Chapter 3
Making Decisions

To make our programs more interesting we need to be able to make actions and behaviors change. That requires the ability to make decisions, and that's the topic for this chapter.

3.1 Logical Expressions

A **logical expression** is an expression that is either true or false, such as "Today we have a hard rain" or "John is a member of the union".

We can combine logical expressions with the logical operators ‘and’, ‘or’ and ‘not’ to make more complex logical expressions such as "John is a member of the union and John is male", “Leanna has been promoted to director or assistant director”, or "I'm not going to the dinner or the wine tasting".

3.1.1 Truth Tables for And, Or and Not

When we combine expressions with logical operators, the truth value of the expression (meaning is the statement true or false?) clearly depends on the truth value of the sub-expressions that have been combined.

For example, consider the statement, in English, “We reviewed on Wednesday and Thursday”. First let's translate to the more precise logical expression “We reviewed on Wednesday and we reviewed on Thursday”. Now take a look at the possibilities:

- Suppose that we reviewed on both Wednesday and Thursday. Then the statement is clearly true.
- On the other hand, if we reviewed on Wednesday but did not review on Thursday, the statement is not true because it said we reviewed both days.
- If we reviewed on Thursday but not on Wednesday, the statement is also not true.
- Finally, if we didn't review on either Wednesday or Thursday the statement is certainly not true.

The point is that the truth value of the sub-expressions gives us the truth value of the overall expression.

More generally, if P is a logical expression and Q is a logical expression, then the truth value of the expression “P and Q” depends on the truth value of P and the truth value of Q. There are
four possibilities for the truth values of P and Q: true and true, true and false, false and true or false and false.

We can represent these combinations using a truth table, with T for true and F for false.

<table>
<thead>
<tr>
<th>P</th>
<th>Q</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>T</td>
<td>T</td>
<td>both expressions are true</td>
</tr>
<tr>
<td>T</td>
<td>F</td>
<td>one expression is true, the other is false</td>
</tr>
<tr>
<td>F</td>
<td>T</td>
<td>one expression is false, the other is true</td>
</tr>
<tr>
<td>F</td>
<td>F</td>
<td>both expressions are false</td>
</tr>
</tbody>
</table>

A *truth table* provides a convenient, organized way to look at all of the combinations of truth values and the resulting truth value of the final expressions.

Here’s a truth table for all of the possibilities of P and Q.

<table>
<thead>
<tr>
<th>and</th>
</tr>
</thead>
<tbody>
<tr>
<td>P</td>
</tr>
<tr>
<td>------</td>
</tr>
<tr>
<td>T</td>
</tr>
<tr>
<td>T</td>
</tr>
<tr>
<td>F</td>
</tr>
<tr>
<td>F</td>
</tr>
</tbody>
</table>

Now assume the following statements are factually correct.
- the sun is shining
- the professor is bald

The first line of the "and" table above expresses the fact that if I tell you "The sun is shining and the professor is bald" that I have told you the truth—I made two assertions, both of which were true under our assumptions, and connected them with an "and". The resulting statement is true.

On the other hand, the second line expresses the fact that if I tell you "The sun is shining and the professor has a beautiful head of hair" I am (unfortunately) lying. The first assertion, "the sun is shining" is true according to our assumptions, but the second, "the professor has a beautiful head of hair", is false. And so connecting them with an "and" makes the resulting statement false.

The third and fourth lines of the table are analyzed similarly and in general, we understand that an "and" statement is true only if both assertions are true. Otherwise it is false.
Consider the first line of the table, which represents the statement "the sun is shining or the professor is bald". The separate assertions, according to our assumptions, are both true. What does connecting them with an "or" do? That depends—if the "or" is an exclusive "or", then we must have one assertion true or the other assertion true, but not both, to have a true resulting statement, and so "the sun is shining or the professor is bald" would be a false statement.

If "or" is not an exclusive "or" it is called an inclusive "or". We can have one assertion, or the other, or both, true to have a resulting statement that is true. This is the case in the truth table above and in most programming languages, including Java.

The second line of the "or" table expresses the fact that if I tell you "the sun is shining or the professor has beautiful hair" I am telling you the truth—I made two assertions, the first is true, the second false, and connected them with an "or". The resulting statement is true. The third line is analyzed similarly.

On the other hand, the last line of the "or" table expresses the fact that if I tell you "it is raining or the professor has a beautiful hair" I've made two assertions, both of which are false. Connecting them with an "or" doesn't improve things. The resulting statement is still false.

Here is the truth table for the logical not operator, which negates the logical value of the expression. It’s never absolutely required but it’s often convenient.

<table>
<thead>
<tr>
<th>P</th>
<th>not P</th>
</tr>
</thead>
<tbody>
<tr>
<td>T</td>
<td>F</td>
</tr>
<tr>
<td>F</td>
<td>T</td>
</tr>
</tbody>
</table>

More complex logical expressions are built up from the smaller logical expressions. Let’s examine the statement “Today I’ll either go to the mall and the movie or I’ll go to the mall and the game”.

Let P be the logical expression "today I’ll go to the mall". Let Q be the logical expression "today I’ll go to the movie". Let R be the logical expression "today I’ll go to the game".

The original expression can thus be written as (P and Q) or (P and R) and we can now evaluate the truth table for all of the possibilities.
### 3.1.2 Order of Evaluation of Logical Operators

Just as arithmetic expressions have an order of evaluation (in the expression $2 + 3 \times 5$, $\times$ is evaluated before $+$), so do logical expressions. For logical expressions, not is evaluated first, then and, then or. So the expression above, $(P \text{ and } Q) \text{ or } (P \text{ and } R)$, could have been written $P \text{ and } Q \text{ or } P \text{ and } R$, but the parenthesis greatly improve readability.

### 3.2 Boolean Data and Expressions

**Boolean expression** is merely the fancy programming language terminology for a logical expression. The name boolean comes from the name George Boole, a mathematician of the 1800s that studied logic extensively.

#### 3.2.1 the Primitive Date Type Boolean

Recall that we’ve already studied two primitive numeric data types – `int` and `double`, which are used with numbers. Boolean expressions are so common that many programming languages provide a primitive data type specifically for representing the boolean values. In Java this is the data type `boolean`. `booleans` can have only the value `true` or the value `false`, both of which are Java keywords.

```java
boolean done;  // no value assigned
boolean quitNow = false;
```

#### 3.2.2 Boolean Operators

The logical operators and, or and not have their Java `boolean` equivalents

- `&&` is Java for logical and
- `||` is Java for logical or
- `!` is Java for logical not
The order of evaluation is !, followed by &&, followed by ||, the same as for logical expressions. This means that the expression


\(!P \parallel Q \&\& \!S\)

is evaluated as if we’d put in parenthesis as shown here:


\((!P) \parallel (Q \&\& (!S))\)

To write the most readable code, and because you may work in other programming languages with different orders of evaluation, it is always a good idea to put in parenthesis when there is likely to be confusion. So the above expression would typically be parenthesized as


\(!P \parallel (Q \&\& \!S)\)

There are several mathematical comparison operators that give boolean results.

- > - test for greater than, as in: \(x > 10\)
- < - test for less than, as in: \(x < y\)
- >= - test for greater than or equal to, as in: \(x \geq y\)
- <= - test for less than or equal to, as in: \(x \leq y\)
- == - test for equal to, as in: \(x == y\) (Note: this is two equal signs)
- != - not equal to, as in: \(x \neq y\)

As before, complex expressions should be written with parenthesis to make the order of evaluation clear:


\((x > y) \parallel (x < 11)\)
\(((y \neq 10) \&\& (p \geq 7)) \parallel (z == 4)\)

Truth tables for boolean expressions are evaluated exactly the same as those for logical expressions.

### 3.2.3 Some Boolean Expressions that are Tricky

Some expressions in English can tricky to write in a programming language. Here are some common problems and the correct solutions.

- \(x\) is between 4 and 10
  - equivalent math expression: \(4 < x < 10\)
  - equivalent Java expression: \((4 < x) \&\& (x < 10)\)
  - Note that \(x > 4 \&\& x < 10\) and other similar variations mean exactly the same thing, but probably aren’t as easy to understand.

- \(grade\) is either an 88 or an 89
WRONG!
- grade == 88 || 89
Correct
- (grade == 88) || (grade == 89)
- the difference between x and y is less than 5
WRONG!
- x – y < 5 ← why is this wrong? Well, suppose x is 2 and y is 9. The difference between them is certainly more than 5 (the difference between them is 7 in fact), yet x – y < 5 is true because 2 – 9 is -7, which is less than 5.
Correct
- Math.abs(x - y) < 5 ← correctly asks the question “Is the difference between them, no matter which is larger, less than 5?” Math.abs() is a function from the Java Math library, used to calculated absolute values. Using the Math library doesn’t require an import.

3.2.4 Short Circuit Boolean Evaluation
Assume x has the value 0 and consider the expression (x != 0) && (y/x > 3). Notice that the second subexpression includes y/x, which with our assumption is a division by 0. Will this expression make the program crash due to division by 0 in the right hand subexpression?

The answer is no, there will not be division by 0 or a program crash. Java avoids this problem with short circuit boolean evaluation, which means that a boolean expression is evaluated from left to right and only as much of the expression as needed to determine its truth value is evaluated. Java and many other programming languages use short circuit boolean evaluation.

The expression (x != 0) && (y/x > 3) is evaluated in this manner:
- x != 0 is evaluated. Since x is 0 (by assumption), x != 0 is false.
- the entire expression is therefore false && (y/x > 3).
- after examining the truth table for && (which is actually the truth table for and), we know the entire expression is false without evaluating the second part of the expression (y/x > 3).
- Thus y/x is never executed, and the program does not crash due to division by zero.

3.3 if and if-else Statements
The primary Java mechanisms for making a decision are the if and if-else statements. These statements are used to express ideas such as
- if the shot hit the target kill the monster
- if the password is correct allow entry to the database
• if the employee is a member of the union increase the rate of pay by 5% otherwise increase pay by 4%
• if the space ship is visible make it invisible otherwise destroy it

3.3.1 if Statements
The basic Java mechanism for making a decision is the if statement. Below are simple examples.

if (x > 10)
{
    done = true;
}
println(“Finished!”);

if (jobFinished == true)
{
    count = count + 1;
    x = 4;
}
...next statements...

In general an if statement is in the form

if (boolean expression)
{
    statement1
    statement2
    ...more statements...
}
...next statements...

Below is a simple program that demonstrate the use of an if statement to correctly display the absolute value of a number that has been read from the keyboard.

AbsoluteValue
//AbsoluteValue.java
import acm.program.*;

public class AbsoluteValue extends ConsoleProgram
{
    public void run( )
    {

```java
double x = readDouble("x?");
if (x < 0)
{
    x = -x;
}
println("Absolute value is: "+x);
}
/*run
*/
/*AbsoluteValue*/
```

**Warning!** The ; (semi-colon) is a legitimate, though vacuous statement. Beware of writing something like this:

```java
if (a > b) ;
{
    println("Found max value!");
}
```

### 3.3.2 simple if else Statements

Sometimes we need to express, in code, statements like this.

*if the goblin is holding a laser gun then duck behind shelter*  
*else*  
*blow that dude away*

In Java, as in many languages, this capability is provided with the `if else` statement. Here is an example.

```java
if (Math.abs(value – PI*2) < EPS)
{
    p = 1.2;
}
else
{
    p = 1.3;
    torqueV = p * m;
}
```

An `if else` statement takes the following form

```java
if (boolean expression)
{
    statement1
if the boolean expression is true, execute the statements inside the first set of braces, then jump to next statements
```
...more statements...
}
else
{
    statementN
    statementN+1
    ...more statements...
}
...next statements...

3.3.3 Example Programs

A console example - the user tries to guess a number from 1 to 20, with the program checking for success. When the guess is correct, the program prints a congratulatory message and then quits.

GuessANumber

//GuessANumber
import acm.program.*;

public class GuessANumber extends ConsoleProgram
{
    public void run()
    {
        final int CORRECT_GUESS = 8;
        int guessCount = 0;
        int guess = readInt("Pick a number from 1 to 20: ");

        while(true)
        {
            if (guess != CORRECT_GUESS)
            {
                guessCount++;
                print("You've guessed" + guessCount + "times");
                print("and you've been wrong, wrong, wrong");
                println("every time!");
                guess = readInt("Try again nimrod:");
                break; //get a guess and check it
            }
            else
            {
                println("Congratulations, you're a genius!");
            }
        } //run
    } //run

    break causes execution to exit the loop
    if the boolean expression is false, execute the statements inside the second set of braces, then jump to the next statements.
    either way execute next statements
An animation example
Our task is to move a green ball down and right from the upper left of the screen. Stop the ball when it touches the bottom of the application window. This program is identical to the *MoveGreenBall1* program in Chapter 2 but for the *if* statement inside the *while* endless loop.

**MoveGreenBall2**

```java
//MoveGreenBall2
import acm.program.*;
import acm.graphics.*;
import java.awt.*;
public class MoveGreenBall2 extends GraphicsProgram
{
    public static final int APPLICATION_WIDTH = 500;
    public static final int APPLICATION_HEIGHT = 300;

    public void run()
    {
        setBackground(Color.BLACK);
        GOval ball = new GOval(0, 0, 50, 50);
        ball.setFilled(true);
        ball.setColor(Color.GREEN);
        add(ball);

        waitForClick();
        while(true)
        {
            ball.move(1, 1);
            pause(50);
            // is the ball touching bottom of screen?
            if (ball.getY() + 50 >= APPLICATION_HEIGHT)
            {
                break;  // exit the loop
            }
        }  // move the ball
    }  // run
}  //MoveGreenBall2
```

### 3.3.4 Cascading if-else

More test alternatives may be created with cascading *if-else* statements.
Consider the problem of displaying a letter grade, based on an int grade, using the familiar 90, 80, 70 and 60 scale. Let’s look at several approaches.

Here’s a first try – this works but is inefficient. Why is it inefficient?

```java
if (grade >= 90 && grade <= 100) {
    println("A");
} else if (grade >= 80 && grade <= 89) {
    println("B");
} else if (grade >= 70 && grade <= 79) {
    println("C");
} else if (grade >= 60 && grade <= 69) {
    println("D");
} else if (grade <= 59) {
    println("F");
}
```

Consider what happens if the grade is 92. Then an A is displayed and no further grade determinations are needed.

But the comparisons continue, checking the 92 to see if it is a B, a C, a D and finally an F. All of these comparisons are not necessary.

The code, though it works, is very inefficient.

Second try – somewhat more efficient. Why is this better? What can still be improved?

```java
if (grade >= 90 && grade <= 100) {
    println("A");
} else if (grade >= 80 && grade <= 89) {
    println("B");
} else if (grade >= 70 && grade <= 79) {
    println("C");
} else if (grade >= 60 && grade <= 69) {
    println("D");
} else if (grade <= 59) {
    println("F");
}
```

This code is better because it is written as an if else.

Once the correct match is made (for example, 84 results in the second boolean expression being true), execution jumps past the rest of the comparisons.
println("F");
}

Third try – much more efficient.

if (grade >= 90)
{
    println("A");
}
else if (grade >= 80)
{
    println("B");
}
else if (grade >= 70)
{
    println("C");
}
else if (grade >= 60)
{
    println("D");
}
else
{
    println("F");
}

Fourth try – where there is only one statement in a block, the { } are not necessary, and so we can write this cascading else as

if (grade >= 90)
    println("A");
else if (grade >= 80)
    println("B");
else if (grade >= 70)
    println("C");
else if (grade >= 60)
    println("D");
else
    println("F");

or even as

if (grade >= 90) println("A");
else if (grade >= 80) println("B");
else if (grade >= 70) println("C");
else if (grade >= 60) println("D");

If grade ≥ 90 it must be an A, so why check to see if it is ≤ 100?

If grade ≥ 80 (and it can't be ≥ 90 because the first if would have found it) it must be a B.

If grade ≥ 70 (and can't be ≥ 80 because the first or second if would have found it) it must be a C.

The D comparison is similar.

It grade is not an A, a B, a C or a D, it must be an F, so why check?
else println("F");

The first general form of a cascading if-else statement is

```java
if (boolean expression)
{
    ...statements...
}
else if (boolean expression)
{
    ...statements...
}
else if ... more tests ...
... more statements in the program...
```

and the second general form of a cascading if-else is

```java
if (boolean expression)
{
    ...statements...
}
else if (boolean expression)
{
    ...statements...
}
else if ...
else
{
    ...statements...
}
... more statements in the program...
```

Note the else - if none of the boolean conditions were matched then execute this block

always execute the statements after the cascading if else

3.3.5 Blocks & Scope

A **block** is a sequence of statements contained in a set of braces ( {} ). The body of **run** is a block. The body of the program itself is a block. Each of the clauses in the if else statements above has its own block defined by braces. The **while** loop has its own block also.

The **scope of an identifier** is the portion of the program for which that identifier is defined – that is, where the identifier can be used.

- Variables and constants are often declared at the beginning of a program, right below the **public class** NameOfClass extends NameOfProgram statement. An identifier declared like this may be used from the point of declaration on in the entire program.
A variable declared inside a block has **block scope**, which means it exists only inside that block. We say that the identifier is **local** to the block.

```java
if (a > b)
{
    double mean = (a + b)/2;
    ...statements...
}
```

In the above example, `mean` is defined inside the `{ }`, it has scope in that block, and cannot be used in any way outside the `{ }`.

### 3.3.6 the Dangling Else Question

As might be expected, **if else** code can get complex and the complexity can make it hard to understand.

Which indentation scheme accurately reflects the logic of the code?

<table>
<thead>
<tr>
<th>if (x &gt; 10)</th>
<th>if (x &gt; 10)</th>
</tr>
</thead>
<tbody>
<tr>
<td>if (x &lt; 20)</td>
<td>if (x &lt; 20)</td>
</tr>
<tr>
<td>print(&quot;M&quot;);</td>
<td>print(&quot;M&quot;);</td>
</tr>
<tr>
<td>else</td>
<td>else</td>
</tr>
<tr>
<td>print(&quot;T&quot;);</td>
<td>print(&quot;T&quot;);</td>
</tr>
</tbody>
</table>

In other words, the **else** belongs to which **if**?

This situation, where an **else** follows an **if** that is part of another **if**, is called a **dangling else**, and can easily be confusing to human reading the code.

Human confusion is resolved by a simple rule: assuming there are no braces that change the evaluation, an **else** is associated with the nearest previous **if** that is not already paired with an **else**. Thus the correct indentation is the second, which indicates that the **else** is part of the **if (x < 20)** statement.

Of course, only the programmer may be confused—Java will not be.

We can avoid this difficulty with good use of `{ }` to group statements and make the intention evident.

```java
if (x > 10)
{
    if (x < 20)
    {
        print("M");
    }
}
```
3.3.7 About Braces and Indentation

Simple if and if else statements can be written in several different styles, as illustrated here.

The if statement

```java
if (boolean expression)
{
    statement;
}
```

can be written as

```java
if (boolean expression)
    statement;
```

or even as

```java
if (boolean expression) statement;
```

However, the first style is often the best. It takes a bit more room and typing but makes the intent of the expression absolutely clear, and this is the style we will typically use.

if else statements can be styled similarly, as in

```java
if (boolean expression)
{
    statement1;
}
else
{
    statement2;
}
```

which can also be written as

```java
if (boolean expression)
    statement1;
else
statement2;

or as

```java
if (boolean expression) statement1;
else statement2;
```

In general a block that consists of one and only one statement can be written without the braces, but be very careful to write for clarity first.

### 3.3.8 Testing Real Numbers for Equality

Real numbers (doubles in Java) are not guaranteed to be represented exactly so we must be careful when comparing them. Here's an example of an unexpected problem.

**WRONG!**
```java
double r = 1.1;
if (r == (1.005 + .095))
{
    print("Match");
}
else
{
    print("No Match!");
}
```

This code displays "No Match" even though mathematically 1.1 and 1.005 + .095 are equal. Why doesn’t it work the way we expect? The inherent imprecision associated with real numbers is the reason.

The correct technique for testing real numbers for equality is to test for 'are the values very close?', using the absolute value method from the Java Math library, as shown below.

**CORRECT!**
```java
final double EPS = .001;
double r = 1.1;
if (Math.abs(r - (1.005 + .095)) < EPS)
{
    print("Match");
}
else
{
    print("No Match!");
}
```
3.3.9 Example Programs

**FindRoot** Consider \( f(x) = x^3 - 4 \). Below is a graph of this function with \( x \) values from -1 to 2.5. The point where it crosses the X axis is a solution to the equation \( x^3 - 4 = 0 \). On the graph we see that it crosses the X axis at about 1.5, but exactly where?

Let’s write a program to find the value of this solution. Note that we won’t actually find the exact value, but we’ll find something very, very close—which is good enough for government work.

A general outline of the technique is:

*start with a guess of 1.25, which is somewhat close but too low
while that guess isn’t close enough to a solution*

\[
\text{\{ increment the guess by .01 \}}
\]

*output the guess*

And here’s a more detailed look at the logic.

\[
\text{let guess = 1.25}
\text{let fOfX = guess}^3 - 4
\text{let diff = | fOfX - 0 | \; \leftarrow \text{how close are we to the solution?}}
\text{while (diff > 1/1000) \; \leftarrow \text{while we are not within 1/1000 of the desired value 0}}
\text{\{ increment guess by .0001}
\text{calculate a new fOfX, using same mathematics as before}
\text{calculate a new diff, using same mathematics as before \}}
\]

And finally here’s the Java program that implements this logic.

**FindRoot**
public class FindRoot extends ConsoleProgram
{
    public void run( )
    {
        final double EPS = .001;
        double guess = 1.25;
        double fOfX = guess*guess*guess - 4;
        double diff = Math.abs(fOfX - 0);
        while (diff > EPS)
        {
            guess = guess + .001;
            fOfX = guess*guess*guess - 4;
            diff = Math.abs(fOfX - 0);
        }
        println("Root: \" + guess);
    }
}

FindRoot is a simple example of numerical analysis, a really interesting area of mathematics and computer science that might appeal to the math geeks in the class.

BounceAround1
A program that will bounce a red ball around the application window. It does this by comparing the coordinates of the ball edges with the dimension of the window and adjusting the ball’s movement to represent a bounce.

BounceAround1
import acm.program.*;
import acm.graphics.*;
import java.awt.*;
public class BounceAround1 extends GraphicsProgram
{
    public static final int APPLICATION_WIDTH = 800;
    public static final int APPLICATION_HEIGHT = 300;
    public final int DIAM1 = 30;
    public final int PAUSE1 = 2;
    public void run( )
    {
        Note the new use of the while loop. This loop will run as long as the boolean expression diff > EPS is true, and will stop running as soon as the boolean expression is tested and is no longer true.
        This is a very common way to use while loops.
    }

The important part of the program is the cascading if else statement, where the branches of the statement test the location of the ball to see if the ball is at the edge of the window. Let’s examine each of the tests.

if \((ball1.getY() \leq 0)\) \(\leftarrow\) if the top of the ball is at the top of the window or above the top of the window, then set the value of the vertical jump to 1, so that the next movement is downward.

else if \((ball1.getY()+DIAM1 \geq \text{APPLICATION\_HEIGHT} - 1)\) \(\leftarrow\) if the bottom of the ball is at the bottom of the window or below, set the value of the vertical jump to -1, so that next movement is upward.
else if \((ball1.getX() <= 0)\) \(\leftarrow\) if the left edge of the ball is at the left edge of the window or farther to the left, adjust the horizontal jump to 1 so that the next movement is to the right.

else if \((ball1.getX() + DIAM1 >= APPLICATION_WIDTH - 1)\) \(\leftarrow\) if the right edge of the ball is at the right edge of the window or farther to the right, set the horizontal jump to -1 so that the next movement is to the left.

**Objective Questions**

1. Evaluate the truth table for each expression
   a. \(!P \&\& Q\)
   b. \(!P \&\& Q\)
   c. \(!P || !Q\)
   d. \(P \&\& !(P || Q)\)
   e. \(!P || !(P \&\& Q)\)
   f. \(!P || (Q \&\& R)\)
   g. \(P \&\& !(Q \&\& R)\)
   h. \!(P || Q) \&\& !(P \&\& R)\)

2. Assume \(x\) has the value 10 and \(y\) has the value 20. Evaluate each expression:
   a. \((x > 10) \&\& (y == 20)\)
   b. \(x = y\)
   c. \((x != 5) \&\& (y < -4)\)
   d. \((x == y) \&\& (x != y)\) \(\leftarrow\) Stop! before constructing the truth table, think!
   e. \((x == y) || (x != y)\) \(\leftarrow\) think!

3. Write the Java boolean expression equivalent to the expressions in English.
   Example: hourly rate of pay is more than $12 \(\iff hrp > 12\)
   a. grade is a 70 or 71
   b. temp is no more than 78
   c. temp is between 75 and 80 inclusive
   d. temp is between 75 and 80 exclusive
   e. z is not 80, though it is at least 75
   f. distance from Denver to Chicago is less than 1,000
   g. q is no more than 50 or it is more than 80
   h. t is more than -80 or it is less than -100
   i. t is not equal to 17 but it is at least 20
   j. t is not equal to 17 or 18
   k. t is greater than 50 but is not equal to 7 \(\leftarrow\) think!
   l. t is greater than 50 but is not equal to 75

4. Write Java statements that are equivalent to the English statement
   Example: If the employee is a member of the union increase pay by 10% otherwise increase pay by 8%
   if \((union == true)\)
   {
       \(pay = pay + 0.1 \times pay;\)
   }
else
{
    pay = pay + .8*pay;
}

a. If the value of test is at least 85 increase grade by 4.
b. If the value of test is at least 875 increase grade by 4, otherwise increase grade by 2.
c. if x is 17, 18 or 19 output “fair” otherwise output “poor” and assign x the value 0.
d. if x and y, both real numbers, are equal then assign v the value x-4, assign t the value 0 and output “Score”, Otherwise output “Not equal” and set both x and y to 0.
e. if x is between 30 and 36 output “excellent”, between 25 and 29 output “very good”, between 19 and 24 “good” and less than 19 output “not acceptable”.

5. Find the syntax error in each statement. Of course, there may be no error.
a. boolean done = x > 20;
b. if (x > 10) println(x) else println(“done”);
c. if (x > 10)
    {
    }
else
    {
    p = x * 4;
    }
d. if (x > 10);
e. if (x > 10) print(“More”); else println(“Stop”);

Short Answer Problems

Console Program Problems

1. **PayRaise** Write a program that inputs a real number representing a current rate of pay and an integer representing an employee class and outputs the new rate of pay after computation according to this scale

<table>
<thead>
<tr>
<th>employee code</th>
<th>pay increase</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>3%</td>
</tr>
<tr>
<td>2</td>
<td>5%</td>
</tr>
<tr>
<td>3</td>
<td>7%</td>
</tr>
</tbody>
</table>

2. **SmallPrime** Write a program that inputs an integer between 1 and 10 and, using a cascading if-else, tells the user if this value is a prime number.

3. **SmallPrime2** Rewrite SmallPrime so that instead of a cascading if-else you use one if statement with several or operators.

4. **FindRoot2** The equation \( x^2 - 8.1 = 0 \) has a solution near 2. Write a program to find that solution, and the other solution which is near -2, using the algorithm in the FindRoot program above.

Graphics Program Problems
1. **SunSetting** Simulate the setting of a yellow sun against a blue sky. The screen turns black when the sun is completely off the screen, and after some delay the phrase “Good night” appears in large white letters, and then disappears.

2. **FancyMoon** Against a black background, a white moon moves from top to bottom of the screen, angling slightly to the right as it does so. As it moves, the size should shrink until it is a small circle. When it hits the bottom, the moon stops. After a slight delay it turns orange. After another slight delay the background then turns red.

3. **BounceAround2** Modify the BounceAround program so that two balls bounce around the graphic window, with the second ball moving more slowly than the first.

4. **ZigZag** Write a program that bounces a ball from the left edge of the window to the right edge, moving downward with each bounce. Stop when the ball hits the bottom of the window. The bounce pattern is shown below.

```
  o  
 o o  
```

5. **UFOCrash** A UFO flies across the screen. As it flies it changes in some manner (perhaps the exhaust flame flickers or lights flash). To the right is a space mine, which doesn't move but it does change shape, color or some other characteristic. The UFO hits the space mine. When it does the UFO blows up into several pieces but the alien parachutes to the ground. After the alien has landed the program concludes with the message *The End* which scrolls right to left, starting at the right side.