Introduction to Interactive Programming
by Lynn Andrea Stein
A Rethinking CS101 Project

Statements and Rules

Chapter Summary

- How do I tell the computer how to do something?

This chapter introduces **statements**, the simplest forms of complete executable instructions. Statements are fragments of Java code that have neither value nor type; instead, they have effects. Statements can be are combined to form rules, or services that one object can provide to another. Statements and rules form the backbone of the peanut-butter and jelly model of programming.

Statements can be built out of expressions. However, unlike expressions, which have both type and value, statements are used for their effect -- to get something done. Examples of this are asking a thing to do something or assigning a name to keep track of a value. In addition to declarations, assignments, and method invocation, this chapter introduces simple control flow statements. More advanced statement types are introduced later in the book.

The chapter ends with a discussion of methods, the rules implementing behavior. Method invocation provides the basis for virtually all inter-object interaction.

This chapter is supplemented by a reference chart on the syntax and semantics of java statements.

Objectives of this Chapter

1. To appreciate the difference between evaluating an expression and executing a statement.
2. To be able to read and understand basic statements including assignments, method invocations, declarations, blocks, conditionals, and loops.
3. To learn how to combine statements to construct rules that implement method behavior.

Statements and Instruction-Followers

In the first chapter of this book, we saw that computations are made of communities of interacting entities. Each of these entities may be a community of smaller entities, until eventually an entity can be subdivided no more. At that point, an entity is a simple instruction-follower that provides behavior -- often in the form of ongoing services -- to the other members of its community. This chapter is about how those instructions work. Towards the end of the chapter, we will begin to see how instructions can be combined to form special sequences that articulate how service requests can be fulfilled.

In the previous chapter, we saw how to create Java expressions. An expression is a piece of Java code with a value and a type. The process of producing the value from an expression is called evaluating that expression. The purpose of evaluating an expression is generally to produce its value.

In contrast, statements are all about their side effects. A statement is a piece of executable Java code without either a type or a value. That is, a statement does something (changes something, produces some visible behavior, etc.). It has an effect. It does not have a value. A statement is **executed** (producing an effect), not evaluated (producing a value).

In order to evaluate an expression, you must evaluate its subexpressions, then use the evaluation rule for that kind of expression to produce an appropriate value of an appropriate type. If you understand the evaluation rules for each type of expression, you understand how expressions work.

Understanding how to execute a statement is similar. A statement is not defined by a type and a value (it doesn't have either!), but by its *effects* and by *what happens next*. That is, statements do things; they change the values associated with names. And statements can also cause you to skip around in the instructions that you are following. This is called **flow of control**: what instruction to follow next. Some of these control flow statements involve conditions (*if* it's raining, do this) or loops (keep doing this *until* the light changes color). And many statements involve either subexpressions--which must be evaluated--or substatements--which must be executed in order to execute the superstatement.

Simple Statements

Perhaps the simplest kind of statement is one built directly out of an expression, such as

```
this.who = name;
```

or

```
Console.println( "Hello" );
```

Note the trailing semicolon following the ends of these expressions. It is this semicolon that converts these expressions into statements.

What kinds of expressions can be used to form statements? Only side-effecting expressions. Many expressions are useful solely because of the value that they compute. But a statement doesn't have a value; it has effects on state and control flow. So an expression whose primary purpose is the value it produces doesn't make a very good basis for a statement on its own.[Footnote: These expressions may find use in other, more complex statements, though.] In fact, it is not legal in Java to make an expression-semicolon statement out of a non-side-effecting expression. (For example, x + 3; is not a legal statement.)

However, some expressions do more than just produce values when they are evaluated. For example, an expression like `x = 3` has the value `3` (and the type `int`, assuming that `x` is an `int`). It also (and more importantly) has the effect of storing the value 3 in the shoebox named `x`. This effect (of evaluating the expression) is called a **side effect**. All assignment expressions (including compound assignments) are side effecting. Autoincrement and autodecrement are also side-effecting expressions. Method invocation expressions are also side-effecting, although not every method invocation actually has a side effect. Instance creations -- `new` expressions -- are also side-effecting.

So, for example, a simple assignment statement can be made by adding a semicolon to the end of the assignment expression `x = 3`

```
x = 3;
```

The semicolon turns this into a statement. It no longer has a value or a type; it just does its work.

To execute an expression-semicolon statement, simply evaluate the expression. Of course, this expression may have complicated subexpressions that must be evaluated according to the rules described in the previous chapter. Since the expression is a side-effecting one, something will happen -- an effect will be produced -- during the evaluation.

After executing a side-effecting-expression-plus-semicolon statement, execution proceeds at the following statement.

Declarations and Definitions

We have also already seen declarations in Chapter 3. A declaration creates a new name that can be used to store (in the case of primitive types) or label (in the case of reference types) a value. A declaration follows the *type-of-thing name-of-thing* rule: It consists of a Java type followed by a Java name, then a semicolon. For example,

```
int i;
Object thing;
```

A declaration (or definition) statement creates a kind of name called a **local variable**.

You can actually declare multiple names of a single type with one declaration statement. The syntax for this is *`type-of-thing name-of-thing1, name-of-thing2,`* and so on, with commas between the names and a semicolon at the end:

```
int i, j, k;
Object thingOne, thingTwo;
```

The same type is associated to each of the comma-separated names, so the declarations above are identical to

```
int i;
int j;
int k;
```

and

```
Object thingOne; Object thingTwo;
```

respectively.

Style Sidebar

Formatting Declaration Statements

Remember that Java doesn't care how much white space you leave between things, so there is no difference in meaning between putting the multiple declarations on one line or many. It is definitely easier to read on multiple lines, though, so the convention is to put each declaration on its own line.

When one declaration statement is used to declare many names, you can put the names on one line or on several. It's good style to indent all of the names on subsequent lines of a single declaration so that they line up with the first name declared:

```
Object thingWithALongName,
       anotherThingWithALongName;
```

This way, it's easy to see that `anotherThingWithALongName` is involved in the same declaration statement as `thingWithALongName`.

Although it is technically correct to mix declarations and definitions of a single type using the comma-separated multiple declaration notation, this is not good style. It is too easy to miss a definition among the declarations; mixing the two makes your code unnecessarily harder to read.

A declaration makes it legal to use the name to hold/label appropriately typed values. But the declaration, by itself, doesn't explicitly assign a value to the name. In fact, for the most generic kind of name--a local variable--it is illegal to use a name without first assigning it a value. [Footnote: It is, however, legal to assign a label-name local variable the special non-value `null`. Assigning `null` to a name means that the name doesn't refer to anything. Not assigning forces the computer to guess. The rule is that you just can't leave the computer to guess.] You can assign this value directly in the declaration (making it a definition), or you can assign it before the first time that you try to use the name's associated value.

A variant on a declaration statement is a definition. A definition is a declaration statement with `= `*`expr`* between the *`name-of-thing`* and the semicolon (or comma). This statement declares the name, but it also assigns it the value of *`expr`*. For example:

```
int     i = 2;
String  who = "Pat";
double  pi = 3.14159,
        ninetyDegrees = pi / 2;
```

Note that the final statement here assigns the value 1.570795 to the name `ninetyDegrees`. First 3.14159 is put into the shoebox named `pi.` Next, the expression `pi / 2` is evaluated: its value is the value inside the `pi` shoebox divided by 2. Finally, this value is assigned to (stored in) the (newly created) shoebox named `ninetyDegrees`.

It is legal to mix declarations and definitions in a single statement -- assigning initial values to only some of the names -- but this can make your code hard to read. It is usually better to use multiple statements in this case.

Executing a declaration statement creates a shoebox or label associated with the name declared. Executing a definition is the same as declaring a name, plus immediately afterwards executing an assignment

statement. Note that this assignment is an expression and may have subexpressions, causing a significant amount of evaluation before execution is complete.

After executing a declaration or definition statement, execution proceeds at the immediately following statement.

Sequence Statements

You can also make a bigger statement out of a collection of statements. You do this by enclosing them in **braces**:

```
{
    int i = 3;
    Console.println( "i is " + i );
    int j = i + 1;
    j = i + 5;
}
```

This statement-made-of-statements is a **block**, and it mostly serves to organize your code. Some other statements -- such as `if`, described below -- are often used together with blocks.

Any statement can be used at any point inside a block. In particular, declarations and definitions may appear anywhere in a block. This is useful as it allows you to declare a name immediately before you need it. Doing so makes it easier to read your code as the reader is less likely to have forgotten what you mean by that name.

Blocks also have implications for scoping of names: a variable has scope (its name can be used) from the point in the code where it is declared until the end of the first enclosing block. [Footnote: Remember, not all names are variables. We will learn more about parameters and fields in subsequent chapters. Type names have scope everywhere that they are visible.] So if we declare a name at the top of the block, it has scope for the whole block, as `i` does in the example above. But `j` is not declared until *after* the call to println, so the definition of `i` and the call to println are outside of `j`'s scope:

```
{                                   _
    int i = 3;                      |
    Console.println( "i_is " + i );   |
    int j = i + 1;      | scope of j   | scope of i
    j = i + 5;          |              |
}                       X              X
```

This means, for example, that it would be illegal to use `j` in `i`'s definition:

```
{
    int i = j;    // illegal use of j outside its scope!
    Console.println( "i is " + i );
    int j = i + 1;
    j = i + 5;
}
```

Beware: The scope of a local variable only persists until the end of the enclosing block. This means that a local variable must be declared at the same level as (or at a level enclosing) each of its uses.

```
{
    {
        // A variable declared here...
        String name;
    }
    // ...is invisible here, making this reference
    name = "Pat";
    // illegal!
}
```

// ...and so on.

The rules for executing a block statement are: execute each substatement in turn, from the top (beginning) of the block to the bottom (end) of the block.

After a block, execution continues at the next statement.

Style Sidebar

Formatting Blocks

The open brace of a block should generally appear on its own line. If the block is part of a compound statement (such as an if), its opening brace can appear as the last character on a line. However, studies have found code using this convention harder for programmers to scan than code in which the open brace appears alone on a line.

Text within a block should always be indented (typically by two or four characters). This makes the left-hand margin of code in a block line up. The text -- but not the braces -- of an interior block is indented further; the original indent is resumed when the interior block is closed, i.e., after the closing brace.

The closing brace of a block should always begin its own line. If the closing brace completes the statement, as in a simple block, it should appear alone on that line.

```
// Some statements...
{
    // Statements in a block
    // all line up.
    {
        // Interior block statements
        // are indented further.
    }
    // Close brace exits the block
    // and restores earlier indent.
}
// ...and so on.
```

Flow of Control

So far, we have seen declarations, definitions, and a few executable statements made out of side-effecting expressions such as method invocation and assignment. You can write some interesting programs using only these constructs, but typical programs involve more complex structures. One of the most important features is the ability to control which code is executed when. This is called flow of control. These statements have execution rules that do not always cause the next statement to be executed in turn. Instead, a statement may be executed more than once or not at all.

Simple Conditionals

One of the simplest forms of control flow is conditional execution. Conditional execution refers to a situation in which a block of code may or may not be executed, depending on the value of an expression. It is analogous to a set of instructions that says

> Step 1. If your gizmo is not already assembled, you must assemble it before going on to step 2. To assemble your gizmo, first....
>
> Step 2. Now that your gizmo is fully assembled, ...

In Java, conditional execution is most often and most generally embodied in the `if` statement. For example:

```
if ( theLight.isOn() )
{
    theRoom.isLit = true;
}
```

Let's dissect this statement. It begins with the java keyword if. After the if is a boolean expression that *must* be enclosed in parentheses. The closing parentheses are followed by a block statement.[Footnote: There are other kinds of statements that can appear in place of this block, but in this book we will restrict ourselves to the cases in which the if body is a block.] This block is sometimes called the if statement's **body** or the **consequent**; the boolean expression is called the if statement's **test** or **condition**.

Execution of the if statement proceeds as follows. First, the boolean condition expression is evaluated. If the value of this expression is true, the if's body block is executed. If the value of the boolean condition expression is false, the if's body block is skipped.

In either case, execution proceeds at the next statement following the if's body.

The if statement, as defined, is very useful when you want to do something or skip it. But often you want to do one of two things. We can express this using two if statements with inverse conditions:

```
if ( theLight.isOn() )
{
    theRoom.isLit = true;
}

if ( ! (theLight.isOn() ) )
```

```
{
    theRoom.isLit = false;
}
```

This is poor code in three ways. The first is that it invokes the same method -- `theLight.isOn()` -- twice, but the code would not work as we want if the value returned were different in the two invocations. (Imagine that the light were off the first time you asked and on the second time. The value of `theRoom.isLit` would never get set!)

We could fix this problem by temporarily assigning this value to a boolean name, and then testing the name twice:

```
boolean itIsLight = theLight.isOn();

if ( itIsLight )
{
    theRoom.isLit = true;
}

if ( ! itIsLight )
{
    theRoom.isLit = false;
}
```

But this makes a second problem with the code even more apparent. This code is testing a boolean expression (`theLight.isOn()` or `itIsLight`, depending on which version) in order to set another boolean expression. It would be cleaner just to write

```
theRoom.isLit = theLight.isOn();
```

This statement is equivalent to the whole previous example (using `itIsLight`), and much easier to read. For more on this stylistic point, see the sidebar on Using Booleans.

Of course, we can write other code that's not subject to these two problems. For example, we could use this idea to write code to compute absolute value of a given `int`, `x`.

```
int absValue;

if ( x > 0 )
{
    absValue = x;
}

if ( x < 0 )
{
    absValue = - x;
}

if ( x == 0 )
{
    absValue = 0;
}
```

This code has neither of the previous problems -- x doesn't change, so we can test it repeatedly, and the value assigned is an int, not a boolean, so we can't write the shorter assignment statement. But this code doesn't make it clear that these are really three cases of the same test. There is a form of an if statement that allows us to make this clearer. It uses the Java keyword else to denote a situation in which we know that these conditions are mutually exclusive, i.e., at most one of them can hold.

So, for example, we could rewrite our light-tester (verbosely) as:

```
boolean itIsLight = theLight.isOn();

if ( itIsLight )
{
    theRoom.isLit = true;
}
else
{
    theRoom.isLit = false;
}
```

This still isn't as nice as the one-line version, but it gives us the opportunity to illustrate control flow in an if/else statement. To execute an if/else statement:

1. Evaluate the boolean condition expression.
2. If the value of the condition is true, execute the if body block, then skip to the end of the entire if/else statement (i.e., to step 4).
3. Else (the value of the condition statement is false, so) execute the else body block. An else body is sometimes called an **alternative**.
4. Execution continues at the following statement.

Since there might be more than two mutually exclusive conditions -- as in the absolute value code -- else is allowed to have its own condition. An else with a condition is like an if, except that you only execute that part of the statement if all previous conditions in this if/else statement have been false. An else with no condition is always executed if no previous condition in this if/else statement has been true.

```
if ( x > 0 )
{
    absValue = x;
}
else if ( x < 0 )
{
    absValue = - x;
}
else
{
    absValue = 0;
}
```

Note that this is all one statement, not three as in the previous version. Exactly one of the assignment statements will be executed, no matter what the value of x at the beginning of the if statement.

Even now, this is not the most elegant absolute value code we could write; for example, the final case is redundant and could be folded into the first case using `>=` instead of `>`. It does, however, illustrate the syntax of cascaded `if`s. We will return to examine if statements, and other conditionals, in the chapter on Dispatch.

Style Sidebar

Using Booleans

There are only two `boolean` values, `true` and `false`. There can be lots of `boolean` labels, but each label is attached to either `true` or `false`; there is nothing else. This means that testing whether a `boolean` is the same as `true` --

```
(boolVal == true)
```

-- is redundant. You can just use `boolVal`, since it's either `true` or `false`. Similarly, you don't need to use an `if` statement to test a `boolean` if you're generating a `boolean` value. For example,

```
if (boolVal) {
    return true;
} else {
    return false;
}
```

is also redundant: just `return boolVal;`. The same thing applies if you're assigning to a variable instead of `return`ing: `otherBoolVal = boolVal;` (or `otherBoolVal = ! boolVal;` if you want to reverse its sense).

Simple Loops

Another flow-of-control construct is `while`. `While` takes a condition and a block, just like the simple form of `if`. Execution of a while statement first evaluates its boolean condition expression. If the condition is true, the while body block is executed. When execution of each statement in the body is complete, the while's condition is checked again. Again, if the condition is true, the body is executed. This continues until the evaluation of the condition expression yields false; at this point, execution continues at the next statement *after* the while body.

There are several uses of a while loop. One is to continually test something until it becomes true:

```
int i = 1;

while ( i < 100 )
{
    Console.println( "I'm up to  " + i );
    i = i + 1;
}
```

This loop prints the numbers from 1 to 99. (Why doesn't it print 100?)

Another use is for a loop that keeps going essentially forever. (It will stop when something stops the program, but not before:

```
while ( true )
{
    myOutput.writeOutput( myInput.readInput() );
}
```

This loop continually passes whatever input it gets to its output. Since the value of true doesn't change, this loop won't end until something nasty happens to it. Writing loops like this one -- that go on essentially forever -- is much easier than writing loops like the counting loop, above, because in the counting loop you have to keep track of what's true each time you go around the loop. For example, the value of i when you exit the loop above will always be one *more* than the last value printed.

Here's an even more tricky one:

```
while ( x < 25 )
{
  x = x + 3;
  x = x - 2;
}
```

If x's value is 20 when we reach the beginning of this **loop**, what will its value be when we exit? Remember that the test expression is only checked at the beginning of each pass through the loop, not in the middle.

There is another looping construct in Java, called do/while statement or just a do loop. It is much like the while loop, except that the loop body is always executed once before the condition is tested:

```
int i = 1;

do
{
    Console.println( "I'm up to  " + i );
    i = i + 1;
} while ( i < 100 )
```

As with a while loop, once the loop exits, execution proceeds at the statement following the entire do statement.

Statements and Rules

Programs are not simply sequences of instructions to be executed. Instead, the instruction-followers executing these statements are embedded in a community of other instruction-followers. A program is a community of interacting entities providing ongoing behavior and services. In this section, we look at how those interactions too rely on statements.

When one Thing needs to communicate with another, this is commonly accomplished through method invocation. Method invocation is an expression in which one object supplies another with information (in the form of arguments), and the second supplies the first with other information (in the form of the return

value). These mechanisms are the major means of inter-object communication and coordination. Of course, method invocation can also be used within an object, allowing one part of the object to communicate with another.

We have previously seen how interfaces specify methods that an object provides. Now, we turn to the question of how method behavior is actually implemented. Statements provide the key. Performing a method amounts to following the instructions associated with that method, i.e., stepping through the instructions for that rule. Statements are the steps of those instructions. By sequencing statements, you can build a rule that the computer can follow to accomplish a desired task. Some rules require information in order to accomplish their tasks. (For example, a rule that doubles a number needs the number to be doubled.) Some rules produce results. (For example, the doubling rule might produce the doubled number.) Some rules behave differently under different circumstances. (This uses a conditional statement).

In order to use a rule -- to interact with it -- you need to know whose rule it is, what information you need to supply in order for the rule to do its work, and what the rule will give you in return. This prefigures the idea of method signature. There are other things you'd like to know about a rule -- such as the relationship between the rule's input and its output -- and these form the basis of the rule's documentation.

For example, here is a rule for printing a brief form letter:

to `printFormLetter` using (`String title`,
`String firstName`,
`String lastName`)

1. `print "Dear "`
2. `if ( title` isn't `null ) print title + lastName`
   `else print firstName`
3. `println ":\nWe are tremendously pleased to inform you that "`
4. `println "you have won!".toUpperCase()`
5. `println "Not much, but what did you expect?"`
6. `println " Sincerely,\n me"`

It's just a short hop from this pseudocode rule to real Java:

```
void printFormLetter( String title,
                      String firstName,
                      String lastName )
{
    if ( title != null )
    {
        Console.print( title + lastName );
    }
    else
    {
        Console.print( firstName );
    }
    Console.print( ":\nWe are tremendously pleased "
                   + "to inform you that " );
    Console.println( "you have won!".toUpperCase() );
```

```
        Console.println( "Not much, but what did you expect?" );
        Console.println( "                Sincerely,\n"
                         + "                  me" );
    }
```

Method Invocation Execution Sequence

Method invocation is, as we have seen, an expression. To invoke the printFormLetter, we need to know whose method it is. We follow this object expression with a dot, then the name of the method, then the parentheses-enclosed parameter list:

```
theWidgetCompany.printFormLetter( "Prof.", "Pat", "Smith" )
```

To evaluate this expression, we need to invoke `theWidgetCompany`'s `printFormLetter` method (using the rule, or instructions, or method body, provided above) with the arguments "Prof.", "Pat", and "Smith".

The first step in method invocation is parameter binding. In this step, each parameter name (title, firstName, and lastName) is treated as though it were newly declared and it is given the value of the corresponding argument. (Recall that parameters are the names in the method declaration, while arguments are the values supplied in the method invocation expression.) In order for this to work, each value must be assignable to the corresponding parameter's declared type.

After parameter binding, method invocation proceeds as though the method body were a simple block. The block is, however, within the scope of the parameter bindings, so that inside the block the parameter names can be used to refer to the provided argument values. For example, in the body of the printFormLetter, title is bound to "Prof", firstName is bound to "Pat", and lastName is bound to "Smith".

Now the body statements are executed in turn. In this case, the first statement is an if, so its test expression is evaluated to determine whether to execute the consequent block or the alternative block. When the test expression

```
title != null
```

is evaluated, title is bound to "Prof", so it is not null, causing the consequent to execute.

This argument-value-providing is one way in which method invocation implements inter-entity communication: the value is communicated from the method-invoker to the method owner.

Return

This special statement can only be used inside method bodies. It is used to terminate the execution of the method body. It is also what is responsible for making a method body -- which is essentially a block statement -- return a value -- which is a necessary property of a method invocation expression (unless the method's return type is void).

The need for this statement arises when the sequence of instructions that you are writing is turned into a method body. In this case, you need to say what the method *returns*. This return value becomes the value

produced by evaluating a method invocation expression. This is accomplished using a **return** statement. The syntax of a return statement is

```
return expression;
```

where *expression* can be any arbitrary Java expression. Remember: the return statement -- a statement -- does not have a value, but the method invocation - an expression -- does.

To execute a return statement, evaluate the expression. Then, exit the enclosing method, providing the value of the expression as the return value of the method invocation expression. Exiting the enclosing method means both exiting from the block that is the method body and also exiting the scope of the parameter/argument bindings.

After a return statement, execution proceeds at the method invocation whose method body contained the return statement; evaluation of this expression is complete (with its value the value supplied by the return statement) and execution of the statement containing the method invocation continues.

For example, if we execute

```
String transformed = this.transform( "Knock, knock" );
```

and the transform method of this object ends with the line

```
return "Who's there?";
```

then the value of the invocation `this.transform( "Knock, knock" )` is `"Who's there?"`. Execution continues by assigning the value of the invocation (`"Who's there?"`) to the name `transformed`.

Another example is the doDouble(int) method mentioned above. The code for doDouble might read:

```
int doDouble( int whatToDouble )
{
    return whatToDouble * 2;
}
```

To evaluate the application of doDouble to 7,

1. The parameter name whatToDouble is bound to 7.
2. Within the scope of this binding, the body block of doDouble is executed.
    a. Each statement in the block is executed in turn. Since there is only one statement, it is executed.
        i. The expression whose value is to be returned is evaluated. This requires evaluating the subexpressions (name whatToDouble and literal 2) and then applying the operator to these values.
        2. The value produced by the operator expression (14) is returned by the method

3. This exits both the method body block and the parameter scope, providing the value (14) as the value of the method invocation expression.

There is also an alternate form of return that does not take an expression. This form is used in methods whose return type is void. In this case, a return statement executes by exiting the method (and, with it, the scope of the parameter names). Since the simple return statement is used only in methods whose return type is void, there is no value for it to supply.

This return statement can also be left implicit certain methods. For example, in the printFormLetter method that we saw above, there was no explicit return statement. In Java, a method without a return statement is presumed to have a return statement as its final statement. This return statement is a simple `return;` -- it is the form that does not return a value. So the end of that method body was equivalent to saying

```
    //...
    Console.println( "                Sincerely,\n"
                   + "                  me" );
    return;
}
```

In a method whose return type is not void, an explicit return statement must always be executed in order to provide the method's return value. Value-returning is another example of inter-object communication.

Chapter Summary

- Statements combine expressions to produce useful behavior.
- A statement does not have a value or a type.
- A statement is executed to produce an effect.
- A side-effecting expression followed by a semicolon is a simple statement.
- Declarations and definitions are also simple statements.
- A sequence of statements can be grouped into a block by surrounding the sequence with braces { }
- Conditional statements allow you to write code containing alternative execution sequences. The execution sequence of a conditional statement depends on the result of evaluating a boolean expression.
- A loop allows the same block of code to be executed repeatedly, until an exit condition -- a boolean expression -- is true.
- A return statement is used to exit from a method, with or without a value.
- Method bodies, or rules, use sequenced statements -- including loops and conditionals -- to produce chunks of executable behavior. A method is specified by its name, the information it needs, and the value (if any) that it produces.

Exercises

1. Using Java's if statement, write instructions for determining which team returns an out-of-bounds ball to play in a soccer game. In soccer, the team that did not last touch the ball receives possession of the ball and returns it to play.

a. You may presume that you have a method, `lastTouch()`, that returns either `homeTeam` or `visitTeam`, and that the goal of your code is to assign the correct team value (either `homeTeam` or `visitTeam`) to the already-defined name `possessingTeam`.

b. In addition, make your code determine whether `returnBallToPlayMethod` is `sideThrow`, `cornerKick`, or `goalKick`. You may make use of the `ballOutLine()` method to determine whether the ball exited via the `sideLine`, the `homeEndLine`, or the `visitEndLine`. [Footnote: If the ball has exited via the side line, the return is by side throw. If the ball exits via the home end line and is last touched by the home team, the visitors return the ball to play by means of a corner kick. A ball that is pushed beyond the home end line by the visiting team is returned by the home team via a goal kick. The situation at the visitor's end line is the opposite.]

2. Using Java's while statement, give instructions for building a tall tower of blocks.

3. Using Java's while statement, give instructions for blowing up a balloon.

4. Which of the following are expressions, which statements, and which illegal? For the expressions, indicate the type and value. For the statements, indicate the effect (if known) and the execution sequence. You may assume that x is an int, b a boolean.

a. `int x = 5`
2. `boolean b;`
3. `x + 3`
4. `x = x + 3`
5. `x = x + 3;`
6. `x == 3`
7. `x == 3;`
8. `b = x == 3;`
9.
```
{
Console.print( "What is your name? " );
String name = Console.readln();
String cap = name.toUpperCase();
}
```

5. What will the value of d be after each of the following statements? Also, indicate any other changes that may occur as a result of executing the statement. You may assume that they are executed in the order given.

a. `double d = 3.5;`
2. `d = d * 3;`
3. `if ( d < 8 )`

```
   {
   Console.println( "d is pretty small" );
   }
4. d = 2.0
5. while ( d < 30 )
   {
   d = d * 2;
   }
```


This chapter is excerpted from a draft of *Introduction to Interactive Programming In Java*, a forthcoming textbook. It is a part of the course materials developed as a part of Lynn Andrea Stein's Rethinking CS101 Project at the Computers and Cognition Laboratory of the Franklin W. Olin College of Engineering and formerly at the MIT AI Lab and the Department of Electrical Engineering and Computer Science at the Massachusetts Institute of Technology.

Questions or comments:
<webmaster@cs101.org>