C:\java>java ShowScope
This is the value of the local variable x: 55
This is the value of the member variable x: 10

C:\java>
```

10 Compile the Java code and then execute the program.

■ The value of the member variable and the value of the local variable are displayed.