

Conditional Control Statements

(CISS-110 William Jojo)

Relational Operators

Up to this point our statements have been executing sequentially, that is, they have been processed in order without any of the sequence changing. The program will simply execute statements in the order they are written. There are times, however, that we would like to execute statements *conditionally* rather than unconditionally.

Statements can be executed in three ways:

- Sequential (unconditional)
- Conditional (selective or branch)
- Iterative (looping)

Programs may contain all three forms of statement execution but, as we have seen, are not required to do anything more than sequential. Iterative statements will be covered in a successive document. This document will concentrate on conditional execution of our programs.

When deciding what should be done when a condition is met, we must first determine the condition that needs to be detected. In other words, what condition must be met and how will we know when that condition or event has occurred? Some examples are:

```
if 18 is evenly divisible by 2
    then print "the number is even"

if -8 is less than 0
    then print "the number is negative"

if the Sun is shining
    note that it is daytime
otherwise
    note that it is NOT daytime
```

Of course, none of these are actual Java statements. These statements are known as *pseudocode*. Pseudocode is used to describe the details of a piece of logic that is yet to be written. The beauty of pseudocode is that we can express anything we like without having to actually write in a specific language. Therefore, pseudocode assists in conceptualizing ideas while remaining language neutral.

Determining if a condition has been met often requires us to make a comparison of a particular value with some known quantity. To make these comparisons, we will need a set of

tools that perform the comparison work and give a yes or no result, or more to the point, a `true` or `false` result. The set of Java *relational operators* gives us a `true` or `false` result based on comparisons. There are six relational operators and they are shown in *Table 1*.

Relational Operator	Description
<code>==</code>	equal to
<code>!=</code>	not equal to
<code><</code>	less than
<code><=</code>	less than or equal to
<code>></code>	greater than
<code>>=</code>	greater than or equal to

Table 1: Relational operators for comparison.

As we will see, the relational operators are binary operators just like our arithmetic operators. The operators can involve integral and floating-point primitive data types. Only *equal to* (`==`) and *not equal to* (`!=`) may be applied when comparing variables of the `boolean` primitive data type.

One should be cautious when testing floating-point values for equality since there is a possibility for loss of precision when these numbers are stored in memory and is more likely to occur when these values are involved in complex arithmetic expressions.

One-way and two-way selection

Let us take the first two examples from above and rewrite them in actual Java code. The first two examples demonstrate the concept of one-way selection, that is, there is only one possible outcome based on the test being performed.

The rewrite is shown below.

```
if ( 18 % 2 == 0 )
    System.out.println("The number is even.");

if ( -8 < 0 )
    System.out.println("The number is negative.");
```

These examples show a conditional test that must be satisfied before performing the indented portion of code. The line below the `if` test is indented to show that the code will only be executed if the test evaluates to `true`. The condition test itself is always surrounded by

parentheses.

```
public class dayornight
{
    public static void main(String[] args)
    {
        boolean daytime, sunShining;

        sunShining = true;

        if ( sunShining == true )
            daytime = true;
        else
            daytime = false;

        System.out.println("sunShining = " + sunShining);
        System.out.println("daytime = " + daytime);
    }
}
```

Example 1: Program demonstrating the else clause of the if statement.

It is now established that statements will be done if conditions are true. The third example, however, shows an opportunity where work may need to be done when the condition is `true` or alternate work done when the condition is `false` – a **two-way selection**. *Example 1* shows a complete program for the third example. The original pseudocode that has been converted is shown in bold.

The program output is as expected:

```
sunShining = true
daytime = true
```

It was easy to predict the output without really knowing what the `else` clause would do for our program. Since `sunShining` was `true`, we would expect the value of `daytime` to be set to `true`. The `else` clause of the `if` statement allows the programmer to offer an alternative if the condition test fails to evaluate to `true`. Therefore, if the condition test failed, the statement indented beneath the `else` clause would be executed setting `daytime` to `false`.

The `else` portion of an `if` is matched with the closest unfinished `if`, that is, one that has no `else` portion.

Compound Statements

Let's change our `dayornight` class and add another boolean variable, `nighttime`. The product of which is *Example 2*. In this example we can see that two assignment statements will be executed for both the true and false portion of our `if/else` construct.

The `if` portion and the `else` portion, by default, can only have one statement associated to each of them. When we need either of them to perform more than one statement worth of work, we use a *compound statement*. A compound statement is formed the same way our main method body was formed – with curly braces. Note the closing of the curly braces before the `else` portion begins.

As we will see in upcoming sections and future chapters, the compound statement is used by many constructs.

```
public class dayornight2
{
    public static void main(String[] args)
    {
        boolean daytime, nighttime, sunShining;

        sunShining = true;

        if ( sunShining == true )
        {
            daytime = true;
            nighttime = false;
        }
        else
        {
            daytime = false;
            nighttime = true;
        }

        System.out.println("sunShining = " + sunShining);
        System.out.println("daytime = " + daytime);
        System.out.println("nighttime = " + nighttime);
    }
}
```

Example 2: Two-way selection with a compound statement.

Multi-way selection

Multi-way selection is often needed when the data tested falls into more than two response categories. Consider the code in *Example 3* and *Example 4* where we want to test if a number is *positive*, *negative* or *zero*. Clearly a single `else` is not enough, but we also cannot have more than one `else` per `if` statement. The concept of a nested `if` can be applied to allow a test with more than two options.

```
import java.util.*;

public class numtype
{
    static Scanner kb = new Scanner(System.in);

    public static void main(String[] args)
    {
        int x;

        System.out.print("Enter an integer: ");
        x = kb.nextInt();

        System.out.print("The value " + x + " is ");
        if ( x < 0 )
            System.out.println("Negative.");
        else if ( x > 0 )
            System.out.println("Positive.");
        else
            System.out.println("Zero.");
    }
}
```

Example 3: Multi-way if statement.

Technically speaking, a nested `if` actually looks like:

```
if ( x < 0 )
    System.out.println("Negative.");
else
    if ( x > 0 )
        System.out.println("Positive.");
    else
        System.out.println("Zero.");
```

The format of *Example 3* is preferred for ease of writing and is accepted conventional format in several programming languages.

```

import javax.swing.*;

public class numtypegui
{
    public static void main(String[] args)
    {
        int x;
        String s;

        s = JOptionPane.showInputDialog("Enter an integer:");
        x = Integer.parseInt(s);

        s = "The number " + x + " is ";
        if ( x < 0 )
            s = s + "Negative.";
        else if ( x > 0 )
            s = s + "Positive.";
        else
            s = s + "Zero.";

        JOptionPane.showMessageDialog(null, s, "Results",
        JOptionPane.PLAIN_MESSAGE);
    }
}

```

Example 4: Multi-way if statement with dialog boxes.

The Dangling else

Consider a multi-way selection where there is an else portion for the outer condition, but not for the inner one. Here is a piece of code to demonstrate:

```

if ( age >= 18 )
    if ( age >= 21 )
        System.out.println("Legal Drinking Age.");
else
    System.out.println("Ineligible To Vote.");

```

The issue is that although the indentation shows the `else` belongs to the test of `age >= 18`, Java (and other languages) associates the `else` with the closest unfinished `if` (shown in bold). As a result, the `else` actually belongs to the test for legal drinking age. Those who are ages 18 to 20 are therefore ineligible to vote! This clearly needs fixing and can be done with a simple compound statement to correctly associate the `else` as shown below:

```

if ( age >= 18 )
{
    if ( age >= 21 )
        System.out.println("Legal Drinking Age.");
    }
else
    System.out.println("Ineligible to vote.");

```

Logical Operators

The set of logical operators allows a joining of logical expressions into larger ones. The logical operators are listed in *Table 2*.

Operator	Description
&&	and
	or
!	not (unary operator)

Table 2: Logical operators.

The ! (not) unary operator simply inverts the boolean value. Therefore !true is false and !false is true.

The two binary operators && (and) and || (or) work according to standard boolean algebra where result of && is only true if both operands are true. Likewise || is only false when both operands are false. A standard truth table is shown in *Table 3*.

Boolean	Op	Boolean	Result
true	&&	true	true
true	&&	false	false
false	&&	true	false
false	&&	false	false
true		true	true
true		false	true
false		true	true
false		false	false

Table 3: Truth table showing && and ||.

The `char` type can be used with relational operators since it is an integer primitive type, so let's add some characters to our examples of selection. Here are a few:

```
if ( ch >= '0' && ch <= '9' )
    System.out.println("The character is a digit.");

if ( ch >= 'A' && ch <= 'Z' )
    System.out.println("The character is an uppercase letter.");

if ( (ch >= 'A' && ch <= 'Z') ||
     (ch >= 'a' && ch <= 'z') )
    System.out.println("The character is a letter.");

if ( (ch >= 'A' && ch <= 'Z') ||
     (ch >= 'a' && ch <= 'z') ||
     (ch >= '0' && ch <= '9') )
    System.out.println("The character is a alphanumeric.");
```

The first two selection statements show simple determination of digit (in the range '0' through '9') or uppercase letter (in the range 'A' through 'Z').

The next one joins the concept of letters being upper or lowercase. Note the use of parentheses to separate the uppercase test from the lowercase test. Also, see that **or** was used to join the two tests since a character can't be both uppercase *and* lowercase.

Our final example takes on the more generalized classification of a character being alphanumeric, that is, the character is uppercase, lowercase or numeric.

Comparing Strings

Comparing strings is actually quite simple once you understand the principles of lexicographical analysis. Simply put, lexicographical analysis is the character by character comparison of two strings using the Unicode collating sequence until we either:

- Run out of characters in one of the strings.
- Two characters in the same position of the two strings are different.
- Both strings have been tested and no mismatches were found.

In other words, if the strings are different length, they cannot be a match. If two characters from the same position, say position 12, in each string are different, they cannot be a match.

Otherwise, they are a perfect match and we say the strings are equal.

The `compareTo()` method of the `String` class returns three possible values as a result of this character by character comparison. This is shown in *Table 4* and it is assumed that `str1` and `str2` are valid `String` reference variables.

Expression	Value	Description
<code>str1.compareTo(str2)</code>	<code>< 0</code>	<code>str1</code> is less than <code>str2</code>
<code>str1.compareTo(str2)</code>	<code>> 0</code>	<code>str1</code> is greater than <code>str2</code>
<code>str1.compareTo(str2)</code>	<code>== 0</code>	<code>str1</code> and <code>str2</code> are equal

Table 4: Possible results for the `compareTo()` `String` method.

Why does the `compareTo()` method return these values? Well, one way to look at this is to realize that when you subtract one value from another if the result is zero, then the values were the same, so if the difference of the last character of each string is zero, then the two strings match since to reach the last character of the strings, they must have matched all along.

Now consider when two characters don't match. If we take the characters from each string at the same position and subtract the Unicode value of the `str2` character from the Unicode value of the `str1` character, there will be either a positive or negative result.

If the result is negative, then the Unicode value from `str1` was less than the the Unicode value from `str2`. That single character comparison is all that was needed to determine that `str1` is less than `str2`.

The `switch` Statement

The multi-way selection of `if-else-if-else-if-else` is very reliable. There exists another method for accomplishing a similar task – the `switch` statement. First, let's review the multi-way `if` selection using an integer, `month`, to reflect the month of the year. As you can see in *Example 5*, the integer is used to determine the string value for the month. This approach can be rewritten using a `select` statement as in *Example 6*.

The `switch` statement uses a parenthesized integer expression or *selector*. In this example the selector is simply `month`. The `switch` statement takes the selector value and compares it to the list of possible action cases. The selector value and the subsequent list of cases must be an integer. Each case represents a selection or path to be taken based on the value of the

selector. In addition, all cases have a particular value to be matched followed by a colon.

The optional `default` case is used to catch selector values that have not matched any other cases.

```
import java.util.*;

public class monthifelse
{
    static Scanner kb = new Scanner(System.in);

    public static void main(String[] args)
    {
        int month;
        String monthStr;

        System.out.print("Enter the month (1-12): ");
        month = kb.nextInt();

        if ( month == 1 )
            monthStr = "January";
        else if ( month == 2 )
            monthStr = "February";
        else if ( month == 3 )
            monthStr = "March";
        else if ( month == 4 )
            monthStr = "April";
        else if ( month == 5 )
            monthStr = "May";
        else if ( month == 6 )
            monthStr = "June";
        else if ( month == 7 )
            monthStr = "July";
        else if ( month == 8 )
            monthStr = "August";
        else if ( month == 9 )
            monthStr = "September";
        else if ( month == 10 )
            monthStr = "October";
        else if ( month == 11 )
            monthStr = "November";
        else if ( month == 12 )
            monthStr = "December";
        else
            monthStr = "AN UNKNOWN MONTH";

        System.out.println("The month selected is " + monthStr);
    }
}
```

Example 5: Recap of the if-else multi-way selector.

```

import java.util.*;

public class monthselect
{
    static Scanner kb = new Scanner(System.in);

    public static void main(String[] args)
    {
        int month;
        String monthStr;

        System.out.print("Enter the month (1-12): ");
        month = kb.nextInt();

        switch ( month )
        {
            case 1: monthStr = "January";
                    break;
            case 2: monthStr = "February";
                    break;
            case 3: monthStr = "March";
                    break;
            case 4: monthStr = "April";
                    break;
            case 5: monthStr = "May";
                    break;
            case 6: monthStr = "June";
                    break;
            case 7: monthStr = "July";
                    break;
            case 8: monthStr = "August";
                    break;
            case 9: monthStr = "September";
                    break;
            case 10: monthStr = "October";
                    break;
            case 11: monthStr = "November";
                    break;
            case 12: monthStr = "December";
                    break;
            default: monthStr = "AN UNKNOWN MONTH";
                    break;
        }

        System.out.println("The month selected is " + monthStr);
    }
}

```

Example 6: Multi-way selection using a select statement.

All of the statements that follow the colon immediately after a matched case are executed and are **not** enclosed in curly braces as you would expect a compound statement would normally

be. Instead of encasing the statement block in curly braces a `break` statement is used to signify both the end of the block and to leave the `switch` statement. If a `break` statement is absent, execution would continue through the next case.

The act of leaving a `break` statement out could either be accidental or intentional. The accidental omission of the `break` statement would result in code that is logically incorrect in one or more cases of the `switch` statement. However, the intentional omission of the `break` statement serves to combine cases together where appropriate.

An accidental example would be to leave the `break` statement out of case 3. If this were done, the value of `monthStr` would be initially set to "March", then the program would *fall through* to case 4 and set `monthStr` to "April" only then encountering the `break` to leave the `switch` statement. Clearly the `break` statement is making sure we only set the value of `monthStr` appropriate to the integer stored in `month`.

An intentional example would be to construct a situation where multiple values are acceptable or differ only in case. With variables `letterGrade` as a `char` and `qp` as an `int`, consider the following:

```
switch (letterGrade)
{
    case 'a':
    case 'A':
        qp = 4;
        break;

    case 'b':
    case 'B':
        qp = 3;
        break;

    case 'c':
    case 'C':
        qp = 2;
        break;

    case 'd':
    case 'D':
        qp = 1;
        break;

    default:
        qp = 0;
        break;
}
```

This code determines the quality points a a step in calculating the cumulative average of a student. If `letterGrade` had not been determined to be in a specific alphabetic case prior to arriving in the `switch` statement, it can handle either one. This code can also be written as follows:

```
switch (letterGrade)
{
    case 'a': case 'A':
        qp = 4; break;

    case 'b': case 'B':
        qp = 3; break;

    case 'c': case 'C':
        qp = 2; break;

    case 'd': case 'D':
        qp = 1; break;

    default:
        qp = 3; break;
}
```

As you can see, the format is pretty free form. The only remaining issue is when do you choose `if-else` over `switch`? For a possible answer to that question, consider the earlier code example where we were identifying a character as alphanumeric. With 52 letters and 10 digits, it would be unlikely one would choose `switch`.