Chapter 5
Selection

Goals

It is sometimes appropriate for certain actions to execute one time but not at other times. Sometimes the specific code that executes must be chosen from many alternatives. This chapter presents statements that allow such selections. After studying this chapter, you will be able to:

- see how Java implements the Guarded Action pattern with the `if` statement
- implement the Alternative Action pattern with the Java `if else`
- implement the Multiple Selection pattern with nested the `if else` statement

---

5.1 Selection

Programs must often anticipate a variety of situations. For example, an automated teller machine (ATM) must serve valid bank customers, but it must also reject invalid access attempts. Once validated, a customer may wish to perform a balance query, a cash withdrawal, or a deposit. The code that controls an ATM must permit these different requests. Without selective forms of control—the statements covered in this chapter—all bank customers could perform only one particular transaction. Worse, invalid PINs could not be rejected!

Before any ATM becomes operational, programmers must implement code that anticipates all possible transactions. The code must turn away customers with invalid PINs. The code must prevent invalid transactions such as cash withdrawal amounts that are not in the proper increment (of 10.00 or 20.00, for instance). The code must be able to deal with customers who attempt to withdraw more than they have. To accomplish these tasks, a new form of control is needed—a way to permit or prevent execution of certain statements depending on the current state.

The Guarded Action Pattern

Programs often need actions that do not always execute. At one moment, a particular action must occur. At some other time—the next day or the next millisecond perhaps—the same action must be skipped. For example, one student may make the dean's list because the student's grade point average (GPA) is 3.5 or higher. That student becomes part of the dean's list. The next student may have a GPA lower than 3.5 and should not become part of the dean's list. The action—adding a student to the dean's list—is guarded.

Algorithmic Pattern 5.1

Pattern:	Guarded Action
Problem:	Do something only if certain conditions are true.
Outline:	if (true-or-false-condition is true) execute this action
Code Example:	`if (GPA >= 3.5)` `System.out.println("Made the dean's list");`

The `if` Statement

This Guarded Action pattern occurs so frequently it is implemented in most programming languages with the **`if`** statement.

***General Form:* `if` statement**

`if` `(`*Boolean-expression***`)`**
    true-part

A *Boolean-expression* is any expression that evaluates to either true or false. The *true-part* may be any valid Java statement, including a block. A block is a sequence of statements within the braces `{` and `}`.

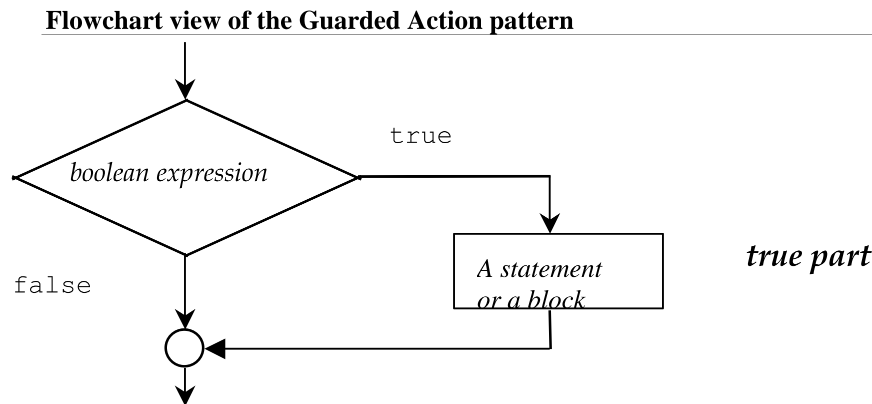
Examples of `if` Statements

```
if (hoursStudied > 4.5)
  System.out.println("You are ready for the test");

if (hoursWorked > 40.0) {
  // With a block with { } for the true part so both statements may execute
  regularHours = 40.0;
  overtimeHours = hoursWorked - 40.0;
}
```

When an `if` statement is encountered, the boolean expression is evaluated to false or true. The “true part” executes only if the boolean expression evaluates to true. So in the first example above, the output `"You are ready for the test"` appears only when the user enters something greater than 4.5. When the input is 4.5 or less, the true part is skipped—the action is guarded. Here is a flowchart view of the Guarded Action pattern:

Flowchart view of the Guarded Action pattern



A test method for `withdraw` illustrates that a `BankAccount` object should not change for negative arguments.

```
@Test
public void testGetWithdrawWhenNotPositive() {
  BankAccount anAcct = new BankAccount("Angel", 100.00);
  // Can't withdraw amounts <= 0.0;
  anAcct.withdraw(0.00);
  // Balance remains the same
  assertEquals(100.00, anAcct.getBalance(), 0.1);
  anAcct.withdraw(-0.99);
  // Balance remains the same
  assertEquals(100.00, anAcct.getBalance(), 0.1);
}
```

Nor should any `BankAccount` object change when the amount is greater than the balance.

```
@Test
public void testGetWithdrawWhenNotEnoughMoney() {
  BankAccount anAcct = new BankAccount("Angel", 100.00);
  // Do not want withdrawals when the amount > balance;
  anAcct.withdraw(100.01);
  // Balance should remain the same
  assertEquals(100.00, anAcct.getBalance(), 0.1);
}
```

The if statement in this modified `wthdraw` method guards against changing the balance—an instance variable—when the argument is negative or greater than the balance

```
public void withdraw(double withdrawalAmount) {
  if (withdrawalAmount > 0.00 && withdrawalAmount <= balance) {
    balance = balance - withdrawalAmount;
  }
}
```

Through the power of the `if` statement, the same exact code results in two different actions. The `if` statement controls execution because the true part executes only when the Boolean expression is true. The `if` statement also controls statement execution by disregarding statements when the Boolean expression is false.

Self-Check

5-1 Write the output generated by the following pieces of code:

-a
```
int grade = 45;
if(grade >= 70)
  System.out.println("passing");
if(grade < 70)
  System.out.println("dubious");
if(grade < 60)
  System.out.println("failing");
```

-b
```
int grade = 65;
if( grade >= 70 )
  System.out.println("passing");
if( grade < 70 )
  System.out.println("dubious");
if( grade < 60 )
  System.out.println("failing");
```

-c
```
String option = "D";
if(option.equals("A"))
  System.out.println( "addRecord" );
if(option.equals("D"))
  System.out.println("deleteRecord");
```

5.2 The Alternative Action Pattern

Programs must often select from a variety of actions. For example, say one student passes with a final grade that is ≥ 60.0. The next student fails with a final grade that is < 60.0. This example uses the Alternative Action algorithmic pattern. The program must choose one course of action or an alternative.

Algorithmic Pattern: Alternate Action

Pattern:	Alternative Action
Problem:	Need to choose one action from two alternatives.
Outline:	if (true-or-false-condition is true) execute action-1 otherwise execute action-2

Code Example:

```
if(finalGrade >= 60.0)
   System.out.println("passing");
else
   System.out.println("failing");
```

The `if else` Statement

The Alternative Action pattern can be implemented with Java's **`if else`** statement. This control structure can be used to choose between two different courses of action (and, as shown later, to choose between more than two alternatives).

***The* `if else` Statement**

`if``(`*boolean-expression*`)`
  true-part
`else`
  false-part

The `if else` statement is an `if` statement followed by the alternate path after an `else`. The *true-part* and the *false-part* may be any valid Java statements or blocks (statements and variable declarations between the curly braces `{` and `}`).

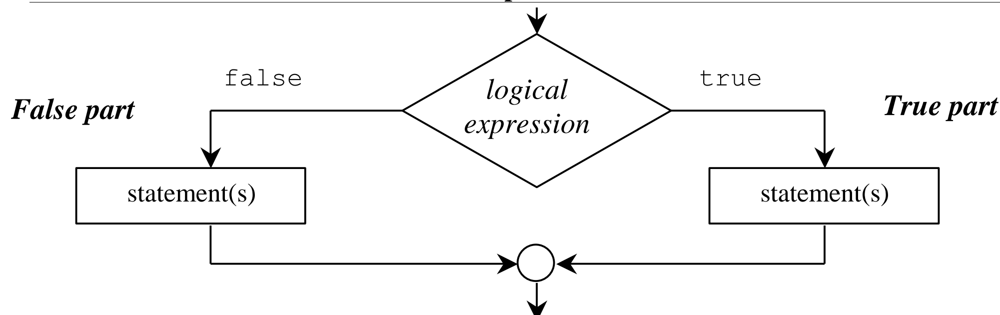
Example of if else Statements

```
if (sales <= 20000.00)
  System.out.println("No bonus");
else
  System.out.println("Bonus coming");


if (withdrawalAmount <= myAcct.getBalance()) {
  myAcct.withdraw(withdrawalAmount);
  System.out.println("Current balance: " + myAcct.getBalance());
}
else {
  System.out.println("Insufficient funds");
}
```

When an `if else` statement is encountered, the Boolean expression evaluates to either `false` or `true`. When `true`, the true part executes—the false part does not. When the Boolean expression evaluates to `false`, only the false part executes.

Flowchart view of the Alternative Action pattern



Self-Check

5-2 Write the output generated by each code segment given these initializations of `j` and `x`:

```
int j = 8;
double x = -1.5;
```

```
-a  if(x < -1.0)
      System.out.println("true");
    else
      System.out.println("false");
      System.out.println("after if...else");

-b  if(j >= 0)
      System.out.println("zero or pos");
    else
      System.out.println("neg");

-c  if(x >= j)
      System.out.println("x is high");
    else
      System.out.println("x is low");

-d  if(x <= 0.0)
      if(x < 0.0) // True part is another if...else
        System.out.println("neg");
      else
        System.out.println("zero");
    else
      System.out.println("pos");
```

5-3 Write an `if else` statement that displays your name if `int option` is an odd integer or displays your school if `option` is even.

A Block with Selection Structures

The special symbols `{` and `}` have been used to gather a set of statements and variable declarations that are treated as one statement for the body of a method. These two special symbols delimit (mark the boundaries) of a block. The block groups together many actions, which can then be treated as one. The block is also useful for combining more than one action as the true or false part of an `if else` statement. Here is an example:

```
double GPA;
double margin;
// Determine the distance from the dean's list cut-off
Scanner keyboard = new Scanner(System.in);
System.out.print("Enter GPA: ");
GPA = keyboard.nextDouble();

if(GPA >= 3.5) {
  // True part contains more than one statement in this block
  System.out.println("Congratulations, you are on the dean's list.");
  margin = GPA - 3.5;
  System.out.println("You made it by " + margin + " points.");
}
else {
  // False part contains more than one statement in this block
  System.out.println("Sorry, you are not on the dean's list.");
  margin = 3.5 - GPA;
  System.out.println("You missed it by " + margin + " points.");
}
```

The block makes it possible to treat many statements as one. When `GPA` is input as `3.7`, the Boolean expression (`GPA >= 3.5`) is true and the following output is generated:

Dialog

```
Enter GPA: 3.7
Congratulations, you are on the dean's list.
You made it by 0.2 points.
```

When `GPA` is `2.9`, the Boolean expression (`GPA >= 3.5`) is false and this output occurs:

Dialog

```
Enter GPA: 2.9
Sorry, you are not on the dean's list.
You missed it by 0.6 points.
```

This alternate execution is provided by the two possible evaluations of the boolean expression `GPA >= 3.5`. If it evaluates to true, the true part executes; if false, the false part executes.

The Trouble in Forgetting { and }

Neglecting to use a block with `if else` statements can cause a variety of errors. Modifying the previous example illustrates what can go wrong if a block is not used when attempting to execute both output statements.

```
if(GPA >= 3.5)
  margin = GPA - 3.5;
  System.out.println("Congratulations, you are on the dean's list.");
  System.out.println("You made it by " + margin + " points.");
else  // <- ERROR: Unexpected else
```

With `{` and `}` removed, there is no block; the two bolded statements no longer belong to the preceding `if else`, even though the indentation might make it appear as such. This previous code represents an `if` statement followed by two `println` statements followed by the reserved word `else`. When `else` is encountered, the Java compiler complains because there is no statement that begins with an `else`.

Here is another example of what can go wrong when a block is omitted. This time, `{` and `}` are omitted after `else`.

```
else
  margin = 3.5 - GPA;
  System.out.println("Sorry, you are not on the dean's list.");
  System.out.println("You missed it by " + margin + " points.");
```

There are no compiletime errors here, but the code does contain an intent error. The final two statements always execute! They do not belong to `if else`. If `GPA >= 3.5` is false, the code does execute as one would expect. But when this boolean expression is true, the output is not what is intended. Instead, this rather confusing output shows up:

```
Congratulations, you are on the dean's list.
You made it by 0.152 points.
Sorry, you are not on the dean's list.
You missed it by -0.152 points.
```

Although it is not necessary, always using blocks for the true and false parts of `if` and `if else` statements could help you. The practice can make for code that is more readable. At the same time, it could help to prevent intent errors such as the one above. One of the drawbacks is that there are more lines of code and more sets of curly braces to line up. In addition, the action is often only one statement and the block is not required.

5.3 Multiple Selection

"Multiple selection" refers to times when programmers need to select one action from many possible actions. This pattern is summarized as follows:

Algorithmic Pattern: Multiple Selection

Pattern:	Multiple Selection
Problem:	Must execute one set of actions from three or more alternatives.

Outline:	if (condition-1 is true)
	  execute action-1
	else if(condition-2 is true)
	  execute action-2
	else if(condition n-1 is true)
	  execute action n-1
	else
	  execute action-n

Code Example:

```
// Return a message related to the "comfyness"
// of the size of the string argument
public String comfy(String str) {
  String result = "?";
  int size = str.length();

  if (size < 2)
    result = "Way too small";
  else if (size < 4)
    result = "Too small";
  else if (size == 4)
    result = "Just right";
  else if (size > 4 && size <= 8)
    result = "Too big";
  else
    result = "Way too big";

  return result;
}
```

The following code contains an instance of the Multiple Selection pattern. It selects from one of three possible actions. Any grade point average (GPA) less than 3.5 (including negative numbers) generates the output "Try harder." Any GPA less than 4.0 but greater than or equal to 3.5 generates the output "You made the dean's list." And any GPA greater than or equal to 4.0 generates the output "You made the president's list." There is no upper range or lower range defined in this problem.

```
// Multiple selection, where exactly one println statement
// executes no matter what value is entered for GPA.
Scanner keyboard = new Scanner(System.in);
System.out.print("Enter your GPA: ");
double GPA = keyboard.nextDouble();
if (GPA < 3.5)
  System.out.println("Try harder");
else {
  // This false part of this if else is another if else
  if (GPA < 4.0)
    System.out.println("You made the dean's list");
  else
    System.out.println("You made the president's list");
}
```

Notice that the false part of the first `if else` statement is another `if else` statement. If `GPA` is less than 3.5, `Try harder` is output and the program skips over the nested `if else`. However, if the `boolean` expression is false (when `GPA` is greater than or equal to 3.5), the false part executes. This second `if else`

statement is the false part of the first `if else`. It determines if `GPA` is high enough to qualify for either the dean's list or the president's list.

When implementing multiple selection with `if else` statements, it is important to use proper indentation so the code executes as its written appearance suggests. The readability that comes from good indentation habits saves time during program implementation. To illustrate the flexibility you have in formatting, the previous multiple selection may be implemented in the following preferred manner to line up the three different paths through this control structure:

```
if (GPA < 3.5)
  System.out.println("Try harder");
else if (GPA < 4.0)
  System.out.println("You made the dean's list");
else
  System.out.println("You made the president's list");
```

Another Example — Determining Letter Grades

Some schools use a scale like the following to determine the proper letter grade to assign to a student. The letter grade is based on a percentage representing a weighted average of all of the work for the term. Based on the following table, all percentage values must be in the range of 0.0 through 100.0:

Value of Percentage	Assigned Grade
90.0 ≤ percentage ≤ 100.0	A
80.0 ≤ percentage < 90.0	B
70.0 ≤ percentage < 80.0	C
60.0 ≤ percentage < 70.0	D
0.0 ≤ percentage < 60.0	F

This problem is an example of choosing one action from more than two different actions. A method to determine the range `weightedAverage` falls into could be implemented with unnecessarily long separate `if` statements:

```
public String letterGrade(double weightedAverage) {
  String result = "";

  if(weightedAverage >= 90.0 && weightedAverage <= 100.0)
    result = "A";
  if(weightedAverage >= 80.0 && weightedAverage < 90.0)
    result = "B";
  if(weightedAverage >= 70.0 && weightedAverage < 80.0)
    result = "C";
  if(weightedAverage >= 60.0 && weightedAverage < 70.0)
    result = "D";
  if(weightedAverage >=  0.0 && weightedAverage < 60.0)
    result = "F";

  return result;
}
```

When given the problem of choosing from one of six actions, it is better to use multiple selection, not guarded action. The preferred multiple selection implementation—shown below—is more efficient at runtime. The solution above is correct, but it requires the evaluation of six complex `boolean` expression every time. The solution shown below, with nested `if else` statements, stops executing when the first boolean test evaluates to true. The true part executes and all of the remaining nested `if else` statements are skipped.

Additionally, the multiple selection pattern shown next is less prone to intent errors. It ensures that an error message will be returned when `weightedAverage` is outside the range of 0.0 through 100.0 inclusive. There is a possibility, for example, an argument will be assigned to `weightedAverage` as 777 instead of 77. Since 777 >= 90.0 is `true`, the method in the code above could improperly return an emtpy String when a `"C"` would have likely been the intended result.

The nested `if else` solution first checks if `weightedAverage` is less than 0.0 or greater than 100.0. In this case, an error message is concatenated instead of a valid letter grade.

```
if ((weightedAverage < 0.0) || (weightedAverage > 100.0))
  result = weightedAverage + " not in the range of 0.0 through 100.0";
```

If `weightedAverage` is out of range—less than 0 or greater than 100—the result is an error message and the program skips over the remainder of the nested `if else` structure. Rather than getting an incorrect letter grade for percentages less than 0 or greater than 100, you get a message that the value is out of range.

However, if the first `boolean` expression is `false`, then the remaining nested `if else` statements check the other five ranges specified in the grading policy. The next test checks if `weightedAverage` represents an A. At this point, `weightedAverage` is certainly less than or equal to 100.0, so any value of `weightedAverage >= 90.0` sets `result` to `"A"`.

```
public String letterGrade(double weightedAverage) {
    String result = "";
    if ((weightedAverage < 0.0) || (weightedAverage > 100.0))
      result = weightedAverage + " not in the range of 0.0 through 100.0";
    else if (weightedAverage >= 90)
      result = "A";
    else if (weightedAverage >= 80.0)
      result = "B";
    else if (weightedAverage >= 70.0)
      result = "C";
    else if (weightedAverage >= 60.0)
      result = "D";
    else
      result = "F";
    return result;
  }
```

The return value depends on the current value of `weightedAverage`. If `weightedAverage` is in the range and is also greater than or equal to 90.0, then "A" will be the `result`. The program skips over all other statements after the first `else`. If `weightedAverage == 50.0`, then all `boolean` expressions are `false` and the program executes the action after the final `else`; `"F"` is concatenated to `result`.

Testing Multiple Selection

Consider how many method calls should be made to test the `letterGrade` method with multiple selection—or for that matter, any method or segment of code containing multiple selection. To test this particular example to ensure that multiple selection is correct for all possible percentage arguments, the method could be called with all numbers in the range from -1.0 through 101.0. However, this would require an infinite number of method calls for arguments such as 1.000000000001 and 1.999999999999, for example. With integers, it would be a lot easier, but still tedious. Such testing is unnecessary.

First consider a set of test data that executes every possible branch through the nested if else. Branch coverage testing means observing what happens when every statement (including the true and false parts) of a nested if else executes once.

Testing should also include the cut-off (boundary) values. This extra effort could go a long way. For example, testing the cut-offs might avoid situations where students with 90.0 are accidentally shown to have a letter grade of B rather than A. This would occur when the Boolean expression `(percentage >= 90.0)` is accidentally coded as `(percentage > 90.0)`. The arguments of 60.0, 70.0, 80.0, and 90.0 complete the boundary testing of the code above.

The best testing strategy is to select test values that combine branch and boundary testing at the same time. For example, a percentage of 90.0 should return `"A"`. The value of 90.0 not only checks the path for returning an A, it also tests the boundary—90.0 as one cut-off. Counting down by tens to 60 checks all

boundaries. However, this still misses one path: the one that sets result to `"F"`. Adding 59.9 completes the test driver. These three things are necessary to correctly perform branch coverage testing:

- Establish a set of data that executes all branches (all possible paths through the multiple selection) and boundary (cut-off) values.
- Execute the portion of the program containing the multiple selection for all selected data values. This can be done with a test method and several assertions.
- Observe that the all assertions pass (green bar).

For example, the following data set executes all branches of `letterGrade` while checking the boundaries:

```
101.1  -0.1  0.0  59.9  60.0  69.9  70.0  79.9  80.0 89.9  90.0  99.9  100.0
```

These two methods do branch and boundary testing.

```
@Test
public void testLetterGradeWhenArgumentNotInRange() {
  assertEquals("100.1 not in the range of 0.0 through 100.0", letterGrade(100.1));
  assertEquals("-0.1 not in the range of 0.0 through 100.0", letterGrade(-0.1));
}

@Test
public void testLetterGradeWhenArgumentIsInRange() {
  assertEquals("F", letterGrade(0.0));
  assertEquals("F", letterGrade(59.9));
  assertEquals("D", letterGrade(60.0));
  assertEquals("D", letterGrade(69.9));
  assertEquals("C", letterGrade(70.0));
  assertEquals("C", letterGrade(79.9));
  assertEquals("B", letterGrade(80.0));
  assertEquals("B", letterGrade(89.9));
  assertEquals("A", letterGrade(90.0));
  assertEquals("A", letterGrade(99.9));
  assertEquals("A", letterGrade(100.0));
}
```

Self-Check

5-4 Which value of `weightedAverage` detects the intent error in the following code when you see this feedback from JUnit org.junit.ComparisonFailure: expected:<[A]> but was:<[B]>?

```
if(weightedAverage > 90)
  result = "A";
else if(weightedAverage >=80)
  result = "B";
else if(weightedAverage >= 70)
  result = "C";
else if(weightedAverage >= 60)
  result = "D";
else
   result = "F";
```

5-5 What `String` would be incorrectly assigned to `letterGrade` for this argument (answer to 5-4)?

5-6 Would you be happy if your grade were incorrectly computed in this manner?

Use method `currentConditions` *to answer the questions that follow*

```
    public String currentConditions(int currentTemp) {
      String result;
      if (currentTemp <= -40)
        result = "dangerously cold";
      else if (currentTemp <= 0)
        result = "freezing";
      else if (currentTemp <= 10)
        result = "cold";
      else if (currentTemp <= 20)
        result = "mild";
      else if (currentTemp <= 30)
        result = "warm";
      else if (currentTemp <= 40)
        result = "hot";
      else if (currentTemp <= 45)
        result = "very hot";
      else
        result = "dangerously hot";
      return result;
    }
  }
```

5-7 List the range of integers that would cause `currentConditions` to return `warm`.

5-8 List a range of integers that would cause `currentConditions` to return `freezing`.

5-9 Establish a list of arguments that tests the boundaries in `currentConditions`.

5-10 Establish a list of arguments that tests the branches in `currentConditions`.

5-11 Write in the correct expected value so each assertion passes.

```
import static org.junit.Assert.*;
import org.junit.Test;

public class LittleWeatherTest {

  @Test
  public void testLittleWeather() {

    assertEquals("________________", currentConditions(-41));

    assertEquals("________________", currentConditions(-40));

    assertEquals("________________", currentConditions(-39));

    assertEquals("________________", currentConditions(0));

    assertEquals("________", currentConditions(1));

    assertEquals("________", currentConditions(10));

    assertEquals("________", currentConditions(11));

    assertEquals("________", currentConditions(20));

    assertEquals("________", currentConditions(21));

    assertEquals("________", currentConditions(30));

    assertEquals("________", currentConditions(31));

    assertEquals("________", currentConditions(40));

    assertEquals("________", currentConditions(41));

    assertEquals("________", currentConditions(45));

    assertEquals("________", currentConditions(46));
  }
```

Answers to Self-Check Questions

5-1 -a `dubious`
`failing`
-b `dubious`
-c `deleteRecord`

5-2 -a `true`
`after if else` *The last prinltn is not part of the else. It always executes*
-b `zero or pos`
-c `x is low`
-d `neg`

5-3
```
if(option % 2 == 0)
  System.out.println( "Your School" );
else
  System.out.println( "Your name" );
```

5-4 90

5-5 B (instead of the deserved A).

5-6 I wouldn't be happy; I doubt you would either.

5-7 21 through 30 inclusive

5-8 -39 through 0 inclusive

5-9 -40 0 10 20 30 40 45

5-10 any integer < -41, -15 (or any integer in the range of -30 through -1), 5, 15, 25, 35, 42, and any integer > 46

5-11
```
assertEquals("dangerously cold", currentConditions(-41));
assertEquals("dangerously cold", currentConditions(-40));
assertEquals("freezing", currentConditions(-39));
assertEquals("freezing", currentConditions(0));
assertEquals("cold", currentConditions(1));
assertEquals("cold", currentConditions(10));
assertEquals("mild", currentConditions(11));
assertEquals("mild", currentConditions(20));
assertEquals("warm", currentConditions(21));
assertEquals("warm", currentConditions(30));
assertEquals("hot", currentConditions(31));
assertEquals("hot", currentConditions(40));
assertEquals("very hot", currentConditions(41));
assertEquals("very hot", currentConditions(45));
assertEquals("dangerously hot", currentConditions(46));
```