SAMPLE CHAPTER

Get Programming with JavaScript

John R. Larsen

manning

Get Programming with JavaScript

by John R. Larsen

Chapter 12

Copyright 2016 Manning Publications

brief contents

12 Conditions: choosing code to run

This chapter covers

- Comparing values with comparison operators
- Checking conditions that are `true` or `false`
- The `if` statement—running code only if a condition is met
- The `else` clause—running code when a condition is not met
- Making sure user input won't break your code
- Generating random numbers with `Math.random()`

So far, all of your code follows a single path. When you call a function, every statement in the function body is executed. You've managed to get a lot done and covered quite a few core ideas in JavaScript but your programs have lacked flexibility; they haven't been able to decide whether or not to execute blocks of code.

In this chapter you learn how to run code only when specified conditions are met. Suddenly, your programs can branch, providing options, flexibility, and richness. You can increase a player's score *if* they splat a kumquat, move to a new location *if* the user specifies a valid direction, or post a tweet *if* it's less than 141 characters long.

If you want to find out how your programs can make decisions, read on, else, … well, read on anyway. You really need to know this stuff!

12.1 Conditional execution of code

To start, create a simple program that asks the user to guess a secret number. If they guess correctly, the program says, "Well done!" An interaction at the console might look like this:

```
> guess(2)
  undefined
> guess(8)
  Well done!
  undefined
```

What's with the ugly appearances of `undefined`? When you call a function at the console, its code is executed and then its return value is displayed. The `guess` function in the following listing doesn't include a return statement so it automatically returns `undefined`.

Listing 12.1 Guess the number
http://jsbin.com/feholi/edit?js,console

```
var secret = 8;

var guess = function (userNumber) {
  if (userNumber === secret) {
    console.log("Well done!");
  }
};
```

Assign a number to the secret variable

Define a function that accepts the user's number and assign it to the guess variable

Check if the user's number matches the secret number

Log "Well done!" to the console if the numbers match

The `guess` function checks to see if the user's number is equal to the secret number. It uses the *strict equality operator*, `===`, and an *if statement* so that you display the "Well done!" message only if the numbers match. The following sections look at the strict equality operator and the `if` statement in more detail and introduce the *else clause*.

12.1.1 The strict equality operator, ===

The strict equality operator compares two values. If they're equal it returns `true`; if they're not equal it returns `false`. You can test it at the console:

```
> 2 === 8
  false
> 8 === 8
  true
> 8 === "8"
  false
> "8" === "8"
  true
```

In the third example, you can see that the strict equality operator doesn't consider the number `8` and the string `"8"` to be equal. That's because numbers and strings are different types of data. The values `true` and `false` are a third type of data; they're called *boolean* values. In fact, `true` and `false` are the only possible boolean values. Boolean values are useful when deciding what a program should do next; for example, by using an `if` statement.

12.1.2 The if statement

To execute a block of code only when a specified condition is met, you use an `if` statement.

```
if (condition) {
    // Code to execute
}
```

If the condition in parentheses evaluates to `true`, then JavaScript executes the statements in the code block between the curly braces. If the condition evaluates to `false`, then JavaScript skips the code block. Notice there's no semicolon after the curly braces at the end of an `if` statement.

Listing 12.1 used the strict equality operator to return a `true` or `false` value for the condition.

```
if (userNumber === secret) {
    console.log("Well done!");
}
```

The code logs the "Well done!" message to the console only if the value of `userNumber` is equal to the value of `secret`. For example, say the `secret` is `8` and the user chooses `2`:

```
if (2 === 8) {                  // The condition is false.
    console.log("Well done!");  // Not executed
}
```

If the user chooses `8`, the `if` statement becomes

```
if (8 === 8) {                  // The condition is true.
    console.log("Well done!");  // This is executed.
}
```

12.1.3 The else clause

Sometimes we want different code to be executed if the condition in an `if` statement evaluates to `false`. We can make that happen by appending an `else` clause to the `if` statement (figure 12.1).

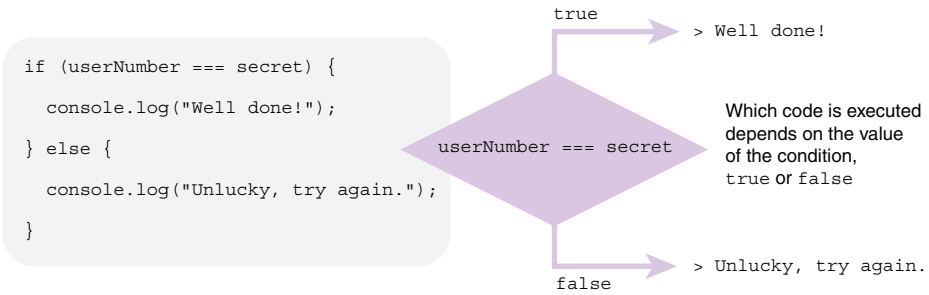

Figure 12.1 Executing code depending on the value of a condition, with `if` and `else`

From listing 12.2:

```
if (userNumber === secret) {
    console.log("Well done!");
} else {
    console.log("Unlucky, try again.");
}
```

If `userNumber` and `secret` are equal, JavaScript displays "Well done!" Otherwise, it displays "Unlucky, try again." Notice there's no semicolon after the curly braces at the end of an `else` clause. Once again, say the `secret` is `8` and the user chooses `2`:

```
if (2 === 8) {                              // The condition is false.
    console.log("Well done!");              // Not executed.
} else {
    console.log("Unlucky, try again.");     // This is executed.
}
```

If the user chooses `8`, the `if` statement becomes

```
if (8 === 8) {                              // The condition is true.
    console.log("Well done!");              // This is executed.
} else {
    console.log("Unlucky, try again.");     // Not executed.
}
```

A guessing game interaction at the console might now look something like this:

```
> guess(2)
  Unlucky, try again.
  undefined
> guess(8)
  Well done!
  undefined
```

Listing 12.2 Guess the number—the `else` clause (http://jsbin.com/nakosi/edit?js,console)

```
var secret = 8;

var guess = function (userNumber) {
    if (userNumber === secret) {

        console.log("Well done!");

    } else {

        console.log("Unlucky, try again.");

    }
};
```

Add an if statement with the condition to check in parentheses

Execute this code only if the condition is true

Add an else clause for when the condition is false

Execute this code only if the condition is false

Next, you use local variables to make `secret` secret.

12.1.4 Hide the secret number inside a function

In listing 12.2, both the `secret` and the `guess` variables are declared outside any function. You saw in chapter 11 how that makes them global variables, accessible at the console and throughout the program. That's great for `guess`—you want users to be able to guess numbers—but it's a disaster for `secret`—users can peek and tweak its value at will. If you run the code in listing 12.2, you can then perform the following actions at the console:

```
> secret          // You can access secret. It's a global variable.
  8
> guess(8)
  Well done!
  undefined
> secret = 20     // You can reset secret to whatever you want.
  20
> guess(20)
  Well done!
  undefined
```

That's not much of a guessing game!

Chapter 11 also discussed how functions are used in JavaScript to create a local scope, a collection of variables accessible only within the function. Listing 12.3 uses the `getGuesser` function to hide the secret number. The function returned by `getGuesser` is assigned to the `guess` variable (figure 12.2).

`guess` is a global variable, available at the console:

```
> guess(2)
  Unlucky, try again
  undefined
```

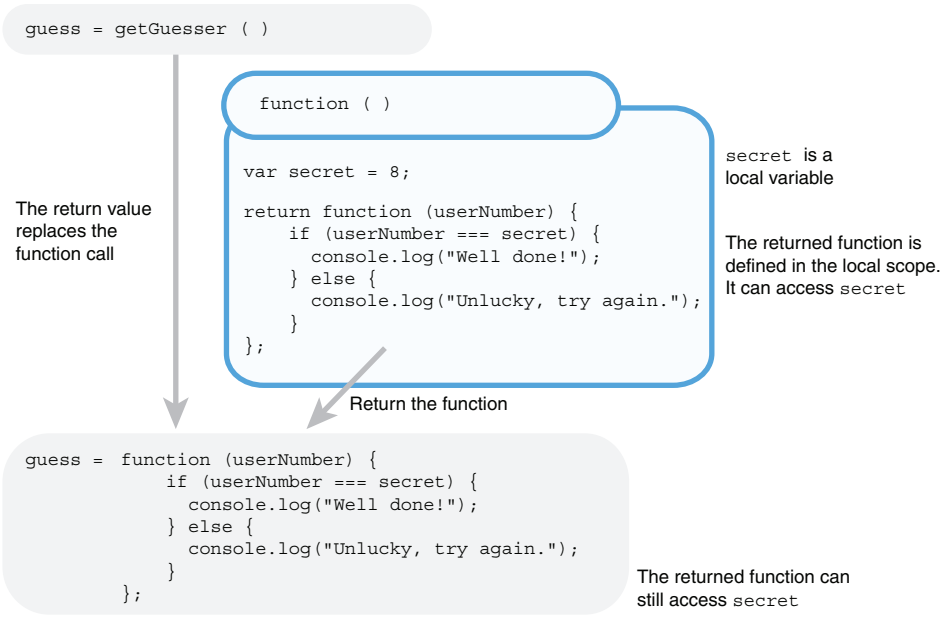

Figure 12.2 The function returned by `getGuesser` is assigned to the `guess` variable.

Listing 12.3 Guess the number—using local scope (http://jsbin.com/hotife/edit?js,console)

```
var getGuesser = function () {
  var secret = 8;

  return function (userNumber) {
    if (userNumber === secret) {
      console.log("Well done!");
    } else {
      console.log("Unlucky, try again.");
    }
  };
};

var guess = getGuesser();
```

Use a function to create a local scope

Hide the secret number within the local scope

Return a function the user can use to guess the number

Check to see if the user's guess is equal to the secret number

Log "Well done!" only if the numbers match

… otherwise log "Unlucky, try again"

Call getGuesser and assign the function it returns to the `guess` variable

The function assigned to `getGuesser` creates a local scope that lets you protect the `secret` variable from the user. It returns another function that lets the user guess a number. That function is assigned to the `guess` variable. Because the guess-checking function is defined in the local scope created by the `getGuesser` function, it has access to the `secret` variable and is able to do its checking.

You have a guessing game but it's always the same secret number. Really, it's a not-so-secret number! Let's make use of a couple of methods from JavaScript's `Math` namespace to inject some extra mystery into our guessing game.

12.2 Generating random numbers with Math.random()

The `Math` namespace provides you with a `random` method for generating random numbers. It always returns a number greater than or equal to 0 and less than 1. Give it a whirl at the console prompt:

```
> Math.random()
  0.7265986735001206
> Math.random()
  0.07281153951771557
> Math.random()
  0.552000432042405
```

Obviously, your numbers will be different because they're random! Unless you're really into guessing games and have a lot of free time, those random numbers are probably a bit too tricky for your purposes.

To tame the numbers, scale them up to fall in a range you want and then convert them to whole numbers (integers). Because they start off less than 1, multiplying by 10 will make them less than 10. Here's a series of assignments using `Math.random`:

```
var number = Math.random();             //  0 <= number < 1
```

To scale the possible numbers, multiply:

```
var number = Math.random() * 10;        //  0 <= number < 10
```

To shift the possible numbers up or down, add or subtract:

```
var number = Math.random() + 1;         //  1 <= number < 2
```

To scale and then shift, multiply and then add:

```
var number = Math.random() * 10 + 1;    //  1 <= number < 11
```

Notice for the last assignment the numbers will be between 1 and 11; they can equal 1 but will be less than 11. The `<=` symbol means less than or equal to, whereas the `<` symbol means less than. The inequality `0 <= number < 1` means the number is between 0 and 1 and can equal 0 but not 1 (see section 12.3.1).

Okay, so you've scaled up the random numbers, but they're still a trifle tricky. At the console you can see the kind of numbers you're generating:

```
> Math.random() * 10 + 1
  3.2726867394521832
> Math.random() * 10 + 1
  9.840337357949466
```

The last step is to lose the decimal fraction part of each number, to round the numbers down to integers. For that you use the `floor` method of the `Math` namespace.

```
> Math.floor(3.2726867394521832)
  3
> Math.floor(9.840337357949466)
  9
```

The `floor` method *always rounds down*, whatever the decimals are: 10.00001, 10.2, 10.5, 10.8, and 10.99999 are all rounded down to 10, for example. You use `floor` to get an expression that returns a random integer between 1 and 10 inclusive:

```
var number = Math.random() * 10 + 1               // 1 <= number < 11
var number = Math.floor(Math.random() * 10 + 1)   // 1 <= number <= 10
```

There's also a `Math.ceil` method that always rounds up and a `Math.round` method that rounds up or down, following the usual rules for mathematical rounding. More information about JavaScript's `Math` object can be found on the *Get Programming with JavaScript* website: http://www.room51.co.uk/js/math.html.

Listing 12.4 puts the `Math` methods into practice. The `guess` function now *returns* strings rather than logging them; the console automatically displays the return values, tidying up the interactions:

```
> guess(2)
  Unlucky, try again.
> guess(8)
  Unlucky, try again.
> guess(7)
  Well done!
```

Listing 12.4 Guess the random number (http://jsbin.com/mezowa/edit?js,console)

```
var getGuesser = function () {
    var secret = Math.floor(Math.random() * 10 + 1);

    return function (userNumber) {

        if (userNumber === secret) {

            return "Well done!";

        } else {

            return "Unlucky, try again.";

        }
    };
};
var guess = getGuesser();
```

Use Math.random and Math.floor to generate a whole number between 1 and 10 inclusive

Use an if statement to execute commands only if the condition evaluates to true

The return value will be displayed on the console when the function is called.

Include an else clause with commands to be executed if the condition evaluates to false

Call getGuesser and assign the function that's returned to the `guess` variable

Using random numbers has made your guessing game more interesting. But there isn't a great deal of strategy involved; it's just straight guessing. The game could be improved by giving better feedback after each guess.

12.3 Further conditions with else if

By receiving better feedback for each guess, users can develop more efficient strategies when battling your guessing game. And strategy games are always more fun than guessing games. If a user's guess is incorrect, tell them if it's too high or too low.

```
> guess(2)
  Too low!
> guess(7)
  Too high!
> guess(5)
  Well done!
```

Figure 12.3 shows the conditions used to produce the three possible types of feedback for a user's guess.

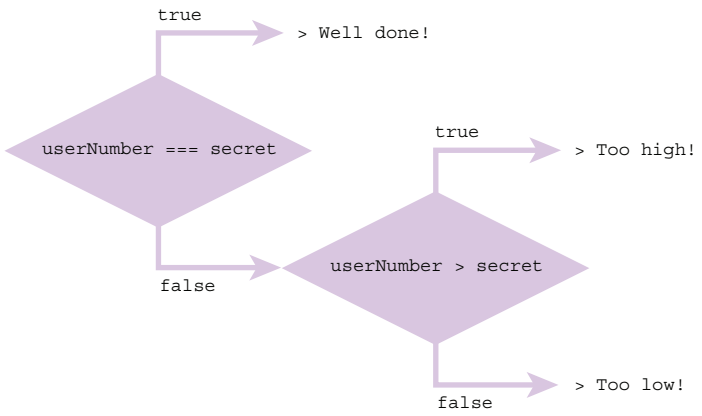

Figure 12.3 Nesting conditions can provide multiple options.

The following listing shows how an extra `if` statement can be used to differentiate between the two types of incorrect answer.

Listing 12.5 Higher or lower (http://jsbin.com/cixeju/edit?js,console)

```
var getGuesser = function () {
    var secret = Math.floor(Math.random() * 10 + 1);

    return function (userNumber) {
        if (userNumber === secret) {
            return "Well done!";
```

Use a condition to check if the user has guessed the secret number

```
        } else {
            if (userNumber > secret) {
                return "Too high!";
            } else {
                return "Too low!";
            }
        }
    };
};

var guess = getGuesser();
```

Execute the first else clause if the guess is incorrect

Check if the user's guess is greater than the secret number

Include code for when the incorrect guess is not greater than the secret number

If a code block contains a single statement, JavaScript lets us leave out the curly braces; the following three statements are equivalent:

```
if (userNumber === secret) {
    return "Well done!";
}

if (userNumber === secret)
    return "Well done!";

if (userName === secret) return "Well done!";
```

As `if` and `else` clauses get more complicated and when code gets updated over time, if you leave out the curly braces it can sometimes be hard to spot which statements go with which clauses. Many programmers (including me) recommend that you always use curly braces for the code blocks (apart from the case of nested `if` statements, as shown shortly). Others aren't so strict. Ultimately, it can come down to personal (or team) preferences. For now, I'd go with whatever you find easiest to understand.

An `if` statement, even with an `else` clause, counts as one statement. When an `else` clause contains a single `if` statement, it's common to leave out the curly braces. The following three code snippets are equivalent:

First, the code as shown in listing 12.5. The nested `if-else` statement is inside a pair of curly braces.

```
else {                              // Curly braces at start
    if (userNumber > secret) {
        return "Too high!";
    } else {
        return "Too low!";
    }
}                                   // Curly braces at end
```

The inner `if-else` is a single statement, so it doesn't need to be wrapped in curly braces.

```
else                                // No curly braces
    if (userNumber > secret) {
        return "Too high!";
```

```
    } else {
        return "Too low!";
    }
                                    // No curly braces
```

And finally, because JavaScript mostly ignores spaces and tab characters, the inner `if-else` statement can be moved to follow on from the initial `else`.

```
else if (userNumber > secret) {      // if moved next to else
    return "Too high!";
} else {
    return "Too low!";
}
```

The last version is the format most commonly seen. The next listing shows the neater `else-if` block in context.

Listing 12.6 A neater else-if block (http://jsbin.com/cidoru/edit?js,console)

```
var getGuesser = function () {
    var secret = Math.floor(Math.random() * 10 + 1);

    return function (userNumber) {
        if (userNumber === secret) {
            return "Well done!";
        } else if (userNumber > secret) {
            return "Too high!";
        } else {
            return "Too low!";
        }
    };
};

var guess = getGuesser();
```

The second `if` statement is shown in bold for comparison with listing 12.5. You've removed the curly braces for the first `else` block and moved the second `if` next to the first `else`. Listing 12.6 shows the most common way of writing `else-if` blocks. If you prefer the longer version in listing 12.5, feel free to stick with it; there are no *Block Judges* waiting to sentence you for syntax abuse. (*At this point the author is called away to deal with a ruckus—some very loud banging on the door to his apartment … it's The Law!*)

All possible outcomes are catered for in the guessing game; the guess could be correct or too high or too low. If the guess is not correct and it's not too high, then it must be too low.

12.3.1 Comparison operators

Listings 12.5 and 12.6 both make use of the *greater than operator*, `>`. It operates on two values and returns `true` or `false`. It's one of a family of operators that compare two values. Some of the operators are shown in table 12.1.

Table 12.1 Comparison operators

Operator	Name	Example	Evaluates to

`>`	Greater than	`5 > 3`
		`3 > 10`
		`7 > 7`
			`true`
			`false`
			`false`

`>=`	Greater than or equal to	`5 >= 3`
		`3 >= 10`
		`7 >= 7`
			`true`
			`false`
			`true`

`<`	Less than	`5 < 3`
		`3 < 10`
		`7 < 7`
			`false`
			`true`
			`false`

`<=`	Less than or equal to	`5 <= 3`
		`3 <= 10`
		`7 <= 7`
			`false`
			`true`
			`true`

`===`	Strictly equal to	`5 === 3`
		`7 === 7`
		`7 === "7"`
			`false`
			`true`
			`false`

`!==`	Not strictly equal to	`5 !== 3`
		`7 !== 7`
		`7 !== "7"`
			`true`
			`false`
			`true`

Because the operators in table 12.1 return `true` or `false`, they can be used in the condition for `if` statements. You may be wondering about the strict part of the strict equality operator—something we'll be sticking to throughout the book—and whether there's a non-strict version. Yes, there is. For non-strict equality you can use `==`. See the "Loose equality and coercion" sidebar.

Loose equality and coercion

The *loose equality operator*, `==`, is allowed to *coerce* values into different types in order to compare them for equality.

Coercion is the process of converting a value from one type to another, for example, from a string to a number.

So, whereas the strict comparison `7 === "7"` evaluates to `false`, because one value is a number and the other is a string, the loose comparison `7 == "7"` evaluates to `true`, because the string is first coerced to a number and `7 == 7` is `true`.

The rules for coercion are beyond the scope of this book (although ignore people who say they're not worth learning), and we'll stick to strict equality comparisons.

Now, obviously, guessing numbers is great fun, but you can learn more from a fact-based quiz like the one you've considered a few times earlier in the book. Adding the ability to check your answers will help raise the quiz app above a mere trivial pursuit.

12.4 Checking answers in the quiz app

Now that you can check conditions in an `if` statement, you're finally able to keep a score for the number of questions a user gets right in the quiz program. A typical console interaction could be this:

```
> quiz.quizMe()
  What is the highest mountain in the world?
> quiz.submit("Everest")
  Correct!
  Your score is 1 out of 1
> quiz.quizMe()
  What is the highest mountain in Scotland?
> quiz.submit("Snowdon")
  No, the answer is Ben Nevis
  You have finished the quiz
  Your score is 1 out of 2
```

The code for the quiz program is shown in the next listing. The `getQuiz` function contains the implementation of the quiz and returns an interface object with only two methods, `quizMe` and `submit`. You take a good look at how the program works after the listing.

Listing 12.7 Checking quiz answers (http://jsbin.com/hidogo/edit?js,console)

```
var getQuiz = function () {
  var score = 0,
      qIndex = 0,
      inPlay = true,
      questions,
      next,
      getQuestion,
      checkAnswer,
      submit;

  questions = [
    {
      question: "What is the highest mountain in the world?",
      answer: "Everest"
    },
    {
      question: "What is the highest mountain in Scotland?",
      answer: "Ben Nevis"
    }
  ];

  getQuestion = function () {
    if (inPlay) {
      return questions[qIndex].question;
    } else {
      return "You have finished the quiz.";
    }
  };
```

Use a single var keyword to declare all the variables in the getQuiz function

Define the getQuestion method to return the current question

```
  next = function () {
    qIndex = qIndex + 1;

    if (qIndex >= questions.length) {
      inPlay = false;
      console.log("You have finished the quiz.");
    }
  };

  checkAnswer = function (userAnswer) {
    if (userAnswer === questions[qIndex].answer) {
      console.log("Correct!");
      score = score + 1;
    } else {
      console.log("No, the answer is " + questions[qIndex].answer);
    }
  };

  submit = function (userAnswer) {
    var message = "You have finished the quiz.";

    if (inPlay) {
      checkAnswer(userAnswer);
      next();
      message = "Your score is " + score + " out of " + qIndex;
    }

    return message;
  };

  return {
    quizMe: getQuestion,
    submit: submit
  };
};

var quiz = getQuiz();
```

Define the next method to move to the next question and check if any questions remain

Define the checkAnswer method to check if the answer is correct and update the score

Define the submit method to handle the user's submitted answer

Return an interface object with two methods, quizMe and submit

Call the getQuiz function and assign the interface object it returns to the quiz variable

Your new quiz program has a number of moving parts; let's break it down into smaller pieces.

12.4.1 Multiple declarations with a single var keyword

Up until now you've been using a `var` keyword for each variable you've declared:

```
var score;
var getQuestion;
var next;
var submit;
```

JavaScript allows you to declare a list of variables with a single `var` keyword. Commas separate the variables, with a semicolon ending the list. The previous declarations can be rewritten in the following shorter form:

```
var score,
    getQuestion,
    next,
    submit;
```

You could even declare the variables on a single line:

```
var score, getQuestion, next, submit;
```

Most programmers prefer one variable per line. You can include assignments too:

```
var score = 0,
    getQuestion,
    next,
    submit = function (userAnswer) {
        // function body
    };
```

The aim is to make sure all variables are declared and the code is easy to read and understand. The style in listing 12.7 is what I tend to prefer; I find it slightly easier to read and it's slightly less typing. Some programmers declare each variable on its own line with a `var` keyword, just as we've been doing in our listings up until now; it's easier to cut and paste variables if each has its own `var` keyword. It's not worth worrying about—you'll probably settle on one style over time.

12.4.2 Displaying a question

The `getQuestion` function returns a question from the `questions` array. It uses the `qIndex` variable to pick the current question-and-answer object from the array. It returns the `question` property of the question-and-answer object.

```
return questions[qIndex].question;
```

But it returns the question only if the quiz is still in progress. Otherwise, it returns a string to say the quiz is finished:

```
return "You have finished the quiz.";
```

The program uses the `inPlay` variable to flag when the quiz is in progress and when it has finished. The `inPlay` variable has a value of `true` while the quiz is in progress and `false` when it has finished. The `getQuestion` function uses the `inPlay` variable as the condition in an `if` statement:

```
if (inPlay) {
  return questions[qIndex].question;
```

```
} else {
  return "You have finished the quiz.";
}
```

When `inPlay` is `true`, the question is returned. When `inPlay` is `false`, the message is returned. (Remember, when you call a function at the console prompt, the console automatically displays the return value.)

12.4.3 Moving to the next question

The program calls the `next` function to move from one question to the next. It moves by incrementing the `qIndex` variable.

```
qIndex = qIndex + 1;
```

The program stores the index of the current element in the `questions` array in `qIndex`. Remember that the array index is zero based, so for an array of length 4 the index could be 0, 1, 2, or 3. An index of 4 would be past the end of the array (3 is the last index). In general, if the index is greater than or equal to the length of the array, you're past the end of the array. All arrays have a `length` property. In the quiz, it represents the number of questions.

The `next` function checks the index to see if it is past the last question:

```
if (qIndex >= questions.length)
```

If the index is past the end of the array, then all the questions have been asked and the quiz is over, so `inPlay` is set to `false`.

```
if (qIndex >= questions.length) {
  inPlay = false;
  console.log("You have finished the quiz.");
}
```

12.4.4 Checking the player's answer

The `checkAnswer` function is straightforward. If the player's submitted answer is equal to the current answer from the questions array, the player's score is incremented. Otherwise, the correct answer is displayed.

```
if (userAnswer === questions[qIndex].answer) {
  console.log("Correct!");
  score = score + 1;
} else {
  console.log("No, the answer is " + questions[qIndex].answer);
}
```

12.4.5 Handling a player's answer

The `submit` function orchestrates what happens when a player submits an answer. It returns either a message with the player's score or a message to say the quiz is over.

```
Your score is 1 out of 2      // If inPlay is true

You have finished the quiz.   // If inPlay is false
```

If the quiz is still in progress, `submit` calls two other functions, `checkAnswer` and `next`. Each will execute its code in turn. You're using the functions to run code on demand.

```
if (inPlay) {
  checkAnswer(userAnswer);
  next();
  message = "Your score is " + score + " out of " + qIndex;
}
```

12.4.6 Returning the interface object

You've kept the interface object returned by `getQuiz` simple. It has no implementation code of its own. You assign its two properties functions from within the local scope of `getQuiz`.

```
  return {
    quizMe: getQuestion,
    submit: submit
  };
```

As discussed in chapter 11, the interface object allows you to maintain a consistent interface over time, even if the implementation within `getQuiz` is changed. The user will always call `quiz.quizMe()` and `quiz.submit()`. You can change which functions are assigned to those two properties of the interface object and how those functions work, but you never remove or rename those properties.

Notice how the program is made up of small pieces working together to build its functionality. As ever, your aim is to make the code readable, understandable, and easy to follow. The `if` statement and its `else` clause help you to direct the flow of the program to take the appropriate actions at each stage.

It's time to put these new ideas to work in *The Crypt*.

12.5 The Crypt—checking user input

In chapter 11, you created a `getGame` function that returns a public interface for *The Crypt*. Players can call a `go` method to move from place to place and a `get` method to pick up items:

```
return {
    go: function (direction) {
        var place = player.getPlace();
        var destination = place.getExit(direction);
        player.setPlace(destination);
```

```
        render();
        return "";
    },

    get: function () {
        var place = player.getPlace();
        var item = place.getLastItem();
        player.addItem(item);
        render();
        return "";
    }
};
```

12.5.1 Step by step through the go method

Let's step through the first three lines of the `go` method. See if you can spot where a problem could arise.

Retrieve the player's location

You start with the `getPlace` method. It returns the player's current location.

```
var place = player.getPlace();
```

You then assign the location to the `place` variable. If the player is currently in the kitchen, then the code is equivalent to

```
var place = kitchen;
```

The program assigned the player's starting location earlier, using the `setPlace` method:

```
player.setPlace(kitchen);
```

Use the direction to find the destination

Now that you have the current place, you can call its `getExit` method to retrieve the destination for a given direction.

```
var destination = place.getExit(direction);
```

When the player calls the `go` method, the argument is assigned to the `direction` parameter.

```
> game.go("south")
```

The previous command will execute code equivalent to the following:

```
var destination = place.getExit("south");
```

If The Library is south of The Kitchen, then the code is equivalent to

```
var destination = library;
```

MOVE THE PLAYER TO THE DESTINATION

You have the destination; you only need to update the player's location.

```
player.setPlace(destination);
```

Fantastic! The user can decide where to go in the game. So, can you let them loose in your carefully crafted castles? Well, no. You see, users are evil. Pure evil!

12.5.2 Never trust user input

Sorry, I panicked. Of course users aren't evil. But they do make mistakes. And they sometimes own cats. And most cats can't type. Whenever a user is expected to provide input for a program, we must guard against mistakes, whether typos (possibly of cat origin), misunderstandings (which may be our fault), or curiosity-driven explorations of what the program can do.

The `go` method expects the user to enter a valid direction as a string. It uses that direction to find a destination, the place to which the player is to be moved. If the user enters a direction that doesn't exist, the whole game breaks!

```
> game.go("snarf")
```

Figure 12.4 shows what happened on JS Bin when I entered the previous command while playing the chapter 11 version of *The Crypt* at http://jsbin.com/yuporu/edit?js,console.

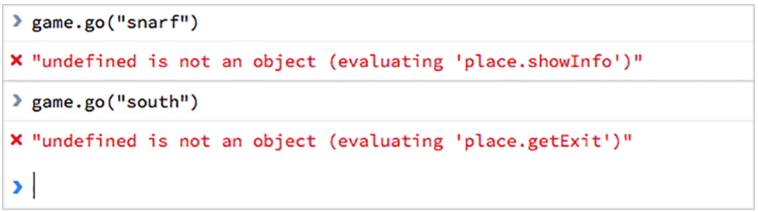

Figure 12.4 Specifying a direction that doesn't exist breaks *The Crypt*.

Error messages on your browser might be slightly different. Even entering a valid direction after the mistake doesn't fix things. From the errors in figure 12.4 it looks like there may be a problem with the `place` variable. The key statement in the `go` method is the one that uses the user input:

```
var destination = place.getExit(direction);
```

If the specified direction is not one of the place's exits, then the `getExit` function will return `undefined`. The program assigns `undefined` to the `destination` variable and sets that value as the new place for the player:

```
player.setPlace(destination);
```

So the player's location is now `undefined`, not a place constructed with the `Place` constructor. `undefined` has no `showInfo` or `getExit` methods; it has no methods at all! The errors in figure 12.4 should now make more sense.

So how can you guard against users (and their cats) making errors?

12.5.3 Safe exploration—using the if statement to avoid problems

You can use an `if` statement to check that you have a valid destination before updating the player's location:

```
go: function (direction) {
    var place = player.getPlace();
    var destination = place.getExit(direction);

    if (destination !== undefined) {
        player.setPlace(destination);
        render();
        return "";
    } else {
        return "*** There is no exit in that direction ***";
    }
}
```

The `getExit` method returns `undefined` if the current place doesn't have an exit for the specified direction. You just need to check that the destination is not `undefined` before calling `setPlace`.

```
if (destination !== undefined) {
    // There is a valid destination.
}
```

Remember from table 12.1 that the `!==` operator returns `true` when two values are *not* equal and `false` when they *are* equal. You can add an `else` clause to catch the cases where the destination *is* `undefined`.

```
if (destination !== undefined) {
    // There is a valid destination.
} else {
    // There is no exit in the direction specified.
}
```

Listing 12.8 shows an updated version of the `go` and `get` methods returned from the `getGame` function. Entering a nonexistent direction at the console now looks like this:

```
> game.go("snarf")
  *** You can't go in that direction ***
```

Calling `get` when there are no items to pick up looks like this:

```
> game.get()
  *** There is no item to get ***
```

Only partial code is shown in this listing. The full listing with `Player` and `Place` constructors and more places is on JS Bin.

Listing 12.8 Checking user input (http://jsbin.com/zoruxu/edit?js,console)

```
var getGame = function () {
  var spacer = { … };
  var Player = function (name, health) { … };
  var Place = function (title, description) { … };
  var render = function () { … };

  var kitchen = new Place("The Kitchen", "You are in a kitchen…");
  var library = new Place("The Old Library", "You are in a library…");

  kitchen.addExit("south", library);
  library.addExit("north", kitchen);

  // Game initialization
  var player = new Player("Kandra", 50);
  player.addItem("The Sword of Doom");
  player.setPlace(kitchen);

  render();

  return {
    go: function (direction) {
      var place = player.getPlace();
      var destination = place.getExit(direction);

      if (destination !== undefined) {

        player.setPlace(destination);
        render();
        return "";

      } else {

        return "*** You can't go in that direction ***";

      }
    },

    get: function () {
        var place = player.getPlace();
        var item = place.getLastItem();

        if (item !== undefined) {
            player.addItem(item);
            render();
            return "";
        } else {
            return "*** There is no item to get ***";
        }
    }
  };
};

var game = getGame();
```

Return an interface object with go and get methods

Use getExit to find the destination for the direction specified by the user

Check that the destination is not undefined; that is, check that it is valid

Only set and show the new place if the destination exists

Add an else clause to handle when the destination is undefined

Give the user feedback about the invalid direction specified

Update the get method to make similar checks to the go method

In the printed listing 12.8, the details of the `Player` and `Place` constructors were left out to make it easier to focus on the changes to the `go` and `get` methods. In chapter 13 you'll move each constructor function to its own file and see how to import the files in JS Bin. Such increased modularity can help you focus on one thing at a time and make it easier to reuse code across multiple projects.

12.6 Summary

- Use comparison operators to compare two values. The operators return `true` or `false`, boolean values:

```
>  5 === 5    // Strict equality
   true
>  10 > 13    // Greater than
   false
```

- Use an `if` statement to execute code only if a condition is met:

```
if (condition) {
    // Execute code if condition evaluates to true
}
```

- Set conditions using comparison operators and/or variables:

```
if (userNumber === secret) {
    // Execute code if userNumber and secret are equal
}

if (inPlay) {
    // Execute code if inPlay evaluates to true
}
```

- Add an `else` clause to execute code when a condition is not met:

```
if (condition) {
    // Execute code if condition evaluates to true
} else {
    // Execute code if condition evaluates to false
}
```

- Include extra `if` statements in the `else` clause to cover all possibilities:

```
if (userNumber === secret) {
    console.log("Well done!");
} else if (userNumber > secret) {
    console.log("Too high!");
} else {
    console.log("Too low!");
}
```

- Generate random numbers with `Math.random()`. The numbers generated are between 0 and 1. They can equal 0 but not 1:

```
> Math.random()
  0.552000432042405
```

- Scale up the random numbers to the range you want:

```
Math.random() * 10                    // 0 <= decimal < 10
Math.random() * 10 + 1                // 1 <= decimal < 11
```

- Round the random numbers to integers with `Math.floor()`:

```
Math.floor(Math.random() * 10 + 1)    // 1 <= integer <= 10
```

- Never trust user input. Put checks in place to make sure any input is valid.

JAVASCRIPT / PROGRAMMING

Get Programming with JavaScript

John R. Larsen Foreword by Remy Sharp

Are you ready to start writing your own web apps, games, and programs? You're in the right place! **Get Programming with JavaScript** is a hands-on introduction to programming for readers who have never written a line of code.

Since you're just getting started, this friendly book offers you lots of examples backed by careful explanations. As you go along, you'll find exercises to check your understanding and plenty of opportunities to practice your new skills. You don't need anything special to follow the examples—just the text editor and web browser already installed on your computer. We even give you links to working online code so you can see how everything should look live on your screen.

WHAT'S INSIDE

- All the basics—objects, functions, responding to users, and more
- Think like a coder and design your own programs
- Create a text-based adventure game
- Enhance web pages with JavaScript
- Run your programs in a web browser

No experience required! All you need is a web browser and an internet connection.

John Larsen is a web developer and professional teacher in the UK who has many years of experience working with students of all levels, helping them to successfully write their first lines of code.

"Provides the guidance you need to get started ..., the support to keep practicing, and the encouragement to enjoy the adventure."
—From the Foreword by Remy Sharp
Founder of JS Bin

"A great book for the new programmer who wants to learn JavaScript."
—Alvin Raj, Oracle

"An approachable and interactive way of learning JavaScript."
—Giselle Stidston, Breville Pty Ltd

"Great interactive code examples! Building a computer game was my favorite part of the book."
—Ivan Rubelj, Vipnet

To download their free eBook in PDF, ePub, and Kindle formats, owners of this book should visit manning.com/books/get-programming-with-javascript

ISBN-13: 978-1-61729-310-8
ISBN-10: 1-61729-310-5
53999
9 781617 293108

MANNING US $ 39.99 | Can $45.99