Chapter 3

Making Choices

In this chapter we show how to make choices in Java using the **if** statement. *Conditional* statements like the **if** statement are programming constructs capable of choosing between blocks of code to execute. These statements provide enormous expressive power to programmers. Despite their strength, however, these statements are very easy to use and understand because they mimic the way we think. For instance, the following sentences form a conditional statement in English:

```
``If it's sunny outside then we will play frisbee.
  Otherwise we will play cards.''
```

An example of a conditional statement that is a bit more relevant to our concerns with programming might be:

```
``If the mouse location is contained in the rectangle then increment counter1.
  Otherwise increment counter2.''
```

In Java, the **if** statement is the most commonly used conditional statement. It comes in a variety of forms that we will explore, each of which is useful for certain situations. With the help of conditionals we will write programs to determine a win or loss in the game of craps and to figure out what to do on a weekend based on the weather and your finances.

After presenting a brief example illustrating the use of the **if** statement, we formally introduce several of its variations that will allow us to handle more complex situations. We also introduce the `boolean` data type for expressions that can be either *true* or *false*. Finally, we provide advice on how to use conditionals clearly and effectively so that your programs can be understood correctly by the Java compiler and, more importantly, by other programmers.

3.1 A brief example: using the if statement to count votes

We illustrate the use of conditionals with a simple example of a program to count votes in an election. To accomplish this we will divide the program's canvas in half vertically so that the left and right sides represent candidates A and B respectively (see Figure 4.1). A mouse click on either side will be treated as a vote for that half's candidate.

Recall the program `ClickCount`, which keeps track of the number of times the user has clicked the mouse (see chapter ???). The code for its `onMouseClick` method is given below:

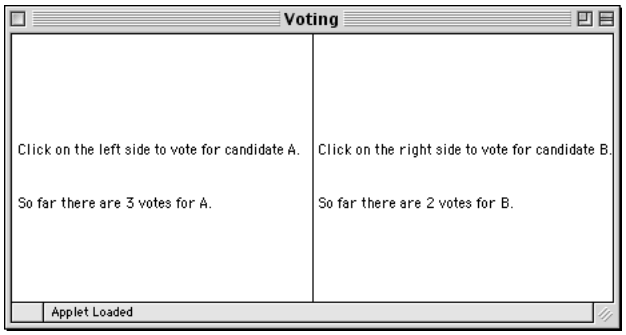

Figure 3.1: Screen shot of Voter program.

```
public void onMouseClick( Location point ) {
   count++;
   display.setText("You have clicked " + count + " times.");
}
```

Whereas this method simply counts *all* mouse clicks on the canvas, by using the **if** statement our voting program's `onMouseClick` method will be able to discriminate between votes for candidate A or B.

The code for class `Voting` is given in Figure 4.2. The `begin` method should be familiar by now. It creates four new `Text` objects, the top two of which display voting instructions while the bottom two display the current tally for each candidate. The last line of the method draws the vertical line dividing the canvas.

The `onMouseClick` method, on the other hand, contains the new programming construct. The **if** statement is used to determine which candidate gets each vote and then updates the appropriate `Text` object to display the new total. To decide who receives each vote, the program compares the x-coordinate of the location of the mouse click to the middle of the canvas. Note that we have defined the constant `MID_X` to refer to the x-coordinate of the middle of the canvas.

The **if** statement allows the programmer to make choices about which statements are executed in a program based on a *condition*. In this case, when the mouse is clicked the program must decide whether to give a vote to candidate A or B. The *condition* is whether the x-coordinate of the mouse click, obtained by evaluating `point.getx()`, is less than `MID_X`. The condition is written in Java as `point.getX() < MID_X`. If the condition is *not* satisfied (*i.e.*, if `point.getX()` is greater than or equal to `MID_X`) then the code after the `else` keyword is executed.

The two lines of code following the line containing the **if** are grouped together by a pair of matching curly braces. A series of statements surrounded by curly braces in this manner is called a *block*. Similarly, the two statements immediately following the `else` also form a block. If the *condition* of an **if** statement is true, the block of statements immediately after the condition is executed. Otherwise the block immediately after the `else` is executed.

In the rest of this chapter we will formally define the **if** statement and some of its variations as well as provide examples using more complex conditional statements.

3.2 if statements

Now that we have seen the **if** statement in action, let's carefully examine its syntax and meaning.

```
public class Voting extends WindowController {
    // coordinates of canvas, including x-coord of middle
    private static final int MID_X = 300;
    private static final int TOP = 0;
    private static final int BOTTOM = 400;

    // x-coordinates of A and B text messages
    private static final int TEXT_A_X = 20;
    private static final int TEXT_B_X = MID_X + 20;

    // y coordinates of instructions and vote count info
    private static final int INSTRUCTION_Y = 180;
    private static final int DISPLAY_Y = 220;

    private int countA = 0;       // number of votes for A
    private int countB = 0;       // number of votes for B

    private Text infoA;           // display of votes for A
    private Text infoB;           // display of votes for B

    // Create displays with instructions on how to vote
    public void begin() {
        new Text( "Click on the left side to vote for candidate A.",
                              TEXT_A_X, INSTRUCTION_Y, canvas );
        new Text( "Click on the right side to vote for candidate B.",
                              TEXT_B_X, INSTRUCTION_Y, canvas );
        infoA = new Text( "So far there are " + countA + " vote(s) for A.",
                              TEXT_A_X, DISPLAY_Y, canvas );
        infoB = new Text( "So far there are " + countB + " vote(s) for B.",
                              TEXT_B_X, DISPLAY_Y, canvas );
        new Line( MID_X, TOP, MID_X, BOTTOM, canvas );
    }

    // Update votes and display vote counts
    public void onMouseClick( Location point ) {
        if ( point.getX() < MID_X ) {
            countA++;
            infoA.setText( "So far there are " + countA + " vote(s) for A." );
        }
        else {
            countB++;
            infoB.setText( "So far there are " + countB + " vote(s) for B." );
        }
    }
}
```

Figure 3.2: Code for Voting class.

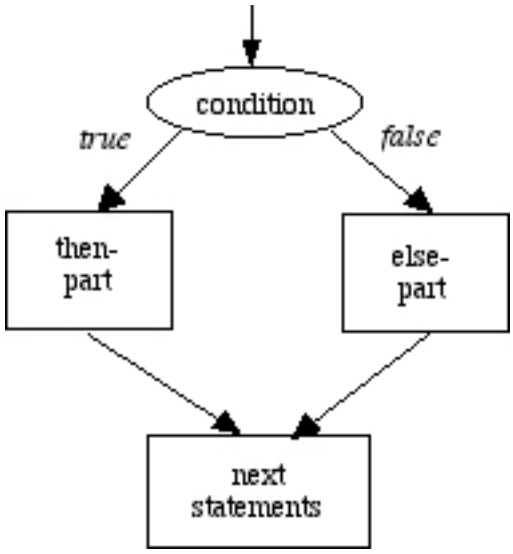

Figure 3.3: Semantics of the **if-else** statement.

The code in the `Voting` class example given in Figure 4.2 contains a form of conditional statement called the **if-else** statement. Its syntax is:

```
if ( condition ) {
   statements  // if-part, executed when condition is true
}
else {
   statements  // else-part, executed when condition is false
}
```

We've included comments to make it clear when each part is executed, even though these are not part of the formal syntax. The occurrences of *statements* in the syntax represent sequences of statements in Java.

When the **if-else** statement is executed, the computer determines whether *condition* is true. If so, it executes the statements in the block of code surrounded by the first pair of curly braces, "{" and "}", called the **if-part**, and then skips over the rest of the statement. Otherwise (*i.e.*, if *condition* is false), it skips over the **if-part** and will instead execute the block of statements after the `else` keyword, called the **else-part**.This execution sequence is illustrated in Figure 3.3.

Therefore, when the **if-else** statement is executed, exactly one of the two blocks of code is processed. When that block is completed, execution resumes immediately after the **if-else** statement. The following example illustrates this behavior.

Suppose we want to modify our `Voting` program so that it always displays the total number of votes. Let `infoTotal` be a variable of type `Text` that has been initialized in the `begin` method. Here is a revised version of `onMouseClick` that displays the current vote total:

```
// Update votes and display vote counts
public void onMouseClick( Location point ) {
    if ( point.getX() < MID_X ) {
```

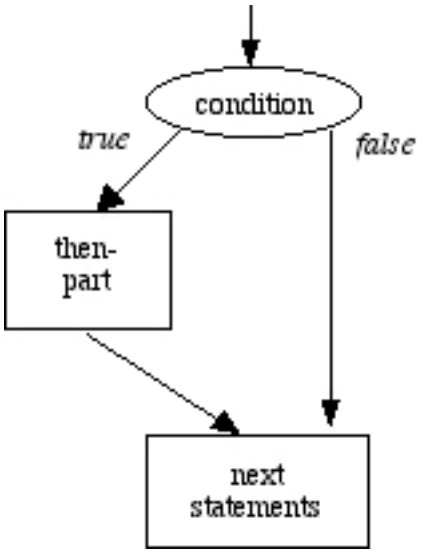

Figure 3.4: Semantics of the `if` with no `else`.

```
        countA++;
        infoA.setText( "So far there are " + countA + " votes for A." );
    }
    else {
        countB++;
        infoB.setText( "So far there are " + countB + " votes for B." );
    }

    infoTotal.setText( "Votes so far: " + ( countA + countB ) );
}
```

Each time the user clicks on the canvas, the line sending the `setText` message to `infoTotal` will be executed regardless of which half of the screen the mouse was clicked on.

This seems pretty clear, but there are many situations in which we don't need an **else-part**. Fortunately, there is a simple variant of the **if-else** statement, the **if** statement, which does not have the `else` keyword or the **else-part**.

```
if ( condition ) {
   statements  // if-part, executed when condition is true
}

next statement  // executed after the if statement
```

If *condition* is true, then the **if-part** is executed as before. If it is false, however, the program simply moves on to the *next statement* after the **if-part** because there is no **else-part** to execute. The execution sequence is illustrated in Figure 3.4.

3.2.1 Example: using the if statement with 2-D objects

Suppose we want to write a program that begins by displaying a square on the canvas. If the user clicks inside the square then the computer moves the square 50 pixels to the right. If the click is not inside the square, nothing happens.

Any attempt to write a program that satisfies this specification requires an answer to the following question: How can we determine if a point is inside a square? While we could compare the coordinates of the point to the locations of the left, right, top, and bottom edges of the square, we are saved that effort because all of the two dimensional geometric objects (*e.g.*, `FramedRect`, `FilledRect`, `FramedOval`, `FilledOval`) provide the method `contains` that does this for us.

For example, let `square` be a variable of type `FramedRect`, and let `point` be a variable of type `Location`. Then the expression `square.contains( point )` will be *true* if the object held in `square` *contains* the `point`, and *false* otherwise. In our example, `geomObj` is a square and `point` is the location of the mouse click.

For the sake of brevity we will only write out the `onMouseClick` method of this program. We assume that `square` has been declared elsewhere to be a variable of type `FramedRect` and that it is initialized in the `begin` method.

```
public void onMouseClick( Location point ) {
    if ( square.contains( point ) ) {
        square.move( X_OFFSET, 0 );
    }
}
```

An English translation of the above method would read: "*If* the `point` where the mouse was clicked is contained in the `square` *then* move the `square` to the right by `X_OFFSET` pixels. Otherwise do nothing." (Of course, the computer code doesn't say, "Do nothing." Instead it simply omits the **else-part**. At run-time if the condition fails then execution simply proceeds to the next statement after the **if-statement**.)

We will present further variations of **if** statements later in this chapter, but first we will first explore the kinds of expressions that can be used to form the *condition* part of these statements.

3.3 Understanding conditions

Comparison operators like "<" are used in expressions that evaluate to either *true* or *false*. Java contains several comparison operators. They include:

```
< , > , ==, <=, >=, !=
```

We must use `==` to test for equality because `=` has already been used as the assignment operator. The symbol `!=` stands for inequality. Because keyboards do not generally include the symbols $\leq$ or $\geq$, Java uses the combinations `<=` and `>=` in their places.

Using these comparison operators, we can write `x > 4`, `y != z+17`, and `x+2 <= y`. Depending on the values of the variables, each of these expressions will evaluate to either *true* or *false*. Suppose the current value of `x` is 3, `y` is 6, and `z` is -10. Here are the results of evaluating the above expressions:

- `x > 4` is *false* because 3 is not greater than 4.

- `y != z+17` is *true* because 6 is different from -10+7.
- `x+2 <= y` is *true* because 3+2 is less than or equal to 6.

Exercise 3.3.1 *Suppose the current value of* `x` *is 2,* `y` *is 4, and* `z` *is 15. For each of the following conditions, determine whether they evaluate to true or false.*

1. `x + 2 < y`.

2. `z - 3 * x != y + 5`.

3. `x * y == z - 9`.

4. `z >= 3 * y`.

Before we consider more complex conditions for the **if** statement, we first step back and look at these expressions in a slightly different way.

3.3.1 The `boolean` data type

Java contains a primitive data type called `boolean`. Unlike other primitive types, such as `int`, that have a large number of elements, the `boolean` type has only two values: *true* and *false*. However, just as one can write down integer values directly as `17`, `-158`, or `47`, we can write down boolean values directly in Java as `true` or `false`. We can also declare variables of type `boolean`.

There are a large number of expressions in Java that return values of type `boolean`. As we have just seen, expressions consisting of two integer valued expressions separated by one of the comparison operators, `<`, `>`, `<=`, `>=`, `==`, and `!=`, return a result of type `boolean`. We have also seen the method `contains` that returns a value of type `boolean`.

Example: analyzing `WhatADrag` We will use both the `contains` method and `boolean` variables in the implementation of a new class `WhatADrag`. The complete code listing for this program is given in Figure 3.5. The program begins by displaying a box on the screen. If the user presses the mouse down while it is pointing inside of the box and then drags the mouse, the box will follow the mouse on the canvas. If the mouse is not pointing in the box when the mouse button is pressed then dragging the mouse will have no effect, even if the mouse happens to cross the box at some point during the drag.

Let's look at the code in the mouse handling methods to see how we can program this behavior. As soon as the mouse button is pressed, the `onMousePress` method is executed. The first assignment in `onMousePress`,

```
lastPoint = point;
```

results in saving the current location of the mouse (as held in parameter `point`) as the value of variable `lastPoint`. The location is saved to be used when `onMouseDrag` is executed later.

The second assignment in `onMousePress`,

```
boxGrabbed = box.contains( point );
```

determines and then remembers whether the `box` contained the `point` where the mouse was initially pressed. Because the result of evaluating `box.contains( point )` is a value of type `boolean`, it can be stored in `boxGrabbed`, a variable of type `boolean`. The value of `boxGrabbed`, which reflects

```
public class WhatADrag extends WindowController {
    ...   // Constant declarations omitted

    private FilledRect box;     // box to be dragged

    private Location lastPoint;   // point where mouse was last seen

    // whether the box has been grabbed by the mouse
    private boolean boxGrabbed = false;

    // make the box
    public void begin() {
        box = new FilledRect( START_LEFT, START_TOP,
                              BOX_WIDTH, BOX_HEIGHT, canvas );
    }

    // Save starting point and whether point was in box
    public void onMousePress( Location point ) {
        lastPoint = point;
        boxGrabbed = box.contains( point );
    }

    // if mouse is in box, then drag the box
    public void onMouseDrag( Location point ) {
        if ( boxGrabbed ) {
            box.move( point.getX() - lastPoint.getX(),
                      point.getY() - lastPoint.getY() );
            lastPoint = point;
        }
    }
}
```

Figure 3.5: Code for dragging a box.

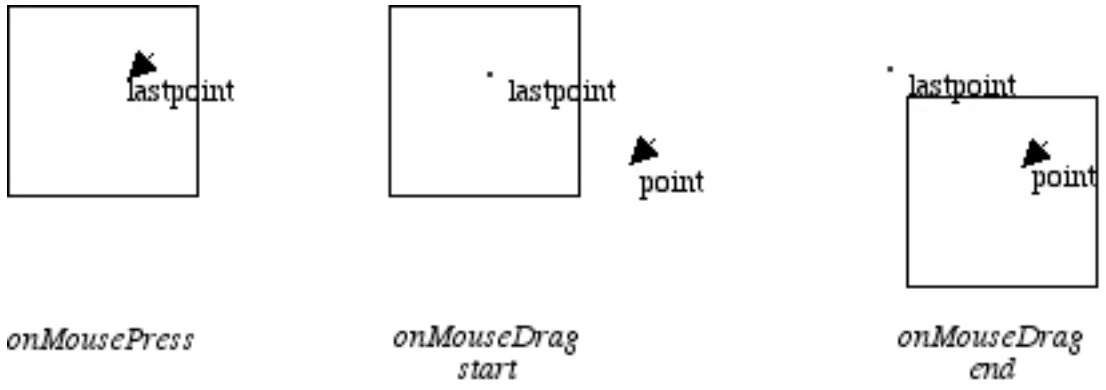

Figure 3.6: Three stages of dragging a rectangle.

whether or not the user actually pressed the mouse down inside `box`, will be used in `onMouseDrag` to determine whether or not the `box` should be moved when the user drags the mouse.

As usual, the `onMouseDrag` method will be executed when the user drags the mouse. At that time the values of the variables `boxGrabbed` and `lastPoint` become relevant.

If the value of `boxGrabbed` is true, `box` will be moved by the distance between the last location of the mouse, saved in `lastPoint`, and the current position, held in `point`. The distance in each of the horizontal and vertical directions is needed to actually perform the move. The horizontal and vertical distances are obtained by evaluating `point.getX() - lastPoint.getX()` and `point.getY() - lastPoint.getY()`, respectively.

Figure 3.6 illustrates three stages of dragging `box` in class `WhatADrag`. In the leftmost picture, the mouse button has been pressed with the mouse inside the rectangle. After the execution of `onMousePress`, the location of the mouse is stored in instance variable `lastPoint`. In the middle picture, the mouse has been dragged down and to the right, and `onMouseDrag` has just begun execution. The current location of the mouse is held in the parameter, `point`, but the **if** statement has not yet been executed. The rightmost picture shows what has happened immediately after the `move` message has been sent to `box` in the **if** statement during the execution of `onMouseDrag`. The rectangle has been dragged to the right and down by the difference between the x-coordinates and by the difference between the y-coordinates of `point` and `lastPoint`. The update of the value of `lastPoint` to the location held in `point` is not shown.

Exercise 3.3.2 *Rather than saving the results of* `box.contains( point )` *in a* `boolean` *variable, we could have simply sent the* `contains` *message to* `box` *every time we dragged the mouse.*

```
public void onMouseDrag( Location point ) {
    if ( box.contains(point) ) {
        box.move( point.getX() - lastPoint.getX(),
                  point.getY() - lastPoint.getY() );
        lastPoint = point;
    }
}
```

However, this would have made the computer do more work in rechecking whether `box` *contains* `point` *each time* `onMouseDrag` *is executed, rather than only when the mouse button is pressed.*

(Typically, `onMouseDrag` *will be executed many times between executions of* `onMousePress` *if the user is dragging the mouse.)*

Beyond the difficulty of making the `onMouseDrag` *method more complex, using the* `contains` *method as part of the condition for the* **if** *statement is also tricky. Please explain why having the test be* `box.contains( point )`*, as written above, produces a different result from using* `box.contains( lastPoint )` *in the same* **if** *statement? Which of these produces exactly the same behavior as the* `onMouseDrag` *method in the* `WhatADrag` *class? In what circumstances and how does the other one behave differently?*

Exercise 3.3.3 *Suppose we replace method* `onMouseDrag` *in class* `WhatADrag` *by the following code:*

```
// if mouse is in box, then drag the box
public void onMouseDrag( Location point ) {
    if ( boxGrabbed ) {
        box.moveTo( point );
    }
}
```

This is simpler than the code in Figure 3.5. For example, we no longer need to keep track of `lastPoint`*. However, it does not result in as nice behavior. Explain why!*

Exercise 3.3.4 *Suppose the statement* `lastPoint = point;` *inside of method* `onMouseDrag` *in Figure 3.5 was placed after the end of the* **if** *statement rather than inside the* **if-part***. Would it make any difference in the visible behavior of the program? If so, how would it change the appearance on the screen during execution? If not, why put the statement inside of the* **if-part***?*

3.3.2 Comparing objects versus comparing primitive types

Comparisons of objects generated from classes raise more subtle issues than comparisons of values of primitive types like `int` and `double` in Java. With objects, the "`==`" operator checks to see if two expressions refer to exactly the same object rather than to similar objects. An example will make the differences clear.

Recall the discussion from 2.9 on the difference between numbers and object types. There it was emphasized that an assignment statement between variables of object types results in both variables referring to the same object.

Suppose we execute the following three statements:

```
FramedRect firstVar = new FramedRect(25,25,100,100,canvas);
FramedRect secondVar = new FramedRect(25,25,100,100,canvas);
FramedRect otherVar = firstVar;
```

The first two statements generate two new `FramedRect` objects that happen to be at the same location and have the same width and height. The third assignment results in `otherVar` referring to the same `FramedRect` as `firstVar`.

Which of these variables should be treated as being equal? Certainly `otherVar` and `firstVar` should be equal because they refer to exactly the same object. However, `firstVar` and `secondVar` refer to different objects that happen to be at the same location and have the same size. Hence they are in some sense equivalent, even though they are not the same objects.

In particular, suppose we execute the following statements:

```
firstVar.move(5,10);
```

Then `firstVar` and `otherVar` still refer to the same object, but `firstVar` and `secondVar` are no longer in the same place. Thus the relationship between the values of `firstVar` and `otherVar` is stronger than the relationship between the values of `firstVar` and `secondVar` before they were moved. A good analogy is to think of `firstVar` and `otherVar` as being two handles attached to the same framed rectangle. If the framed rectangle is moved or resized, both continue to refer to it. Because `secondVar` is attached to a different `FramedRect` at the same location, if the first `FramedRect` is moved, the other will stay where it is.

Because there are two reasonable definitions of equality between objects, Java supports two different ways of comparing objects. The "`==`" operator is true between object expressions if the expressions refer to the same object. Thus after executing the three assignment statements above, `firstVar == otherVar` will evaluate to true, while `firstVar == secondVar` will return false. Of course `firstVar == otherVar` also remains true after sending a `move` message to `firstVar`.

All objects also support an `equals` method that can be used for the weaker sort of equality. After the three assignments above, all three of `firstVar.equals(secondVar)`, `firstVar.equals(otherVar)`, and `secondVar.equals(otherVar)` will be true. After sending a `move` message to `firstVar`, the expression `firstVar.equals(otherVar)` remains true, but the comparisons with `secondVar` both become false because the `FramedRect` referred to by `firstVar` and `otherVar` is now at a different location from the `FramedRect` referred to by `secondVar`.

Most of the time you will want to use the `equals` method rather than "`==`" when comparing objects. This is especially important with strings. If `string1` and `string2` refer to two different string objects that both hold the same sequence of characters, then `string1 == string2` would return false, but `string1.equals(string2)` would return true.

The only time it is appropriate to use `==` with objects is when you want to know if the expressions refer to the same objects rather than to objects with equivalent features.

Here are some simple rules of thumb:

- When comparing objects, you should almost always use the `equals` method rather than "`==`". In particular, always use `equals` with strings.
- Only use "`==`" with objects if you really want to know whether they are the same object, rather than that they represent objects that look or behave the same.
- You cannnot send an `equals` method to an integer or double because they are not objects. Thus "==" is always the correct thing to use in testing numbers for equality.

3.4 Selecting among many alternatives

The **if-else** statements discussed earlier in this section are ideally suited when we have to choose between executing two different blocks of statements at some point in a program. However, sometimes there are more than two alternatives that have to be considered.

A simple example might include assigning letter grades based on numeric scores on an examination. Scores greater than or equal to 90 are assigned an "A," scores from 80 to 89 (inclusive) are assigned a "B," scores from 70 to 79 are assigned a "C," and those below 70 are assigned "no credit." Suppose that a variable `score` of type `int` contains the numeric score of a particular examination. We would like to display the appropriate letter grade in a `Text` item, `gradeDisplay`. In this situation we have not just two, but *four* different possibilities to worry about, so it is clear that a simple **if-else** statement is insufficient.

Java allows us to extend the **if-else** statement for more than two possibilities by including one or more `else if` clause(s) in an **if** statement. Thus we can display the appropriate grade using the following statement:

```
if ( score >= 90 ) {
    gradeDisplay.setText( "The grade is A" );
}
else if ( score >= 80 ) {
    gradeDisplay.setText( "The grade is B" );
}
else if ( score >= 70 ) {
    gradeDisplay.setText( "The grade is C" );
}
else {
    gradeDisplay.setText( "No credit is given");
}
```

When we execute this **if** statement, the computer first evaluates the `boolean` expression `score >= 90`. If that is true, an "A" will be displayed and execution will continue after the last **else-part** of the statement. However, if it is false the program will evaluate the next `boolean` expression, `score >= 80`. If that is true, a grade of "B" will be displayed and execution will continue after the last **else-part**. If not, the expression `score >= 70` will be evaluated. If that is true then a grade of "C" will be displayed and execution will continue after the **else-part**. Otherwise, the statement in the **else-part** will be executed and a grade of "no credit" will be displayed.

This seems pretty intuitive, but we must wonder why, when checking to see if the student should be given a "B," we didn't also have to check whether `score < 90`. The reason is that to get to the condition `score >= 80`, the previous test, `score >= 90`, must have already *failed*. That is, we *only* execute the **else** clauses if `score` is less than 90. The same reasoning shows that we do not need to check if `score < 80` when we determine whether to give a "C" as the grade. We can always count on the fact that the conditions within the **if-else** statements are evaluated sequentially, and therefore that all previous tests in the **if** statement have failed before determining whether to execute the next block.

We can summarize the execution of an **if** statement including `else if`'s as follows:

- Evaluate the conditions after the `if`'s in order until one is found to be true.
- Execute the statements in the block following that `if` and then resume execution with the first statement after the entire **if** statement.
- If none of the conditions are true and there is an **else-part**, then execute the statements in the **else-part**. If there is no **else-part**, don't execute any of the statements in the **if** statement.
- Resume execution with the first statement after the **if** statement.

If there is an `else` clause in an **if** statement, then it must be the very last part of the **if**. That is, no further `else if`'s are allowed after a plain `else` clause.

We will see later that this more complex **if** statement is actually a special case of a more general way of *nesting* **if** statements. However, it is convenient for the moment to consider it by itself.

3.5 More on boolean expressions

The **if-else** statement in the `Voting` class at the beginning of this chapter worked very well because there were only two candidates to consider when assigning a new vote. However, for more than two candidates the `else if` clause introduced in the last section becomes necessary.

In the next example our program must choose between 3 candidates, A, B, and C, who each have been allotted a vertical third of the canvas. We will not rewrite the entire program here, but will instead focus on the `onMouseClick` method.

Let `LEFT_SEP` and `RIGHT_SEP` be constants representing the x-coordinates of the vertical lines that divide the canvas into the three pieces. The `int` variables `countA`, `countB`, and `countC` will keep track of the number of votes for the three candidates. Here is the code:

```
// Update votes and display vote counts for 3 candidates
public void onMouseClick( Location point ) {
    if ( point.getX() < LEFT_SEP ) {        // clicked in left section
        countA++;
        infoA.setText( "So far there are " + countA + " votes for A." );
    }
    else if ( point.getX() < RIGHT_SEP ) {  // clicked in center
        countB++;
        infoB.setText( "So far there are " + countB + " votes for B." );
    }
    else {                                  // clicked in right section
        countC++;
        infoC.setText( "So far there are " + countC + " votes for C." );
    }
}
```

When determining whether the click was in the center section, why didn't we have to check that `point.getX()` was to the right of `LEFT_SEP`? As with our earlier example, the reason is that to even get to the second condition we *know* that the test `point.getX() < LEFT_SEP` must have failed (that is, it must have evaluated to *false*).

This same style solution works for determining whether clicks are in 4 or more vertical strips. However, once we get to four candidates it might make sense to divide the screen both vertically and horizontally rather than into 4 narrow, vertical strips. A picture is given in Figure 3.7.

Suppose we draw vertical and horizontal lines bisecting the canvas. We will count clicks in the upper-left hand corner as votes for A, in the upper-right hand corner as votes for B, lower-left for C, and lower-right for D. Thus to be a vote for A, the location where the user clicked must be *both* above the horizontal line *and* to the left of the vertical line. How can we write this as a condition?

In Java we use the *and* operator, `&&`, between two `boolean` expressions to indicate that *both* must be true for the entire expression to be true. For example, we can ensure that `x` is postive and `y` is negative by writing `x > 0 && y < 0`.

The `&&` is an example of a logical, or `boolean`, operator in Java. Just as the "+" operator on `int`s takes two `int` values and returns an `int` value, `boolean` operators take two `boolean` values and return a `boolean` value. `&&` returns true exactly when both `boolean` values are true. Thus `x > 0 && y < 0` will be true only if *both* `x` is greater than 0 *and* `y` is less than 0.

In order to ensure that `point` is in the upper-left corner of the canvas we would need to write:

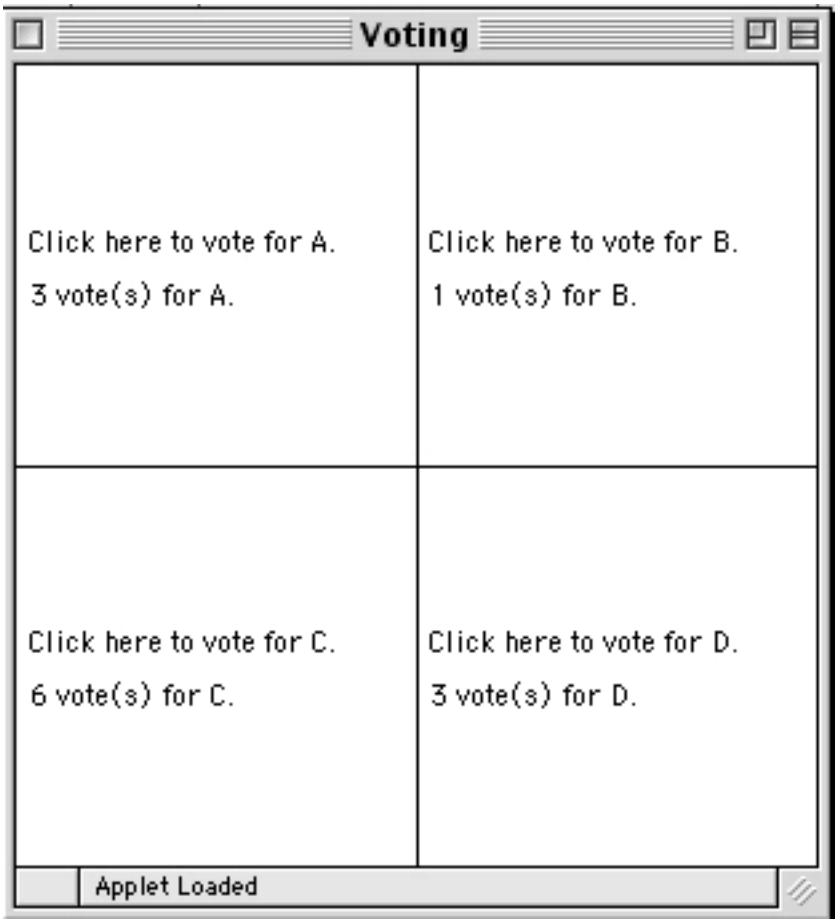

Figure 3.7: Voting for four candidates.

```
if ( point.getX() < MID_X && point.getY() < MID_Y ) { // upper-left
    ...
}
```

The **if-part** is only executed if *both* `point.getX() < MID_X` *and* `point.getY() < MID_Y`. If either one of those `boolean` expressions is false, then the entire condition evaluates to false and the **if-part** is not executed.

Here is the complete code to assign votes to four candidates:

```
// Update votes and display vote counts
public void onMouseClick( Location point ) {
    if ( point.getX() < MID_X && point.getY() < MID_Y ) {       // upper-left
        countA++;
        infoA.setText( "So far there are " + countA + " votes for A." );
    }
    else if ( point.getX() >= MID_X && point.getY() < MID_Y ) { // upper-right
        countB++;
        infoB.setText( "So far there are " + countB + " votes for B." );
    }
    else if ( point.getX() < MID_X && point.getY() >= MID_Y ) { // lower-left
        countC++;
        infoC.setText( "So far there are " + countC + " votes for C." );
    }
    else {                                                      // lower-right
        countD++;
        infoD.setText( "So far there are " + countD + " votes for D." );
    }
}
```

Unfortunately computer programming languages do not usually allow us to combine two inequalities, as in the expression `1 <= x <= 10`. Instead we must write it out as the combination of the tests `1 <= x` and `x <= 10`.

Just as Java uses `&&` to represent the logical *and* operation, it uses `||` to represent the logical *or* operator. Thus `x < 5 || y > 20` will be true if `x < 5` *or* `y > 20`. In general, if b_1 and b_1 are boolean expressions, then b_1 `||` b_2 is true if *one* or *both* of b_1 and b_2 are true.

A good example illustrating the use of the *or* operation is determining whether someone playing the game of craps has won or lost after his or her first roll of the dice. The "shooter" in craps throws two dice. If the numbers on the face of the dice add up to 7 or 11, then the shooter wins. A sum of 2, 3, or 12 results in an immediate loss. With any other result, play continues in a way that will be described later in the chapter.

The **if** statement below has three branches that encode the relevant outcomes of the first roll of the dice. The value of `status` is simply a `Text` object that, as usual, has been created in the `begin` method of the program.

```
if ( roll == 7 || roll == 11 ) {  // 7 or 11 wins on first throw
    status.setText( "You win!" );
}
else if ( roll == 2 || roll == 3 || roll == 12 ) {  // 2, 3, or 12 loses
    status.setText( "You lose!" );
```

operator	meaning
`&&`	*and*
`||`	*or*
`!`	*not*
`==`	*equal*
`!=`	*not equal*
`<`	*less than*
`<=`	*less than or equal*
`>`	*greater than*
`>=`	*greater than or equal*

Figure 3.8: A summary of the `boolean` and comparison operators in Java

```
}
else {                                  // play must continue
    status.setText( "The game continues" );
}
```

The `if` portion determines if the player has won by checking if the `roll` was 7 or 11. The `else if` portion determines if the player has lost by checking if `roll` was 2, 3, or 12. Finally the `else` portion is executed if further rolls of the dice will be required to determine whether the player wins or loses.

The last `boolean` operator to be introduced here is `!`, which stands for "not". For example, the expression `!box.contains( point )` will be true exactly when `box.contains( point )` is false, *i.e.*, when `box.contains( point )` is *not* true.

Although we can use `!` with equations and inequalities, it is usually clearer to rewrite the statement using a different operator. For instance, `!( x == y )` is more simply written as `x != y`, and `!( x < y )` is simplified to `x >= y`. Similarly, `!( x <= y )` is equivalent to `x > y`, which is much easier to read.

Figure 3.8 summarizes the most common operators in Java that give a `boolean` result.

Exercise 3.5.1 *Suppose that the current value of* `x` *is 6,* `y` *is -2, and* `z` *is 13. For each of the following conditions, determine whether they evaluate to true or false.*

1. `x - 6 < y && z == 2 * x + 1`.

2. `!(x - 6 < y && z == 2 * x + 1)`.

3. `x - 6 < y || z == 2 * x + 1`.

4. `!(x - 6 < y || z == 2 * x + 1)`.

Before we move on, we should note a few last points about `&&` and `||`. The first is to be sure and use the double version of each of the symbols `&` and `|` because the single versions represent quite different operators. Unfortunately, your Java compiler is unlikely to warn you if you make this mistake and will instead let you *discover* it (if you are lucky!) through nasty runtime errors.[1]

[1]Runtime errors are errors that occur while the program is executing. We prefer to have errors reported by the compiler because they are much easier to find and fix than runtime errors. Runtime errors will frequently just cause your program to crash with a poor explanation, if any, describing why it died, so be careful!

Second, both `&&` and `||` in Java are implemented as "short circuit" operations. What this means is that Java will cease evaluating an expression involving one of these operators as soon as it can determine whether the entire expression is true or false.

For example, suppose a program includes a declaration of an `int` variable, `x`, and that it contains the expression `(x > 10) && (x <= 20)`. If the computer evaluates that expression when `x` has value 3, it will only evaluate `x > 10` without even considering `x <= 20`. Because `x > 10` is false, Java knows that the final value of the `&&` expression must be false no matter what the value of `x <= 20`. However, if `x > 10` had been true the rest of the expression would have had to be evaluated to determine whether the entire `&&` expression evaluates to true or false.

The `||` expression behaves in exactly the opposite manner. If the left side evaluates to true then Java knows that the entire `||` expression *must* evaluate to true, so it does not bother to evaluate the right side. Conversely, if the left side is false then the right side must be evaluated in order to determine the final value of the entire `||` expression.

At this point, the only impact of the "short circuit" evaluation of `boolean` expressions is that your program may run a little faster. However, later we will see examples where short circuit evaluation of conditional expressions can have important consequences on the correctness of your program.

3.6 Style guidelines for if statements

In this section we provide some style guidelines for the most appropriate use of **if** statements. The "bad" examples we illustrate are legal Java code. However, they can be replaced by much simpler and easier to understand code.

It is extremely important to write clear and concise code not only for the sake of others who might read it (including your professors!), but also for *yourself*. Programmers sometimes waste a great deal of time trying to understand code that they wrote themselves months, weeks, or even days earlier because they did not pay much attention to clarity and style. As a result we consider it essential to learn and use good coding style.

Avoid empty if-parts. We mentioned earlier that an **if** statement can omit the `else-part` if it is not needed. However, what if we need the `else-part`, but not the **if-part** of an **if** statement? For example, suppose we want to increment variable `counter` exactly when `point` is not contained in `box`, but do nothing otherwise. We could write:

```
if ( box.contains( point ) ) {
   // do nothing here
}
else {
   counter++;
}
```

However, writing it that way is considered very bad style. We should instead rewrite it as:

```
if ( !box.contains( point ) ) {
   counter++;
}
```

Here we have used `!` to negate the value of the condition so the statement we wish to execute now belongs in the **if-part**. Notice that this version of the statement is much shorter and simpler. In general, we will be looking for short, clear ways of writing code in order to make it more understandable to a reader.

Don't be afraid to use boolean expressions in assignments. Recall the class `WhatADrag` contained in Figure 3.5. In that program, `box` was a `FilledRect`, `point` was a `Location`, and `boxGrabbed` was a `boolean` variable.

Many beginning programmers are more comfortable writing

```
if (box.contains( point )) {
    boxGrabbed = true;
} else {
    boxGrabbed = false;
}
```

rather than the code we actually wrote in that program

```
boxGrabbed = box.contains( point );
```

However, you should become comfortable enough with boolean variables, that this kind of assignment makes sense to you. The assignment is simpler and easier to understand than the **if-else** statement for an experienced Java programmer.

Don't use true or false in conditions Again, let `boxGrabbed` be a `boolean` variable. Suppose you want the program to do something exactly when `boxGrabbed` is true. Look at the following code excerpt:

```
if ( boxGrabbed == true ) {
    ...
}
```

Can you determine why this code is considered bad style? The code can, and should, be written more simply as:

```
if ( boxGrabbed ) {
    ...
}
```

Both code segments do exactly the same thing. However the first is overly verbose. It is like the difference between saying "if it is true that it is raining then I will stay home" and "if it is raining then I will stay home". The second version is clearly preferable.

Similarly, rather than writing

```
if ( boxGrabbed == false ) {
    ...
}
```

or

```
if ( boxGrabbed != true ) {
    ...
}
```

we prefer

```
if ( !boxGrabbed ) {
    ...
}
```

Again, this is like someone saying "if it is false that it is raining then ..." or "if it is not true that it is raining then ..." rather than "if it is not raining then ...".

A good general rule is to avoid using either `true` or `false` in a condition, as their presence represents verbose and redundant code. The literals `true` and `false` are most commonly used to initialize `boolean` variables, and never in conditional expressions.

Warning about == versus =. One of the most common errors made by beginning Java programmers (and even many experienced programmers) is to use "=" in a condition where "==" is intended. Most of the time Java will produce a compile-time error message warning that the left-hand side of the condition cannot be converted to a `boolean`. Recognize that when this happens the most likely cause is that you have used an assignment operator, "=" rather than a comparison operator "==".

There is one case, however, where Java will not find your error! That is when you are comparing two `boolean` expressions. For example, if you write:

```
if (boxGrabbed = true) { ... }
```

or

```
if (boxGrabbed = false) { ... }
```

Java will not give you an error message. In the first case, it will treat `boxGrabbed = true` as an assignment statement, and it will set the value of `boxGrabbed` to be true. It will then check to see if `boxGrabbed` is true, and of course it will be. Because `boxGrabbed` is true, the computer will execute the **then-part** if the **if** statement. In the second case above, `boxGrabbed` will be assigned the value false, and, because the value of `boxGrabbed` is false, it will always skip the **else-part** of the **if** statement.

Thus in the first case, it will *always* execute the **then-part** of the **if** statement, while in the second, it will alway skip the **then-part**. This was surely not what was intended, but Java will not give you an error message because `boxGrabbed` is itself a `boolean` variable.

We do not want to go into all of the details here, but the root of the problem is that Java allows assignment statements in the middle of expressions. The value obtained from an assignment statement used in an expression is the value of the variable *after* the assignment. Thus, in each of the two examples above, the value used as the condition in the **if** statement is the value of `boxGrabbed` after the assignment.

Of course this is all extremely confusing, and virtually always represents an error. Fortunately, if you follow the rule we suggested in never including `true` or `false` in a condition, then it is highly unlikely that you will ever need to test `boolean` expressions for equality!

3.7 Operator precedence with `boolean` expressions

In Section 2.4, we introduced precedence rules for arithmetic expressions. Recall that precedence rules help us make sense out of complex expressions by telling us in which order operations should be performed. In this section we extend the precedence rules to apply to boolean expressions and to expressions involving both arithmetic and boolean operators.

Suppose a computer program contains the expression:

```
size < BOX_RIGHT - BOX_LEFT
```

Fortunately, the computer will always do the subtraction before the comparison using "`<`". If this were not the case, the computer would end up trying to subtract `BOX_LEFT` from a `boolean` value, the results of comparing `size` to `BOX_RIGHT`. For example, if `size` is less than `BOX_RIGHT`, it might attempt to compute `true - 5`, clearly an error!

Recall from Section 2.4, the precedence rules that specify the order in which arithmetic expressions should be evaluated.

1. First perform all unary minus operations (e.g., `-x`) from left to right.
2. Then perform all multiplications, divisions, and remainder operations from left to right.
3. Then perform all additions and subtractions from left to right.

Thus the expression

```
2 + 3 * 8 / 4
```

will be evaluated as follows:

1. First compute `3 * 8`, yielding 24.
2. Divide this result (24), by 4, yielding 6.
3. Finally add 2 to that result (6), yielding 8.

Notice that if we simply (and incorrectly) performed all operations from left to right, we would first add together 2 and 3, getting 5. Multiply that by 8, getting 40. Finally we would divide that by 4, getting 10, a different *and incorrect* answer.

As noted earlier, these precedence rules for arithmetic expressions are essentially the same ones commonly used in everyday mathematics, and, just as in mathematics, we can override them by using parentheses. Thus in evaluating the expression

```
(2 + 3) * 8 / 4
```

we must add together 2 and 3 before multiplying it by 8 and then dividing by 4.

Now let us extend these rules to extend to expressions containing comparison operators and boolean operators. Because Java allows a programmer to mix comparison, logical, and arithmetic operations in the same expression, the precedence rules apply to all of these operators.

1. Unary operators, unary minus: `-`, and not: `!`.
2. Multiplication, division, and remainder operations: `*`, `/`, and `%`.
3. Addition and subtraction operators: `+` and `-`.

4. Comparison operators: `<`, `<=`, `>`, `>=`.
5. Equality and inequality: `==`, `!=`
6. And: `&&`
7. Or: `||`
8. Assignment operators: `=`.

As usual, parentheses can be used to change the order of operations. All operations of the same precedence level are performed from left to right.

We can use these rules to understand the order in which the computer evaluates the following expression:

```
size < BOX_RIGHT - BOX_LEFT
```

It first evaluates `BOX_RIGHT - BOX_LEFT` because arithmetic operators have higher precedence than comparison operators. It then determines if that number is greater than the value of `size`.

Similarly, when the computer evaluates

```
point.getX() >= LEFT_SEP && point.getX() <= RIGHT_SEP
```

it first checks the two inequalities, because comparison operators have higher precedence than `&&`. It then returns true only if both inequalities are true.

This detailed understanding of the way Java evaluates expressions helps explain why it does not allow the following expression, as we discussed earlier:

```
1 <= x <= 10
```

Let us assume that the value of `x` is 6. To evaluate the above expression the computer would:

- Determine that 1 is less than or equal to `x`, yielding `true`. It evaluates this operation first because it is the left-most of the two comparison operators, which have equal precedence.
- Attempt to determine if `true` is less than or equal to 10, resulting in an error, as comparing two different types of values – a `boolean` and an `int` – is not allowed.

Thus the difficulty arises because comparing two integers results in a `boolean` value, and this `boolean` value may not be compared with an integer.

The following diagram illustrates what happens during the computation:

```
1 <= x <= 10
  ⇓
 true  <= 10
       ⇓
       ??
```

Clearly the comparison between `true` and 10 makes no sense!

Fortunately, the compiler recognizes and reports this kind of error, as the problem would arise no matter what the value of `x` is. The error message given will generally fail to suggest how to rewrite the expression correctly, but we can see from the above example that all we would have to do is to break the expression into two comparisons and to combine the results using `&&`:

```
1 <= x && x <= 10
```

Sometimes even legal expressions are so complicated that determining the order of operations can be difficult. A good rule of thumb is to use parentheses wherever there is a question regarding the order of operations. The added parentheses will not only force the computer to evaluate the expression in the order you want, but will also aid readability, making life much easier for readers of your program.

3.8 DeMorgan's rules and complex boolean expressions*

We have already seen a few cases where the condition in an **if** statement can be fairly complicated. In order to ensure that the conditions in **if** statements mean what you want, it is worth taking a small excursion into propositional logic to study DeMorgan's laws. DeMorgan's laws are rules for understanding complicated boolean expressions by identifying logically equivalent expressions. Of particular interest will be DeMorgan's laws involving negations and `&&` or `||`.

Recall that if `A` and `B` are boolean expressions, then the expression `A && B` will be true exactly when both `A` and `B` are true. We can illustrate this with a diagram called a truth table:

A	B	A && B
true	true	true
true	false	false
false	true	false
false	false	false

A truth table is similar to an addition or multiplication table in that it indicates the results of an operation for given values of the operands.

The first column of the table shows possible values of `A`, the second column shows possible values of `B`, while the third column shows the corresponding value of `A && B` for those values of `A` and `B`. Thus the first row shows that if `A` and `B` are true then `A && B` is also true. The second row shows that if `A` is true and `B` is false then `A && B` is false.

We can make similar truth tables for the boolean operator `||`:

A	B	A || B
true	true	true
true	false	true
false	true	true
false	false	false

This table show that `A || B` is only false if both `A` and `B` are false.

The truth table for the boolean operator `!` is shorter:

A	!A
true	false
false	true

It simply shows that the negation of an expression is always the opposite of the original value.

The table for `!A` only has two rows because `!` is a unary operator. That it, it only takes a single operand. Because the single operand has only two possible values, `true` and `false`, only two rows are necessary. The tables for binary operations `&&` and `||` have four rows because each operation

has two operands and there are a total of four distinct combinations of boolean values for those two operands.

We can build more complicated truth tables by adding extra columns to the tables. The following table is formed by starting with the table for `A && B` and adding an extra column for the negation, `!(A && B)`

A	B	A && B	!(A && B)
true	true	true	false
true	false	false	true
false	true	false	true
false	false	false	true

The values for the last column are obtained by negating the values in the previous column – the values of `A && B`.

Next we build the table for `!A || !B`. To build this table, we will need to compute the values of `!A` and `!B` before we compute the *or*.

A	B	!A	!B	!A || !B
true	true	false	false	false
true	false	false	true	true
false	true	true	false	true
false	false	true	true	true

The columns for `!A` and `!B` are obtained by negating the values in the corresponding positions in columns `A` and `B`. The values for column `!A || !B` are obtained from the values in the columns for `!A` and `!B`. Because an *or* only fails when both operands are false, only the first row results in a value of false for `!A || !B`.

It is interesting to observe that the column for `!A || !B` is exactly the same as for `!(A && B)`. This tells us that for all possible combinations of values of `A` and `B`, those two boolean expressions have exactly the same resulting value. In other words those two boolean expressions are equivalent. This equivalence is one of DeMorgan's laws of logic.

This should make sense intuitively. Here is a simple example in English that is roughly equivalent. Suppose I say that I had wanted to take both Math and Computer Science, but I failed. Then it must be that I didn't take Math or I didn't take Computer Science.

Following the truth table, we see that the expression `!(A && B)` is true exactly when `A && B` is false, and `A && B` is false exactly when one or more of `A` and `B` is false. Of course `!A || !B` is true exactly when one or more of `!A` and `!B` is true, which is equivalent to having one or more of `A` and `B` being false. Thus each is true exactly when one or more of `A` and `B` are false.

Below we write the truth tables for the expressions `!(A || B)` and `!A && !B`.

A	B	A || B	!(A || B)
true	true	true	false
true	false	true	false
false	true	true	false
false	false	false	true

A	B	!A	!B	!A && !B
true	true	false	false	false
true	false	false	true	false
false	true	true	false	false
false	false	true	true	true

Because the final columns for each of these tables are the same, we know that the boolean expressions `!(A || B)` and `!A && !B` are equivalent. This is another of DeMorgan's laws.

Exercise 3.8.1 *Give an intuitive argument for the equivalence of* `!(A || B)` *and* `!A && !B`.

We can use DeMorgan's laws to simplify boolean expressions. For example, according to the first equivalence above `!(x > 0 && x <= 10)` is equivalent to `!(x > 0) || !(x <= 10)`. We can simplify this further by remembering that the negations of inequalities can be simpified. Thus `!(x > 0)` is equivalent to `x <= 0` while `!(x <= 10)` is equivalent to `x > 10`. Thus, `!(x > 0 && x <= 10)` is equivalent to `x <= 0 || x > 10`, where the second expression is much easier to understand than the first.

Exercise 3.8.2 *Convince yourself of the above equivalence by drawing a number line and shading in the regions represented by* `x > 0 && x <= 10` *and* `x <= 0 || x > 10`. *The two regions do not overlap and contains the entire number line between them. Hence* `!(x > 0 && x <= 10)` *is equivalent to* `x <= 0 || x > 10`.

Whenever possible we will use DeMorgan's laws and the rules for simplifying the negations of comparison operators to simplify complex boolean expressions.

Exercise 3.8.3 *Use DeMorgan's laws and the rules for simplifying the negations of comparison operators to simplify the following boolean expression:*

```
!(x = 0 || x >= 100)
```

3.9 Nested conditionals

Occasionally we run into problems with quite complicated logic, often requiring very complex boolean conditions if they are handled using **if-else if** statements as we have seen them used so far. Rather than constructing these very complex conditions, we will introduce alternative structures for supporting the program logic. Happily, we don't need to introduce any more syntax in order to handle them; we just need to combine **if** statements in more complicated ways.

Suppose it is a summer weekend and you are trying to figure out what to do. Your choice of recreation will depend on the weather and how much money you have to spend. The following table lists the various options and choices, where the row headings represent your possible financial situation and the column headings represent the weather possibilities.

	sunny	not sunny
rich	outdoor concert	indoor concert
not rich	ultimate frisbee	watch TV

The table entries represent the suggested recreational activity given the financial situation represented by the row and the weather as represented by the column. Thus if you are feeling rich and it is not sunny, you might want to go to an indoor concert. If you are not feeling rich and it is sunny, you might play some ultimate frisbee.

How can we represent these choices with an **if** statement? Let `rich` and `sunny` be variables of type `boolean`, and let `activityDisplay` be a variable of type `Text` that will display the selected activity. The **if** statement below uses `else if` clauses to represent the four choices.

```
if ( sunny && rich ) {
    activityDisplay.setText( "outdoor concert" );
}
```

```
else if ( !sunny && rich ) {
    activityDisplay.setText( "indoor concert" );
}
else if ( sunny && !rich ) {
    activityDisplay.setText( "play ultimate" );
}
else  { // !sunny && !rich
    activityDisplay.setText( "watch TV" );
}
```

Recall that because `!` has higher precedence than `&&`, the condition `!sunny && rich` is evaluated by first evaluating `!sunny` and then using the `&&` operation to determine whether both `!sunny` and `rich` are true.

This code correctly represents all four options, but is rather verbose and loses the nice structure of the table. A related problem is that by the time we arrive at the last case the program has evaluated 3 fairly complex `boolean` expressions.

We can write this so that only two evaluations of boolean variables are ever made, and those without the added complication of *negations* or *and* operators. This is accomplished by *nesting* **if** statements.

```
if ( sunny ) {
    if ( rich ) {
        activityDisplay.setText( "outdoor concert" );
    }
    else {  // not rich
        activityDisplay.setText( "play ultimate" );
    }
}
else {    // not sunny
    if ( rich ) {
        activityDisplay.setText( "indoor concert" );
    }
    else {  // not rich
        activityDisplay.setText( "watch TV" );
    }
}
```

The organization of these **nested if** statements is actually quite similar to that of the table. There is an outer **if-else** statement that determines whether or not `sunny` is true. This corresponds to choosing either the first or second column of the table. Inside the outer **if-part** there is an **if-else** statement that determines whether or not `rich` is true. This corresponds to figuring out which row of the table applies. For example, if `rich` is false, the outcome should correspond to the first column and second row of the table, and hence the `activity` should be "play ultimate". The **else-part** corresponding to it not being sunny is handled in a similar fashion.

Style Note: The nested **if-else** statements are indented from the outer ones in order to make the code easier to read and understand. While Java compilers ignore the layout of code,[2] human

[2]Java uses the term "whitespace" to refer to the spacing present between code elements. Java ignores all white space, so you could theoretically write an entire program on one line, but this would lead to unreadable code.

readers appreciate the cues of indenting to understand complex code like this.

Aside from the indenting, another thing that makes this code easy to understand is the inclusion of comments. In particular, notice how each `else` clause includes comments indicating under which conditions the **else-part** is executed. This has the advantage of making it absolutely clear to the reader under what circumstances this code is executed. The more complex the conditional, the more important these comments become. We strongly urge all programmers to include such comments.

Therefore, while using nested **if-else** statements yields code that is not quite as compact and simple to understand as the table, it is much easier to see its correspondence to the table than the version involving only `else if` clauses. Notice in particular that no matter what the values of `sunny` and `rich` are, only two `boolean` variables are ever evaluated during the execution of this code, so it is not only clearer but faster as well!

Exercise 3.9.1 *Use nested conditionals to rewrite the* `onMouseClick` *method for tabulating votes of four candidates from Section 3.5.*

In Figure 3.9 we provide another example of complex choices being represented by nested **if-else** statements. This class provides the code to simulate a complete game of craps. The rules of craps are as follows:

> The shooter rolls a pair of dice. If the shooter rolls a 7 or 11, it is a win. If the shooter rolls a 2, 3, or 12, it is a loss. If the shooter rolls any other number, that number becomes the "point". The shooter then must roll the "point" value again before rolling a 7. If the shooter rolls the "point" before a 7, it is a win. Otherwise it is a loss.

The program simulates a roll of the dice by using a random number generator every time the user clicks the mouse. In order to implement the rules given above, we must organize the game logic in a way that can be represented using **if** statements. Notice that the rules are quite different depending on whether this is the player's first throw or not. For instance, if it is the first throw then rolling a 7 results in a win, but if it is a second or subsequent throw then 7 results in a loss. Therefore we will organize the first level of conditional to lead to different nested conditionals based on whether or not it is the first throw.

In order to make such a choice, the `Craps` class in Figure 3.9 declares a `boolean` variable, `newGame`, to remember whether or not this is the first throw of a new game. This variable is initialized to be `true` in its declaration because, by default, the user is starting a new game.

The outer **if** statement in the method `onMouseClick` has the following structure:

```
if ( newGame ) {   // starting a new game
    ...
}
else {             // continuing trying to make the point
   ...
}
```

The **if-part** of this code is a **nested if** statement with three branches, each of which encodes the relevant actions to be taken based on the first roll of the dice. Recall that we saw a simplified version of this example earlier in the chapter. The new code is reproduced below:

```
if ( roll == 7 || roll == 11 ) {  // 7 or 11 wins on first throw
    status.setText( "You win!" );
```

```
public class Craps extends WindowController {
   // Generator for roll of a die
   private RandomIntGenerator dieGenerator = new RandomIntGenerator( 1,6 );
   private boolean newGame = true;     // True if starting new game
   private Text status,                 // Display status of game
                message;               // Display dice roll value
   private int point;                  // number to roll for win

   // Create status message on canvas
   public void begin() {
       status = new Text( "", 10, 70, canvas );
   }

   // For each click, roll the dice and report the results
   public void onMouseClick( Location pt ) {
      // get values for both dice and display sum
      int roll = dieGenerator.nextValue() + dieGenerator.nextValue();
      message.setText( "You rolled a " + roll + "!" );

      if ( newGame ) {               // starting a new game
         if ( roll == 7 || roll == 11 ) {  // 7 or 11 wins on first throw
             status.setText( "You win!" );
         }
         else if ( roll == 2 || roll == 3 || roll == 12 ) {  // 2, 3, or 12 loses
             status.setText( "You lose!" );
         }
         else {                      // Set the roll to be the new point to be
made
             status.setText( "Try for your point!" );
             point = roll;
             newGame = false;         // no longer a new game
         }
      }
      else {                         // continuing trying to make the point
         if ( roll == 7 ) {          // 7 loses when trying for point
             status.setText( "You lose!" );
             newGame = true;         // set to start new game
         }
         else if ( roll == point ) { // making the point wins!
             status.setText( "You win!" );
             newGame = true;
         }
         else {                      // keep trying
             status.setText( "Keep trying for " + point + " ..." );
         }
      }
   }
}
```

Figure 3.9: Craps class illustrating nested conditionals.

```
}
else if ( roll == 2 || roll == 3 || roll == 12 ) {  // 2, 3, or 12 loses
    status.setText( "You lose!" );
}
else {                        // Set the roll to be the new point to be made
    status.setText( "Try for your point!" );
    point = roll;
    newGame = false;       // no longer a new game
}
```

Rather than having a separate branch for each possible value of the roll of the dice, there are only three. These branches correspond to winning, losing, and establishing a point to be made on subsequent rolls. The variable `newGame` remains true in the first two branches, so it need not be updated. Only the third branch requires setting `newGame` to false.

Let us now examine the **else-part** of the outer **if** statement. Like the **if-part**, this nested **if** statement also has 3 branches, though the second and third conditions are quite different from those that handle the first roll:

```
if ( roll == 7 ) {             // 7 loses when trying for point
    status.setText( "You lose!" );
    newGame = true;      // set to start new game
}
else if ( roll == point ) {    // making the point wins!
    status.setText( "You win!" );
    newGame = true;
}
else {                         // keep trying
    status.setText( "Keep trying for " + point + " ..." );
}
```

In this statement, both of the first two choices result in setting `newGame` back to `true` because they represent the end of a game with either a win or a loss. The third statement merely asks the player to continue rolling.

Exercise 3.9.2 *Try writing out this program using only a single* **if** *statement with many* `else if` *clauses. It should become painfully clear why nested* **if** *statements are useful in situations with complex logic.*

3.10 Simplifying syntax in conditionals

You have probably noticed that many lines of code in this chapter have been taken up by curly braces, which simply indicate the beginning and end of code blocks. Java allows the programmer to drop these braces when a block includes only a single statement. For example, the first attempt at coding the decision table for activities can be simplified as follows:

```
if ( sunny && rich )
    activityDisplay.setText( "outdoor concert" );
else if ( !sunny && rich )
```

```
    activityDisplay.setText( "indoor concert" );
else if ( sunny && !rich )
    activityDisplay.setText( "play ultimate" );
else // !sunny && !rich
    activityDisplay.setText( "watch TV" );
```

Because each of the blocks contains only a single line of code, all of the curly braces can be dropped.

Similarly, the code for the same problem using **nested if** statements can be simplified to:

```
if ( sunny )
    if ( rich )
        activityDisplay.setText( "outdoor concert" );
    else   // not rich
        activityDisplay.setText( "play ultimate" );
else // not sunny
    if ( rich )
        activityDisplay.setText( "indoor concert" );
    else   // not rich
        activityDisplay.setText( "watch TV" );
```

While it is easy to see why we can drop the curly braces around all of the invocations of `activityDisplay.setText(...)`, it may not be clear why we can drop the curly braces in the second case. The reason we can drop them is that the **if-part** is a single nested **if** statement:

```
if ( rich )
    activityDisplay.setText( "outdoor concert" );
else   // not rich
    activityDisplay.setText( "play ultimate" );
```

While this statement looks complex, Java treats it as a *single statement*. As a result the curly braces around it may be dropped. The same reasoning holds true for the lack of curly braces around the **else-part**.

While dropping curly braces can result in more *compact* code, it does not always result in more *readable* code, and the latter is certainly more important to anyone who might look at the code, including the programmer! To see how removing curly braces can hurt readability, compare the original **nested if** example and the more compact code above. The original has more white space, making its structure more apparent.

Warning! Aside from hurting readability, dropping curly braces makes it easier for errors to creep in when revising code. Suppose we begin with the following code:

```
if ( score >= 60 )
   gradeDisplay.setText( "Pass" );
else
   gradeDisplay.setText( "Fail" );
```

In this case it is fine and convenient to leave off the curly braces. However, suppose we now want to add a new statement to the `else` clause to warn the student to work harder:

```
if ( score >= 60 )
   gradeDisplay.setText( "Pass" );
else
   gradeDisplay.setText( "Fail" );
   message.setText( "You are now on academic probation!" );
```

Although this *looks* perfectly fine, when it is executed the user is told that he or she is on academic probation no matter what the score is!

The problem is that the Java compiler ignores the indenting of the program and instead merely looks at the code. According to the rules stated above, if curly braces are not used, the blocks of code associated with the **if** and **else parts** are taken to consist of only one line. Thus the above code is equivalent to:

```
if ( score >= 60 ) {
   gradeDisplay.setText( "Pass" );
}
else {
   gradeDisplay.setText( "Fail" );
}
message.setText( "You are now on academic probation!" );
```

rather than the intended code below:

```
if ( score >= 60 ) {
   gradeDisplay.setText( "Pass" );
}
else {
   gradeDisplay.setText( "Fail" );
   message.setText( "You are now on academic probation!" );
}
```

This mistake would be less likely to occur if the original version included the curly braces. While this kind of error is easy to correct once it is discovered, detecting why your program does not function properly is often very, very difficult in the first place. The loss of time in trying to track down the error is usually significantly greater than any time saved in not inserting the curly braces.

Yet another problem associated with leaving out curly braces arises when using **nested if** statements. Suppose we write the following code:

```
if ( sunny )
    if ( rich )
        message.setText( "Go to outdoor concert" );
else
    message.setText( "Try something else" );
```

From the indenting, it appears that the message "Try something else" should be displayed when `sunny` is false. However, recall that Java compilers ignore indenting and all white space. Thus, to the Java compiler, the above code is exactly the same as:

```
if ( sunny )
    if ( rich )
        message.setText( "Go to outdoor concert" );
    else
        message.setText( "Try something else" );
```

In this case, the indenting seems to indicate that the message "Try something else" is displayed when `sunny` is true but `rich` is false.

Which of these is the correct interpretation? This problem is so common and confusing that it has even been given a name, the *dangling else* problem. It occurs when there are two **if** clauses in a row, followed by a single `else` clause.

The second indenting above represents the way Java will interpret the **nested if** statement. The rule Java uses is that **else-parts** are always associated with the nearest possible **if** statement unless, of course, curly braces indicate otherwise. Thus in this example the `else` is associated with the second `if`.

Should you memorize this rule? We don't think it is necessary. Instead, if you always use curly braces to ensure that the code is interpreted the way that you want it, you will never experience the *dangling else* problem. Thus to ensure the interpretation associated with the first indenting above, you should write:

```
if ( sunny ) {
    if ( rich ) {
        message.setText( "Go to outdoor concert" );
    }
}
else {
    message.setText( "Try something else" );
}
```

To ensure the second interpretation, you should write:

```
if ( sunny ) {
    if ( rich ) {
        message.setText( "Go to outdoor concert" );
    }
    else {     // not rich
        message.setText( "Try something else" );
    }
}
```

When you use curly braces, you are forced to write the statement in an unambiguous fashion that the computer cannot misinterpret.

There is one time when it is very helpful to drop the curly braces. Suppose we have an **if** statement nested within the **else-part** of another **if** statement as follows:

```
if ( temperature >= 100 ) {
    display.setText(``Water is in a gaseous phase'');
}
else {
```

```
        if ( temperature >= 0 ) {
            display.setText(``Water is in a liquid phase'');
        }
        else {    // temperature < 0
            display.setText(``Water is in a solid phase'');
        }
    }
```

If the temperature is above 100 degrees Celsius, water is in a gaseous phase. Otherwise another **if** statement is used to determine whether or not the temperature is above 0 degrees to decide whether water is liquid or solid.

With this structure we have an **if-else** nested in an **else-part**. Even the indenting gets complex as each successive **else-part** is moved farther and farther to the right. Because the first **else-part** consists of a single, albeit complex, **if** statement, we may drop the curly braces around it as follows:

```
    if ( temperature >= 100 ) {
        display.setText(``Water is in a gaseous phase'');
    }
    else
        if ( temperature >= 0 ) {
            display.setText(``Water is in a liquid phase'');
        }
        else {    // temperature < 0
            display.setText(``Water is in a solid phase'');
        }
```

If we now move the **if** statement immediately following the `else` keyword up to be next to the `else` and fix the indentations, we get:

```
    if ( temperature >= 100 ) {
        display.setText(``Water is in a gaseous phase'');
    }
    else if ( temperature >= 0 ) {
        display.setText(``Water is in a liquid phase'');
    }
    else {    // temperature < 0
        display.setText(``Water is in a solid phase'');
    }
```

These simplifications have given us the same **else if** structure that we saw in Section 3.4!

We see that there is really nothing special about the `else if` clause. It is simply obtained by dropping the curly braces around a **nested if** statement that happens to be the only statement in an **else-part**.

This is the only instance in which we recommend dropping the curly braces around an **if** or **else-part**, as it is equivalent to creating a clearer, more concise `else if` clause. Otherwise we strongly recommend that curly braces be used with all other **if** and **else-parts**, as it is far too easy to make errors when they are omitted.

3.11 The conditional expression and switch*

Java contains two other conditional expressions and statements. One is a conditional expression of the form

```
boolExp ? tVal : fVal.
```

It returns the value of either `tVal` or `fVal`, depending on whether `boolExp` evaluates to true or false. It is hard to read and is used only rarely by Java programmers, so we encourage beginning programmers not to use it.

The **switch** statement is much more useful, but unfortunately it has an error-prone syntax. Thus we do not encourage our beginning students to use it until they are more comfortable with Java syntax. For completeness, we include a brief introduction to `switch` in this section.

The **switch** statement is useful in situations where different actions are to be taken based on the value of an expression. Here is a simple example that uses the value of an integer variable, `gradePoint`, to display a message about the corresponding letter grade:

```
switch (gradePoint)
{
    case 4:
        letterGradeDisp.setText("You got an A.";
        break;
    case 3 :
        letterGradeDisp.setText("You got a B.";
        break;
    case 2 :
        letterGradeDisp.setText("You got a C.";
        break;
    case 1 :
        letterGradeDisp.setText("You got a D.";
        break;
    case 0 :
        letterGradeDisp.setText("You failed.";
        break;
    default:
        letterGradeDisp.setText("Illegal value for gradePoint.";
        break;
}
```

In the above example `letterGrade` is a variable of type `char`, while `gradePoints` is a variable of type `double`. When this statement is executed, the value of `letterGrade` is obtained. If the value of `letterGrade` is one of the values listed after the `case` keyword, then the statements after the corresponding `case` up until the following `break` statement are executed. When the break statement is executed, execution jumps to immediately after the closing brace for the `switch` statement. The `default` clause indicates what should be done if the value of `letterGrade` does not correspond to any of the values given after the `case` keywords.

There are a number of restrictions to the case statement that a programmer must be aware of. First, the expression immediately after the `switch` keyword must have type either `int` or `char`. No other possibilities for types are allowed, though the expression need not be a variable.

Second, there may only be one value provided after each occurrence of `case` (and there may not be any duplicates). If you wish to do the same thing with several distinct values of the switch expression, you may stack up several case clauses. We included the following code in our initial discussion of the craps example in Section 3.5

```
if ( roll == 7 || roll == 11 ) {  // 7 or 11 wins on first throw
    status.setText( "You win!" );
}
else if ( roll == 2 || roll == 3 || roll == 12 ) {  // 2, 3, or 12 loses
    status.setText( "You lose!" );
}
else {                              // play must continue
    status.setText( "The game continues" );
}
```

We can rewrite this with a `switch` statement as follows:

```
switch (roll)
{
    case 7:
    case 11:                    // 7 or 11 wins on first throw
        status.setText( "You win!" );
        break;
    case 2:
    case 3:
    case 12:                    // 2, 3, or 12 loses
        status.setText( "You lose!" );
        break;
    default:                     // play must continue
        status.setText( "The game continues" );
        break;
}
```

Because there is no break between `case 7` and `case 11`, if the value of `roll` is 7, it will begin immediately after the `case 7` and continue until it hits the first break, which is also the break for `case 11`.

Finally, the most important restriction on `switch` statements is that the values after the `case` keywords must be constants. That is, they either must be literals like `'A'` and `17` or they must be declared constants of type `int` or `char` (e.g., `public static final int ...`). In particular, you may never put a relation in a `case` clause. Thus a clause like `case x > 10` will generate a compile-time error.

We caution students about using `switch` statements because it is easy to make errors in writing these statements that the compiler will not report as errors. The most common error is to omit the `break` statement at the end of a `case`. As illustrated in the example above, omitting the `break` is sometimes desirable. However, if a break was omitted after the case for `'A'` in the previous example, the computer would blithely continue and assign the value of `3.0` to `gradePoints`, which is certainly not what was wanted. Experience shows that this omission is very easy to make, so programmers must be on guard for this.

Another common error is to omit the `default` statement when it might catch an error. For example, if the value of `letterGrade` was 'G' when the first `switch` statement executed, and the `default` clause was omitted, then the erroneous value would not have been caught. In some cases, (for example if the total number of values is incremented before the switch statement) this might result in an incorrect value being computed (e.g., for a student's gpa).

A switch statement is often used in a context where an `if-else if` compound statement could also be used. If the decision on which block of statements to execute is based on equality comparisons with constants, then a `switch` statement may be the best choice, both from the point of view of readability and efficiency. (Though we do not discuss the reasons here, the `switch` statement is often compiled to a form that is more efficient than an `if-else if` statement when there are many cases to consider.)

However, we again caution the reader about the dangers of the `switch` statement syntax. The idea of the `switch` statement is a good one, and other languages have similar constructs that avoid the syntactic pitfalls of Java's `switch`. However, we feel that the possibility of inadvertently omitting the `break` without receiving an error message should make programmers very cautious when using this contruct.

3.12 Summary

In this chapter we introduced conditional statements and the `boolean` data type. The major points discussed were:

- The **if-else** statement is used when different code is to be executed depending on the value of a condition.
- **if** statements without an `else` clause are used when extra code is to be executed in one case, but nothing extra is needed in the other.
- **if** statements with `else if` clauses are used if there are more than two cases to be considered in a choice. **if** statements with `else if` clauses may or may not be terminated with an **else-part**, at the programmer's option.
- Nested **if** statements can be used to represent complex logic.
- While curly braces can be omitted from **if** or **else-parts** that consist of only a single statement, this practice is somewhat error prone. We strongly recommend keeping the curly braces even when optional.

We have also discussed good and bad ways to write conditional statements. It is important to remember that, although these constructs provide immense power and flexibility to programmers, unless used intelligently they can produce confusing, unreadable code. If you follow the suggestions we have made, however, you will almost never find yourself having this problem.

Here are the tips summarized for your convenience:

1. Don't negate the result of evaluating a comparison operator – change the operator instead! For example, replace `!(a == b)` by `a != b`, replace `!(a > b)` by `a <= b`, and replace `!(a <= b)` by `a > b`. Use DeMorgan's laws to simplify complex boolean expressions.
2. Never omit the **if-part** of a conditional. Negate the condition so what used to be in the **else-part** now belongs in the **if-part**.

3. Never use the `boolean` literals `true` or `false` in a condition.

Index