CPS109 Course Notes 5

Alexander Ferworn

Unrelated Facts Worth Remembering

- Make a "to do" list every day. Crossing things off the list is very satisfying and if items stay on the list, they become very annoying.

Table of Contents

1 Introduction

If nothing interferes, a Java program begins executing at the first line of code in the main() method and stops when the last line is executed. While this is useful it can be cumbersome without the ability to do things like change what will be done next based on conditions that are encountered while the program is running, or do things over and over again.

This is an easy sell for most people. Imagine being on the 10m board of a diving pool and you discover that there is no water below. If you couldn't change your course of action you would probably have what old programmers called an ABEND or "ABnormal ENDing".

Most programming languages provide the ability to change the course of execution through a series of constructs commonly called ***selection*** constructs—allowing you to select what will be executed next. In addition they also provide a set of ***iteration*** constructs—allowing you to choose how many times something will be executed. This document discusses Java features supporting both concepts.

2 if

The if statement is the first and simplest ways of changing what will be done next based on testing a condition.

Definition:

- An if statement allows a program to choose whether or not to execute an associated statement.

Syntax:

- The syntax for an if statement is;

```
if(<condition>)
      <statement1>;
[else
      <statement2>;]
```

Rules:

- <condition> must be a boolean expression that evaluates to true or false.
- If the condition is true then <statement1> is executed.
- If the condition is false then <statement1> is skipped and <statement2> is executed if else is present. If there is no else, execution continuing after <statement1>.
- Only a single statement can follow the <condition> and the **else** (we will fix this shortly).

Here is an example program illustrating how if works.

```
/* Program demoing if. A decision is made about an input
temperature. If the temperature is below 20 a message is
output. */

import java.util.Scanner;

class Temperature
{
        final static int COLD = 20;
        public static void main(String[] args)
        {
                Scanner In = new Scanner(System.in);
                System.out.println
                  ("Enter the temperature outside.");
                int temperature = In.nextInt();
                if(temperature <= COLD)
                        System.out.println
                              ("It's cold out there!");
                        System.out.println("Bye!");
        }
}
```

As output the program could produce several results;

```
Enter the temperature outside.
21
```

```
Bye!
```

or

```
Enter the temperature outside.
19
It's cold out there!
Bye!
```

Note that only a single statement can be associated with the condition.

2.1 Boolean Expressions

Definition:

- Anything that evaluates to true or false is a **boolean expression**. For example;

```
true
total == val
4 > 3
```

will all evaluate to one or the other.

It is also possible to assign the results of a boolean expression to a boolean variable as in,

```
…
boolean ans = (total == val);
…
```

Java provides a set of relational operators each with its own precedence. The operators are shown in the table below.

Operator	Meaning
==	equal to (note: not assignment)
!=	not equal to
<	less than
<=	less than or equal to
>	greater than
>=	greater than or equal to

2.2 Blocks

Recall that an if statement allows only one statement after the condition. You can avoid this problem by making a block of statements.

Definition:

- A **block** is a list of statements enclosed in curly brackets

We have already seen blocks in the main method and a class definition. We revisit the above example below.

```
// Program demoing if. A decision is made
// about an input temperature. If
// the temperature is below 20 a message is output.

import java.util.Scanner;

class Temperature1
{
        final static int COLD = 20;
        public static void main(String[] args)
        {
                Scanner In = new Scanner(System.in);
                System.out.println
                        ("Enter the temperature outside.");
                int temperature = In.nextInt();
                if(temperature <= COLD)
                {
                        System.out.println
                              ("It's cold out there!");
                        System.out.println
                              ("But this is Canada!");
                }
                System.out.println("Bye!");
        }
}
```

2.3 if-else

The following program illustrates the use of the if-else combination as well as blocks.

```
/* Program demoing if-else. A decision is made  about input.
*/
import java.util.Scanner;

class Decision
{
       public static void main(String[] args)
       {
              Scanner In = new Scanner(System.in);
              System.out.println
                     ("Enter 1 or 2.");
              int answer = In.nextInt();
              if(answer == 1)
              {
                     System.out.println
                            ("You entered 1!");
                     System.out.println
                            ("Ha, I knew you would");
              }
              else
```

```
                {
                        System.out.println
                                ("You entered 2!");
                        System.out.println
                                ("I thought you would enter 1 !");
                }
                System.out.println("Bye!");
        }
}
```

Notes:

- The "Bye!" always gets printed
- There is a problem…what if they don't enter 1 or 2?

2.4 Nesting

Now, an if statement is, in particular, a statement. This means that either statement-1 or statement-2 inside an if statement can itself be an if statement. (Note: If statement-1 is an if statement, then it has to have an else part; if it does not, the computer will mistake the "else" of the first if statement for the missing "else" of statement-1. This is called the *dangling else problem*. You can avoid this problem by enclosing statement-1 between { and }, making it into a block.)

An if statement in which the else part is itself an if statement would look like this (perhaps without the final else part):

```
if (<boolean-expression-1>)
     <statement-1>
else
     if (<boolean-expression-2>)
          <statement-2>
     else
          <statement-3>
```

You should think of this as a single statement representing a three-way branch. When the computer executes this, one and only one, of the three statements, statement-1, statement-2, and statement-3, will be executed. The computer starts by evaluating boolean-expression-1. If it is true, the computer executes statement-1 and then jumps all the way to the end of the big if statement, skipping the other two statement's. If boolean-expression-1 is false, the computer skips statement-1 and executes the second, nested if statement. That is, it tests the value of boolean-expression-2 and uses it to decide between statement-2 and statement-3.

Here is an example that will print out one of three different messages, depending on the value of a variable named temperature:

```
if (temperature < 50)
```

```
            System.out.println("It's cold.");
      else
            if (temperature < 80)
                  System.out.println("It's nice.");
            else
                  System.out.println("It's hot.");
```

If temperature is, the computer prints out the message "It's cold", and skips the rest -- without even evaluating the second condition.

You can go on stringing together "else-if's" to make multiway branches with any number of cases:

```
if (boolean-expression-1)
      statement-1
else if (boolean-expression-2)
      statement-2
else if (boolean-expression-3)
      statement-3

      // (more cases)

else if (boolean-expression-N)
      statement-N
else
      statement-(N+1)
```

You should just remember that only one of the statements will be executed and that the computer will stop evaluating boolean-expressions as soon as it finds one that is true. Also, remember that the final else part can be omitted and that any of the statements can be blocks, consisting of a number of statements enclosed between { and }. (Admittedly, there is lot of syntax here; as you study and practice, you'll become comfortable with it.)

3 Switch (sometimes known as “case”)

Java also provides a control structure that is specifically designed to make multiway branches of a certain type: the switch statement. A switch statement allows you to test the value of an expression and, depending on that value, to jump to some location within the switch statement. The positions 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 has the form:

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

3.1 A bit about break

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. 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:
            System.out.println("The number is 1.");
            break;
      case 2:
      case 4:
      case 8:
            System.out.println("The number is 2, 4, or 8.");
            System.out.println("(That's a power of 2.)");
            break;
      case 3:
      case 6:
```

```
        case 9:
              System.out.println("The number is 3, 6, or 9.");
              System.out.println("(That's a multiple of 3.)");
              break;
        case 5:
              System.out.println("The number is 5.");
              break;
        default:
              System.out.println("The number is 7,");
              System.out.println(" or not in the range 1-9.");
        }
```

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++. The switch statement is certainly one place where the designers of Java should have introduced some improvements.

4 Conditional Operator

The conditional operator (? :) is a ternary operator. The operator selects one of two expressions for evaluation, based on the value of its first operand. In this way, the conditional operator is similar to an if statement.

The conditional operator produces a pure value. Conditional expressions group from right to left. Consider the following expression:

```
g?f:e?d:c?b:a
```

It is equivalent to

```
g?f:(e?d:(c?b:a))
```

The first operand of the conditional operator must be of type boolean, or a compile-time error occurs. If the first operand evaluates to true, the operator evaluates the second operand (i.e., the one following the ?) and produces the pure value of that expression. Otherwise, if the first operand evaluates to false, the operator evaluates the third operand (i.e., the one following the :) and produces the pure value of that expression. Note that the conditional operator evaluates either its second operand or its third operand, but not both.

The second and third operands of the conditional operator may be of any type, but they must both be of the same kind of type or a compile-time error occurs. If one operand is of an arithmetic type, the other must also be of an arithmetic type. If one operand is of type boolean, the other must also be of type boolean. If one operand is a reference type, the other must also be a

reference type. Note that neither the second nor the third operand can be an expression that invokes a void method.

The types of the second and third operands determine the type of pure value that the conditional operator produces. If the second and third operands are of different types, the operator may perform a type conversion on the operand that it evaluates. The operator does this to ensure that it always produces the same type of result for a given expression, regardless of the value of its first operand.

If the second and third operands are both of arithmetic types, the conditional operator determines the type of value it produces as follows:

- If both operands are of the same type, the conditional operator produces a pure value of that type.
- If one operand is of type short and the other operand is of type byte, the conditional operator produces a short value.
- If one operand is of type short, char, or byte and the other operand is a constant expression that can be represented as a value of that type, the conditional operator produces a pure value of that type.
- Otherwise, if either operand is of type double, the operator produces a double value.
- Otherwise, if either operand is of type float, the operator produces a float value.
- Otherwise, if either operand is of type long, the operator produces a long value.
- Otherwise, if either operand is of type int, the operator produces an int value.
- If the second and third operands are both of type boolean, the conditional operator produces a pure boolean value.
- If the second and third operands are both reference types, the conditional operator determines the type of value it produces as follows:
- If both operands are null, the conditional operator produces the pure value null.
- Otherwise, if exactly one of the operands is null, the conditional operator produces a value of the type of the other operand.
- Otherwise, it must be possible to cast the value of one of the operands to the type of the other operand, or
- a compile-time error occurs. The conditional operator produces a value of the type that would be the target of the cast.