CSCE 120: Learning To Code

Module 4: Making Decisions

Introduction

This module introduces you to logical statements and operators and their use in conditional control structures in a program.

Conditional Statements

When writing code, its important to be able to distinguish between one or more situations. Based on some *condition* being true or false, you may want to perform some action if its true, while performing another, different action if it is false. Alternatively, you m ay simply want to perform one action if and only if the condition is true, and do nothing (move forward in your program) if it is false.

Normally, the *control flow* of a program is *sequential*: each statement is executed top-to-bottom one after the other. A *conditional* statement interrupts this normal control flow and executes statements only if some specified condition holds. The usual way of achieving this in a programming languages is through the use of the key words, `if`, `else`, and `else if`.

The If Statement

The `if` statement is the simplest form of a conditional statement. The basic syntax is presented here:

```
if(⟨condition⟩) {
  //Conditioned Code
  //code inside this block will execute only if
  //the condition above evaluates to true
}
//rest of code...
```

The `if` keyword is followed by parentheses inside of which we put a condition. The ⟨*condition*⟩ part of the example above is *not* valid JavaScript. It is just a placeholder that we will use temporarily; we will discuss logical statements in the sections below. Following the condition is a *code block* denoted with opening and closing curly brackets. Anything you place inside this code block will execute if and only if the condition that you provide evaluates to true. Otherwise, if the condition is not satisfied, (evaluates to false) then the code block is *not* executed and the program continues executing any code after the closing curly bracket.

The If-Else Statement

The `if-else` statement is a generalization of the `if` statement. With an `if` statement, code is executed or it is not. However, with an `if-else` statement, two blocks of code are provided. Exactly one of these blocks of code will execute whether or not the condition evaluates to true or false. In particular, if the condition evaluates to true, the first block is executed. Otherwise, if the condition evaluates to false, the second block is executed. It will *never* be the case that both (or neither) of the blocks of code will execute. These blocks of code are usually referred to as being *mutually exclusivity*, as the execution of one block precludes the execution of the other. The syntax for writing an `if-else` statement is as follows.

```
if(⟨condition⟩) {
  //Conditioned Code A
  //code inside this block will execute if
  //the condition above evaluates to true...
} else {
  //Conditioned Code B
  //otherwise this block will execute when
  //the condition above evaluates to false
}
```

The If-Else-If Statement

Yet another generalization is the `if-else-if` statement. Here, more than one condition can be specified. The code block associated with the first condition that evaluates to true with be executed. No other code blocks will be executed.

```
if( ⟨condition 1⟩ ) {
  //Conditioned Code A
  //code inside this block will execute if
  //condition 1 above evaluates to true
} else if ( ⟨condition 2⟩ ) {
  //Conditioned Code B
  //code inside this block will execute if
  //condition 1 above evaluates to false and
  //condition 2 above evaluates to true
} else {
  //Conditioned Code C
  //otherwise this block will execute when
  //both of the conditions evaluate to false
}
```

In this example, we've only provided 2 conditions with 3 code blocks. However, you can generalize this and provide as many `else if` statements as you wish. Realize that since the first satisfied statement executes, the ordering of your conditions matters! Another thing to note about `if-else-if` statements is that the final `else` block is optional. If none of the conditions is satisfied and you don't wish for the program to do anything, then you do not need to include the final `else` line.

For all of the previous examples, notice that each inner block of code was indented. This is the preferred style of writing code as it is cleaner and easier to read. It is similar to when writing an outline or lists/sublists: each subsection or sublist is indented, making it more organized and easier to read. When blocks of code are properly indented it is clear which conditional statements they are associated with. Failure to consistently indent code doesn't change the behavior of the code (in general, whitespace does not matter). However, proper and consistent styling makes your code a lot more readable, easier to maintain, and easier to find bugs.

Nesting

You can also *nest* conditional statements within other conditional statements. An example:

```
if( ⟨condition A⟩ ) {
  if( ⟨condition 1⟩ ) {
    //Conditioned Code A-1
  } else {
    //Conditioned Code A-2
  }
} else
  //Conditioned Code B
}
```

In this example, if ⟨*condition A*⟩ evaluates to true, ⟨*condition* 1⟩ is then checked and either code A-1 or code A-2 is executed. However if ⟨*condition A*⟩ evaluates to false, then code B is executed.

Comparison & Logical Operators

Comparison Operators

We now discuss how to write the conditions used in the conditional statements. There are several *comparison operators* that we can use on numeric and string types. These are summarized in Table 1.

Operator	Meaning	Examples

`===`	Equality, true if the two operands are the same type and have the same value, false otherwise.	`a === 5`
		`a === "five"`
		`a === b`

`!==`	Inequality, true if the two operands are either different types or do not have the same value, false otherwise.	`a !== 5`
		`"5" !== 5`
		`a !== b`

`<`	Strictly Less Than, true if the left-hand-size is strictly less than the right-hand-side, false otherwise. Works for both numeric and string types.	`a < 5`
		`a < b`
		`"a" < "b"`
		`a < "five"`

`>`	Strictly Greater Than, true if the left-hand-size is strictly greater than the right-hand-side, false otherwise. Works for both numeric and string types.	`a > 5`
		`a > b`
		`"a" > "b"`
		`a > "five"`

`<=`	Less-than-or-equal-to, true if the left-hand-side is less than or equal to the right-hand-side, false otherwise.	`a <= 5`
		`a <= b`
		`"a" <= "b"`
		`a <= "five"`

`>=`	Greater-than-or-equal-to, true if the left-hand-side is greater than or equal to the right-hand-side, false otherwise.	`a >= 5`
		`a >= b`
		`"a" >= "b"`
		`a >= "five"`

Table 1: Comparison Operators

Lexicographic Ordering

When applied to numbers (either literals or variables whose value is a number), the operators in Table 1 act as expected. When applied to string types (either literals or

variables whose value is a string), the operators rely on something called *lexicographic ordering*. For the most part, this is simply alphabetic ordering so that `"alpha"` would come before `"beta"` (that is, `"alpha" < "beta"` is true). However, strings can include other types of characters. The exact ordering of characters is determined by the ASCII text table (see http://en.wikipedia.org/wiki/ASCII for details). For our purposes just know that: numerical characters come before uppercase letters which come before lowercase letters. Moreover, each of these three types of characters is ordered in the usual manner. Finally, just as in alphabetic ordering, shorter words come before longer words when equal (`"race"` comes before `"racer"`).

Boolean Variables

Recall that we can use the keywords `true` and `false` in JSON data. We can also assign these values to a variable *or* we can assign the result of an expression to a variable which then takes on the values `true` or `false`. This allows us to use a single variable as a *flag* variable in a conditional statement:

```
1 var a = true;  //example of assigning a boolean value to a variable
2 ...
3 var isAdult = (age >= 18);
4 if(isAdult) {
5   console.log("Get a job!");
6 }
```

In line 3, the expression `(age >= 18)` is evaluated (and evaluates to either true or false) with the result (`true` or `false`) being placed into the variable `isAdult`.

Other Equality Operators

You may see code snippets or other examples that use the equality operators `==, !=` (sometimes referred to as loose equality operators) which have a similar meaning to the two equality operators presented in Table 1 (called *strict* equality operators). However there is a subtle (and potentially dangerous) difference. When applied to variables or expressions of the same type (that is they are both numbers or both strings), there is no difference. However, when applied to variables or expressions that have different types, weird things happen.

For example, the expression, `"5" == 5` would evaluate to `true`! When these operators are used, a complex set of rules involving something called *type coercion* occurs. The situations in which you'd actually want this behavior are rare. Given the pitfalls, we'll be avoiding these issues altogether. As a general rule, always use `===` and `!==`.

Logical Operators

We now create more complex logical statements using *logical* operators. Suppose you want to condition a block of code on *two* conditions or expressions. That is, you want a block of code to execute if condition A is true *and* if condition B is true. Alternatively, you could condition a block of code on two conditions such that the code will execute if condition A is true *or* if condition B is true.

Both of these statements can be achieved using the logical *and* operator using two ampersands, `&&` and the logical *or* operator using two vertical bars, `||` respectively. The syntax and details for these two operations are summarized in Table 2.

Operator	Meaning	Examples
`!`	Logical *not* operator. Applied to one expression, it "flips" the value of the expression; true to false, false to true.	`!(x === 0)` `!isAdult`
`&&`	Logical *and* operator. Applied to two other expressions, the result is true only if both expressions are true.	`x >= 0 && x <= 10` `a === 0 && b !== 0`
`||`	Logical *or* operator. Applied to two other expressions, the result is true if *at least one* of the expressions is true.	`a > 0 || b != 0` `x < 0 || x > 10`

Table 2: Logical Operators

The *not* logical operator is similar to the inequality operator: it *negates* the statement it is applied to. If the expression evaluates to true, negating it makes it false and vice versa. In contrast to the logical and/or operators, the negation is only applied to one expression. Table 3 gives the truth values for the logical and and or operators for each of the 4 possible combinations.

If `a` is...	and `b` is...	then `a && b` is...	and `a || b` is...
`false`	`false`	`false`	`false`
`false`	`true`	`false`	`true`
`true`	`false`	`false`	`true`
`true`	`true`	`true`	`true`

Table 3: Truth values of logical and and or operators.

Order Of Precedence

Recall that some arithmetic operators are evaluated before others (multiplication and division is done before addition and subtraction). Something similar happens when using logical operators: any negation is evaluated first. Then, any logical and operators are

evaluated, finally logical or operators are evaluated last. As with arithmetic expressions, it is best to avoid these issues altogether and write your expressions using parentheses.