10

More on Conditionals

ORACLE

Objectives

After completing this lesson, you should be able to:

- Use a `ternary` statement
- Test equality between strings
- Chain an `if/else` statement
- Use a `switch` statement
- Use the NetBeans debugger

Topics

- Relational and conditional operators
- More ways to use `if/else` statements
- Using a `switch` statement
- Using the NetBeans debugger

Review: Relational Operators

Condition	Operator	Example
Is equal to	==	`int i=1; (i == 1)`
Is not equal to	!=	`int i=2; (i != 1)`
Is less than	<	`int i=0; (i < 1)`
Is less than or equal to	<=	`int i=1; (i <= 1)`
Is greater than	>	`int i=2; (i > 1)`
Is greater than or equal to	>=	`int i=1; (i >= 1)`

By way of review, here you see a list of all the relational operators. Previously, you used the == operator to test equality for numeric values. However, `String` variables are handled differently because a `String` variable is an object reference, rather than a primitive value.

Testing Equality Between `String` variables

Example:

```
public class Employees {

    public String name1 = "Fred Smith";
    public String name2 = "Sam Smith";

    public void areNamesEqual() {
        if (name1.equals(name2)) {
            System.out.println("Same name.");
        }
        else {
            System.out.println("Different name.");
        }
    }
}
```

If you use the == operator to compare object references, the operator tests to see whether both object references are the same (that is, do the `String` objects point to the same location in memory). For a `String` it is likely that instead you want to find out whether the characters within the two String objects are the same. The best way to do this is to use the `equals` method.

Testing Equality Between `String` variables

Example:

```
public class Employees {

    public String name1 = "Fred Smith";
    public String name2 = "fred smith";

    public void areNamesEqual() {
        if (name1.equalsIgnoreCase(name2)) {
            System.out.println("Same name.");
        }
        else {
            System.out.println("Different name.");
        }
    }
}
```

There is also an `equalsIgnoreCase` method that ignores the case when it makes the comparison.

Testing Equality Between `String` variables

Example:

```
public class Employees {

    public String name1 = "Fred Smith";
    public String name2 = "Fred Smith";

    public void areNamesEqual() {
        if (name1 == name2) {
            System.out.println("Same name.");
        }
        else {
            System.out.println("Different name.");
        }
    }
}
```

- Depending on how the `String` variables are initialized, == might actually be effective in comparing the values of two `String` objects, but only because of the way Java deals with strings.
- In this example, only one object was created to contain "Fred Smith" and both references (`name1` and `name2`) point to it. Therefore, `name1 == name2` is true. This is done to save memory. However, because `String` objects are immutable, if you assign `name1` to a different value, `name2` is still pointing to the original object and the two references are no longer equal.

Testing Equality Between `String` variables

Example:

```
public class Employees {

    public String name1 = new String("Fred Smith");
    public String name2 = new String("Fred Smith");

    public void areNamesEqual() {
        if (name1 == name2) {
            System.out.println("Same name.");
        }
        else {
            System.out.println("Different name.");
        }
    }
}
```

- When you initialize a `String` using the new keyword, you force Java to create a new object in a new location in memory even if a `String` object containing the same character values already exists. Therefore in the following example, `name1 == name2` would return `false`.
- It makes sense then that the safest way to determine equality of two string values is to use the `equals` method.

Common Conditional Operators

Operation	Operator	Example

If one condition AND another condition	`&&`	`int i = 2;`
		`int j = 8;`
		`((i < 1) && (j > 6))`

If either one condition OR another condition	`||`	`int i = 2;`
		`int j = 8;`
		`((i < 1) || (j > 10))`

NOT	`!`	`int i = 2;` `(!(i < 3))`

Relational operators are often used in conjunction with conditional operators. You might need to make a single decision based on more than one condition. Under such circumstances, you can use conditional operators to evaluate complex conditions as a whole.

The table in the slide lists the common conditional operators in the Java programming language. For example, all of the examples in the table yield a `boolean` result of `false`.

Discussion: What relational and conditional operators are expressed in the following paragraph?

- If the toy is red, I will purchase it. However, if the toy is yellow and costs less than a red item, I will also purchase it. If the toy is yellow and costs the same as or more than another red item, I will not purchase it. Finally, if the toy is green, I will not purchase it.

Ternary Conditional Operator

Operation	Operator	Example

If `some condition` is true, assign the value of `value1` to the result. Otherwise, assign the value of `value2` to the result.	`?:`	`condition ? value1 : value2`
		
		`Example:`
		`int x = 2, y = 5, z = 0;`
		
		`z = (y < x) ? x : y;`

Equivalent statements

```
if(y<x){
   z=x;
 }
 else{
   z=y;
 }
```

```
z = (y < x) ? x : y;
```

The ternary operator is a conditional operator that takes three operands. It has a more compact syntax than an `if/else` statement.

Use the ternary operator instead of an `if/else` statement if you want to make your code shorter. The three operands shown in the example above are described here:

- `(y < x)`: This is the boolean expression (condition) being evaluated.
- `? x` : If (y < x) is true, z will be assigned the value of x.
- `: y` : If (y < x) is false, z will be assigned the value of y.

Using the Ternary Operator

Advantage: Usable in a single line

```
int numberOfGoals = 1;
String s = (numberOfGoals==1 ? "goal" : "goals");

System.out.println("I scored " +numberOfGoals +" "
+s );
```

Advantage: Place the operation directly within an expression

```
int numberOfGoals = 1;

System.out.println("I scored " +numberOfGoals +" "
+(numberOfGoals==1 ? "goal" : "goals") );
```

Disadvantage: Can have only two potential results

```
(numberOfGoals==1 ?      "goal" :    "goals" :    "More goals");
```

boolean — true — false — ???

Based on the number of goals scored, these examples will print the appropriate singular or plural form of “goal.”

The operation is compact because it can only yield two results, based on a boolean expression.

Exercise 10-1: Using the Ternary Operator

In this exercise, you use a ternary operator to duplicate the same logic shown in this `if/else` statement:

```
01      int x = 4, y = 9;
02      if ((y / x) < 3){
03          x += y;
04      }
05      else x *= y;
```

Perform this exercise by opening the project **Exercise_10-1** or create your own project with a **Java Main Class** named `TestClass`.

Topics

- Relational and conditional operators
- More ways to use `if/else` statements
- Using a `switch` statement
- Using the NetBeans debugger

Handling Complex Conditions with a Chained `if` Construct

The chained `if` statement:

- Connects multiple conditions together into a single construct
- Often contains nested `if` statements
- Tends to be confusing to read and hard to maintain

Determining the Number of Days in a Month

```
01  if (month == 1 || month == 3 || month == 5 || month == 7
02       || month == 8 || month == 10 || month == 12) {
03      System.out.println("31 days in the month.");
04  }
05  else if (month == 2) {
06      if(!isLeapYear){
07         System.out.println("28 days in the month.");
08     }else System.out.println("29 days in the month.");
09  }
10  else if (month ==4 || month == 6 || month == 9
11          || month == 11) {
12      System.out.println("30 days in the month.");
13  }
14  else
15      System.out.println("Invalid month.");
```

- The code example above shows how you would use a chained and nested `if` to determine the number of days in a month.
- Notice that, if the month is 2, a nested `if` is used to check whether it is a leap year.

Note: Debugging (covered later in this lesson) would reveal how every `if/else` statement is examined up until a statement is found to be true.

Chaining `if/else` Constructs

Syntax:

```
01  if <condition1> {
02     //code_block1
03  }
04  else if <condition2> {
05     // code_block2
06  }
07  else {
08     // default_code
09  }
```

You can chain `if` and `else` constructs together to state multiple outcomes for several different expressions. The syntax for a chained `if/else` construct is shown in the slide example, where:

- Each of the conditions is a boolean expression.
- `code_block1` represents the lines of code that are executed if condition1 is true.
- `code_block2` represents the lines of code that are executed if condition1 is false and condition2 is true.
- `default_code` represents the lines of code that are executed if both conditions evaluate to false.

Exercise 10-2: Chaining `if` Statements

1. Open the project `Exercise_10-2` in NetBeans.

In the `Order` class:

2. Complete the `calcDiscount` method so it determines the discount for three different customer types:
   - Nonprofits get a discount of 10% if total > 900, else 5%.
   - Private customers get a discount of 7% if total > 900, else 0%.
   - Corporations get a discount of 8% if total < 500, else 5%.

In the `ShoppingCart` class:

3. Use the `main` method to test the `calcDiscount` method.

In this exercise, you write a `calcDiscount` method that determines the discount for three different customer types:

- Nonprofits get a discount of 10% if total > 900, else 5%.
- Private customers get a discount of 7% if total > 900, else no discount.
- Corporations get a discount of 8% if total < 500, else 5%.

Topics

- Relational and conditional operators
- More ways to use `if/else` statements
- Using a `switch` statement
- Using the NetBeans debugger

Handling Complex Conditions with a `switch` Statement

The `switch` statement:

- Is a streamlined version of chained `if` statements
- Is easier to read and maintain
- Offers better performance

Coding Complex Conditions: `switch`

```
switch (month) {
    case 1: case 3: case 5: case 7:
    case 8: case 10: case 12:
        System.out.println("31 days in the month.");
        break;
    case 2:
        if (!isLeapYear) {
           System.out.println("28 days in the month.");
        } else
           System.out.println("29 days in the month.");
        break;
    case 4: case 6: case 9: case 11:
        System.out.println("30 days in the month.");
        break;
    default:
        System.out.println("Invalid month.");
}
```


Here you see an example of the same conditional logic (from the previous chained `if` example) implemented as a `switch` statement. It is easier to read and understand what is happening here.

- The `month` variable is evaluated only once, and then matched to several possible values.
- Notice the `break` statement. This causes the `switch` statement to exit without evaluating the remaining cases.

Note: Debugging (covered later in this lesson) reveals why the switch statement offers better performance compared to an `if/else` construct. Only the line containing the true case is executed in a switch construct, whereas every `if/else` statement must be examined up until a statement is found to be true.

`switch` Statement Syntax

Syntax:

```
switch (<variable or expression>) {
   case <literal value>:
       //code_block1
       [break;]
    case <literal value>:
       // code_block2
       [break;]
     default:
        //default_code
{
```


The `switch` construct helps you avoid confusing code because it simplifies the organization of the various branches of code that can be executed.

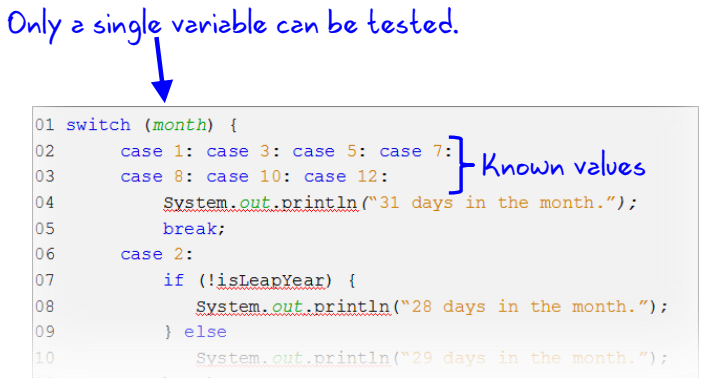
The syntax for the `switch` construct is shown in the slide, where:

- The `switch` keyword indicates a `switch` statement
- `variable` is the variable whose value you want to test. Alternatively, you could use an expression. The `variable` (or the result of the expression) can be only of type `char`, `byte`, `short`, `int`, or `String`.
- The `case` keyword indicates a value that you are testing. A combination of the `case` keyword and a `literal value` is referred to as a *case label*.
- `literal value` is any valid value that a variable might contain. You can have a case label for each value that you want to test. Literal values can be constants (final variables such as `CORP`, `PRIVATE`, or `NONPROFIT` used in the previous exercise), literals (such as '`A`' or `10`), or both.
- The `break` statement is an optional keyword that causes the code execution to immediately exit the `switch` statement. Without a `break` statement, all `code block` statements following the accepted case statement are executed (until a `break` statement or the end of the `switch` construct is reached).

When to Use `switch` Constructs

Use when you are testing:

- Equality (not a range)
- A *single* value
- Against fixed known values at compile time
- The following data types:
  - Primitive data types: `int`, `short`, `byte`, `char`
  - `String` or `enum` (enumerated types)
  - Wrapper classes (special classes that wrap certain primitive types): `Integer`, `Short`, `Byte` and `Character`




```
01 switch (month) {
02      case 1: case 3: case 5: case 7:
03      case 8: case 10: case 12:
04          System.out.println("31 days in the month.");
05          break;
06      case 2:
07          if (!isLeapYear) {
08             System.out.println("28 days in the month.");
09          } else
10             System.out.println("29 days in the month.");
```


If you are not able to find values for individual test cases, it would be better to use an `if/else` construct instead.

Exercise 10-3: Using `switch` Construct

1. Continue editing **`Exercise_10-2`** or open **`Exercise_10-3`**.

In the `Order` class:

2. Rewrite `calcDiscount` to use a `switch` statement:
   - Use a ternary expression to replace the nested if logic.
   - For better performance, use a `break` statement in each case block.
   - Include a `default` block to handle invalid `custType` values.

In the `ShoppingCart` class:

3. Use the main method to test the `calcDiscount` method.

In this exercise, you modify the `calcDiscount` method to use a `switch` construct, instead of a chained `if` construct:

Use a ternary operator instead of a nested `if` within each case block.

Quiz

Which of the following sentences describe a valid case to test in a `switch` construct?

a. The `switch` construct tests whether values are greater than or less than a single value.
b. Variable or expression where the expression returns a supported `switch` type.
c. The `switch` construct can test the value of a `float`, `double`, `boolean`, or `String`.
d. The `switch` construct tests the outcome of a `boolean` expression.

Answer: b

- Answer a is incorrect because you must test for a single value, not a range of values. Relational operators are not allowed.
- Answer b is correct.
- Answer c is incorrect. The `switch` construct tests the value of types `char`, `byte`, `short`, `int`, or `String`.
- Answer d is incorrect. The switch construct tests of value of expressions that return `char`, `byte`, `short`, `int`, or `String`—not `boolean`.

Topics

- Relational and conditional operators
- More ways to use `if/else` statements
- Using a `switch` statement
- Using the NetBeans debugger

Working with an IDE Debugger

Most IDEs provide a debugger. They are helpful to solve:

- Logic problems
  - (Why am I not getting the result I expect?)
- Runtime errors
  - (Why is there a `NullPointerException`?)

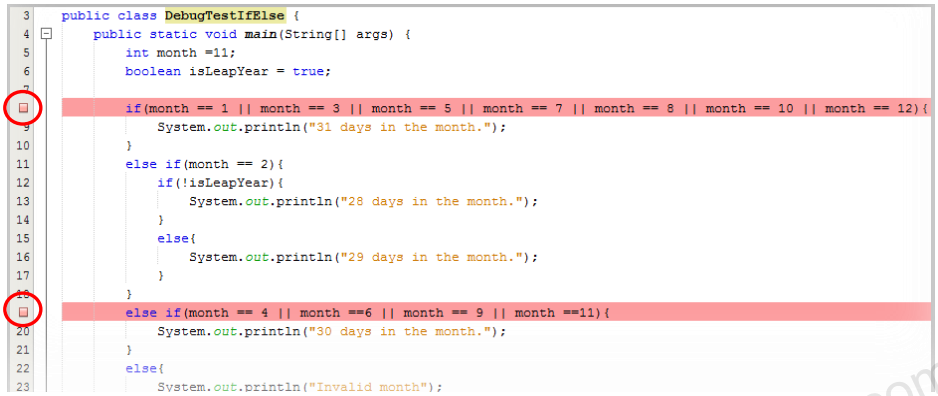
Debugging can be a useful alternative to print statements.

Debugger Basics

- Breakpoints:
  - Are stopping points that you set on a line of code
  - Stop execution at that line so you can view the state of the application
- Stepping through code:
  - After stopping at a break point, you can “walk” through your code, line by line to see how things change.
- Variables:
  - You can view or change the value of a variable at run time.
- Output:
  - You can view the System output at any time.

Setting Breakpoints

- To set breakpoints, click in the margin of a line of code.
- You can set multiple breakpoints in multiple classes.



```
3  public class DebugTestIfElse {
4      public static void main(String[] args) {
5          int month =11;
6          boolean isLeapYear = true;
7
8          if(month == 1 || month == 3 || month == 5 || month == 7 || month == 8 || month == 10 || month == 12){
9              System.out.println("31 days in the month.");
10         }
11         else if(month == 2){
12             if(!isLeapYear){
13                 System.out.println("28 days in the month.");
14             }
15             else{
16                 System.out.println("29 days in the month.");
17             }
18         }
19         else if(month == 4 || month ==6 || month == 9 || month ==11){
20             System.out.println("30 days in the month.");
21         }
22         else{
23             System.out.println("Invalid month");
```

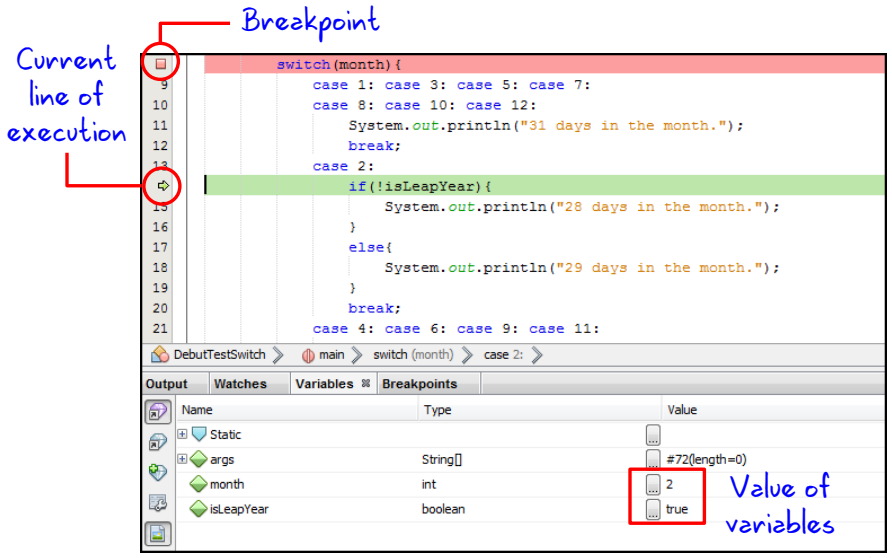
The Debug Toolbar

1. Start debugger
2. Stop debug session
3. Pause debug session
4. Continue running
5. Step over
6. Step over an expression
7. Step into
8. Step out of

Here you see the Debug toolbar in NetBeans. Each button is numbered and the corresponding description of the function of that button appears in the list on the left.

1. Start the debug session for the current project by clicking button 1. After a session has begun, the other buttons become enabled. The project runs, stopping at the first breakpoint.
2. You can exit the debug session by clicking button 2.
3. Button 3 allows you to pause the session.
4. Button 4 continues running until the next breakpoint or the end of the program.
5. Buttons 5 through 8 give you control over how far you want to drill down into the code. For example:
   - If execution has stopped just before a method invocation, you may want to skip to the next line after the method.
   - If execution has stopped just before an expression, you may want to skip over just the expression to see the final result.
   - You may prefer to step into an expression or method so that you can see how it functions at run time. You can also use this button to step into another class that is being instantiated.
   - If you have stepped into a method or another class, use the last button to step back out into the original code block.

Viewing Variables



Here you see a debug session in progress. The debugger stopped at the breakpoint line, but then the programmer began stepping through the code. The current line of execution is indicated by the green arrow in the margin.

Notice that the `isLeapYear` variable on the current line appears in the Variables tab at the bottom of the window. Here you can view the value or even change it to see how the program would react.

Note: Debugging reveals why the `switch` statement offers better performance compared to an `if/else` construct. Only the line containing the true case is executed in a switch construct, whereas every `if/else` statement must be examined up until a statement is found to be true.

Summary

In this lesson, you should have learned how to:

- Use a `ternary` statement
- Test equality between strings
- Chain an `if/else` statement
- Use a `switch` statement
- Use the NetBeans debugger

Practices Overview

- 10-1: Using Conditionals
- 10-2: Debugging

In this practice, you are asked to enhance the soccer league application to keep track of the points and goals of each team. Features are best implemented using conditional statements, but it's up to you to figure out the implementation details.