A Tennis Applet

Introduction

In the old procedural paradigm, writing a program was essentially the construction of an algorithm. To design a program, the entire task was broken down into a series of subtasks. To implement that design, subprograms (also called subroutines) were written for each subtask. Typically there was a main loop which repeatedly called various of the subroutines depending on conditions. Looping and conditional statements were central to understanding and building programs, so they appeared early in programming texts; little could be done without them.

By contrast, in the object paradigm presented here, writing a program involves designing and implementing a class structure and a GUI. The finished program still implements an algorithm, but its complexity is distributed across the various classes. In a properly implemented object program, every class and method is simple. Once classes are written they may be reused with a minimum of labor. Thus far, the programs in this text have accomplished conditional and repeated action by relying on the user and the event loop. The event loop is the mechanism built into the Java VM to handle user events. If the user wanted to make several withdrawals, they pushed the withdraw Button several times. To move the Eye left, they pressed the moveLeft Button. Nonetheless, object programs do need looping and conditional statements. This chapter and the next will introduce those two elements of control structure.

Different actions depending on conditions - Conditional execution

The if Statement -- Do Something or Don’t

Every program thus far has run the same way every time you ran it. But there are times when a program needs to choose between actions to perform on the basis of the current conditions. For instance, ATM machines usually won’t let you withdraw more money than you have in your account. The one in Chapter 3 would pretend to give out money even if the resulting balance was negative. The code for withdraw() is shown in Listing 7.1.

Listing 7.1 The withdraw method for the Account class (from Listing 3.4).

```
1 public void withdraw(int amountToWithdraw) {
2     balance = balance - amountToWithdraw;
3 }
```

An if statement can be used to prevent the Account from being overdrawn, as in Listing 7.2.

Listing 7.2 A withdraw method that prevents overdraughts.

```
1 public void withdraw(int amountToWithdraw) {
2     if (balance >= amountToWithdraw) {
3         balance = balance - amountToWithdraw;
4     }
```

**Lines 2-3:** This if statement causes the assignment on line 3 to only occur if the balance is at least as big as the amount to withdraw.
An if statement starts with the word if, then a boolean expression in parentheses, then a statement to execute if that expression evaluates to true. This syntax is shown in BNF 7.1.

**BNF 7.1 The if Statement**

<if stmt> ::= if (<boolean expression>) <stmt> [else <stmt>]

<table>
<thead>
<tr>
<th>Semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Evaluate the &lt;boolean expression&gt;</td>
</tr>
<tr>
<td>2. If the value of the expression is true, execute the &lt;stmt&gt; after &lt;expression&gt;</td>
</tr>
<tr>
<td>3. If the value is false and there is an else part, execute the &lt;stmt&gt; in the else part.</td>
</tr>
</tbody>
</table>

Remember that every legal if statement must match this syntax exactly. The expression must be of type boolean (since only the values true and false make sense here) and must be enclosed in ()s. Then there must be exactly one statement (if there are several things you want to do in the if part you must enclose them in {}s to transform them into a single block statement).

**if-else -- Do One Thing or Another**

An if statement (without an else) is used when you want an action performed only under certain conditions; it either executes the statement following the expression or does nothing, depending on the value of the expression. By contrast, an if-else statement chooses between two actions.

**Example: Preventing Overdrafts While Alerting the Customer to the Problem**

If a person tries to withdraw more money than they have, the code in Listing 7.2 would simply ignore them; which could be a bit unsettling. It might be better to let them know something had gone wrong. I.e. either make the withdrawal, or print an error message. That’s a choice between two things, so the construct to use is the if-else statement, as in Listing 7.3.

**Listing 7.3 A withdraw() Method with an if-else Statement**

1    public void withdraw(int amountToWithdraw) {
2        if (balance >= amountToWithdraw) {
3            balance = balance - amountToWithdraw;
4        } else System.out.println("Oops! You don't have that much!");
5    }

The else part (line 4) executes if amountToWithdraw > balance. Thus, if the user is trying to withdraw more than they have, instead of ignoring them, it will print an error message.

**More Complex boolean Expressions**

The boolean expression in an if statement may, like any expression, be arbitrarily complex. Numbers may be compared by any of the relational operators (see BNF 5.13 “binary operators”, in Chapter 5) Two boolean expressions may be conjoined with || (or), and && (and), and negated with ! (not).
Examples:

\[
\begin{align*}
x &> 0 & \quad \text{// true if } x \text{ is } > 0 \\
x &\geq 0 \ \&\ x \leq 100 & \quad \text{// true if } 0 \leq x \leq 100, \ i.e. \ a \ legal \ exam \ score \\
x &= 7 \ || \ x &= 11 & \quad \text{// true if } x \text{ is } 7 \text{ or } 11, \ i.e. \ a \ winner \ in \ craps \\
x &!= 7 \ \&\ \&\ x != 11 \ \&\ \&\ x != 2 \ \&\ \&\ x != 12 & \quad \text{//true if } x \text{ is not } 7, 11, 2, \text{ or } 12 \\
\end{align*}
\]

Truth Tables

It is not always obvious exactly how to write boolean expressions, especially in complicated situations. When in doubt, a heavy-handed, but inevitably correct technique is to make a truth table. A truth table lists (or, as computer scientists like to say, enumerates) all the possible combinations of values for the boolean clauses in a boolean expression, along with the values of the expression under those conditions. For instance, assume there are two boolean variables, \(p\) and \(q\). Each may take on either the value true (T), or false (F). Thus, for the pair, there are four possible values; TT, TF, FT, and FF. These are shown in the leftmost two columns of Table 7.1.

### Table 7.1: Truth table I

| \(p\) | \(q\) | \(p \&\& q\) | \(p || q\) | \(!p\) |
|-------|-------|---------------|-------------|--------|
| T     | T     | T             | T           | F      |
| T     | F     | F             | T           | F      |
| F     | T     | F             | T           | T      |
| F     | F     | F             | F           | T      |

The rightmost column has the values for not-\(p\) (!\(p\)); notice that they are just the opposite of the values for \(p\). The values for \(p\&\&q\) and \(p||q\) are shown in between; notice that || is inclusive-or, it includes the case when both \(p\) and \(q\) are true. There is another or operator, exclusive-or, which is true just if \(p\) or \(q\) is true, but not both. Java’s or is inclusive-or.

DeMorgan’s Law

The ! operator seems straightforward, it turns true to false and false to true. But, there is a peculiarity of applying ! to expressions including operators. The ! operator distributes across parentheses, but it changes || to &\& and &\& to ||. See Table 7.2 for an example.

### Table 7.2: Truth table 2

| \(p\) | \(q\) | \(p \&\& q\) | \(!p\ \&\& \!q\) | \(!p\ || \!q\) |
|-------|-------|---------------|-------------------|-----------------|
| T     | T     | T             | F                 | F               |
| T     | F     | F             | T                 | F               |
| F     | T     | T             | F                 | F               |
| F     | F     | F             | T                 | T               |

The fact that \(!p\&\&q\)=!\(p\)||!\(q\) and that \(!p||q\)=!\(p\&\&!q\) is called DeMorgan’s Law; forget it and you will run into some nasty bugs.
Problem Solving Technique -- Analysis By Cases
It is very common in writing a program, and in problem solving in general, that one must do
different things in different cases. For instance, if you are running under a frisbee, if it was
thrown forehand, you expect it to tail off one way, if it was thrown backhand, the other. If it was
thrown as a hammer (up-side-down) you expect it to slow down rapidly and tail off abruptly. In
each case you do different things to catch it.

Analysis By Cases (ABC for short) is a problem solving technique designed especially for
problems with multiple cases.

Problem Solving Technique: Analysis By Cases (ABC)
Identify the various cases. For each, answer these (making a table if it is complicated): 1) How
can you distinguish this case? 2) What action do you wish to take in this case?

Once you have identified each case, decided how to distinguish each case from the others, and
what action to perform in each case, you are ready to write code. The examples will illustrate the
use of this technique.

Example -- A Robot Bouncer
Imagine going to a club and encountering a robot bouncer. The job of the robot bouncer is to
only let in people who are at least 21 and to charge them each the cover charge. Write a method
that is passed a Person as a parameter, and that outputs as a message to System.out what the
robot would say to that person. Assume that the Person is passed as a parameter and that a
Person object has an age and a balance variable with standard accessors.

To start with write a method that only checks their age. There are just two cases here. If the
person’s age is greater than or equal to 21, it should say “Welcome;” otherwise say “Sorry.” That
was the ABC method in a very simple context; so simple it is, well, trivial. The two cases were
under 21 and not. They are distinguished by the age of the person. To write code for this requires
an if-else statement, as in Listing 7.4.

Listing 7.4 Robot Bouncer that Only Checks age
1    void checkAge(Person aPerson) {
2        if (aPerson.getAge() >= 21)
3            System.out.println("Right this way!");
4        else System.out.println("I'm sorry, you must be 21 to enter.");
5    }

This would work fine if there were no cover charge to collect. Next, one might reason that if the
person is 21 or older, then the bouncer, instead of waving them in, should check if they also have
the cover charge. This is shown in Listing 7.5.
Listing 7.5 Robot Bouncer that Checks age and balance
1    void checkAgeAndBalance(Person aPerson) {
2        if (aPerson.getAge() >= 21)
3            if (aPerson.getBalance() > 5)
4                System.out.println("Right this way!");
5            else System.out.println("Sorry, you don't have the cash");
6        else System.out.println("I'm sorry, you must be 21 to enter.");
7    }

This example replaces line 3 in the previous with an if-else to check if they have the money to pay the cover charge. This is legal syntactically, since an if-else statement is a statement (check the BNF if you have any doubts about this). This would work; but perhaps it is cleaner to use a compound boolean expression to check both conditions at once. The person is allowed in if their age and their balance meet certain conditions. This is illustrated in Listing 7.6.

Listing 7.6 Robot Bouncer that Checks age and balance using &&
1    void checkAgeAndBalance2(Person aPerson) {
2        if (aPerson.getAge() >= 21 && aPerson.getBalance > 5)
3            System.out.println("Right this way!");
4        else System.out.println("You must be 21 and have $5 to enter.");
5    }

Cascaded if-elses -- Do One of a Number of Things

Sometimes a program must do exactly one of a number of things. In that case you can build a structure called a cascaded if-else by repeatedly using an if-else statement as the statement following the else. The code in Listing 7.5 could be rewritten into a cascading if-else as shown in Listing 7.7. This is exactly equivalent to Listing 7.5 but is perhaps easier to read.

Listing 7.7 Robot Bouncer that Checks age and balance Using a Cascaded if-else
1    void checkAgeAndBalance(Person aPerson) {
2        if (aPerson.getAge() < 21)
3            System.out.println("I'm sorry, you must be 21 to enter.");
4        else if (aPerson.getBalance() > 5)
5            System.out.println("Right this way!");
6        else System.out.println("Sorry, you don't have the cash");
7    }

The order that conditions are checked may be important; careful thinking is required to make sure a it will work properly in every case.

First Example -- A Robot Aspirin Bottle

Imagine that you are assigned to program an aspirin bottle to announce the correct dosage given a person’s age. The dosage for aspirin is as follows: under 5, consult with a doctor; 6-12, one; 13-65, two; over 65, one. Listing 7.8 shows this coded as a cascading if-else.
Listing 7.8 Cascaded if-else for Aspirin Dosage
1    if (age<6)
2        System.out.println("consult with your physician");
3    else if (age<13)
4        System.out.println("dosage=1");
5    else if (age<66)
6        System.out.println("dosage=2");
7    else System.out.println("dosage=1");

Notice that in the else clause it must be the case that the previous condition was false (otherwise the else clause would not execute). Thus if execution reaches line 3, age must be >=6, and if execution reaches line 7, age>=66.

Each else-if clause executes only if the previous boolean expression was false (because of the semantics of an if-else; see BNF 7.1 “The if statement,” if you have any doubts about this. Once you get used to this notion, it will be very obvious.). It is also possible to make this a bit shorter as shown in Listing 7.9.

Listing 7.9 A Slightly Shorter Cascaded if-else for Aspirin Dosage
1    if (age<6)
2        System.out.println("consult with your physician");
3    else if (age<13 || age>65)
4        System.out.println("dosage=1");
5    else System.out.println("dosage=2");

Notice that if execution reaches line 5, !(age<13 || age>65) must be true, in other words it must be true that (age>=13 and age <=65).

Which is the right way to structure this? There is not one clear answer. It depends on which way makes sense to the programmer, and which way is clearer to a reader. It’s a matter of style.

Second Example -- Reporting the Score of a Tennis Game
Imagine you were assigned to build a robot tennis score announcer. In tennis, scores of zero, one, two, and three, are announced as love, fifteen, thirty and forty. Assuming the scores are kept as ints, your robot score announcer would be passed two ints. It must convert the two int scores to the appropriate String. E.g. if the score were 3 for the server and 2 for the receiver, the correct output is forty-thirty. So a method that announced the score might make use of a method that converted an int to a tennis score for both of the scores, as in Listing 7.10.

Listing 7.10 An announceScore() Method that Uses convert() Twice
1    void announceScore(int serverScore, int receiverScore) {
2        System.out.println(convert(serverScore) + "-" + convert(receiverScore));
3    }

In that case the task is to write and test the convert() method. As usual, the simplest way to generate a test driver is to create a class with a main() method as in Listing 7.11.
Listing 7.11 Testing the convert() Method

```java
String convert(int score) {
    if (score==0)
        return "love";
    else if (score==1)
        return "fifteen";
    else if (score==2)
        return "thirty";
    else if (score==3)
        return "forty";
    else return "value out of range:" + score;
}
```

The `convert()` method is implemented using a cascaded if-else. It returns the String representing the int it is passed. It does not use `System.out.println()` to display the score because `System.out.println()` is only for debugging or experimenting with code.

The Switch Statement

Another construct that selects between a number of possibilities is the `switch` statement. The `switch` statement does not add any power to the cascaded if-else, but it is a little neater and easier to read. The syntax is shown in BNF 7.2

BNF 7.2 The Switch Statement

<table>
<thead>
<tr>
<th>BNF</th>
<th>Semantics:</th>
</tr>
</thead>
</table>
| `<switch stmt>` ::= switch `<enumerable expression>` { | 1. Evaluate the `<enumerable expression>`
| `[<case clause>]*` | 2. If the value of the expression equals any of the `<constant>s in the `<case clauses>`,
| `[default: <stmt>]` | then execute the `<stmt>s after that `<constant>`, and all the rest of the cases! |
| } | 3. Otherwise execute the `<stmt>s after default (if it appears).` |
| `<case clause>` ::= case `<constant>`: `<statement>`* | |

This is the most complicated statement in Java, it is inherited, so to speak, from C++ and old C. The convert method from Listing 7.11 is shown in Listing 7.12, and is rewritten using a `switch` statement in Listing 7.13.
Listing 7.12 convert() Using Cascaded if-else
1    String convert(int score) {
2        if (score==0)
3            return "love";
4        else if (score==1)
5            return "fifteen";
6        else if (score==2)
7            return "thirty";
8        else if (score==3)
9            return "forty";
10        else return "value out of range:" + score;
11    }

Listing 7.13 convert() Using Switch
1    String convert(int score) {
2        switch (score) {
3            case 0: return "love";
4            case 1: return "fifteen";
5            case 2: return "thirty";
6            case 3: return "forty";
7            default: return "Value out of range: " + score;
8        } // switch
9    }

This latter convert() uses a switch statement instead of a cascaded if-else. Switch adds no power, but some people find it easier to read.

There are two difficulties with the switch statement. One is that it can only switch on types which are enumerable. Enumerable types include int, and char. String and double will not work. The other is that unless the statements in each case clause end with a break or return statement, the following cases are all executed too! This can have surprising (and sometimes upsetting) results.

This week’s lab: Tennis Score Keeping Program

A Description of the Task
The programming task for this lab is to write a Java Applet that will keep track of the score in a tennis game, and post the Applet on your web page. It should announce the score before each point, as well as the winner of the game.

The same player serves for an entire tennis game. Before each point, the score is announced, server’s score first. A game is won when one of the players has at least four points and is at least two points ahead. If the score reaches 4-4, then, until the game is decided, the score is announced as “deuce” for ties and “advantage server” or “advantage receiver”, depending on who is ahead by one.

Assume for now that the user will push one of two Buttons for each point, one if the server gets the point, the other if the receiver gets the point.
GUI Design

The GUI is simple; two Buttons and somewhere to display the output. Either a TextField or a TextArea could work for that. Which would be better? A TextField only has one line. The user might want to be able to see the history of points, so a TextArea seems more sensible.

Here’s how to create a prototype (although, odds are you are familiar with this by now).

Create a new JApplet named TennisApplet.
Add, rename, relabel, resize, connect and test two Buttons (don’t forget to set the Layout to null so they don’t act funny).
Add, rename, and resize a TextArea - write in the TextArea on a Button press to make sure everything is working so far.

Adding Images to the GUI (Optional)
You can make your GUI look much nicer if you add images to it. For instance, the programmer that wrote this Applet to keep himself amused while writing this, copied images of Maria Sharpova and Serena Williams, the two finalists in the 2004 Wimbledon tournament and added them to his Applet. But, due to copyright issues has replaced them, here, with images from his photo album.

To add images to a GUI requires:
1. Find and copy the images you want to have in your GUI (save them in the same directory as the code; otherwise it won’t work).
2. Declare an Image variable for each one.
3. Read each image into its Image variable.
4. Write a public void paint(Graphics) method to draw them.

Step 1 can be accomplished either by finding an image on the web, right-clicking it and selecting “Save” (or some such), and then navigating to the directory your tennis code is in. Or, you could copy them from http://www.willamette.edu/~levenick/SimplyJava/images/ The code to accomplish steps 2-4 is in Listing 7.14.

Listing 7.14 The TennisApplet
1    import java.awt.*;
2    
3    public class TennisApplet extends java.applet.Applet {
4        Image mariaImage, serenaImage;
5        
6        /** Initializes the applet TennisApplet */
7        public void init() {
8            initComponents();
9                // read from code directory
10            mariaImage = getImage(getCodeBase(), "maria.jpg");
11            serenaImage = getImage(getCodeBase(), "serena.jpg");
12        }
13        
14        public void paint(Graphics g) {
Line 1 Imports java.awt.*, so it will know what Graphics and Image are.

Line 4: Declares two Image variables.

Line 8: Notice that this is the NetBeans 3.6 form; with NetBeans 4.0 the Applet code looks like Listing 4.6. They both work.

Lines 10-11: Read in the two images. The files must be named exactly maria.jpg and serena.jpg, or else (of course) they will not be found.

Lines 14-17: Draw the images. The four int parameters specify the rectangle to draw the image in; it will be scaled to that size.

You will need to arrange your Buttons and TextArea so they are not on top of the Images. Or you can change the x and y coordinates of the rectangles the Images are displayed in. Whatever works for you.

Making It Smaller; Start with simple scoring (no deuce)

Listing 7.15 The TennisApplet

```java
  public class TennisApplet extends java.applet.Applet {

    /** Initializes the applet TennisApplet */
    public void init() {
      initComponents();
    }

    private void serButtonActionPerformed(java.awt.event.ActionEvent evt) {
      serverScored();
    }

    private void recButtonActionPerformed(java.awt.event.ActionEvent evt) {
      receiverScored();
    }

    Lines 12 and 16: invoke the appropriate methods when the user pushes a button.

    Keep in mind that if you’ve added the Image code, your TennisApplet will look slightly different.

    You will need variables for each player’s score, and methods to increment each of them when the user pushes that Button. It also needs a way to report the score. You might have some code that looks like...
```
Listing 7.16 Code for simple scoring

```java
3       int receiverScore, serverScore;
5
13      public void receiverScored() {
14          receiverScore++;
15          announceScore();
16      }
17
18      public void serverScored() {
19          serverScore++;
20          announceScore();
21      }
22
23      public void announceScore() {
24          System.out.println(convert(serverScore) + "-" + convert(receiverScore));
25      }
26  }
```

**Lines 2-4:** Declare variables.

**Lines 13-16 and 18-21:** Increment the appropriate score and announce the new score.

**Lines 23-25:** Announces the score.

Compile and run the code to do simple scoring; push Buttons until both scores are out of range. Now it is time to add the code to handle deuce games and the game being over. But, before doing that, it will be more efficient to replace the `System.out.println` in `announceScore()` with code to write to the TextArea in the Applet. That `println` was for debugging, and the more `printlns` we add, the more we’ll have to change for the final product.

**Adding Game Over and Deuce Scoring Code**

The `announceScore()` method in Listing 7.16 assumes that the game is still in progress and the first deuce (40-40) has not been reached. It remains to write code to detect and announce these other two cases, so that we can write `announceScore()`. This is a perfect place to use the ABC method (see the problem solving technique, “Analysis By Cases (ABC)”, earlier in this chapter). If you wanted to make sure you’ve learned this technique, this would be a good time to practice it. On the other hand, it is possible to delay making detailed decisions by pretending there are methods that can distinguish between the cases. This is illustrated in Listing 7.18.

Listing 7.18 `announceScore()` Using Methods

```java
1    public void announceScore() {
2        if (gameOver())
3            announceGameOver();
4        else if (simpleScore())
5            announceSimpleScore();
6        else announceDeuceScore();
7    }
```

There are three different cases for the score in tennis: game over, simple scoring, and deuce scoring. This `announceScore()` distinguishes between them by using methods in a cascaded if-else.
The use of methods makes the code legible. This way the code is simple and easy to understand; the details of whether the game is over or when we can use simple scoring is hidden in the methods.

**The gameOver() Method**

How to write the gameOver() method? Oddly, most of it can be written syntactically. I.e. you can do most of the work of writing it given what you know about syntax. Look at line 2 in Listing 7.18. From that you can infer the type of the gameOver() method. Since its type is not void, it must return a value of that type. If you don’t know the type, look back at the syntax of an if statement (BNF 7.1 “The if Statement.”) With that in mind, a prototype of gameOver() can be written while ignoring the details of its logic; see Listing 7.19.

Listing 7.19 gameOver() Prototype

```java
1    public boolean gameOver() {
2        return false;
3    }
```

The type of gameOver() must be boolean; otherwise it could not be used in the context if(gameOver()). Since it is not void, it must return a value. This prototype method would compile and run, but would always (as you can see) return false.

Now we must fill in the logic so that the method will return true when the game is over.

**Writing Simple Code**

Logically, the game is over if either player has at least 4 points and is at least 2 points ahead. The first thing some programmers think to write is that the game is over

```
if (serverScore >= 4 && serverScore - receiverScore >=2 ||
receiverScore >= 4 && receiverScore - serverScore >=2)
```

which is a mouthful. If you were going to write this, it would be good to enclose it in a method, see Listing 7.20, so that you could use it from various places.

Listing 7.20 gameOver() First Try

```java
1    public boolean gameOver() {
2        return (serverScore >= 4 && serverScore - receiverScore >=2 ||
3                    receiverScore >= 4 && receiverScore - serverScore >=2);
4    }
```

Logically, this method is correct, but compare it for simplicity and ease of understanding with Listing 7.21. This code is not wrong, but it is a bit complicated and difficult to read. It is more elegant to write the test more abstractly as in Listing 7.21 which is a simple, easy to read version of gameOver(). The game is over if either the server has won, or the receiver has won.

Listing 7.21 gameOver() Made Simple

```java
1    public boolean gameOver() {
2        return serverWon() || receiverWon();
3    }
```
This way is less prone to errors and easier to modify when there are errors. This method is far superior in terms of debugging and clarity, but it has a price; now there are two more methods to write.

**Combining Nearly Identical Methods**

One’s first impulse might be to simply copy and paste the logic from the two lines into the two methods as in Listing 7.22.

Listing 7.22 `serverWon()` and `receiverWon()`, Take One

```java
public boolean serverWon() {
    return serverScore >= 4 && serverScore - receiverScore >=2;
}

public boolean receiverWon() {
    return receiverScore >= 4 && receiverScore - serverScore >=2;
}
```

Notice the logic is identical in the two methods, with the variables `serverScore` and `receiverScore` reversed. Anytime you discover nearly identical code like this you can combine it. The resulting methods are shown in Listing 7.23.

Listing 7.23 `serverWon()` and `receiverWon()`, Take Two.

```java
public boolean serverWon() {
    return winner(serverScore, receiverScore);
}

public boolean receiverWon() {
    return winner(receiverScore, serverScore);
}
```

The commonalities are combined by invoking the same method with the variables `serverScore` and `receiverScore` reversed.

Both methods simply return the value of winner, but with the actual parameters exchanged. This is a useful technique to learn; it has the benefit of putting the decision of whether one player or the other has won the game in the same code. This is useful because if the logic is wrong, it is only wrong in one place, and can be fixed in one place. The downside is that now `winner(int, int)` must be written; see Listing 7.24.

Listing 7.24 `winner(int, int)`

```java
public boolean winner(int x, int y) {
    return x >= 4 && x >= y + 2;
}
```

A player with score `x` wins over a player with score `y` if `x` is at least 4 and is at least 2 more than `y`. 


Notice that this method returns true just if the first parameter, x, is at least 4 and at least 2 more than the second, y. This concludes gameOver(). Writing it produced several other methods, but they are all one line long. Looking back at Listing 7.18, it remains to write announceGameOver(), simpleScore(), announceSimpleScore(), and announceDeuceScore().

The simpleScore() Method
As long as we have not reached the first deuce score, then we can use the simple score announcer from before. The first deuce score is 3-3, so as long as either score is less than 3, simple scoring will work (see Listing 7.25).

Listing 7.25 simpleScore()
1    public boolean simpleScore() {
2        return receiverScore < 3 || serverScore < 3;
3    }

The announceSimpleScore() method is just the old announceScore() method, as shown in Listing 7.26.

Listing 7.26 announceSimpleScore()
1    public void announceSimpleScore() {
2        theTA.append(.convert(serverScore) + "-"
3                   + convert(receiverScore) + "\n");
4    }

The announceGameOver() & announceDeuceScore() Methods
The method announceGameOver() is shown in Listing 7.27. It checks who won and announces that.

Listing 7.27 announceGameOver()
1    public void announceGameOver() {
2        if (serverWon())
3            theTA.append("Game Server!\n");
4        else theTA.append("Game Receiver!\n");
5    }

Finally, announceDeuceScore() is shown in Listing 7.28. There are three cases; deuce, advantage server and advantage receiver.

Listing 7.28 announceDeuceScore()
1    public void announceDeuceScore() {
2        if (receiverScore==serverScore)
3            theTA.append(“Deuce
”);
4        else if(receiverScore < serverScore)
5            theTA.append(“Advantage Server
”);
6        else theTA.append(“Advantage Receiver
”);
There are three cases in deuce scoring, so the correct construct is a cascaded if-else. If it were difficult to decide how to write the code, ABC would be relevant.

That’s all the code. There were quite a few methods, but they are all quite simple. There is a trade-off between writing simple, easy to debug code and the number of classes and methods one must type. The typing is rather tedious, but the time spent designing and implementing a simple solution will be paid back many times by the time not spent debugging, especially in complicated programs.

Testing

Now it’s time to test the play one game code. After you fix all the small mistakes, you should see it displaying in the TextArea. There are two problems with the code written above (see Figure
First, it does not announce the score before the first point, i.e. it never says “love-love”. Second, after the game is over, it still allows points to be scored. The output shown is for four clicks of
“point for server” then six of “point for receiver.” It would be better to stop scoring after the game is over! Fortunately, because the code is written well, these are both easy to fix.

**Announcing the Score Before the First Point**

Fixing bugs can be easy or difficult. The better you understand the code the easier it is. The simpler the code, the easier it is to understand what’s going wrong. Well written code has modules that make it easy to modify.

The reason it does not announce the score before the first point is obvious if you look at the code. The only time it announces the score is when the user pushes a Button (check it, it’s true). So, how to announce the score before the first point?

It would be easy to announce the score if there were a method that announced the score. Is there?

Where should the `announceScore` message be sent? Look at the TennisApplet code you’ve written. Where should you send that message?

**Preventing Points After the Game is Over**

When the user pushes a “point for” Button, you don’t always want to add a point; only sometimes. That’s what if statements are for. You should only add a point if the game is not over. Here’s the payoff from having written the `gameOver()` method. You can use it from the Applet to check if the game is over and only add a point if it is not, see Listing 7.30.

Listing 7.30 Only Add Points if the Game is Not Over -- !over

```java
private void serButtonActionPerformed( java.awt.event.ActionEvent evt) {
    if (!theGame.gameOver())
        theGame.serverScored();
}
```

**Conclusion**

The if statement allows different statements to execute depending on the situation. It has two forms, if without an else and if-else. If you want your program to do something only under certain conditions, use an if statement.

```java
if (condition is true)
    doSomething
```

If you want your program to do either one thing or another, use an if-else statement.

```java
if (whatever condition determines when to do the first thing)
    oneThing();
```
else anotherThing();

Multiple cases can be handled either by a cascaded if-else, or a switch statement. In complex cases, the Analysis By Cases (ABC) technique can help clarify your thinking.
Review Questions

7.1 If you want to either execute a statement or not depending on some condition, what statement do you use?

7.2 If you want to execute either one statement or another depending on circumstances, what statement do you use?

7.3 Why is the if statement called a conditional statement?

7.4 Write a statement that prints “yes” if x is greater than zero and “no” otherwise.

7.5 Reverse the logic in Listing 7.6. I.e. exchange the statements before and after the else; then reverse the logic of the boolean expression so it still does the right thing. If in doubt, make a truth table.

7.6 Make a truth table for (!p||q) && !(p&&!q)

7.7 In craps, on the first roll, if you roll 7 or 11, you win, if you roll 2 or 12, you lose. Otherwise, you must roll the same value you rolled the first time before you roll a 7 to win. Write an cascaded if-else statement that tests a variable rollValue and prints one of three things: “you win!,” “sorry, you lose,” or “roll again” depending on its value.

Programming Exercises

7.8 Here’s some code with what may be a subtle bug. What would it print if age were 14? If age were 41? How to fix it?

```java
if (age > 21)
    System.out.println("Major");
else
    System.out.println("Minor");
System.out.println("You may not enter!");
```

7.9 Write a method named exclusiveOr(), which is passed two boolean parameters and returns true if exactly one of them is true.

7.10 An exam is graded as follows: 91-100: A, 81-90: B, ...61-70: D, <61: F. Write a method that is passed an int score and returns the appropriate grade as a String. First use an if-else, then a switch statement. Hint: what values would (score-1)/10 take on for various scores? Remember how int division works.

7.11 Modify announceWinner() to announce the winner’s name. Like “Game Serena!”

7.12 Add an error message if the user pushes a Button after the game is over.

7.13 A better solution is to disable the Buttons when the game is over. Do that. Perhaps the easiest way is to send them the setVisible(false) message.