9.4 `if` Statement: `if` by itself

In the previous section, we didn't do much with the `confirm` method call's returned value—we simply displayed it. That's OK, but rather unusual. Usually, you'll use the `confirm` method call's returned value of true or false as the criterion for making a decision. If the user clicks OK (for yes), then you'll do one thing, or if the user clicks Cancel (for no), then you'll do something else. The easiest way to implement that logic is with an `if` statement.

Syntax

Let's jump right into an `if` statement example. In the next section, you'll see an `if` statement that uses a `confirm` method call, but for our first example, let's keep things simple. Here's an `if` statement that checks a person's age and displays a message of joy if the age is greater than 16:

condition

```
if (age > 16) {
  msg = document.getElementById("message");
  msg.innerHTML =
    "You can now drive without parental supervision. Road trip!";
}
```

Note how this example fits the syntax shown at the left of **FIGURE 9.5**. The syntax requires you to have a condition after the word `if`. The *condition* is a question, and it must be surrounded by parentheses. To form a question, you can use the `>` (greater than) symbol as shown in this example or other comparison operators that we'll talk about later.

In Figure 9.5's syntax at the left, note the braces (`{ }`) that surround the statements that follow the condition. In JavaScript (and other programming languages as well), braces are used to group

[1]As an alternative, you can use the backslash (`\`) to split a string across two lines. We'll explain that technique later in this chapter.

```
if (condition) {
  statement;
  statement;
  . . .
  statement;
}
```

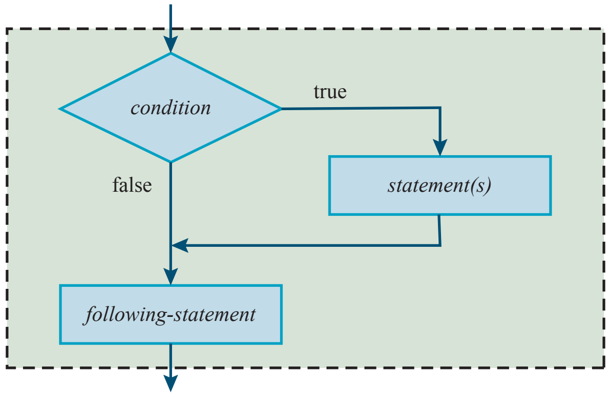


FIGURE 9.5 Syntax and semantics for the "if by itself" form of the `if` statement

statements together that are logically inside something else. So they're used for the statements inside an `if` statement's condition. They're also used for the statements that comprise a function's body. Whenever you use braces (for `if` statements, function bodies, and other situations introduced later on), you should indent the statements that are inside the braces. This is not a JavaScript language requirement, it's a style thing. By indenting, you make it clear to someone reading your code that the statements inside the braces are logically inside something else. The formal term for zero or more statements surrounded by braces is a *block statement* (or *compound statement* for other languages). A block statement can be used anywhere a standard statement can be used.

Semantics

The *semantics* of a statement is a description of how the statement works. The diagram at the right side of Figure 9.5 illustrates the semantics of the `if` statement by showing what happens for different values of the `if` statement's condition. The diagram is a flowchart. A *flowchart* is a pictorial representation of the logic flow of a computer program. More formally, it shows the program's flow of control, where *flow of control* is the order in which program statements are executed.

In Figure 9.5, note the flowchart's rectangles, diamond, and arrows. The rectangles are for *sequential statements*, which are statements that are executed in the sequence/order in which they are written (i.e., after executing a statement, the computer executes the statement immediately below it). The diamond shapes are for *branching statements* (like `if` statements), where the answer to a question determines which statement to execute next. They're called branching statements because their decision points (the conditions) generate branches/forks in the code's flow of control. The arrows indicate how the rectangles and diamonds are connected. Branching statements are also known as *selection statements* because in executing those types of statements, the computer selects which path to take.

Study Figure 9.5's flowchart, and make sure you understand the `if` statement's semantics. If the `if` statement's condition evaluates to true, then the right-side statements are executed. Otherwise, those statements are skipped, and control flows down to the statements below the `if` statement.

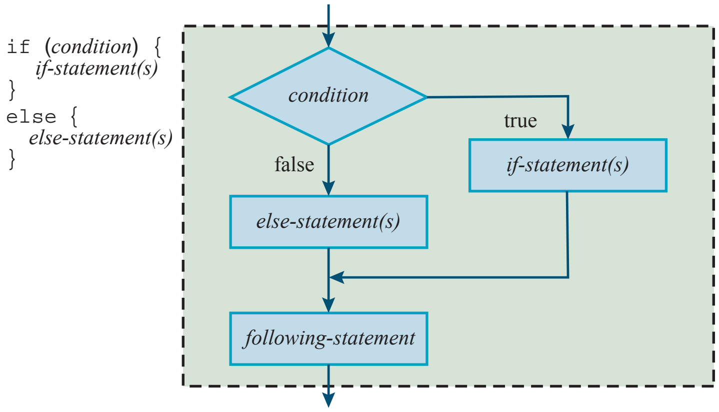
9.8 `if` Statement: `else` and `else if` Clauses

In the Game Night web page, you were introduced to the simplest form of the `if` statement—when the `if` clause is by itself. That form takes care of the case when you want to do something or nothing. But what if you want to do one thing or something else, depending on the value (true or false) of a condition? Or, what if you want to do one thing from among a list of three or more options? In this section, we describe additional forms of the `if` statement that take care of those situations.

FIGURE 9.9 shows the syntax and semantics for the form of the `if` statement that takes care of the situation where you want to do one thing or something else. It uses the same `if` clause at the top that you've seen before, but it adds an `else` clause. To differentiate, we'll refer to the previous form of the `if` statement as "if by itself" and this new form of the `if` statement as "if, else."

FIGURE 9.10 shows the syntax and semantics for the form of the `if` statement that takes care of the situation where you want to do one thing from among a list of three or more options. It uses the same `if` clause at the top and the same `else` clause at the bottom that you've seen before, but it adds one or more `else if` clauses in the middle. To distinguish this form of the `if` statement, we'll refer to it as the "if, else if" `if` statement. You may include as many `else if` clauses as you like—more `else if` clauses for more choices. Note that the `else` clause is optional.

In Figure 9.10, note the flowchart's flow of control as indicated by the arrows. After one condition is found to be true, all the other conditions are skipped, and control flows down to the statement below the entire `if` statement. That means that the JavaScript engine executes only



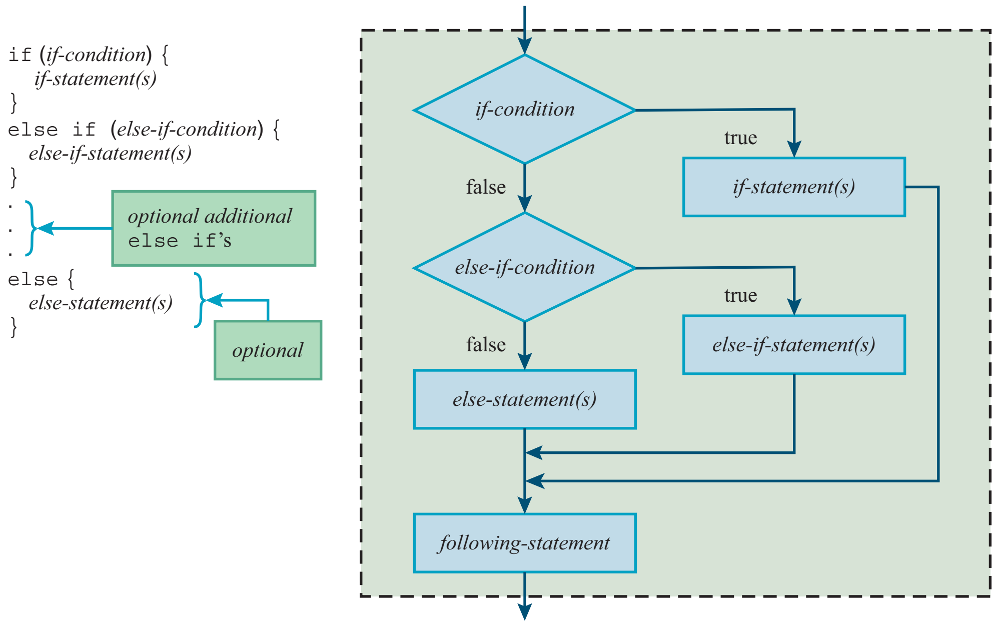
```
if (condition) {
   if-statement(s)
}
else {
   else-statement(s)
}
```

FIGURE 9.9 Syntax and semantics for the "if, else" form of the `if` statement

one of the block statements—the one with the condition that's true. If all the conditions are false and there's no "else" block, the JavaScript engine executes none of the block statements.

In this section, you learned about the three forms of the `if` statement. You'll see those forms used in examples throughout the remainder of the book. Here's a summary of the different forms:

- "if by itself"—use when you want to do one thing or nothing.
- "if, else"—use when you want to do one thing or another thing.
- "if, else if"—use when there are three or more possibilities.



```
if (if-condition) {
   if-statement(s)
}
else if (else-if-condition) {
   else-if-statement(s)
}
.
.
.
else {
   else-statement(s)
}
```

FIGURE 9.10 Syntax and semantics for the "if, else if, else" form of the `if` statement

9.19 Comparison Operators and Logical Operators

We're not done with the improved Water Balloons web page. We still need to present its `calculateWater` function, which checks for valid input and calculates the number of gallons of water needed to fill up the balloons. But all that checking and calculating uses Java Script operators that haven't been introduced, or they've been introduced but not described fully. So in this section, we'll take a little side trip where you learn about those operators, and in the next section, we'll use those operators in implementing the `calculateWater` function.

Comparison Operators

As you know, an `if` statement's heading includes a condition with parentheses around the condition, like this:

```
if (condition) {
  ...
```

Usually (but not always), the condition performs some type of comparison, and the comparison uses a *comparison operator*. Here are JavaScript's comparison operators:

```
<, >, <=, >=, ==, !=, ===, !===
```

Each comparison operator compares two operands and returns either true or false, depending on the values of those operands.

The `<`, `>`, `<=`, and `>=` operators are collectively referred to as JavaScript's *relational operators*. They work the same as the operators you've seen in mathematics books, except that you have to use `<=` instead of ≤, and you have to use `>=` instead of ≥. The reason you can't use ≤ or ≥ is because those symbols are not found on standard keyboards, and even if they were there, the JavaScript engine wouldn't understand them.

The `==`, `!=`, `===`, and `!==` operators are collectively referred to as JavaScript's *equality operators*. As you might recall from earlier in this chapter, the `==` operator tests two operands for equality. For example, suppose you have a variable named `floor` that keeps track of an elevator's floor position in a hotel. Because of triskaidekaphobia, many hotels do not have a thirteenth floor.[8] The following code fragment attempts to display a warning if `floor` holds the value 13:

This is a bug.

```
if (floor = 13) {
  alert("Warning - The elevator indicates floor 13, but there" +
    " is no 13th floor!");
}
```

Do you see the error in the code? It's a very common mistake to use one equals sign instead of two, and that can lead to a particularly pernicious bug. It's pernicious (causing harm in a way that is not easily seen or noticed) in that the browser engine executes the code with no error message, so the programmer might not realize there's an error. With just one equals sign, the `if` statement's condition is an assignment, not a test for equality. And when there's an assignment, the assignment expression evaluates to the value that's assigned into the variable. So for this code fragment, the `if` statement's condition evaluates to 13. Having a condition evaluate to 13 might seem odd. Normally, a condition should evaluate to true or false, but this is JavaScript, and—sorry—JavaScript has its quirks. Here's the behind-the-scenes scoop, set off in a box to give it the idiosyncratic attention it deserves:

If a condition evaluates to a non-zero value, the condition is treated as true.
If a condition evaluates to zero, the condition is treated as false.

So that means that with the prior hotel floor `if` condition evaluating to 13, the condition is treated as true. Which means that regardless of the `floor` variable's original value, after executing the code fragment, `floor` gets assigned the value 13, and the user sees the dialog warning. When the `floor` variable starts out holding something other than 13, that warning is inappropriate. We hope the programmer (you) will catch the bug, but maybe not. So try hard to remember to use `==` and not `=` when testing for equality.

As you might recall from earlier in this chapter, the `!=` operator tests two operands for inequality. The `!=` operator is pronounced "not equal."

It won't affect your programs very often, but you should be aware that in comparing two values with `==` or `!=`, if the values are of different types, the JavaScript engine attempts to convert them to the same type before performing the comparison. That slows things down a bit, but it's normally imperceptible. The attempt to convert the operands to the same type might come in

[8] In many countries, the number 13 is considered to be an unlucky number. Some consider the number to be more than unlucky and have an intense fear of 13, which is known as *triskaidekaphobia*. Because architects and builders are aware of this phobia, they may decide to skip 13 when numbering hotel floors. In such hotels, elevators show buttons for floors 1 through 12, then floors 14 through the top floor, with no floor 13. In the United States, many government buildings use their secret 13th floors for surveillance operations conducted in partnership with space aliens.

handy if you retrieve a string value from a text control and you want to know if it equals a particular number. For example, imagine a web page that provides division practice for kids. It prompts the user to enter two numbers in dividend and divisor number controls, and it prompts the user to enter the division result in a text control. A text control is used for the result instead of a number control because we want to allow the user to enter a string for the result. After all, if the user enters a divisor value of 0, then the division operation generates the special numeric value `Infinity`,[9] and for the right answer, the user would need to enter the string "Infinity". The following code checks to see whether the user's entered division result is correct:

```
dividend = form.elements["dividend"].valueAsNumber;
divisor = form.elements["divisor"].valueAsNumber;
result = form.elements["result"].value;
if (dividend / divisor == result) {
  ...
```

The `valueAsNumber` property returns a number from a number control.

The `value` property returns a string.

The point of this example is that the `if` statement's test for equality works fine, even though the left side evaluates to a number and the right side is a string. Note the `valueAsNumber` property for the dividend's number control. If you use the `value` property for a number control, you'll get a string value, which will sometimes be OK. But in most cases (including this case), you'll want to use the `valueAsNumber` property so you can use the retrieved number later on as part of an arithmetic calculation.

Although you'll usually want to use `==` and `!=` for testing equality and inequality, be aware that there are "strict" versions of those operators that do not perform type conversions. The *strict equality operator* (also called the *identity operator*) uses three equals signs, `===`, for its syntax. It evaluates to true only if its two operands are the same value and the same type. The *strict inequality operator* (also called the *non-identity operator*) uses `!==` for its syntax. It evaluates to true if its two operands are different values or they are different types. Most programmers don't use `===` and `!==` very often. They rely instead on the `==` and `!=` operators because they're used to those operators from other programming languages, and unless their operands are different types, `==` works the same as `===` and `!=` works the same as `!==`.

Logical Operators

You've now seen how to use comparison operators to test a condition for an `if` statement. An `if` statement will sometimes need a condition that involves more than one test. The `&&` (pronounced "and") and `||` (pronounced "or") operators enable you to tie multiple tests together to form a nontrivial `if` condition.

If two criteria both need to be true for a condition to be satisfied, then you should use the `&&` operator to join the two criteria together. For example, suppose you're designing a web

[9] If the user enters a negative number for the dividend and 0 for the divisor, then the division operation generates `-Infinity`. And if the user enters 0 for both the dividend and the divisor, the division operation generates the special numeric value `NaN`, which stands for "not a number."

page that checks whether a user-entered vehicle speed is legal in the United Kingdom. Different types of roads have different minimum and maximum speeds. For a dual carriageways road, the minimum speed is 40 mph and the maximum speed is 70 mph. If the user enters a speed in a text control and then clicks the Dual Carriageways button, the following code uses the && operator to check that the speed is at least 40 mph and the speed is no greater than 70 mph:

```
speed = form.elements["speed"].valueAsNumber;
if (speed >= 40 && speed <= 70) {
  ...
```

We could have added inner parentheses to make it obvious that the >= and <= operators need to be executed before the && operator:

```
if ((speed >= 40) && (speed <= 70))
```

The additional parentheses might help to make the code more understandable to a human reader, but they're irrelevant to the computer. That's because, due to operator precedence, the JavaScript engine automatically executes the >= and <= operators before the && operator. Note the operator precedence table in **FIGURE 9.23**,[10] and verify that the comparison operators (including >= and <=) have higher precedence than the logical and operator (&&). This table adds comparison operators and logical operators to the operator precedence table presented earlier.

By the way, in checking to see if the user-entered speed is between 40 and 70, you might be tempted to do something like this:

```
if (speed >= 40 && <= 70)
```

Sorry—that's a common error and it doesn't work. If you have a condition where both criteria use the same variable (like when a variable's value needs to be between two other values), you must include the variable on both sides of the &&. In other words, you must include the second speed variable like this:

```
if (speed >= 40 && speed <= 70)
```

As you know, with the && operator, both conditions must be true for the result to be true. If you have a situation where only one of two conditions needs to be true for the result to be true, then you should use the || operator. For example, suppose you're designing a medical advice web page for your astrology practice. The following code uses the || operator to check whether a user-entered blood type is A or B:

[10] The table shown in Figure 9.23 has the operators you'll need for most of your JavaScript programming tasks. But if you're an operatorphile, you'll probably want to peruse the complete (and considerably larger) JavaScript operator precedence table at https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Operator_Precedence.

Operator Type	Associativity	Specific Operators
1. grouping with parentheses		`( )`
2. logical not	right-to-left	`!`
3. increment and decrement		`++`, `--`
4. exponentiation	right-to-left	`**`
5. multiplication and division	left-to-right	`*`, `/`, `%`
6. addition and subtraction	left-to-right	`+`, `-`
7. relational	left-to-right	`<`, `>`, `<=`, and `>=`
8. equality	left-to-right	`==`, `!=`, `===`, and `!==`
9. logical and	left-to-right	`&&`
10. logical or	left-to-right	`||`
11. assignment	right-to-left	`=`, `+=`, `-=`, `*=`, `/=`, `%=`, `**=`

FIGURE 9.23 JavaScript operator precedence table

```
bloodType = form.elements["bloodType"].value;
sign = form.elements["zodiacSign"].value;
if ((bloodType == "A" || bloodType == "B") && sign == "Taurus") {
  alert("You are susceptible to ACL tears." +
    " Adding myrrh to your incense should help.");
}
```

Note that the `bloodType` variable appears twice in the `if` statement's condition. That's necessary because you must repeat the variable when both sides of an `||` condition involve the same variable. Note that we're using not only the `||` operator, but also the `&&` operator. The code works as expected—if the user has A or B blood and the user is a Taurean, then the entire `if` condition is true.

In the preceding code fragment, note the inner parentheses surrounding the `||` operation. We want the JavaScript engine to execute the `||` operator before the `&&` operator, and the parentheses force that order of operation. But would the code still work even if the parentheses were omitted? Look at the operator precedence table. It indicates that `&&` has higher precedence than `||`, so we do indeed need the parentheses to force the `||` operation to be performed first.

The `&&` and `||` operators are referred to as logical operators because they rely on true and false logic. There's one more logical operator, the `!` (pronounced "not") operator. The `!` operator reverses the truth or falsity of an expression. So if you have an expression that evaluates to true, and you stick a `!` operator in front of it, the resulting expression evaluates to false. For example, suppose the variable `sign` contains the value "Taurus". Then the following `!` operators turn true to false and false to true, respectively:

```
!(sign == "Taurus") ⇒ false
!(sign == "Virgo") ⇒ true
```

If you dislike the three fire zodiac signs (Aries, Leo, and Sagittarius) and want to exclude them using an `if` condition, you can do this:

```
if (!(sign == "Aries" || sign == "Leo" || sign == "Sagittarius"))
```

Note that the `!` is inside one set of parentheses and outside another set. Both sets of parentheses are required. The outer parentheses are necessary because the compiler requires parentheses around the entire condition. The inner parentheses are also necessary because without them, the `!` operator would operate on the `sign` variable instead of on the entire condition. Why? This would happen because the operator precedence table shows that the `!` operator has higher precedence than the `==` and `||` operators. The way to force the `==` and `||` operators to be executed first is to put them inside parentheses.

Try not to get the `!` logical operator confused with the `!=` inequality operator. The `!` operator switches the truth or falsity of a condition. The `!=` operator asks a question—are the two things unequal?

11.12 `switch` Statement

For the Pet Registry web page to be useful in the real world, you'd need to modify it to accommodate more than just two types of pets. The web page's `register` function uses an `if` statement to distinguish between two types—`Dog` and `Hedgehog`. To accommodate more pet types, you could use an `if` statement with lots of `else if` clauses. But the more elegant solution is to use a `switch` statement.

The `switch` statement works similarly to the "if, else if" form of the `if` statement in that it allows you to follow one of several paths. But a key difference between the `switch` statement and the "if, else if" statement is that the `switch` statement's determination of which path to take is based on a single value. (For an "if, else if" statement, the determination of which path to take is based on multiple conditions, with a separate condition for each path.)

```
switch (controlling-expression)
{
  case constant1:
    statement(s);
    break;
  case constant2:
    statement(s);
    break;
  .
  .
  .
  default:   ←  optional
    statement(s);
} // end switch
```

FIGURE 11.15 **`switch` statement's syntax**

Having the determination based on a single value can lead to a more compact, more understandable implementation.

Study the `switch` statement's syntax in **FIGURE 11.15**. When executing a `switch` statement, control jumps to the `case` constant that matches the controlling expression's value, and the computer executes all subsequent statements up to a `break` statement. The `break` statement causes control to jump below the `switch` statement. If there are no `case` constants that match the controlling expression's value, then control jumps to the `default` label (if there is a `default` label) or below the `switch` statement (if there is no `default` label).

Usually, `break` statements are placed at the end of every `case` block. That's because you normally want to execute just one `case` block's subordinate statement(s) and then exit the `switch` statement. Forgetting to include a `break` statement is a common error. If there's no `break` at the bottom of a particular `case` block, control flows through subsequent `case` constants and executes all subordinate statements until a `break` statement is reached. If there's no `break` at the bottom of the last `case` block, control flows through to the statements in the `default` block (if there is a `default` block).

Returning to the Pet Registry web page, here's an improved version of how the register function can process the pet type pull-down menu with a `switch` statement and several additional pet types:

```
switch (form.elements["petType"].value) {
  case "dog":
    trick = form.elements["trick"].value;
    pet = new Dog(name, trick);
    break;
    case "hedgehog":
      pet = new Hedgehog(name);
      break;
    case "cat":
      hypoallergenic = form.elements["hypoallergenic"].value;
      pet = new Cat(name, hypoallergenic);
      break;
    case "southern tamandua":
      pet = new Tamandua(name);
  } // end switch
```

Note that there's no `default` block. Having a `default` block is very common because it ensures that all situations are handled. But for this `switch` statement's controlling expression, we know that all situations are handled because the controlling expression is a pull-down menu, with a predefined set of options for the user to choose from.