

These notes are adapted from **Introduction to Programming Using Java Version 5.0, David J. Eck**. Modified versions can be made and distributed provided they are distributed under the same license as the original. More specifically: This work is licensed under the Creative Commons Attribution-Share Alike 2.5 License. To view a copy of this license, visit <http://creativecommons.org/licenses/by-sa/2.5/>. The web site for this book is: <http://math.hws.edu/javannotes>

Basic 1: Control Structures

The basic building blocks of programs—variables, expressions, assignment statements, and subroutine call statements—were covered in the previous chapter. Starting with this chapter, we look at how these building blocks can be put together to build complex programs with more interesting behavior.

Since we are still working on the level of “programming in the small” in this chapter, we are interested in the kind of complexity that can occur within a single subroutine. On this level, complexity is provided by **control structures**. The two types of control structures, loops and branches, can be used to repeat a sequence of statements over and over or to choose among two or more possible courses of action. Java includes several control structures of each type, and we will look at each of them in some detail.

Blocks, Loops, and Branches

The ability of a computer to perform complex tasks is built on just a few ways of combining simple commands into control structures. In Java, there are just six such structures that are used to determine the normal flow of control in a program—and, in fact, just three of them would be enough to write programs to perform any task. The six control structures are: the **block**, the **while loop**, the **do..while loop**, the **for loop**, the **if statement**, and the **switch statement**. Each of these structures is considered to be a single “statement,” but each is in fact a structured statement that can contain one or more other statements inside itself.

Blocks

The **block** is the simplest type of structured statement. Its purpose is simply to group a sequence of statements into a single statement. The format of a block is:

```
{  
    <statements>  
}
```

That is, it consists of a sequence of statements enclosed between a pair of braces, “{” and “}”. (In fact, it is possible for a block to contain no statements at all; such a block is called an **empty block**, and can actually be useful at times. An empty block consists of nothing but an empty pair of braces.) Block statements usually occur inside other statements, where their purpose is to group together several statements into a unit. However, a block can be legally used wherever a statement can occur. There is one place where a block is required: As you might have already noticed in the case of the **main** subroutine of a program, the definition of a subroutine is a block, since it is a sequence of statements enclosed inside a pair of braces. I should probably note again at this point that Java is what is called a free-format language. There are no syntax rules about how the language has to be arranged on a page. So, for example, you could write an entire block on one line if you want. But as a matter of good programming style, you should lay out your program on the page in a way that will make its structure as clear as possible. In general, this means putting one

statement per line and using indentation to indicate statements that are contained inside control structures. This is the format that I will generally use in my examples. Here are two examples of blocks:

```
{
println("The answer is ");
println(ans);
}

{ // This block exchanges the values of x and y
int temp; // A temporary variable for use in this block.
temp = x; // Save a copy of the value of x in temp.
x = y; // Copy the value of y into x.
y = temp; // Copy the value of temp into y.
}
```

In the second example, a variable, `temp`, is declared inside the block. This is perfectly legal, and it is good style to declare a variable inside a block if that variable is used nowhere else but inside the block. A variable declared inside a block is completely inaccessible and invisible from outside that block. When the computer executes the variable declaration statement, it allocates memory to hold the value of the variable. When the block ends, that memory is discarded (that is, made available for reuse). The variable is said to be **local** to the block. There is a general concept called the “scope” of an identifier. The **scope** of an identifier is the part of the program in which that identifier is valid. The scope of a variable defined inside a block is limited to that block, and more specifically to the part of the block that comes after the declaration of the variable.

The While Loop

The block statement by itself really doesn’t affect the flow of control in a program. The five remaining control structures do. They can be divided into two classes: **loop statements** and **branching statements**. You really just need one control structure from each category in order to have a completely general-purpose programming language. More than that is just convenience.

A while loop is used to repeat a given statement over and over. Of course, it’s not likely that you would want to keep repeating it forever. That would be an **infinite loop**, which is generally a bad thing. (There is an old story about computer pioneer Grace Murray Hopper, who read instructions on a bottle of shampoo telling her to “lather, rinse, repeat.” As the story goes, she claims that she tried to follow the directions, but she ran out of shampoo. (In case you don’t get it, this is a joke about the way that computers mindlessly follow instructions.))

To be more specific, a **while** loop will repeat a statement over and over, but only so long as a specified condition remains true. A **while** loop has the form:

```
while (<boolean-expression>
<statement>
```

Since the statement can be, and usually is, a block, many while loops have the form:

```
while (<boolean-expression>) {
<statements>
}
```

The semantics of this statement go like this: When the computer comes to a **while** statement, it evaluates the `<boolean-expression>`, which yields either `true` or `false` as the value. If the value is `false`, the computer skips over the rest of the **while** loop and proceeds to the next command in the program. If the value of the expression is `true`, the computer executes the `<statement>` or block of `<statements>` inside the loop. Then it returns to the beginning of the

while loop and repeats the process. That is, it re-evaluates the `<boolean-expression>`, ends the loop if the value is **false**, and continues it if the value is **true**. This will continue over and over until the value of the expression is **false**; if that never happens, then there will be an infinite loop.

Here is an example of a **while** loop that simply prints out the numbers 1, 2, 3, 4, 5:

```
int number; // The number to be printed.
number = 1; // Start with 1.
while ( number < 6 ) { // Keep going as long as number is < 6.
    println(number);
    number = number + 1; // Go on to the next number.
}
println("Done!");
```

The variable `number` is initialized with the value 1. So the first time through the **while** loop, when the computer evaluates the expression `"number < 6"`, it is asking whether 1 is less than 6, which is `true`. The computer therefore proceeds to execute the two statements inside the loop. The first statement prints out "1". The second statement adds 1 to `number` and stores the result back into the variable `number`; the value of `number` has been changed to 2. The computer has reached the end of the loop, so it returns to the beginning and asks again whether `number` is less than 6. Once again this is `true`, so the computer executes the loop again, this time printing out 2 as the value of `number` and then changing the value of `number` to 3. It continues in this way until eventually `number` becomes equal to 6. At that point, the expression `"number < 6"` evaluates to `false`. So, the computer jumps past the end of the loop to the next statement and prints out the message "Done!". Note that when the loop ends, the value of `number` is 6, but the last value that was printed was 5.

The If Statement

An **if statement** tells the computer to take one of two alternative courses of action, depending on whether the value of a given boolean-valued expression is true or false. It is an example of a "branching" or "decision" statement. An **if** statement has the form:

```
if ( <boolean-expression> )
    <statement >
else
    <statement >
```

When the computer executes an **if** statement, it evaluates the boolean expression. If the value is **true**, the computer executes the first statement and skips the statement that follows the **"else"**. If the value of the expression is **false**, then the computer skips the first statement and executes the second one. Note that in any case, one and only one of the two statements inside the **if** statement is executed. The two statements represent alternative courses of action; the computer decides between these courses of action based on the value of the boolean expression. In many cases, you want the computer to choose between doing something and not doing it. You can do this with an **if** statement that omits the **else** part:

```
if ( <boolean-expression> )
    <statement >
```

To execute this statement, the computer evaluates the expression. If the value is `true`, the computer executes the `<statement>` that is contained inside the **if** statement; if the value is `false`, the computer skips that `<statement>`. Of course, either or both of the `<statement>`'s in an **if** statement can be a block, so that an **if** statement often looks like:

```
if ( <boolean-expression> ) {  
  
    <statements>  
}  
else {  
    <statements>  
}
```

or:

```
if ( <boolean-expression> ) {  
    <statements>  
}
```

As an example, here is an **if** statement that exchanges the value of two variables, `x` and `y`, but only if `x` is greater than `y` to begin with. After this **if** statement has been executed, we can be sure that the value of `x` is definitely less than or equal to the value of `y`:

```
if ( x > y ) {  
    int temp; // A temporary variable for use in this block.  
    temp = x; // Save a copy of the value of x in temp.  
    x = y; // Copy the value of y into x.  
    y = temp; // Copy the value of temp into y.  
}
```

Finally, here is an example of an **if** statement that includes an **else** part. See if you can figure out what it does, and why it would be used:

```
if ( years > 1 ) { // handle case for 2 or more years  
    print("The value of the investment after ");  
    print(years);  
    print(" years is $");  
}  
else { // handle case for 1 year  
    print("The value of the investment after 1 year is $");  
} // end of if statement  
printf("%.2f", principal); // this is done in any case
```

For Loops

The **for** statement makes a common type of while loop easier to write. Many while loops have the general form:

```
<initialization>  
while ( <continuation-condition> ) {  
    <statements>  
    <update>  
}
```

For example,

```
years = 0; // initialize the variable years  
while ( years < 5 ) { // condition for continuing loop  
    interest = principal * rate; //
```

```

principal += interest; // do three statements
println(principal); //
years++; // update the value of the variable, years
}

```

This loop can be written as the following equivalent for statement:

```

for ( years = 0; years < 5; years++ ) {
interest = principal * rate;
principal += interest;
println(principal);
}

```

The initialization, continuation condition, and updating have all been combined in the first line of the **for** loop. This keeps everything involved in the “control” of the loop in one place, which helps makes the loop easier to read and understand. The **for** loop is executed in exactly the same way as the original code: The initialization part is executed once, before the loop begins. The continuation condition is executed before each execution of the loop, and the loop ends when this condition is **false**. The update part is executed at the end of each execution of the loop, just before jumping back to check the condition. The formal syntax of the **for** statement is as follows:

```

for ( <initialization>; <continuation-condition>; <update> )
<statement>

```

or, using a block statement:

```

for ( <initialization>; <continuation-condition>; <update> ) {
<statements>
}

```

The `<continuation-condition>` must be a boolean-valued expression. The `<initialization>` can be any expression, but is usually an assignment statement. The `<update>` can also be any expression, but is usually an increment, a decrement, or an assignment statement. Any of the three can be empty. If the continuation condition is empty, it is treated as if it were “true,” so the loop will be repeated forever or until it ends for some other reason, such as a **break** statement. (Some people like to begin an infinite loop with “for (;;)” instead of “while (true)”.)

Usually, the initialization part of a **for** statement assigns a value to some variable, and the update changes the value of that variable with an assignment statement or with an increment or decrement operation. The value of the variable is tested in the continuation condition, and the loop ends when this condition evaluates to **false**. A variable used in this way is called a **loop control variable**. In the **for** statement given above, the loop control variable is **years**. Certainly, the most common type of **for** loop is the **counting loop**, where a loop control variable takes on all integer values between some minimum and some maximum value. A counting loop has the form

```

for ( <variable> = <min>; <variable> <= <max>; <variable> ++ ) {
<statements>
}

```

where `<min>` and `<max>` are integer-valued expressions (usually constants). The `<variable>` takes on the values `<min>`, `<min>+1`, `<min>+2`, . . . , `<max>`. The value of the loop control variable is often used in the body of the loop. The for loop at the beginning of this section is a counting loop in which the loop control variable, `years`, takes on the values 1, 2, 3, 4, 5. Here is an even simpler example, in which the numbers 1, 2, . . . , 10 are displayed on standard output:

```

for ( N = 1 ; N <= 10 ; N++ )
println( N );

```

For various reasons, Java programmers like to start counting at 0 instead of 1, and they tend to use a "<" in the condition, rather than a "<=". The following variation of the above loop prints out the ten numbers 0, 1, 2, . . . , 9:

```
for ( N = 0 ; N < 10 ; N++ )
    println( N );
```

Using < instead of <= in the test, or vice versa, is a common source of off-by-one errors in programs. You should always stop and think, Do I want the final value to be processed or not? It's easy to count down from 10 to 1 instead of counting up. Just start with 10, decrement the loop control variable instead of incrementing it, and continue as long as the variable is greater than or equal to one.

```
for ( N = 10 ; N >= 1 ; N-- )
    println( N );
```

The Switch Statement

A switch statement allows you to test the value of an expression and, depending on that value, to jump directly to some location within the switch statement. Only expressions of certain types can be used. The value of the expression can be one of the primitive integer types or, can be the primitive char type, Or, it can be a boolean as well. In particular, the expression **cannot** be a String or a real number. The positions that you can jump to are marked with **case labels** that take the form: "case <constant>:". This marks the position the computer jumps to when the expression evaluates to the given <constant>. As the final case in a switch statement you can, optionally, use the label "default:", which provides a default jump point that is used when the value of the expression is not listed in any case label. A **switch** statement, as it is most often used, has the form:

```
switch (<expression>) {
    case <constant-1>:
        <statements-1>
        break;
    case <constant-2 >:
        <statements-2>
        break;
    .
    . // (more cases)
    .
    case <constant-N >:
        <statements-N>
        break;
    default: // optional default case
        <statements-(N+1)>
} // end of switch statement
```

The **break** statements are technically optional. The effect of a **break** is to make the computer jump to the end of the switch statement. If you leave out the break statement, the computer will just forge ahead after completing one case and will execute the statements associated with the next case label. This is rarely what you want, but it is legal. (I will note here—although you won't understand it until you get to the next chapter—that inside a subroutine, the **break** statement is sometimes replaced by a **return** statement.)

Note that you can leave out one of the groups of statements entirely (including the **break**). You then have two case labels in a row, containing two different constants. This just means that the computer will jump to the same place and perform the same action for each of the two constants.

Here is an example of a switch statement. This is not a useful example, but it should be easy for you to follow. Note, by the way, that the constants in the case labels don't have to be in any particular order, as long as they are all different:

```
switch ( N ) { // (Assume N is an integer variable.)
  case 1:
    println("The number is 1.");
    break;
  case 2:
  case 4:
  case 8:
    println("The number is 2, 4, or 8.");
    println("(That's a power of 2!)");
    break;
  case 3:
  case 6:
  case 9:
    println("The number is 3, 6, or 9.");
    println("(That's a multiple of 3!)");
    break;
  case 5:
    println("The number is 5.");
    break;
  default:
    println("The number is 7 or is outside the range 1 to 9.");
}
```

The switch statement is pretty primitive as control structures go, and it's easy to make mistakes when you use it. Java takes all its control structures directly from the older programming languages C and C++.