Statements involving binary decisions are referred to as [conditionals](http://learn.flvs.net/webdav/educator_apcsa_v14/glossary.htm) because *if* some condition is true, *then* some consequence or action occurs. Condition statements give programmers the ability to make decisions based on evaluation of relationships using Java’s reserved word **if** in combination with six [**relational operators**](http://learn.flvs.net/webdav/educator_apcsa_v14/glossary.htm) as follows.

|  |  |  |  |
| --- | --- | --- | --- |
| if (average >= 90 ) | | if (age <= 10 ) | |
|  | System.out.println (":Grade = A") |  | admissionFee = 4.50; |
|  | | | |
| if(hours > 40) | | if (balance < 0) | |
|  | overtimePay = (hours - 40) \* rate \* 1.5; |  | fee = 24; |
|  | | | |
| if (side1 != side2) | | if (tempF = = 32) | |
|  | System.out.println("Not a square."); |  | tempC = 0; |

Four (>, <, >=, and <=) of the six relational operator symbols should look very familiar to you from your study of inequalities in algebra, but the other two (= = and !=) are unique to several programming languages. Can you determine which operations these symbols represent?

These six Java statements look very different from the ones you have used in earlier programs. Notice the following:

* **if** statements are written on more than one line.
* The condition is on the first line and the action or consequence is on the second.
* The condition is written inside a pair of parentheses.
* **if** statements always include one of the six relational operators.
* There is no semicolon on the first line.

Conditional statements always involve a branch in a program’s [flow of control](http://learn.flvs.net/webdav/educator_apcsa_v14/glossary.htm). This is most clearly illustrated by several flowchart segments shown below.

Simple **if** statements always exhibit this pattern; the flow of control branches if a binary condition is true, otherwise the branch is skipped.

In its decision-making Java relies on a simple data type called boolean. Boolean variables are declared using the keyword boolean. They are simple in the sense that they can be assigned one of only two values, namely true and false. These values may be entered directly into Java expressions and statements as boolean literals. In fact, in the following code fragment, the variable a is initialized to be a boolean by setting its value to be that of a boolean literal:

 The following summary is concerned with the behavior of the relational operators in expressions of the form a *op* b, where *op* is one of the six relational operators referenced below, each of a and b is either an int or a double, and neither a nor b has a special floating point value (-0.0, NaN, -Infinity, or Infinity). [For information concerning the behavior of these operators when special floating point values are involved, click [here](javascript:secWindow('mainpage?epid=E202817175&cid=162149&s=2','InfoPopUp',640,540,40,40,'menubar,scrollbars,resizable')).]

< (strictly less than)

a < b has the value true if the value of a is numerically *strictly less than* the value of b, and it has the value false otherwise.

<= (less than or equal to)

a <= b has the value true if the value of a is numerically *less than or equal to* the value of b, and it has the value false otherwise.

> (strictly greater than)

a > b has the value true if the value of a is numerically *strictly greater than* the value of b, and it has the value false otherwise.

>= (greater than or equal to)

a >= b has the value true if the value of a is numerically *greater than or equal to* the value of b, and it has the value false otherwise.

== (equal to)

a == b has the value true if the value of a is numerically *equal to* the value of b, and it has the value false otherwise.

!= (not equal to)

a != b has the value true if the value of a is numerically *not equal to* the value of b, and it has the value false otherwise.

The operator == is called *the identity operator*. Its importance lies in the fact that it is used to test whether its operands are — according to Java — the same. For example, 52 == 52, 30 == (2 \* 15) and 3.14 == 3.14 all evaluate to true.

Be sure to bear in mind, however, that, when one of the operands of the identity operator is a floating point number and the other is an integer, the integer is converted to a floating point number before the comparison takes place. So, for example,

* (1 / 2) == 0.0 is true. The expression 1 / 2 has integer operands, so it evaluates to the integer 0, which is converted to 0.0 prior to the comparison being made.
* (1.0 / 2) == 0.0 is false. In the expression 1.0 / 2, 2 is converted to a floating point number before being divided into 1.0. The division therefore yields 0.5, which is *not* equal to 0.0.

A common mistake in Java programming is to omit the second = symbol when typing the identity operator. When this mistake is made, Java interprets the single = symbol as the [assignment operator](http://pages.eimacs.com/eimacs/mainpage?epid=E2247639333&cid=162149#AssignOp). This usually causes Java to generate an error message.

You have noticed that many Java statements end with a semicolon. In fact, you have probably developed the habit of automatically putting a semicolon at the end of almost every line in a program. Be for warned! If you accidentally put a semicolon at the end of an if statement, your program may compile and run, but it will not perform as intended. Placing a semicolon at the end of an if statement can lead to hours of debugging frustration!

The structure of a generic **if-else** statement is shown below along with a corresponding flowchart segment.

…

**if (boolean Condition)**  
**execute Code Block A;**  
**else**  
**execute Code Block B**

…

Syntactically, it is important to note that the condition must be **boolean**; it must be written inside of a pair of parentheses; and each line in an executable block must end with a semicolon. Notice, however, that the actual **if** and **else** statements **do not** end with a semicolon. In terms of style, it is good programming practice to indent statements within each block of code.

In the program below, the statement executed in the case when x is not 0 is a *do nothing* statement. It consists of a single semicolon and nothing else.

    int a = 6;   
    int x = ;   
    if ( x == 0 )   
      a = a + 10;   
    else   
      ;   
  
    System.out.println( a );

In fact, Java provides a shortcut for just this situation, allowing us to omit the else statement altogether. The following abbreviated program behaves in exactly the same way as the one above.

    int a = 6;   
    int x = ;   
    if ( x == 0 )   
      a = a + 10;   
  
    System.out.println( a );

Use a cascading sequence of conditional statements as you complete the program below so that it will calculate the total price of an item when sales tax is calculated in accordance with this table:

|  |  |
| --- | --- |
| itemtype | salestax |
| "food" | 0% |
| "clothing" | 0% |
| "non-essential" | 5% |
| "luxury" | 30% |

Since all the possible values of itemtype are Strings, we remind you that your program should use the [equals](http://pages.eimacs.com/eimacs/mainpage?epid=E1981463220&cid=162149#EqualsMethod) method to check which type of item it is dealing with.

**Exercise 55**

The missing code is highlighted below:

  double salestax = 0.0;  
  
  if ( itemtype.equals( "non-essential" ) )  
    salestax = 5.0;  
  else if ( itemtype.equals( "luxury" ) )  
    salestax = 30.0;  
  
  System.out.println( "Total price is $"  
                        + (cost + (cost \* salestax / 100)) );

Notice that there is no need to check whether itemtype is "food" or "clothing", because, by initializing salestax to 0.0, the program already takes care of those two possibilities.

In the above solution, it is also correct to assign 5 or 30 to salestax in the highlighted conditional statement. Since salestax is declared to be a double, Java will implicitly cast any integer you assign to salestax to a double.

Furthermore, you may optionally omit the keyword else from the highlighted code. As a result you will have two successive conditional statements, neither of which includes an else statement. In this case, the resulting code will behave in exactly the same way as the code we provide above.

**if(boolean Condition 1)**  
**execute Code Block A;**  
**else if(boolean Condition 2)**  
**execute Code Block B;**  
**else if(boolean Condition 3)**  
**execute Code Block C;**  
**else**  
**execute Code Block D;**

The *sign* of a floating point or integer value a may be defined as follows:

* If a is negative, then the sign of a is -1
* If a is zero, then the sign of a is 0
* if a is positive, then the sign of a is 1

So, to find the sign of a, we could first check whether a < 0 is true. If so, then the sign of a is -1. If not, we could go on to test whether a == 0 is true. If so, then the sign of a is 0. If not, then the sign of a must be 1. This *algorithm* (that is, procedure) for calculating the sign of a is employed in the following code fragment, which uses the if ... else keyword combination twice in succession. Study the code and run it a few times to verify that it works correctly.

    int a = ;   
    if ( a < 0  )   
     System.out.println( "The sign of " + a + " is -1" );   
    else if ( a == 0 )   
      System.out.println( "The sign of " + a + " is 0" );   
    else   
      System.out.println( "The sign of " + a + " is 1" );

The **equals()** method takes one parameter, which will be a **String** object for now, and returns a **boolean** value which can only be **true** or **false**. Case matters when evaluating **Strings**, so there are additional methods that deal with uppercase and lowercase information as well as some methods that simply ignore case altogether.

For the sake of simplicity, all of the condition statements you have written so far have only included one line of code in the **if-else** blocks; however, with a minor variation, blocks can consist of multiple statements. In the following segment of code, notice the inclusion of a pair of curly braces around the blocks of code in each section of the conditional statement.

…   
boolean isChild;   
double admissionFee = 6.00;  
…   
isChild = age <= 10;

if(isChild)  
**{**   
    admissionFee = 4.50;  
    category = “Child”;  
**}**  
else  
**{**  
    admissionFee = 6.00;  
    category = “Adult”;  
**}**

admissionFee += admissionFee \* tax;**…**

Each pair of curly braces defines a block of code containing multiple lines. Notice that all lines **within a block** end with a semicolon. Errors will occur if blocks include multiple lines without the curly braces or if semicolons are not included at the end of each statement. Even when a block contains only one statement, many programmers include the curly braces to emphasize the logical structure of the condition as indicated in the following code segment.

if(isChild)  
**{**   
    admissionFee = 4.50;  
**}**  
else  
**{**  
    admissionFee = 6.00;  
**}**

To build consistent programming skills, it is suggested that you always use a pair of curly braces in conditional blocks no matter how many statements are included.

In Java programs that work with text, the need often arises to compare strings. Sometimes, our only interest is in whether the strings are the same. But in other situations our focus is more upon sorting strings into order. The techniques for working with strings in this way are different from those we have just met in the context of numbers. We begin by dealing with the matter of deciding if two strings are the same or different.

For this purpose, Java provides the equals method. This is a one-argument method, just like the indexOf method that we met [earlier](http://pages.eimacs.com/eimacs/mainpage?epid=E2163739133&cid=162149). If a and b are Strings and we want to find out if they are the same string, then we may either send the message equals( b ) to a or we can send the message equals( a ) to b. That is, we may determine the value of either a.equals( b ) or b.equals( a ).

The equals method is a boolean method, that is, it returns one of only two values: true or false. The result is true if and only if the argument is the same string as the string to which the message is sent.

To make the comparison of strings according to this extended lexicographical order possible, Java provides the one-argument method compareTo. (Note that the capitalization of the letter "T" in the name of this method is important.) This is not a boolean method, however. Instead, the value that it returns is always an integer. Specifically,

If a and b are Strings, then the value of a.compareTo( b ) is

* **negative** if and only if a precedes b in the extended lexicographical order;
* **zero** if and only if a and b are the same string;
* **positive** if and only if b precedes a in the extended lexicographical order.

It follows that to determine which of two strings precedes which, we should evaluate a relational expression (whose value, of course, is a boolean):

If a and b are Strings, then

* the value of a.compareTo( b ) < 0 is true if and only if a precedes b in the extended lexicographical order;
* the value of a.compareTo( b ) == 0 is true if and only if a and b are the same string;
* the value of a.compareTo( b ) > 0 is true if and only if b precedes a in the extended lexicographical order.

Notice that the value of a.compareTo( b ) == 0 being true provides exactly the same information as the value of a.equals( b ) being true, namely, that a and b are the same string.

Suppose that a, b, and c are Strings that are initialized as in the code fragment below. Use the code fragment to help you evaluate each of these expressions:

1. a.compareTo( b )
2. b.compareTo( c )
3. c.compareTo( a )
4. b.compareTo( c ) < 0
5. c.compareTo( c ) < 0
   1. 32
   2. -5
   3. -32
   4. true
   5. false
6. By evaluating relational expressions such as

"2".compareTo( "M" ) < 0 and "M".compareTo( "m" ) < 0

we discover that, in Java's extended lexicographical order, numerals precede uppercase letters, which in turn precede lowercase letters.

1. Sorted into increasing extended lexicographical order, the given strings are:

"$1million", "2468", "B4", "LA5T", "LASSIE", "after", "first".

As you might expect, since a block of statements plays the same role in a program as a single statement, there are special rules that apply to statement blocks. Here are three that are concerned with the declaration of variables:

Rules concerning Variable Declarations

1. If a variable is declared outside any statement block, then the variable's value is *visible* at all points of the program — including within statement blocks — that come after the statement that initializes the variable.
2. If a variable is declared within a statement block, then the variable's value is only visible from the statement in which it is initialized to the end of that block.
3. It is an error to declare a variable at any point of a program where another variable of the same name is visible.
4. For example, the following code produces an error:
5. int a = 5;   
       System.out.println( "a is " + a );   
     
       {    
         int a = 8;   
         System.out.println( "a is " + a );    
       }
6. TC1.java:12: error: variable a is already defined in method main(String[])   
     
         int a = 8;   
     
             ^
7. The declaration of a and its initialization to 5 occurs outside any statement block. By Rule 1, the value of a is therefore visible at the point within the statement block where an attempt is made to declare and initialize a again. In light of Rule 3, this provokes an error.
8. Similarly, the following code produces an error:
9. {    
         int a = 8;   
         System.out.println( "a is " + a );    
       }   
     
       int b = a + 1;   
       System.out.println( "b is " + b );
10. TC1.java:13: error: cannot find symbol   
      
        int b = a + 1;   
      
                ^
11. In this case, the variable a is declared and initialized in the opening block of statements. Its value is therefore invisible when the variable b is declared outside the block. As a result, a cannot resolve symbol error is generated.
12. On the other hand, the following code violates none of the above rules, since the first declaration of a, being inside a statement block, is not visible when the second declaration is made (inside a different statement block):
13. {   
          int a = 5;   
          System.out.println( "a is " + a );   
        }   
      
        {    
          int a = 8;   
          System.out.println( "a is " + a );    
        }
14. a is 5   
    a is 8
15. Once you get used to them, these rules are very helpful. They allow us, for example, to create temporary variables that are only of use within a particular block of statements. In the next exercise, the variable t is just such a temporary variable.

#### Exercise 57

1. What are the values of a and b after the following code is executed?
2. int a = -6;   
       int b = 10;   
     
       if ( a < 0 )    
       {    
         int t = a;   
         a = b;   
         b = t;   
       }
3. What are the values of a and b after the following code is executed?
4. int a = -6;   
       int b = 10;   
     
       if ( a < 0 )    
       {    
         int t = a;   
         a = b;   
         b = t;   
       }

a is 10 and b is -6.  
The variable t serves as a temporary store to house the initial value of a while the new value of a is being assigned and before the new value of b is set to the old value of a. Without t, the old value of a would be lost as soon as a is assigned its new value.

**String** class methods are frequently necessary to process alphanumeric data and you will use them throughout the course. Some of the most useful **String** methods include the following.

|  |  |
| --- | --- |
| equals() | equalsIgnoreCase() |
| toUpperCase() | toLowerCase() |
| compareTo() | compareToIgnoreCase() |

Simple decision statements only require evaluation of one **boolean** expression; however, Java also has the ability to handle multiple **boolean** expressions.

For example, what if a student’s eligibility for extracurricular activity depended on having a GPA greater than or equal to 2.0, and no F’s in any courses. This combination condition can easily be handled using one of Java’s [logical operators](http://learn.flvs.net/webdav/educator_apcsa_v14/glossary.htm): **AND, OR, NOT, XOR**. Since both conditions must be satisfied for the student to be eligible, the statement would be written as follows.

if((average >= 2.0) && (numFs = = 0))  
     System.out.println(“Student is eligible);  
else  
     System.out.println(“Student is not eligible);

The **&&** symbol is a **boolean** operator which represents AND.

The logical operator **AND** corresponds to intersection in Boolean algebra, because both parts of the condition must be true for the whole statement to be true.

There are also circumstances where only one part of a condition needs to be true for an entire condition to be true. For example, eligibility could also be determined using the **OR** operator shown below.

if((average < 2.0) || (numFs != 0))  
          System.out.println(“Student is not eligible);  
     else  
          System.out.println(“Student is eligible);

A pair of vertical bars (e.g. **||**) is the **boolean** operator for **OR**. The pipe symbol (a vertical bar) is usually found on the right side of the keyboard on the backslash key (\).

When the logical operator **OR** is used, the statement is true if each separate condition is true and if both conditions are true. Using not equal (**!=**) in the second example makes this statement a little more awkward to interpret, but the **AND** and **OR** versions of the decision statements are equivalent.

There is a third useful logical operator with a strange name, **XOR** (pronounced exclusive **OR**), that will come in handy if you work as a waiter or waitress and ever serve a programmer or a logician. If a restaurant menu says the customer can have ice cream or pie for desert, which of the following Venn diagrams would apply?

To a programmer/logician (like you) ice cream **OR** pie means you could have either dessert, or both! However, to the server, it most likely means you can have one or the other, but certainly not both. Having taken this course, you will be able to have an interesting discussion with the customer about the **XOR** operator which applies in this situation. (Your programming knowledge could also lead to a bigger tip!).

|  |  |
| --- | --- |
| a | !a |
| true | false |
| false | true |

1. !(5 == 3)
2. !(5 != 3)
3. !(3 < 5)
4. !(5 < 3)
5. The operator ! is sometimes referred to as a unary operator because it has only one operand. (Recall that we met this terminology before when considering [unary subtraction](http://pages.eimacs.com/eimacs/mainpage?epid=E206162549&cid=162149).) With the exception of ! and unary subtraction, the other operators we have met so far — the assignment and concatenation operators, the other arithmetic operators, the relational operators — and the other two logical operators we are about to discuss all have two operands. They are referred to as binary operators.
6. The behavior of && is summarized in this truth table:

|  |  |  |
| --- | --- | --- |
| a | b | a && b |
| true | true | true |
| true | false | false |
| false | true | false |
| false | false | false |

1. In words, if a and b are boolean values, then a && b (read as a and b) is true if and only if both a and b are true.
2. Note that, in the expression a && b, if a is known to be false then we can evaluate the expression without even knowing the value of b. If a is false, the value of a && b *must* be false, regardless of b's value. Java makes use of this fact to avoid unnecessary work. In particularly, it does not evaluate the second operand of an && operator if it discovers that the first operand is false. This is called *short-circuit evaluation*.
3. Short-circuit evaluation can be used to advantage. Let us look at a simple example. The intent of the program, the most important part of which is displayed below, is to test whether 12 is divisible by y. The strategy it uses is to check whether or not the remainder is zero when 12 is divided by y. Use the code fragment to help you discover what the value of b is when y is 3 and when y is 5.
4. The program seems to do its job — until, that is, you initialize y to 0. Then something undesirable happens. Try it to find out what occurs.
5. Short-circuit evaluation can be used to sidestep this problem. In the following program, notice how the expression y != 0 is used to "trap" the case when y is 0:
6. int y = ;   
       boolean b = ( y != 0 && ((12 % y) == 0) );   
     
       System.out.println( "b is " + b );

In Java programs it is quite often useful to "and" together a whole collection of boolean values. Consider, for example, the following expression, which is obtained by "and"ing together five boolean expressions:

a != 0 && a < 10 && a > -10 && (a / 5) > 1 && (a % 2) == 0

An extended version of short-circuit evaluation applies. To evaluate the expression, Java works from left to right, evaluating one boolean expression at a time, until either

* it finds an expression with a value of false — in which case evaluation ceases and the whole expression evaluates to false — or
* all the boolean expressions have been evaluated without any of them having a value of false — in which case the whole expression evaluates to true.
* The behavior of the third logical operator, ||, is summarized in this truth table:

|  |  |  |
| --- | --- | --- |
| a | b | a || b |
| true | true | true |
| true | false | true |
| false | true | true |
| false | false | false |

* In words, if a and b are boolean values, then a || b (read as *a or b*) is true if and only if either a *or* b *or* both are true.

Note that, if a is known to be true, then the value of a || b must be true, regardless of b's value. Consequently, Java uses a form of short-circuit evaluation in this case too. Here, however, it looks for a value of true rather than a value of false (as in the case of &&-expressions). We explain the short-circuit procedure directly in the context of expressions that are obtained by "or"ing together several boolean expressions. Here is one, for example, that comprises four boolean expressions:

a == 1 || a == 2 || a > 7 || a < -10

To evaluate such an expression, Java works from left to right, evaluating one boolean expression at a time, until either

* it finds an expression with a value of true — in which case evaluation ceases and the whole expression evaluates to true — or
* all the boolean expressions have been evaluated without any of them having a value of true — in which case the whole expression evaluates to false.

In determining the order of precedence when evaluating complex expressions, logical operators have lower rank than relational operators. But among the logical operators, the unary operator, !, takes precedence over the binary operators && and ||

The relationships between && and || are summarized by [De Morgan's Laws](javascript:void(0);):

* The value of !(a && b) == (!a || !b) is always true.  
  In words, if not both a and b are true, then either a is false or b is false or both.
* The value of !(a || b) == (!a && !b) is always true.  
  In words, if neither a nor b is true, then both a and b must be false.

De Morgan's Laws may be used to "distribute" the ! operator. This often makes the expression easier to understand. Consider, for example, the expression !(x == 5 || x == 6). By the second of De Morgan's Laws this expression has the same value as !(x == 5) && !(x == 6). In this case, we can further simplify the expression by using the not equals arithmetic operator !=, obtaining the expression x != 5 && x != 6.