Glossary

Welcome to the glossary. I hope you find it useful. Some of the definitions have attached code samples that you can find in the sample code for the book in the glossary folder.

Application

An application performs something a user wants to do, whether playing a game or writing a book. Originally, an application was a single piece of software you copied onto your device. However, today, you can also use applications from within your web browser. The browser creates a sandbox, a safe environment for an application to run in. The sandbox provides all the services that the application requires and strictly controls access to the underlying computer. The application itself might comprise multiple cooperating pieces of software that work together to provide the user experience. And the application might use cloud components to do this.

Application Programming Interface (API)

A set of functions or methods is exposed by a piece of software to allow other software to use its facilities. The facilities could be a single action (for example, a mathematical calculation such as the sine of a value) or a more complex transaction (or, for example, loading a text file).

Argument

An argument is a value given to a function call. Within the function, the matching parameter value is replaced by the argument supplied in the call.

```
doAddition (3,4);
```

In the statement above, the arguments are the values `3` and `4`, which are passed into the `doAddition` function.

```
function doAddition(p1, p2) {
    let result = p1 + p2;
    alert("Result:" + result);
}
```

When the function is called, each argument is mapped onto the corresponding parameter in the function body. If an argument is not supplied, the value of the matching parameter will be set to `undefined`. Excess arguments are ignored if a call has more arguments than the receiving function has parameters.

Assign

The assignment operator (`=`) takes the result of an expression and assigns this to a variable.

```
let age = 21;
```

The assignment above assigns the value of `21` to the newly created variable `age`.

Asynchronous

There are two ways you can get your car serviced. You can drive to the garage, hand over the keys, sit in the reception area, and wait until the service is complete. When the car is finished, you drive it back home. This is the synchronous approach to car maintenance. You must wait until the car is ready before getting on with your life. You are synchronized to the speed at which the garage works.

The alternative is to use an asynchronous approach, where you drop off the car at the garage and take a taxi home. The garage calls you when the car is ready, and you get another taxi back to the garage and pick the car up. In this case, you can get on with your life while the car is being serviced.

JavaScript was built for the asynchronous approach to life. At no point should we find a JavaScript program waiting for something. A JavaScript program will not ask the file system to open a file and then hang around, waiting for the file to open. A JavaScript program will ask to open the file and provide a function to be called when the file is ready. That way, the program can do something else while the file system fetches the file from storage. The ease with which you can create and deploy anonymous functions in JavaScript makes it a natural candidate for asynchronous working.

Note that asynchronous operation is not without its complications. If the operating system can't find a file, there needs to be a way that a request can generate events that mean, "I couldn't find this," to which the program asking for the file can respond.

JavaScript provides the `Promise` to help manage asynchronous operations. A `Promise` represents an asynchronous task that is being performed.

Attribute

HTML pages contain elements that describe things to be rendered by the browser. Each type of element is associated with a particular set of attributes. An attribute adds extra information to an element. The HTML below uses the `button` element to create an on-screen button for the user to click. The button contains the text "Throw dice" and has an `id` attribute set to `dicebutton`.

```
<button id="diceButton">Throw dice</button>
```

The `id` attribute can be used in a JavaScript program to locate an element in a document. The `getElementById` method provided by the Document Object Model (DOM) can search for an element in the document with a particular `id` attribute. The statement below would create a variable called `diceButton`, which refers to the DOM object representing the button.

```
let diceButton = document.getElementById("diceButton");
```

A JavaScript program can create new attributes for an element using the `setAttribute` function. The statements above use the `id` attribute to locate a button and then add a `name` attribute to the button: `"Rob"`.

```
diceButton.setAttribute("name", "Rob");
```

Note that the button contains some HTML (in this case, the "`Throw dice`" string). This is not a property of the button element, though it can be accessed in JavaScript using the `innerHTML` property of the element. The statement below would set the `innerHTML` for the dice button to `"Please click to throw the dice"`:

```
let diceButton = document.getElementById("diceButton");
diceButton.innerHTML = "Please click to throw the dice"
```

Note that this is a potential security risk. If the `innerHTML` of an element is entered into the webpage, an attacker could enter malicious HTML into the document. The following code shows how this could happen. The function `doReadName` reads a name from an input element and then displays the name in a paragraph called `helloPar` by setting the `innerHTML` of that paragraph. If visitors to the web page enter HTML code as their name, they can cause new HTML elements to appear on the page. To set the text on an element on a page, you should use the `innerText` property instead.

```
function doReadName() {
    let nameInput = document.getElementById("nameInput");
    let name = nameInput.value;
    console.log(name);
    let helloPar = document.getElementById("helloPar");
    helloPar.innerHTML = name;
}
```

I've provided a sample application you can use to experiment with this in the **Attributes** folder in the glossary samples at *https://begintocodecloud.com/glossary.html*. See *https://developer.mozilla.org/en-US/docs/Web/API/Element/innerHTML* for more details on `innerHTML`.

Block

A block of code in JavaScript is a number of statements that have been lumped together and treated as a whole. You create a block by enclosing statements in braces (curly brackets). You put code into a block so you can use a single conditional statement to control several actions.

The following code logs a message if the age is greater than `70`. It also sets the value of `age` to `70`. Both statements are controlled by the condition testing the value of `age`. You also put code into a block when you create the function body.

```
if ( age>70 ) {
    console.log("Age too large. Using 70");
    age = 70;
}
```

Class

There are two kinds of classes. The first is the one you find on an HTML page. This class is applied to an element to make it pick up a particular style. The statement below is from the clock program we wrote in Chapter 2.

```
<p id="timePar" class="clock">0:0:0</p>
```

It says the element displaying the clock value has been assigned the `clock` class. The `clock` class is defined in a stylesheet, which makes the clock text large and centered on the page.

```
.clock {
  font-size: 10em;
  font-family: 'Courier New', Courier, monospace;
  text-align: center;
}
```

Classes are very useful for separating the design of a page from the elements themselves. A given element can be assigned to multiple classes. Container elements can be assigned a class that will be applied to all items in the container, forming the basis of cascading stylesheets (CSS). (Note, class settings applied to individual items will override those in parent classes.)

The second kind of class appears in JavaScript code, bringing together data and behaviors to create an object. The following code creates a class called `Vehicle`, which we could use to store the color and make of a vehicle. The `Vehicle` class contains two methods—a `constructor` and a method called `logDetails`. These methods work with two properties stored in the class: `color` and `make`.

```
class Vehicle {
  constructor (newValue){
    this.color = newValue.color;
    this.make = newValue.make;
  }
  logDetails(){
    console.log("Color is:"+this.color+" make is:"+this.make);
  }
}
```

The `Vehicle` class is a design for an object. When we create a new class instance, we only get the object. In the context of the methods inside the `Vehicle` class, the keyword `this` is resolved as a reference to the object being manipulated by the method:

- Within the `constructor` method, `this` contains a reference to the object being created.
- Within the `logDetails` method, `this` contains a reference to the object on which the method is being called.

The following code creates two instances of `Vehicle`—referred to by the `v1` and `v2` references—and then logs their details.

```
let v1 = new Vehicle({color:"white",make:"Nissan"});
let v2 = new Vehicle({color:"blue",make:"Ford"});
v1.logDetails();
v2.logDetails();
```

In the first `logDetails` call, the value of `this` will be a reference to `v1`. In the second call, it is set to `v2`. So, the first call will output `"Color is:white make is:Nissan"`. If you think about it, the `this` reference enables what you want to happen when the methods run. We want the first `logDetails` call to output the contents of `v1`.

Classes can contain many methods and data properties. Note that the function keyword does not precede a method declaration in a class. Classes can be used to create class hierarchies, where a class is extended by a child class that adds behaviors and properties based on the parent.

Cloud

The term "The Cloud" refers to a network of remote servers hosted on the Internet and used for storing, managing, and processing data. These servers are typically owned and operated by third-party providers who allow customers to run applications or store and share data.

Closure

Closures can be used to create private variables in a JavaScript program. Consider the function below. The function contains a single variable that contains a string message:

```
function outerFunc(){
  let outerVar = "I'm a var in the outer function";
}
```

If the function is called, the variable `outerVar` is created, and then, because it is declared using `let`, it is deleted when the function completes. Now, let's put a function inside the outer function and make the outer function return that function:

```
 function outerFunc(){
  let outerVar = "I'm a var in the outer function";
  function innerFunc(){
    console.log(outerVar);
  }
  return innerFunc;
}
```

We can declare functions inside functions, and a function running inside a function can access all the variables declared in the enclosing function. So, `innerFunc` can log the value of `outerVar`. Now, let's call `outerFunc` and assign the result to a variable. The statement below creates a variable called `funcRef` that refers to the result of calling `outerFunc`:

```
let funcRef = outerFunc();
```

So, if you are still following this, `funcRef` refers to `innerFunc`. If I call `funcRef`, it will log the value of `outerVar` on the console. This is the part of the process we can call the "closure." The value of `outerVar` has been retained. Normally, JavaScript destroys variables that go out of scope, but in this case, the compiler has determined that the value of `outerVar` is still required, so it has been retained. The statement below calls the function to which `funcRef` refers (`innerFunc`), then displays, `"I'm a var in the outer function"` on the console:

```
funcRef ();
```

This can be a bit confusing but very powerful. The `outerVar` variable is not visible to any other part of the program. It is truly private. Let's see how we could use this. Suppose we want a counter that only counts up. We want to be sure that there is no way that a naughty programmer could change the counter value. The function `countUp`—shown below—will do this for us. The `countUp` function contains a local `count` value, which is incremented and returned by the `nextCount` function declared inside `countUp`. This forms a closure.

```
function countUp(){
    let count = 0;
    function nextCount(){
        count = count + 1;
        return count;
    }
    return nextCount;
}
```

Now, I can call `countUp` to make me a counting behavior, as shown below:

```
let myCounter = countUp();
console.log(myCounter());
```

The `myCounter` variable refers to the `nextCount` function, which increments the count value and then returns it. The first time `myCounter` is called, it will return `1`, then `2`, and so on. There is no way for this count to be reset. Note that if I make a second counter, as shown below, it will have its own copy of the `count` variable. In other words, each `countUp` call makes a new local variable that is then implemented as a closure.

```
let newCounter = countUp();
console.log(newCounter());
```

Code

The word "code" is generally thought to mean the same as "program." However, the original meaning of "code" was the very low-level instructions the computer hardware runs to implement a program. A program might contain a statement that adds 1 to the value of a variable. The code for this might end up being a sequence of instructions to fetch the variable, add 1, and then store it back into random access memory (RAM). The phrase "machine code" refers to the code implemented by a particular type of hardware.

Condition

A condition is used to control the flow of the program. The condition is controlled by a value that can be either "truthy" or "falsey." Some conditions are obvious. The code below checks the `age` value, capping it at `70`:

```
if ( age>70 ) {
    console.log("Age too large. Using 70");
    age = 70;
}
```

Some conditions are less obvious. JavaScript has "built-in" values to represent values such as `Infinity`, `undefined`, `NaN` (not a number), and objects such as an empty array or a `null` reference. Which of these are truthy and which are falsey? The answer is that everything is truthy except

- `false`
- The value `0`
- An empty string
- A `null` reference

- `undefined`
- `NaN`

Console

This word has two general meanings. The first refers to the act of trying to make someone feel better by expressing sorrow, such as, "I'm sorry that your program doesn't work properly." The second refers to a user interface provided by a keyboard and screen. Generally, you type commands into the user interface and get responses. However, there might be some tools, such as text editors, where you use the arrow keys to move the cursor around the screen and interact with the display. The environment in which you type your console commands is called a "shell." Windows PowerShell is a cross-platform shell that provides a comprehensive set of commands and the ability to create and run scripts to automate operations. You can find out more at *https://learn.microsoft.com/en-us/powershell*.

Context

Everything we do takes place in a particular context. You behave one way in a church and another in a football stadium. JavaScript establishes the context of an operation by looking at the type of operands it is working on. For example, the `+` operator will perform numeric addition if applied to two numbers but concatenate if applied to two strings.

Compiler

A compiler is a program that takes in program text and converts it into machine code that can run directly on a computer. The compilation process takes place before the program runs. Actually, this is a bit of an oversimplification. Sometimes, a compiler doesn't make machine code; sometimes, it makes intermediate code, which is performed or compiled when the code runs. However, all languages have a compilation process that happens before a program is allowed to run.

Different programming languages have compilers that apply different levels of rigor to the program code they are looking at. The JavaScript compiler is on the relaxed side. It will accept programs that contain syntax discrepancies that would be rejected in other languages. This can result in a JavaScript program running without error but delivering incorrect results, so we must be mindful of this.

Computer

A computer is a collection of hardware that can run programs. On its own, a computer is of no use to you. It must have software to make it useful. Some computers run only one program. Others allow the user to select programs and add new ones. A computer is frequently part of another device, such as a mobile phone. Some devices also contain computers running "embedded" software to make them work, such as a remote-controlled light bulb.

Const

JavaScript variables can be declared using the `const` keyword, as shown below:

```
const maxAge = 70;
```

A variable declared as `const` must be given an initial value, which cannot be changed. You can protect variables from being changed by declaring them as `const`.

Cursor

In a console, a graphical cursor element—perhaps an underline or a block, which might or might not flash—indicates where text will be entered. In a windowed environment, such as on a Windows or macOS computer, it might appear as a mouse pointer. (Also, a cursor is someone who has just discovered that their program doesn't do what they want it to.)

Data

Data is the stuff that computers work with. It ends up as patterns of bits that are combined to make other patterns of bits that might be presented to humans who decide whether it is information (something that means something).

Declaration

Declaration of a variable tells JavaScript that the variable exists.

```
let x;
```

The above statement tells JavaScript of the existence of an `x` variable. It doesn't give any information about the type of `x` or its initial contents. A declared variable contains the value of `undefined`. You might be wondering why we declare variables like this when JavaScript will automatically declare variables for us. We declare a variable because we want to set the `scope` of the declared variable.

Define

When we define a variable, we tell JavaScript all about it.

```
let age = 21;
```

The statement above defines a variable called `age`. JavaScript can infer that the `age` variable is a number and that the initial value of the variable will be `21`.

Delimiter

A delimiter is something that defines the limits of something. In English, a capital letter and a full stop are used as sentence delimiters. JavaScript programs and HTML documents use delimiters in several different contexts. Blocks of JavaScript code are delimited by braces: `{` and `}`. Strings of text are delimited by single (`'`), double (`"`), or back (`'`) quotes. HTML elements are delimited by their names, such as `<par>` and `</par>`.

Document Object Model (DOM)

You can think of program input and output as a program reading what you typed in and then printing a result. This is how a console application works. It is also how we started using computers once we had got rid of punched cards.

A browser-based application does not perform input and output in this way. Instead, the browser uses the HTML from the web page to build a Document Object Model (DOM) that represents the page. The model contains a set of linked software objects representing the page's elements. A program can obtain inputs from element properties set by the browser. If the user types something into an input element, the element's `value` property will contain what was typed. A program can display output by changing and adding properties to the objects.

The browser renders these changes to change the page's appearance. Objects in the DOM can also generate events, such as when a web page button is clicked. A JavaScript function can be bound to an event so that when the event occurs, the function runs. This allows pages to be responsive to user input.

Element

HTML pages can contain elements. Each element has a particular type (for example, paragraph or heading) and a particular attribute set. An element can contain other elements to allow nesting.

Event

Some software components generate events. An event is a trigger that causes code to run. A JavaScript program will "handle" an event by assigning a function to run when the event occurs. The function can be a named `function`, or it can be an arrow function.

Exception

An exception is an object that describes something bad that has just happened. Exceptions are used in `try-catch` constructions. See Try–Catch, later in this glossary, for more details.

Exploit

An exploit uses knowledge of the internal workings of software to get the software to do things it is not supposed to do. For example, you might discover that entering an empty username and password causes a badly written authentication system to allow you to log in. We can reduce the chance of exploits by properly testing and validating our code.

Function

A function is an object containing a "body" made up of JavaScript statements performed when the function is called. A function also contains a name property that gives the name of that function. Functions can be called by name, at which point, the program execution is transferred into the statements that make up the function body. When the function completes, the execution returns to the statement after the one that called the function. Functions can accept values to work on, and a function can return a result value. The `doAddition` function below accepts two parameters and displays an alert showing the result of adding their values.

```
function doAddition(p1, p2) {
    let result = p1 + p2;
    alert("Result:" + result);
}
```

Functions reduce program size. Rather than repeat a sequence of statements each time a particular behavior is required, you can create a function to be called each time the behavior is needed. Functions make programs easier to maintain because a fault in a function only needs to be fixed once. Functions can also make programs easier to understand and work on. A task can be performed by several different functions. Each function can be written and tested independently of the others, and the functions can be combined to make the finished solution.

Global

The global context exists outside all others. Items declared at the global level will be visible (can be viewed and—more importantly—changed) by any code in the system. Global status should be reserved for data values that absolutely need to be visible to all.

Glossary

Thanks for looking up the word *glossary*. A glossary is a kind of "dictionary for books." A dictionary contains definitions of words. A glossary contains definitions of words for a particular context—in our case, learning how to create cloud-based applications.

Hash

Hashing is widely used during Internet transactions to validate data before it is used. We first saw hash functions in Chapter 8 when we started using the Bootstrap stylesheet. Below is the HTML used to add the Bootstrap stylesheet files to a web page:

```
<link rel="stylesheet"
href=https://cdn.jsdelivr.net/npm/bootstrap@4.3.1/dist/css/bootstrap.min.css
integrity="sha384-ggOyR0iXCbMQv3Xipma34MD+dH/1fQ784/j6cY/iJTQUOhcWr7x9JvoRxT2MZw1T">
```

The `integrity` attribute of the link is the hash code for the stylesheet file. It is there so that the browser can ensure that the stylesheet's contents have not been tampered with. The value of the integrity attribute has been calculated using SHA-3 (Secure Hash Algorithm 3). This takes the file's content and produces the random-looking sequence of characters you can see above. The "hash algorithm" has been designed to produce a unique output for a particular input. The browser uses the SHA-3 algorithm to calculate the hash value of the stylesheet file it receives from the Internet and compares it with the one given in the HTML. If the two values are the same, the stylesheet is valid. Any tampering with the contents of the stylesheet would cause it to have a different "hash code."

Hypertext Transfer Protocol (HTTP)

HTTP is a text-based protocol used by browsers. When a web address is entered into a browser, the browser sends a message starting with `GET`. When a browser wants to send a message to the server (perhaps the contents of a form the user has completed), it sends a message starting with `POST`. HTTP defines headers that add extra detail to the message. In a GET request, the header indicates whether the page was found correctly and the data format of the response. If we wanted to create a web server from scratch, we would have to learn the format of all the messages and responses. Fortunately, there are libraries for JavaScript and `node.js` that can be used to assemble and respond to HTTP-formatted messages.

Identifier

An identifier is a name created by the programmer to refer to something a program needs to keep track of. JavaScript has rules that determine how the identifier is constructed:

- The identifier must start with a letter (`A-Z` or `a-z`), a dollar sign (`$`), or an underscore (`_`) character.
- The identifier can contain letters, digits (`0-9`), a dollar sign (`$`), or an underscore (`_`) character.

It is best if your identifiers describe the thing they are connected to. An identifier called `a` is not very useful, but one called `age` would be more useful, and one called `ageInYears` would be very useful. If you want to use multiple words in an identifier, the JavaScript convention is to capitalize the first letter of each word inside the identifier. This is called “camel case” because the capital letters stick up like the humps on a camel’s back. Note that the identifier’s letter case is significant. JavaScript would regard `AgeInYears` and `ageInYears` as different identifiers. If a program fails with a complaint that it can’t find something, make sure that the identifier you are using is correct.

If you are choosing an identifier for a function or method, using a *verb-noun* structure is a good idea. The `displayMenu` identifier would work well for a function that displayed a menu.

If you choose a class name, the identifier should start with a capital letter.

Infinity

The range of JavaScript numbers includes the value of `Infinity`. A variable will be set to `Infinity` if the calculation results generate that value.

```
var x = 1/0;
```

The above statement sets the value of `x` to `Infinity`. If the value of `x` is printed, it will display the `Infinity` value. The implementation of `Infinity` works how you might expect:

- If you add 1 to `Infinity`, you get `Infinity`.
- If you divide `Infinity` by any value, the quotient retains the value of `Infinity`.
- If you divide any number by `Infinity` you get the result of `0`.
- If you divide `Infinity` by `Infinity`, you receive `NaN` (Not a Number).
- A program can set a variable to the value of `Infinity` value and can test for the value:

```
if(x==Infinity) console.log("result too large");
```

Internet

The Internet is based on a set of open standards called TCP/IP (Transport Control Protocol/Internet Protocol). TCP/IP was created by the Defense Advanced Research Projects Agency (DARPA), a United States Department of Defense division. TCP/IP sets out how to build and connect local networks to create a "network of networks" that can span the globe. The standards provide an addressing scheme for each physical device on the network and a resource naming scheme allowing users to identify services by names rather than their physical addresses. The Internet is not the only network that uses TCP/IP. You can create your own TCP/IP network in your bedroom if you like.

You might think of an Internet connection as a cable between two machines. However, this is not how it works. The sender breaks the data into small packets, which are sent individually. It is a bit like me sending you a loaf of bread by mailing each slice in a different letter. Software in the receiver takes the packets, puts them in the correct order, and passes them up to the application using the connection. The application using the connection (for example, a browser loading a web page) is completely unaware of all this packing and unpacking. It just receives the data and does something with it.

The Internet was designed to provide communications at a time when nuclear war looked distinctly possible. It uses a mesh of systems called "servers," each of which maintains a "map" of the network and passes incoming packets of data across the network to its destination. If one of the servers suddenly becomes unavailable, the servers automatically route packets around it. Client machines connect to the servers to send and receive packets of data.

The Department of Defense decided that the best way to get lots of network connectivity worldwide was to make the TCP/IP standard public and give away all the software they wrote to make it work. This made it much easier for hardware manufacturers (and even hobbyists) to connect their machines. The Internet became very popular very quickly. It provides something that was revolutionary at the time it was introduced. The Internet transfers packets of data anywhere in the world at the same cost—which is zero once you are connected. This was a genuine game-changer for computing. Before the Internet, you had to lay your cables or rent them from the telephone company if you wanted to connect computers. And international communications were extremely expensive.

However, once you have connected a machine to the Internet, you can send packets and expect them to arrive at their destination irrespective of where they're going. A packet might take longer to arrive at a more distant destination as it is passed from system to system on its journey, but longer journeys don't cost more. Two successive packets to the same destination might travel by completely different routes, but they both would get there. The Internet's job is to hide all this complexity and give users the impression that they are directly connected to a machine, even though it is on the other side of the planet. The Internet can function across many forms of physical media, including telephone lines, wired connections, wireless networks, and mobile phones.

You can think of the Internet as a system of rails that connect places, but just like a railway, the Internet is not interesting until you start to move things around on it. Just as trains make a railway interesting, applications make the Internet interesting. Electronic mail (email) was one of the first "killer applications" for the Internet. A user connects to their Internet-connected mail server, which accepts messages and stores them for the user to read. The mail server also sends mail messages to other mail servers. However, the application that did the most to get users onto the Internet was the World Wide Web.

IP address

Every computer on the Internet has a unique address. This is called an Internet Protocol or IP address. You can think of it as the computer's phone number. When you want to call someone on your phone, you must enter their number. When a program wants to call a program on a distant machine, it uses the IP address of that machine. Of course, you hardly ever enter a phone number in real life. The actual numbers are stored behind names in an address book. The Internet does something similar, too, using the Domain Name System (DNS) process that converts names (for example, *robmiles.com*) into IP addresses. If you point your browser at *www.robmiles.com*, the browser will first use DNS to discover the server's IP address that hosts my blog and then send HTTP requests to that server.

Most of the addressing on the Internet uses 32-bit integers to hold IP values. This addressing scheme is called IPv4 and allows for over 4 billion different addresses, which seemed plenty when it was introduced. However, the number of connected devices has increased to the point that we are running out of addresses. A more advanced addressing scheme, IPv6, is being introduced that uses 128-bit integers, allowing for many more connected devices. IPv4 and IPv6 are designed to coexist and will both be around for a while.

An IPv4 address is expressed as four eight-bit values, separated by dots—for example, 158.252.73.252. You can find out the IP address of your machine by searching for "IP address" in your browser.

JSON

JavaScript Object Notation, or JSON, is a textual description of the data contents of a JavaScript object. A JSON document can contain arrays and objects as a series of named value pairs. JSON documents cannot contain references to objects; instead, the contents of the referred object are inserted in the place of the reference. JSON documents can contain numeric values, text strings, and true or false. Many languages contain native support for JSON, which means it can be used to transfer data between different devices and programs. Visit *https://www.json.org/json-en.html* for a definition of the standard.

Let

Programs use `variables` to hold values that the program wants to work on. A program can declare a variable by using the keyword `let`. The useful thing about variables declared using `let` is that they are discarded when program execution leaves the block in which they are declared. We say that such variables have "block scope."

```
{
    let personName = "Rob";
    // do things with the variable personName
}
// at this point in the code the variable personName no longer exists
```

Consider the code above. The `personName` variable is declared inside a block of statements. The `personName` variable can be used within that block, but when the program reaches the end of the statements in the block and leaves it, the `personName` variable is discarded. This means there is no chance of the `name` variable being confused with other variables called `personName`, which might be used in other parts of the program.

Note that if the enclosing block contains a variable called `personName` (it would be declared outside the block shown above), the outer `personName` variable would not be accessible inside the block. But the outer `personName` variable would become accessible when the program leaves the block above.

Once you have learned about `let`, you should look at the glossary entries for `var` and `global`, which are also used in JavaScript to manage where variables can be used.

Local

A local variable has a restricted `scope`. A local variable declared using `let` is local to a particular block of statements. A local variable declared using `var` is local to the `function` or `method` in which it was declared.

Localhost

A device connected to the Internet has an IP address used to locate it. A program will enter the IP address of that machine to send messages to it. A program can also send messages to a localhost address, which is the address of the machine itself. We use the localhost address to test a network service on a machine. In other words, I can run a program that implements a web server on my machine and then put the localhost address into the browser to connect to that server. The IPV4 (see the IP address entry earlier in this glossary for more) localhost address is 127.0.0.1. You can also use localhost as this address. If you wanted to host a website on your computer and then use your computer's browser to connect to that website, you would use the localhost address.

Logical

This term is used in two ways in computing. First, computer programs use logical expressions to make decisions. For example, if a program needed to express, "If you are less than five years old, you can't go on this fairground ride," it would use a logical expression, perhaps using the `<` logical operator to make the decision. In this context, we are talking about elements of a programming environment that allow us to express how decisions are made.

Secondly, the term *logical* also means a way of viewing things. For example, we talk about networks that use "logical" addresses. In this context, *logical* means a thing that might or might not map to a physical device. For example, a logical address might be mapped onto a physical computer. However, a logical address might also be mapped to a process running on a computer that supports many such processes. The idea of "logical" entities is a big part of how we can talk about components in the "cloud." We give the component a logical address, and then the underlying system can determine the actual location when it is used.

Machine code

The computer's processor processes instructions encoded as "machine code," expressing the actions to be performed in a very low-level form the hardware can understand. Machine code programs have a structure and content specific to a particular hardware architecture. Programs written in higher-level languages (such as C++) must be converted into machine code before running on a device. JavaScript programs are not usually converted into machine code before being run. Instead, an interpreter program runs on the hardware. The interpreter works out what each JavaScript statement does and then performs it. Of course, life is not quite as simple as this. In reality, a JavaScript interpreter might convert some JavaScript code into machine code just before it runs it (this is called Just-in-Time compilation).

Metadata

Metadata is "data about data." The metadata for a photograph file might contain the date and time the picture was taken, the type of camera used, and so on. If we represent the photograph as a JavaScript object, each metadata item would be an object property.

Method

Methods are members of a class and are used in exactly the same way as functions. They can accept parameters and can return results. Within a method, the keyword `this` behaves as a reference to the object the method is running within. Below, you can see the definition of a class called `Vehicle`. This class contains two methods: `constructor` and `logDetails`.

```
class Vehicle {
  constructor (newValue){
    this.color = newValue.color;
    this.make = newValue.make;
  }
  logDetails(){
    console.log("Color is:"+this.color+" make is:"+this.make);
  }
}
```

If we create a class instance, we get an object with behaviors provided by the methods in it. The `constructor` method is used to create an instance of the class. When the statement below is performed, the `Vehicle` constructor is called with an argument, which is an object literal containing the information to initialize the object.

```
let v1 = new Vehicle({color:"white",make:"Nissan"});
```

The `logDetails` method can be called to display the data held in an instance of `Vehicle`.

```
v1.logDetails();
```

Within the `logDetails` method, the keyword this refers to `v1`, the object on which `logDetails` has been called.

Nesting

In JavaScript, we nest one construction inside another. In the code below, the statements logging a message and setting the age to 70 are said to be nested inside the `if` construction that controls whether they run.

```
if ( age>70 ) {
    console.log("Age too large. Using 70");
    age = 70;
}
```

Network

Once we had many computers, we started linking them to form networks. Networks make two things possible. Firstly, a network connection gives you remote access to the processing power of a distant computer. You can run your programs on a distant machine. But secondly, networks let you move data between systems. A program running on one machine can access data stored on another. Data can be centralized and made available to connected clients. A service can be provided by several cooperating systems rather than by a process running on a single machine.

The first computer networks were "proprietary." Machines made by company A could not talk to machines made by company B, making it much harder for customers of company A to switch to company B (which the computer companies rather liked). But then, someone made a network that was so compelling that everyone wanted to connect their machines to it—the Internet.

Not a Number (NaN)

JavaScript uses the value Not a Number (`NaN`) to mean that the result of an operation has not generated a numeric result. Consider the following:

```
var x = 1/"fred";
```

This statement sets the value of x to the result of dividing the value 1 by the "fred" string. This is a meaningless calculation. In some programming languages, attempting to divide a number by a string would be rejected when the program was compiled or would produce an error when the program runs. In JavaScript, when a numeric expression cannot be evaluated, the program keeps running, but the value of the result of the expression is set to NaN. You can check to see if a variable contains NaN but not in the way you might think.

The if statement below doesn't work. NaN is not a value as such, so it isn't really meaningful to compare it with anything. But we do know that the value of NaN is not equal to anything, *including itself*.

```
if (x==NaN) console.log("x is not a number");
```

The if statement below checks to see if x is equal to itself. If this test fails, the value of x is NaN. Note that JavaScript expression evaluation is also aware of the concept of infinity. So, perhaps you might like to read the Infinity entry in this glossary next.

```
if (x!=x) console.log("x is not a number");
```

Null

JavaScript programs can contain variables that act as references to objects or functions. The null value is used in a program to express the situation where a reference variable does not refer to any object. For example, if you called a function to find something and that something could not be found, the function could return null to indicate this. A program can check if a value is null, and the value null itself is falsey (meaning a direct test on a value containing a null reference would return false). You can find out more about truthy and falsey earlier in this glossary, under Condition.

Obfuscation

Obfuscation is the process of making something hard to read. This can be useful if you want to stop casual eavesdroppers from being able to look at the JavaScript sent to a browser and work out how the code works before creating an exploit.

Object

A primitive data value, such as a number or a string, can only hold a single value. A JavaScript object is a container that can hold a collection of data values. A value held inside an object is called a property, and each property has a name. Objects can be used to describe physical items with properties for each descriptive item. I could create an object to describe my car properties by using code like this:

The following statement creates an object that contains two properties. The first property is the car's color; the second is the car's make. A variable with the identifier `car` is set to refer to the object that has been created.

```
let car = {color: "white", make: "Nissan"};
```

A JavaScript program can access a property by adding a period (`.`) followed by a property name to the variable's name referring to the object. The following statement would display the message `white` on the console because that is the value of the color property of the car object.

```
console.log(car.color);
```

A program can update the contents of a property by assigning a new value to the property. The statement below would set the `color` property of the car to the value `"blue"`.

```
car.color="blue";
```

A program can also add a new property to an object simply by assigning a value to a new property name. The statement below adds a `model` property to the `car` object. The `car` now has a `model` property set to the `"Cube"` string.

```
car.model="Cube";
```

Properties can be functions as well as values. The statement below adds a new property to the `car`. The new property is a function called `toString`, and it returns a string describing the object contents. The string contains the color and the car's name and model. Note that the keyword `this` is used in the function to get a reference to the object the function is part of. We can now call the `toString` function on the object referred to by car:

```
car.toString = function (){return this.color+" "+this.make+" "+this.model};
```

Once a function property has been added to an object, it can be called. The following statement would log the `blue Nissan Cube` string (returned by `toString`) in the console (if the color property had been updated to `blue` from the original `white`).

```
console.log(car.toString());
```

You can create and manage an object by managing object properties like this, but you can make your objects more cohesive using JavaScript classes. Objects are managed by `reference`.

Open source

Open source code is made available to anyone to view. It is usually made available according to terms set out in a license that specifies how the code can be modified, redistributed, and perhaps (but not always) sold as part of a new product. Before you start building something, you should look for an open-source implementation of the thing you are about to make. GitHub is a great place to start such a search.

Organization

Within GitHub, you can create an organization containing related repositories. Organizations can have multiple managers and contributors, and each organization can have a *github.io* repository that can host a web page for that organization. I created a Building-Apps-and-Games-in-the-Cloud organization to host all the repositories for this book at *https://github.com/Building-Apps-and-Games-in-the-Cloud*.

Parameter

A program can pass data into a function or method call by adding an argument to the call.

Below, you can see a call of a function called `doDisplay`. The function has a single argument—the `"Hello"` string. Within the definition of the function, the data item supplied is called a *parameter*:

```
doDisplay("Hello");
```

Below is the `doDisplay` function. The parameter to the function is called `message`. When the preceding function call preceding runs, the `message` value is set to the `"Hello"` string. The function calls the `alert` function, which displays the message for the user. Note that the parameter to a function is not given a type.

```
function doDisplay(message){
    alert(message);
}
```

The statement below calls doDisplay with an argument of 99, which is a number rather than a string. However, the function would work correctly because the value 99 would be converted into a string when the alert function displayed it:

```
doDisplay(99);
```

The following version of doDisplay sets a default value for the message parameter. If the argument is left off the doDisplay call, the parameter will be set to the "empty" string:

```
function doDisplay(message="empty"){
    alert(message);
}
```

This would display the message “empty” in an alert.

```
doDisplay();
```

Physical

When working with computers, we can split things into the physical and the logical/virtual. The physical element of a system is always the bit you have to plug in and switch on.

Port

A single computer can support many connected clients. Some clients might want to browse a website hosted on the machine; others might want to connect to a mail server running on the machine. When an application creates a connection to the Internet, it specifies a port number that it will use for network input and output. A web server application will use port 80, and the mail server will use 587. Clients connecting to the server know which port to use. Your browser will use port 80 to connect to a server unless you specify a different port number in the URL.

The port number is a 16-bit value, giving 65,655 possible port numbers. The first 1,023 are reserved for well-known ports—for example, email and web. The range from 1,024 to 49,151 is available for registration by organizations wishing to set up specific services of their own. Port numbers greater than 49,151 can be used for ad-hoc connections. Note that this means that if I want to connect to a program on a computer, I need to know the computer's IP address (so I can connect to it) and the port number that the program is sitting behind.

Primitive

The JavaScript language provides eight different data types for holding different kinds of value. Seven of the types, including Number, Boolean, and String, are defined as "primitive." You can't add properties to a primitive type; it just holds a single value. You must use an object if you want a variable that holds multiple values, such as a coordinate that contains `x` and `y`. Objects are managed by `reference`.

Procedure

A procedure is a function that doesn't contain a `return` statement that returns a value to the caller. The function `doDisplay` below displays a message but doesn't return a value:

```
function doDisplay(message="empty"){
    alert(message);
}
```

If a program tries to use the value returned from a procedure call, it will be given the `undefined` value. The result of the statement below would be to set the value of `res` to `undefined`.

```
let res = doDisplay("hello")
```

Program

A *program* is a sequence of instructions you give to a computer that tells it how to perform a specific task. For example, you could write a program to add two numbers and display the result.

Promise

In JavaScript, a Promise is an object describing the intention to perform an asynchronous task. A promise can be kept or broken, and event handlers can be attached to each outcome. It is also possible that a promise is never kept or broken. A Promise object contains a then method that accepts a function reference called when the Promise has been kept.

The Promise object also contains useful functions that work with promises. The Promise.race function is given a list of promises and returns a Promise that will complete when one of the Promise objects completes. You can combine multiple promises into a single Promise by using Promise.resolve. This accepts a list of promises and returns a single Promise that will be resolved when all the promises in the list have been resolved.

Property

A JavaScript object can contain property values. (See the Object entry in this glossary to learn how a JavaScript object is created and properties are added.) Consider that we have an object that describes a car. It has color, model, and make properties. You can delete a property from an object by using the delete operator:

```
delete car.model;
```

This statement would delete the model property from an object referred to by the car variable. The statement above uses the “dot” notation to access a property of an object. The property name is separated from the variable name using a “dot” or period (.). Properties get a lot more interesting when we start to use the “brackets” method of accessing them in an object. The statement below restores the "model" property to the object referred to by car. In this case, the property name (model) has been specified as a string enclosed in brackets:

```
car["model"] = "Cube"
```

The “dot” and the “brackets” mechanisms both generate the same property. You can use the “brackets” mechanism to create properties that have names containing spaces, as shown below:

```
car["number of seats"] = 5;
```

However, if you put spaces in your property name, you won't be able to use the "dot" mechanism to access this property; you will have to use brackets. The statement below displays the number of seats in the car:

```
console.log("Passenger capacity: " + car["number of seats"]);
```

Proprietary

An organization or company owns and promotes a proprietary technology, usually to maintain control of all or part of a market.

Recursion

See Recursion. (Sorry about this trite response.) Recursion happens when a function calls itself directly (Fred calls Fred) or indirectly (Fred calls Jim, who calls Fred). It is extremely useful when parsing grammar or traversing hierarchies.

Reference

A reference is a variable that refers to an object. Variables that are objects are managed by reference, whereas values manage all other types of variables. A reference can be set to the `null` value to explicitly indicate that it doesn't refer to anything.

Render

Render takes something from the logical (structured data) to the more physical (an image on a screen). This happens in games when data objects representing the players and scenery are converted into images and in the browser when the document object model is converted into a page. In most systems, the rendering process is performed separately from the rest of the application.

Repository

A repository is a collection of resources that can be managed as a whole by GitHub. It can contain text and binary files of any type. It has a name and can be owned by an individual or organization.

Return

A JavaScript function or method uses the `return` keyword to return a value or just to return early.

Sandbox

A system can host an application in a "sandbox" environment that strictly controls the application's access to the host machine.

Scope

In JavaScript, scope refers to that part of a program in which it is possible to access a given variable. A variable declared with the `let` keyword has the scope of the code block in which it is declared. A variable declared with the `var` keyword will have the function scope in which it is declared. A variable declared automatically will have global scope, meaning it will be visible to all the functions in the program. When creating nested blocks, it is possible for a newly declared variable to "scope out" a variable with the same name in an outer block. The code below creates two variables called `i`. The first (outer) variable has the value `0`. The second (inner) variable has the value `99`. Code running inside the inner block cannot access the outer version of `i` because any references to `i` will resolve to the inner variable.

```
let i = 0;
{
  let i = 99;
}
```

Statement

Actions to be performed in a JavaScript program are expressed as statements. A statement performs a single action (call a function, perform an assignment, and so on). Statements on the same line must be separated by semicolon characters. Semicolons might be omitted if the statements are separated by a line break or in a block on their own. A statement will generate a value. In the case of an assignment, the result of the statement is the value being assigned.

String

The string type in JavaScript can hold a string of Unicode characters. A JavaScript string can be extremely long—longer than the entire storage space in your computer. A program can create a string-type variable by assigning a string literal to a variable. Strings can be compared for equality. Comparing strings in respect of greater than/less than will result in an alphabetic ordering. Applying the `+` operator between two strings will concatenate them. For a full description of strings and their behaviors, see *https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/String*.

Synchronous

A synchronous operation is something you have to wait for. If a program calls a function to do something, and that function works synchronously, the program will be paused until the action is complete and the function returns. We should avoid using synchronous functions where possible because they can slow down the systems that use them. The alternative to synchronous operation is `asynchronous`.

System software

System software is software that provides a service to other software. The most obvious piece of system software you own is probably your computer's operating system. Other system software pieces include drivers for your graphics card or printer.

`this`

The behavior of `this` keyword can be confusing because it all depends on the context in which `this` is used. In the following examples, we'll examine a few contexts and discuss what `this` means in each. Note that this is a brief overview. Find more detail at *https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/this*.

`this` inside a method

A method is a block of code inside an object providing a behavior that an object can provide. The `Vehicle` class below contains two methods:

- `constructor` for the class
- `logDetails`

Both methods use the `this` keyword, which in this context, means "a reference to the object within which this method is running."

```
class Vehicle {
  constructor (newValue){
    this.color = newValue.color;
    this.make = newValue.make;
  }
  logDetails(){
    console.log("Color is:"+this.color+" make is:"+this.make);
  }
}
```

The this keyword allows the method call to manipulate the object's properties. In the code below, the logDetails method will run with the value of this set to v1:

```
let v1 = new Vehicle({color:"white",make:"Nissan"});
v1.logDetails();
```

this inside a function

In JavaScript, a function is implemented as an object that can contain properties. Within the body of a function, the this reference refers to the function object. The code below shows a function that contains two statements:

- The first statement displays a property called name.
- The second sets the name to "fred".

```
function thisDemo(){
  console.log(this.name);
  this.name="fred";
}
```

The first time we call the thisDemo function, the undefined message is displayed because the function doesn't have a name property. The "fred" value is displayed when we call the function because the thisDemo function's name property has been set to "fred".

this inside an arrow function (=>)

We have seen that within the code body of a function, the this reference refers to the function object. However, this behavior is different if the function is declared as an arrow function, which is declared using the => construction. Arrow functions are frequently created as anonymous functions bound to JavaScript events. Consider the code above (in the "this inside a function section"), which creates a button element in an HTML page. When the button is clicked, a count value is updated and displayed in the console:

```
function makeCountButton() {
    this.count = 0;
    let container = document.getElementById("mainPar");
    let newButton = document.createElement("button");
    newButton.textContent = "Increment Count";
    newButton.addEventListener("click", (e) => {
        console.log(this.count);
        this.count = this.count + 1;
    });
    container.appendChild(newButton);
}
```

The count value is stored within the makeCountButton function. When the button is clicked, the value of count is incremented. This code works because the event handler function bound to the button's click event is declared using arrow notation, which means any this references in the event handler refer to the enclosing object (in this case, the makeCountButton function containing the count value). If the event handler was written as below (as a function), the count value in makeCountButton would not be incremented because this.count refers to a variable in the event handler method:

```
newButton.addEventListener("click", function (e) {
        console.log(this.count);
        this.count = this.count + 1;
    });
```

Throw

When a program reaches a point where it cannot meaningfully continue, it can throw an exception to transfer control to code that will attempt to return things to a known good state. Examples of things that could be used to trigger exceptions include a timeout on an operation or an invalid or missing parameter to a function or method call. The exception object that is thrown can contain a description of what went wrong. The statement below throws an exception that contains a string message.

```
throw("Something bad happened");
```

Take a look at the try–catch definition to find out more.

Try–catch–finally

Code that might throw an exception can be enclosed in a try-catch-finally construction. The code that might throw an exception is placed in the try block. The code to handle the exception is placed in the catch block. Code that will run irrespective of whether the exception is thrown is placed in the finally block. The code below—which attempts to fetch something from a URL—shows how this works. The fetch function provided by JavaScript performs this operation. The fetch function will throw an exception if the resource is not found. The code in the catch clause deals with the error, and the code in the finally block will run regardless of whether the exception is thrown:

```
try {
  // Attempt to fetch some data
  const response = await fetch('https://api.example.com/data');
  const data = await response.json();

  // Do something with the data
  console.log(data);
} catch (error) {
  // Handle the error
  console.error('Error fetching data:', error.message);
} finally {
  // Clean up any resources
  console.log('Cleaning up resources...');
}
```

Undeclared

An undeclared variable is one that you've not declared. It doesn't exist in your program. You could ask, "How can I make one?" but, of course, the answer is you don't. That's the whole point. Undeclared variables are a problem because if JavaScript encounters one, it will throw an exception, stopping that thread of execution. Consider the following completely legal JavaScript code:

```
if(age>70) console.log("too old");
```

This code displays a message in the console if the `age` variable value is greater than `70`. However, if the variable `age` has not been declared, a `ReferenceError` exception will be thrown, stopping the program. `undeclared` sounds a lot like `undefined` (see below), but it is actually quite different. An undeclared variable doesn't exist. An undefined variable exists, but it is set to the value undefined, which means that it has not been given a value.

Undefined

If you create a variable but don't put anything in it, that value is set to "undefined".

```
var x;
console.log(x);
```

If you perform the above statements in the **Developer Tools** console, the "undefined" value is displayed. JavaScript regards undefined as a value that you can assign and test for.

```
x = undefined;
if(x==undefined) console.log("x has not been defined")
```

If you want to mark something as explicitly not set with a value, you can assign undefined to it. A function can test parameters to make sure that they are not undefined. If a program attempts to use an undefined value in a numeric expression, the result of the expression is the special value "not a number" or NaN.

Var

A program can declare a variable by using the var keyword. A variable declared using var within a function will exist for the duration of the function body in which it was declared. The variable v in the varDemo function shown below exists throughout the function body, even though it is declared inside an inner block. This means that the console will display the value 99.

```
function varDemo(){
  {
    // inner block
    var v = 99;
  }
  console.log(v);
}
```

A variable declared using var outside any function will have global scope.

Variable

A variable in a JavaScript program can hold a value. It is given an identifier by which it is referred. A variable can hold a single "primitive" value (for example, a number) or act as a reference to an object. Variables have a type, but JavaScript infers this from their assignment. The code below creates a variable called a that first contains a string and then an integer:

```
let a = "Rob";
a = 99;
```

Virtual

We can use software and computers to create "virtual" versions of things in the real world. Computers can contain virtual files, folders, and even avatars representing human users.

Visibility

The "visibility" of a variable is that part of the program code where the variable can be used in code. See the Scope entry earlier in this glossary.