

# 1

# The Cloud

You are reading this book because you have some programming skills, and you want to learn how to write programs for “the cloud”. If you can program you can take a problem, figure out how to solve it, and then express that solution as a set of instructions in a programming language a computer can be made to understand.

Now you want to take your problem-solving ability and use it to make “cloud based” solutions. But what is “the cloud”? What does “the cloud” do? And what kinds of problems will you have to solve if you are writing a solution that uses “the cloud”?

In this part we will start to answer these questions. We are going to take look at the origins of the cloud and identify what it is that makes an application “cloud based”. Then we’ll move on to consider how to create and manage services in the cloud. Next, we’ll address security and safety, identifying the issues to worry about and steps you can take to reduce risk when you create for the cloud. Finally, we are going to explore how the TypeScript language builds on JavaScript to make programs more secure and manageable. As we explore each topic, we’ll also take a close

look at JavaScript language features by creating some useful and fun applications.

# Chapter 1

# Coding for the cloud

## What you will learn

In this chapter we are going to investigate the fundamentals of cloud computing and discover what makes an application “cloud based”. We are also going to start our journey with the JavaScript language by exploring how JavaScript functions allow code running in the browser to interact with the JavaScript environment. We’ll see how programs run inside a web browser and how we can interact directly with code running in the browser via the Developer Tools, which will even let us view inside our programs as they run.

I’m assuming that you are familiar with programming but just in case there are things that you don’t know (or I have a different understanding of) I’ve added a glossary at the end of this book. Whenever you see a word highlighted like this: “**computer**” it means that the word is defined in the glossary. If something doesn’t quite make sense to you, go to the glossary and check on my definition of the word that I’m using.

# What is the cloud?

The internet now underpins many of our daily activities. Things like booking a table at a restaurant, buying a book, or keeping in touch with our friends are now performed using networked services. Nowadays we refer to these services as “in the cloud”. But what is the cloud? What does it do? And how can we use it? Let’s start with a look at how things were done in before we had the cloud.

## The World Wide Web

The **world wide web** and the **internet** are different things. The internet was invented to make it easy to connect software together over long distances. One early “killer app” for the internet was electronic mail. Internet connected computers acting as “mail servers” managed mailboxes for users, replacing paper messages with digital ones.

The world wide web was created some years after the internet to make it easier to work with documents. Rather than having to fetch and read a paper document you use a **browser** program to load an electronic copy from a web **server**. Documents can contain links to other documents, so that you can follow a reference without having to go and fetch another physical document.

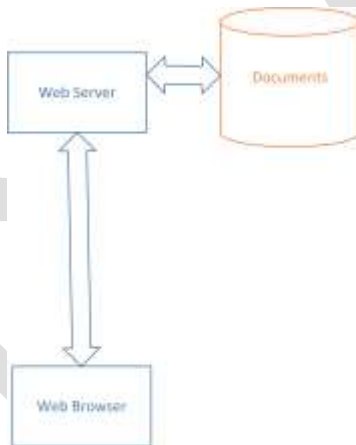


Figure 1.1 Ch01\_Fig\_01 Browser and Server

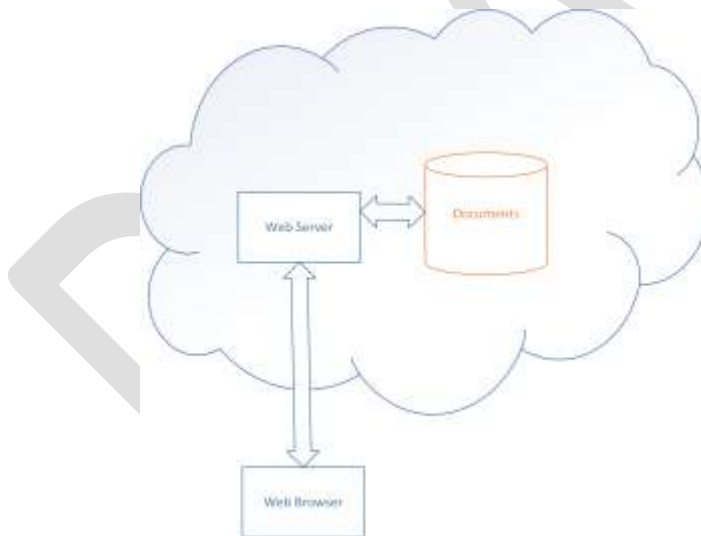
Figure 1.1 above shows how it works. The user sits at the browser and sends the web server computer requests for documents which are then sent back to the browser to be read. Later versions of the web added graphics so that a document could contain pictures.

# Putting the web in the cloud

If you wanted a web site in the early days of the internet you would set up your own **server** computer. If your site became popular you had to increase the power of your server (or get extra ones) to handle the load. Then you might find that all your capacity was only used at times of peak demand. The rest of the time your expensive hardware was sat twiddling its digital thumbs.

The cloud addresses this problem by turning computing resources into a commodity that can be bought and sold. Rather than setting up your own server, you now rent space in the cloud and pay someone else to host your site. The amount you spend on computer resources is proportional to the demand for the service you are providing. You never have to pay for resources that you don't use. What's more, the cloud makes it possible to create and deploy new services without having to set up expensive servers to make the service available. Most of the popular cloud suppliers even have pricing plans that provide free tariffs to help you get started.

So, the server that you connect to when using a network service (including the world wide web) might be owned by the company providing the service (Facebook have invested considerable sums in setting up their own servers) but it is more likely that you will be connected to a system hosted by one of the cloud providers. Later in the book we will discover how to create an account on a cloud provider and set up a service in the cloud.



**Figure 1.2** Ch01\_Fig\_02 HTML browser in the cloud

Figure 1.2 shows how this works. The web server and the documents are hosted in the cloud. Note that you could use a mix of different service providers, you could host the resources in one place and the server somewhere else. From the perspective of the service user, a cloud-based web site works in the same way as a server based one.



Figure 1.3 Ch01\_Fig\_03 Cloud application

Figure 1.3 shows the login page for a service we will be creating in Chapter 11 which connects with Internet of Things devices. My version of the service has the web address: <https://clbportal.azurewebsites.net/> If you enter that address into your browser you will be connected to a process in the cloud hosting this web site.

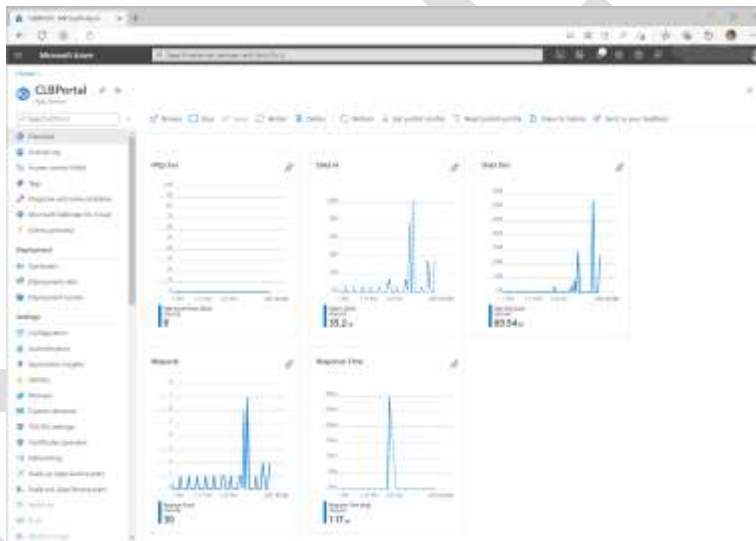


Figure 1.4 Ch01\_Fig\_04 Cloud management

You can manage your cloud services from the web. Figure 1.4 above shows the overview page on the Azure Portal for the service shown in Figure 1.3. It shows the amount of traffic the page is receiving and the time taken for the page to respond. There are also lots of options for service management and diagnostics. You can also use this page to increase the service provision to support many thousands of users. At the moment this service is using a free service level which supports enough users to allow demonstration and testing.

So, to answer our original questions: The **cloud** is a means by which companies can provide computing resources as a service. It hides the **physical** location of computing resources behind a **logical** address which users connect to. We can use the cloud to host our own services and make

them available for others to use via web addresses. A cloud service provider will provide a management interface for each service it hosts. Now, let's move on to take a look at the JavaScript language.

#### Programmer's Point

#### The cloud makes it a great time to be a developer

When I was learning to program it was very nearly impossible to show people what I had done. I could send them my punched cards (each card was punched with holes that contained the text of one line of my code) but it would be unlikely the program would run on the recipient's computer. Today you can invent something and make it available to the whole world by writing some JavaScript and hosting it in the cloud for free. This is tremendously empowering.

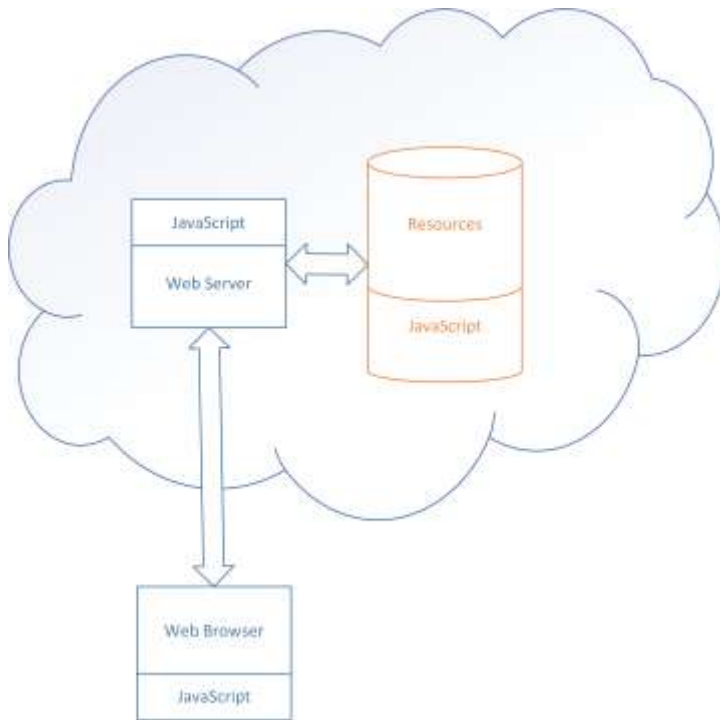
Nowadays the hard part is not making your service available but making people aware that it exists. Take a look at the Programmers Points in chapter 2, "Open-Source projects are a great place to start your career" and "GitHub is also a social network" for hints on how to do this.

## JavaScript

We now know what the cloud does. It provides a means of buying (or even getting for free) space on the internet where we can host our services. JavaScript has been described as "the programming language for the cloud". Let's look at what this means.

Originally the browser just displayed information that had been received from the server. Then it was decided that it would be useful if the browser could run programs that have been loaded from web sites. Putting a program inside a web page makes the page interactive without increasing the traffic to and from the web server. The user can interact with a program running in the browser without the server having to do anything. The program in the browser can animate the display or check user input to make sure it was correct before sending it to the server.

The language developed to run inside the browser was JavaScript. There have been several different versions of the language. Early ones suffered from a lack of standardization. Browsers from different companies provided different sets of features that were accessed in different ways. But this has now settled down. The specification of the language is now based on a worldwide standard managed by a standards organization called ECMA. We are going to be using version ES6 of the language.



**Figure 1.5** Ch01\_Fig\_05 JavaScript powered web

Figure 1.5 shows how a modern, JavaScript enabled browser and server work together. The web pages and the server program both contain JavaScript elements and the document store has been replaced with a range of resources which can also contain JavaScript code.

JavaScript has become extremely popular. Whenever you visit a website, you will almost certainly be running JavaScript code. JavaScript has also become very popular on web servers (the machines on the internet that deliver the information the browser requests). A technology called “**node.js**” allows JavaScript programs to run on a server to respond to requests from browsers. The web page shown in Figure 1.3 is produced by a JavaScript program running in the cloud using node.js. We’ll discover how to do this later, for the rest of this chapter we are going to be running JavaScript in the web browser. We’ll start by looking at a JavaScript hero, the JavaScript Function.

## JavaScript Heroes: functions

This is the first of our “JavaScript heroes”. JavaScript heroes are features of the language which make it super useful for creating cloud applications. A cloud application is all about events. Events happen when the user clicks the mouse button, when a message arrives from a server and when a timer goes tick. In all these situations you need a way of connecting a behavior to what

has just happened. As we shall see, functions in JavaScript simplify the process of connecting code to events. You might think you already know about functions in a programming language. However, I'd advise you to work through this section very carefully anyway. I'm sure that there will be at least one thing here that you didn't know about functions in JavaScript. This is because JavaScript has a very interesting implementation of the function which sets it apart from other programming languages. Let's start with an overview of functions and then drill down into what makes them special in JavaScript.

## The JavaScript function object

A JavaScript function is an object that contains a "body" made up of JavaScript statements that are performed when the function is called. A function object contains a name property which gives the name of that function. Functions can be called by their 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 statement after the one called the function. Functions can be made to accept values to work on and a function can also return a result value.

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

The code above defines a function with the name `doAddition`. The definition is made up of the header (the part with the name of the function and the parameters it accepts) and the body (the block of two statements that are obeyed when the function is called). The function above calculates the sum of two values and displays the result in an alert box. The `alert` function is one of many "built-in" functions that are provided by the browser Application Programming Interface or API. Learning how to use the facilities provided by the API in a system is a huge part of learning how to be an effective developer.

### Programmer's Point

### You don't need to know about every JavaScript API function

There are thousands of different functions available to a JavaScript program. The `alert` function is one of them. You might think you need to know about all of them, but actually you don't. I don't know anyone who knows all the functions available to JavaScript programs. But I know lots of people who know how to use search engines to find things when they need them. Don't be afraid to look things up, and don't feel bad about not knowing everything.



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

We call the `doAddition` function as shown above. The values 3 and 4 are called the **arguments**. Argument values are mapped onto the parameters in the function. When the above call of `doAddition` starts to run the parameter `p1` will hold the value 3 and the parameter `p2` will hold the value 4. This leads to the display of an alert box that displays the value 7.



Figure 1.6 Alert box

Figure 1.6 above shows the alert box that is displayed when the function runs. An alert box always has the name of the originator at the top. In this case the function was running in a page on a website located at `beginintocodecloud.com`. The `alert` function is the first JavaScript function that we have seen. It asks the browser to display a message and then waits for the to click the OK button. From an API point of view, we can say that the `alert` function accepts a string of text and displays it.

# Lifting the lid on JavaScript

Wouldn't it be nice if we could watch the `doAddition` function run? It turns out that we can. Modern browsers contain a "Developer Tools Console" which lets you type in JavaScript statements and view the results. How you start the console depends on the browser you are using:

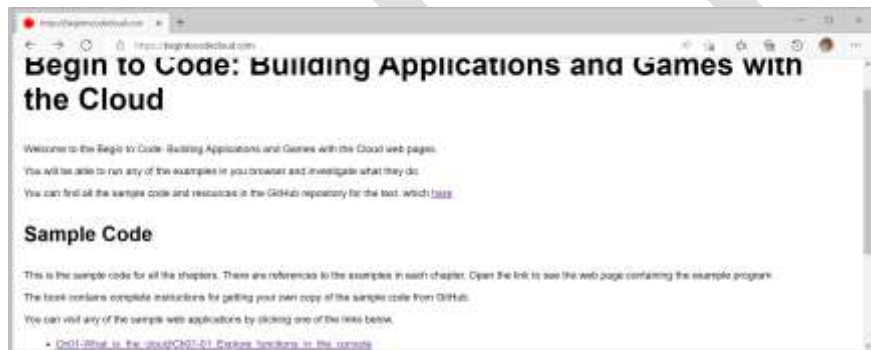
Operating System	Browser	Sequence	Notes
Windows	Edge	F12 or CTRL+SHIFT+J	The first time you do this you will be asked to confirm the action.
Windows	Chrome	F12 or CTRL+SHIFT+J	
Windows	FireFox	F12 or CTRL+SHIFT+J	
Windows	Opera	CTRL+SHIFT+J	
Macintosh	Safari	CMD+OPTION+C	You need to go to Preferences->Advanced->Show Develop Menu and select "Show Develop Menu" to enable it.
Macintosh	Edge	F12 or CTRL+SHIFT+J	
Macintosh	Chrome	CMD+OPTION+J	
Macintosh	FireFox	CMD+SHIFT+J	
Macintosh	Opera	CMD+SHIFT+J	This also works with the Chromium browser on the
Linux	Chrome	F12 or CTRL+SHIFT+J	

The table above gives the shortcut keys for different browsers and operating systems. Note that the console will look slightly different on each browser, but the views that we are going to use are present on all of them. Let's do our first "Make Something Happen" and use the Developer Tools console to explore functions from our web browser.

## Make Something Happen

## Explore functions in the console

All the example code for this book is available in the cloud. Of course. You can find the sample pages at [begintocodecloud.com](http://begintocodecloud.com). Open this web site and scroll down to the Samples section on the page.



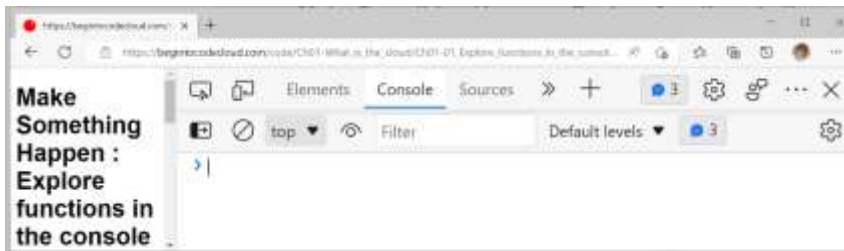
### Ch01\_inset01\_01 Web Site Samples

The sample code is presented as a list of links. There is a link for each sample page. We are going to do the very first sample so click **Ch01-What\_is\_the\_cloud/Ch01-01\_Explore\_functions\_in\_the\_console**.



### Ch01\_inset01\_02 Explore Functions

This is the web page for this exercise. Press the key sequence to open the Developer Tools in your browser. The screenshots for this section are from the Edge browser running in Windows.



Ch01\_inset01\_03 Developer Console

The Developer Tools will open on the right-hand side of your page. You can make the tools area larger by clicking the line separating the sample code from the tools and dragging it to the left as shown above. The web page will automatically resize. The Developer Tools contain several different tabs. We will look at the Elements and Sources tabs later. For now, click the Console tab to open the console.

We can type JavaScript statements into the console, and they will be performed in the browser and the results displayed. The console provides a ' > ' prompt which is where you type commands. Click the console and start typing "doAddition" and watch what happens.



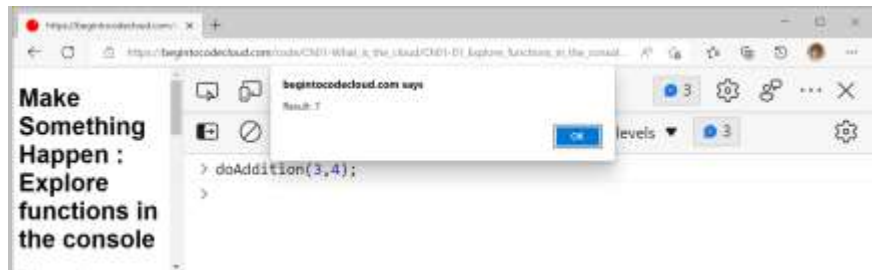
Ch01\_inset01\_04 Typing doAddition

As you type the text the console presents you with a menu of things you could type in. This is a very useful feature. It saves you typing, and it makes it less likely that you will type something incorrectly. You can move up and down the menu of items by using the arrow keys or the mouse. Click [doAddition](#) in the list or press the TAB key when it is selected. Now fill in the rest of the function call by adding 3 and 4, the two arguments:



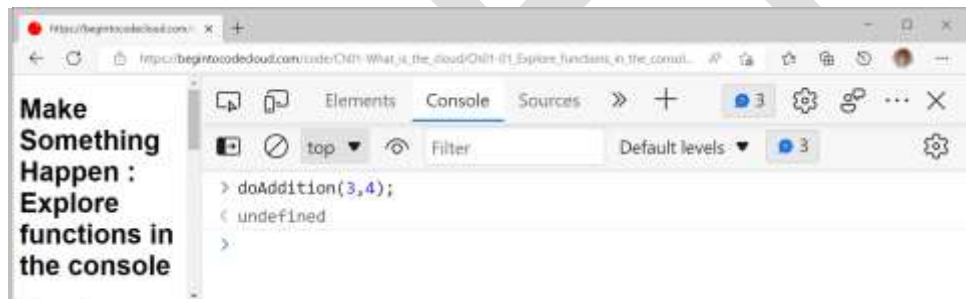
Ch01\_inset01\_05 Calling doAddition

When you have finished the console should look like the above. Now press enter to run the function.



Ch01\_inset01\_06 doAddtion alert

The `alert` function has control and is displaying the alert box containing the result. Note that you can't enter new statements into the console at this point. The only thing you can do next is press the OK button in the alert to clear it. Do this.



Ch01\_inset01\_07 doAddtion completed

When you click on OK the alert box disappears and the `doAddition` function completes. You can now enter further commands in the console.

## CODE ANALYSIS

### Calling functions

A code analysis section is where we examine something that we have just seen and answer some questions you might have about it. There are a few questions you might have about calling functions in JavaScript. Keep the browser open and displaying the console so you can find out the answers.

Question: What does the `undefined` message mean in the console after the call of `doAddition`?

The console takes a JavaScript statement, executes it, and displays the value generated by the statement. If the statement calculates a result the statement will have the value of that

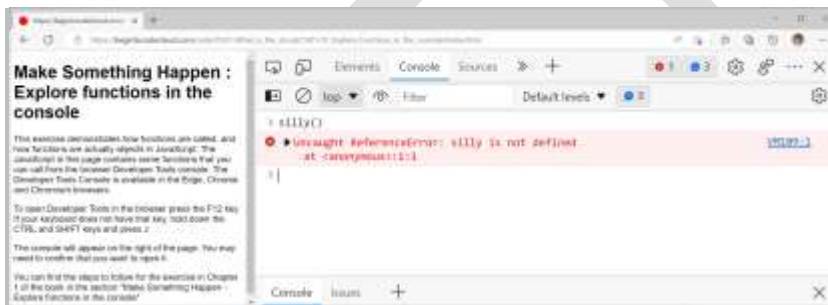
result. This means you can use the console as a calculator. If you type in 2+2 the console will display 4.

```
> 2+2  
< 4
```

The expression 2+2 is a valid JavaScript statement that returns the value of the calculation result. So the console displays 4.

However, the `doAddition` function does not deliver a result. It has no value to return, so it returns a special JavaScript value called **undefined**. Later we will discover JavaScript functions that do return a value.

Question: What happens if I try to call a function that is not available?



Ch01\_inset02\_01 calling missing function

Above you can see what happened when I tried to call a function called “silly”. JavaScript told me that the function has not been defined.

Question: What would happen if I added two strings together?

In JavaScript you can express a **string** of text by enclosing it in double quotes or single quotes. We will discuss JavaScript in detail strings later in the text.

```
> doAddition("hello", "world");
```

The call of `doAddition` above has two string arguments. When the function runs the value of `p1` is set to “hello” and the value of `p2` is set to “world”. The function applies the + operator between the parameters to get the result.

```
let result = p1 + p2;
```

Above you can see the statement in the `doAddition` function that calculates the value of the result of the function. The statement defines a variable called `result` which is then set to sum of the two parameters. We will be looking at the **let** keyword later in the book (or you can look up `let` in the Glossary). The JavaScript selects a + operator to use according to the **context** of the addition. If `p1` and `p2` are numbers JavaScript will use the numeric version of +.

If `p1` and `p2` are strings of text JavaScript will use the string version of `+` and set the value of `result` to a string containing the text "helloworld". You might find it interesting to try adding strings to numbers and watching what JavaScript does. If you look in the `doAddition` function itself you will find a statement does this.

Question: What happens if I subtract one string from another?

Adding two strings together makes sense but subtracting one string from another is not sensible. We can investigate what happens if we do this because the web page for this exercise contains a function called `doSubtraction`. Normally you would give this function numeric arguments. Let's discover what happens if we use text.

```
> doSubtraction("hello", "world");
```

If you make the above call of `doSubtraction` you get the following message displayed by the alert:

```
Result: NaN
```

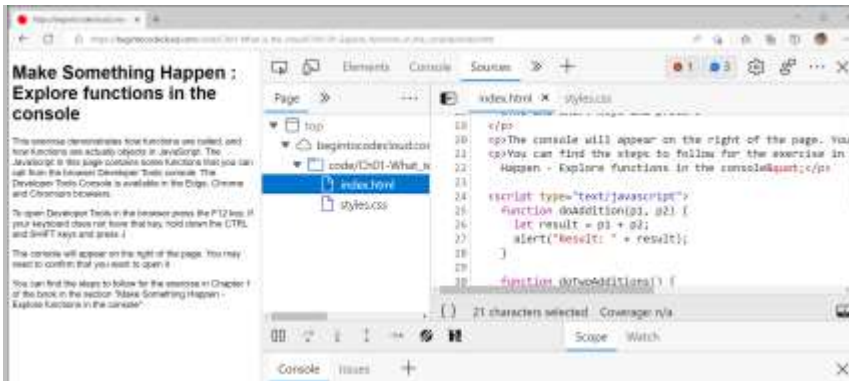
The value "NaN" means "not a number". There is only one version of the `-` operator, the numeric one. However, this can't produce a number as a result because it is meaningless to subtract one string from another. So the result of the operation is to set the value of `result` to a "special" value **NaN** to indicate that the result is not a number. Later we'll discover more about the special values in JavaScript programs.

Question: Why do some strings have `"` around them and some have `'`.

When the debug console shows you a string value it will enclose the string in single quote characters. However, in some parts of the program strings are delimited by double quote characters. In JavaScript you can use either double or single quotes to mark the start and end of a string.

Question: Where do these functions come from?

That's a good question. The function statements are in the web page loaded by the browser from the `begintocodecloud.com` server. We can use the Developer Tools to view this file. We must change the view from Console to Source to do this. Click the Sources tab which is next to the Console tab on the top row.

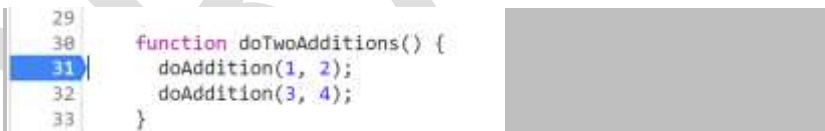


Ch01\_inset02\_02 viewing the web page source

The Sources view shows you all the files behind the website that you are visiting. This site has two files; a styles.css file which contains style definitions (of which more in the next chapter) and an index.html file which contains the text of the web page, along with the JavaScript programs. If you select the index.html file as shown above, you will see the contents of the file including the JavaScript for `doAddition`. There is another function called `doTwoAdditions` which calls `doAddition` twice.

Question: Can we watch the JavaScript run?

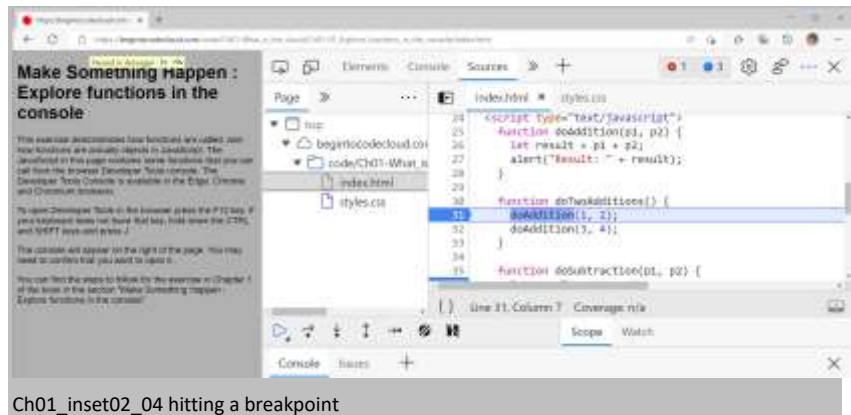
Yes. We can set a “breakpoint” at a statement and when that statement is reached the program will pause and we can go through it one step at a time. This is a wonderful way to see what a program is doing. Put a breakpoint at the first statement of `doTwoAdditions` by clicking in the left margin to the left of the line number:



Ch01\_inset02\_04 setting a breakpoint

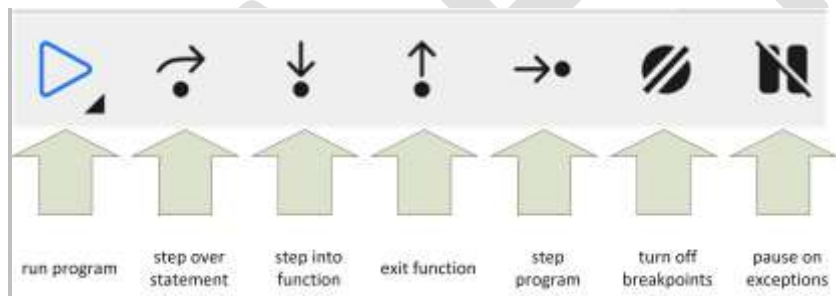
The breakpoint is indicated by the arrow highlighting the line number. It will cause the program to pause when it reaches line 31 in the program. Now we need to call the `doTwoAdditions` function. Select the Console tab, type in the following and press enter:

```
> doTwoAdditions();
```



Ch01\_inset02\_04 hitting a breakpoint

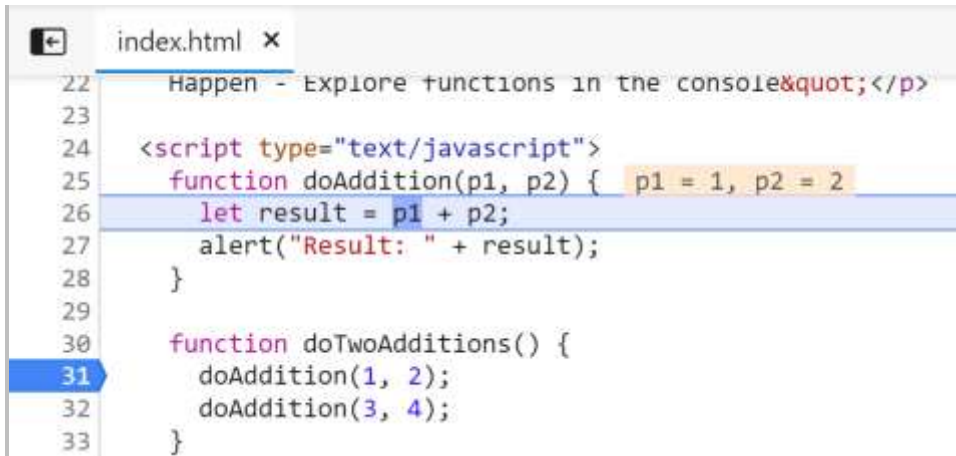
Above you can see what happens when a breakpoint is hit. The browser shows the program paused at the statement with the breakpoint. The most interesting part of this view is the control buttons you can see towards the bottom of the page:



Ch01\_inset02\_05 program controls

These controls might look a bit like cave paintings, but they are very useful. They control how the browser will work through your program. The one that we will use first is "step into function" (the one third from the left). Each time you press this control the browser will perform one statement in your program. If the statement is a function call the browser will step into the function. You can use the "step over statement" control to step over function calls and the "exit function" to leave a function that you have just entered. Press "step into function".





```
index.html x
22   Happen - Explore functions in the console";</p>
23
24   <script type="text/javascript">
25     function doAddition(p1, p2) { p1 = 1, p2 = 2
26       let result = p1 + p2;
27       alert("Result: " + result);
28     }
29
30     function doTwoAdditions() {
31       doAddition(1, 2);
32       doAddition(3, 4);
33     }
```

Ch01\_inset02\_06 viewing program execution

The highlighted line has now moved to the first statement of the `doAddition` function. The debugger shows you the values in the parameters. If you keep pressing the “step into function” button in the control panel you can see each statement obeyed in turn. Note that you will have to click the OK button in the alert when you perform the statement on line 27 that calls `alert`. If you get bored, you can press the “run program” button at the left of the program controls to run the program. You can clear the breakpoint at line 31 by clicking it.

You can add as many breakpoints as you like and you can use this technique on any web page that you visit. It is interesting to see just how much complexity that there is behind a simple site. Leave the browser open at this page, we will be making some more things happen in the following text.

## References to JavaScript function objects

We have seen that function in a JavaScript program is function is represented by a JavaScript function object. A JavaScript function **object** is managed by **reference**. Our programs can contain reference variables that can be made to refer functions.

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

We’ve seen the definition above before. It defines a function called `doAddition` that adds two parameters together and display the result. When JavaScript sees this it creates a function object to

represent the function and creates a variable called `doAddition` that refers to the function object.

```
doAddition(1,2);
```

The statement above calls the `doAddition` function which will display an alert containing a result of 3. JavaScript variables can contain references to objects so you can write statements like this in your program:

```
let x = doAddition;
```

This statement creates a variable called `x` and makes it refer to the same object as the `doAddition` function.

```
x(5,6);
```

This statement calls whatever `x` is referring to and passes it the arguments 5 and 6. This would call the same object that `doAddition` is referring to (because that is what `x` is referring to) resulting in an `alert` with the message "Result:11" being displayed. We can make the variable `x` refer to a different function.

```
x = doSubtraction;
```

The above statement only works if we have previously declared a function called `doSubtraction` which performs subtraction. The statement makes `x` refer to this function.

```
x(5,6);
```

When the statement above calls `x` it will run the `doSubtraction` function and display a result value of -1, because that is the result of subtracting 6 from 5. We can do evil things with function references. Consider the following statement:

```
doAddition = doSubtraction;
```

If you understand how evil this statement is, you can call yourself a “function reference ninja”. It is completely legal JavaScript that means that from now on a call of `doAddition` will now run the `doSubtraction` function.

## Function expressions

You can create JavaScript functions in places where you might not expect to be able to. We are used to setting variables by assigning **expressions** to them:

```
let result=p1+p2;
```

The above statement assigns the expression `p1+p2` to a variable called `result`. However, you can also assign a variable to a *function expression*:

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

The above statement creates a function object which does the same as the `doAddition` function we have been using. I’ve put the entire function on a single line but the statements are exactly the same. The function is referred to by a variable called `codeFunc`. We can call `codeFunc` in the same way as we used `doAddition`. The statement below calls the new function and would display a result of 17

```
codeFunc(10,7);
```

## Function references as function arguments

This is probably the most confusing section title so far. Sorry about that. What we want to do is look at how you can pass function references into functions. In other words, a program can tell a function which function to call. Later in this chapter we are going to tell a timer which function to call when the timer goes tick. This is how it works.

An **argument** is something that which is passed into a function when it is called. We have passed two arguments ([p1](#) and [p2](#)) into the [doAddition](#) function each time we call it (these are the items to be added). We can also use references to functions as arguments to function calls.

```
function doFunctionCall(functionToCall, p1, p2){  
    functionToCall(p1,p2);  
}
```

The function [doFunctionCall](#) above has three parameters. The first ([functionToCall](#)) is a function to call, the second ([p1](#)) and third ([p2](#)) are values to be passed into that function when it is called. All that [doFunctionCall](#) does is call the supplied function with the given arguments. It's not particularly useful, but it does show that you can make a function that accepts function a reference as a parameter. We could call [doFunctionCall](#) like this:

```
doFunctionCall(doAddition,1,7);
```

This statement calls the [codeFunc](#) function. The first argument to the call is a reference to [doAddition](#), the second argument is 1 and the third 7. The result would be an alert that displayed "Result:8". We can get different behaviors by using [doFunctionCall](#) to call different functions.

We can take this further and use function expressions as arguments to function calls as shown in the statement below which defines a function which used as an argument in a call to the [doFunctionCall](#) function. A function created as an argument has no name and so it is called an anonymous function.

```
doFunctionCall(function (p1,p2){ let result=p1+p2; alert("Result: "+result);},1,7);
```

## Make Something Happen

## Fun with Function Objects

Function objects can be confusing. Let's use our debugging skills to take a closer look at how they work. If you have left the browser at the page for the previous "Make Something Happen" just continue with that. Otherwise need to go to the book web page at

[beginnertocloud.com](https://beginnertocloud.com) and then select the sample **Ch01-What\_is\_the\_cloud/Ch01-01\_Explore\_functions\_in\_the\_console**. Then press function key F12 (or CTRL+SHIFT J) to open Developer Tools and select the console.

```
> let x = doAddition;
```

Type the above into the console. Note that we don't put any arguments on the end of [doAddition](#) because we are not calling it, we are specifying the name of the reference to the function. Now press enter.

```
> let x = doAddition;  
< undefined
```

The console shows the value "undefined" because the console always displays the value returned by a statement and the act of assignment (which is what the program is doing) does not return a value. After this statement has been performed the variable `x` now refers to the [doAddition](#) function. We can check this by looking at the **name property** of `x`. A function object has a name property which is the name of the function. Type in "`x.name;`" press enter and look at what comes back:

```
> x.name;  
< 'doAddition'
```

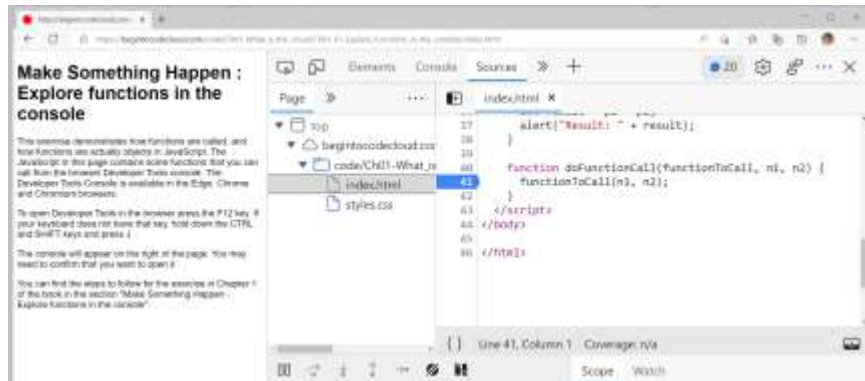
The console displays the value returned by the statement. In this case the statement is accessing the [name](#) property of the variable `x` which is the string `doAddition`.

```
> x.name;  
< 'doAddition'
```

Now let's call `x` with some arguments Type in the following statement and press Enter.

```
> x(10,11);
```

Since `x` refers to [doAddition](#), you will see an alert displaying the value 21. Next, we are going to feed the `x` function reference into a call of [doFunctionCall](#), but before we do that we are going to set a breakpoint so that we can watch the program run. Select the Sources tab and scroll down the `index.htm` source file until you find the definition of [doFunctionCall](#). Click the left margin near the line number 41 to set a breakpoint inside the function at statement 41.

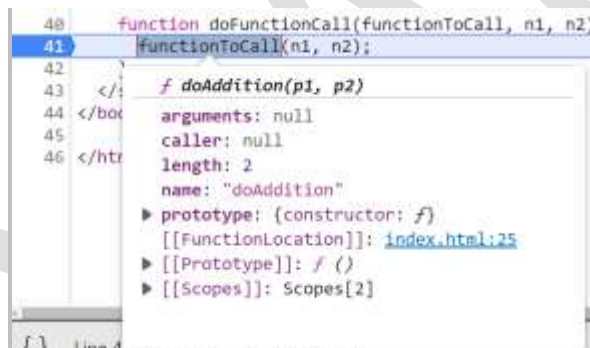


Ch01\_inset03\_01 breakpoint in doFunctionCall

Now return to the Console view, type the following statement and press Enter.

```
> doFunctionCall(x,11,12);
```

When you press Enter the console calls `doFunctionCall`. When it reaches the first statement in the function it hits the breakpoint and pauses.



Ch01\_inset03\_02 program paused in doFunctionCall

Above you can see the program paused. The `doFunctionCall` function has been entered and the parameters to the function have been set to the arguments that were supplied. If you hover the mouse pointer over `functionToCall` in line 41 you will see a full description of the value in the parameter. The description shows that the parameter refers to the `doAddition` function.

Repeatedly press the “step into function” button to watch the program go into the `doAddition` function, calculate the result and display the result in an alert. Then click OK in the alert to clear it and press the clear the alert and press the “Run Program” button (it’s the one on the left) to complete this call. Finally, return to the console for the “grand finale” of this Make Something Happen.

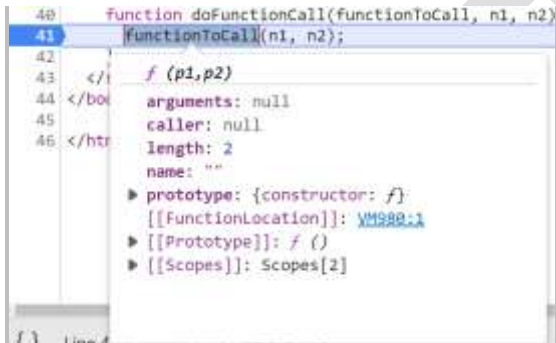
In the “grand finale” we are going to create an anonymous function and pass it into a call of

`doFunctionCall`. The function will be defined as an argument to `doFunctionCall`. The statement we are going to type is a bit long and you must get it exactly right for it to work. The good news is that the console will suggest sensible things to type. If you get an error, you can use the up arrow key to get the line back so that you can edit it and fix any mistakes.

```
> doFunctionCall(function (p1,p2){ let result=p1+p2; alert("Result: "+result);},1,7);
```

Ch01\_inset03\_03 calling anonymous function

Now press enter to execute this statement. The program will hit the same breakpoint as before, but the display will be different:



Ch01\_inset03\_04 anon function in doFunctionCall

This time the name property of the function is an empty string. The function is anonymous. Press the “step into function” button to see what happens when an anonymous function is called.



Ch01\_inset03\_05 stepping through an anon function in doFunctionCall

The browser has created a temporary file to hold the anonymous function while it is in the debugger. The file is called VM980. When you do this exercise, you might see a different name. We can step through the statements in this file using “step into function”. If you want to just run the function to completion you can press the “Run program” button in the program controls.

Anonymous functions are used a lot in JavaScript, particularly when calling API functions to perform tasks. An object from the JavaScript API will signal that something has happened by calling a function. The quickest and most convenient way to create the function to be called is to declare it as an anonymous function.

## Returning values from function calls

Up until now we have just called functions and they have done things. They have not returned a

value. They have returned the value undefined. Now we are going to investigate how a function can return a value and how a program can use the value that is returned.

```
function doAddSum(p1, p2) {  
    let result = p1 + p2;  
    return result;  
}
```

The function `doAddSum` above shows how a function can return a value. The `return` keyword is followed by an expression that gives the value to be returned when the function is called.

```
let v = doAddSum(4,5);
```

The statement above creates a variable called `v` and set the value of this variable to the result of the call of `doAddSum` – in this case the value 9 (the result of adding 4 to 5). The return from a function can be used anywhere you can use a value in a function.

```
let v = doAddSum(4,5) + doAddSum(6,7);
```

In the statement above the `doAddSum` function would be called twice and the value of `v` would be set to 22. We can also use function returns as arguments in function calls.

```
let v = doAddSum(doAddSum(4,5), doAddSum(6,7));
```

The code above looks a bit confusing but JavaScript would not have a problem performing it. The outer call of `doAddSum` would be called first, and then the two further calls would run to calculate the values of the two arguments. Then the outer call would run with these values. Note that the above statement is not an example of **recursion** (where a function calls itself). It shows how function calls can be used to produce values to be used as arguments to function calls.

A function can contain multiple `return` statements so it can return from different places in the function code. You can also use a `return` from a function to “escape” from inside the middle of deeply nested loops. However, if you use `return` like this you might end up making code that is



harder to debug.

## Programmer's Point

### Try to design your code to make it easy to debug and maintain

You will spend at least as much time debugging and maintaining code as you will writing it. Worse still, you will frequently be called on to debug and maintain programs that have been written by other people. Even worse still, six months after you've written a piece of code you become one of the "other people". I've occasionally asked myself "What idiot wrote this code?" only to find out that it was me.

When you write a program, try to make sure that it is going to be easy to debug. Consider the implementation of `doAddSum` below.

```
function doAddSum(p1, p2) {  
  let result = p1 + p2;  
  return result;  
}
```

You might think that it would be more efficient to return the result directly and get rid of the `result` variable:

```
function doAddSum(p1, p2) {  
  return p1 + p2;  
}
```

The above version of the function works fine. And it might even save a few millionths of a second when it runs (although I doubt this because browsers are very good at optimizing code). However, the second one will be harder to debug because you can't easily view the value of the result that it returns. With the original code I can just look at the contents of the `result` variable. In the "improved" version I'll have to mess around a bit to find out what value is being returned to the caller.

It's a similar issue with lots of `return` statements in a function. If the function containing lots of returns delivers the wrong result you have to step through it to work out the route it is following to deliver that particular result. If the function only has one return statement you know exactly where the result is coming from.

If the function just performs a task a good trick is to make a function return a status code so that the caller knows exactly what has happened. I often use the convention that an empty string means that the operation worked, where as a string contains a reason why it failed.

If the function is supposed to return a value you can use the JavaScript values `null` and `undefined` to indicate that something has not worked

Oh, and if you find yourself thinking “What idiot wrote this code?”, don’t be so hard on the “idiot”. They were probably in a hurry, or lacked your experience or maybe, just maybe, there might a reason why they did it that way that you don’t know.

## Returning multiple values from a function call

A problem with functions is that they can only return one value. However, sometimes we would like a function that returns multiple values. Perhaps we need a function to read information about a user. The function returns a name and an address along with status which indicates whether or not the function has succeeded. If the status is an empty string it means that the function worked. Otherwise, the status string contains an error message.

```
function readPerson() {  
  let name = "Rob Miles";  
  let address = "House of Rob in the city of Hull";  
  let status = "";  
}
```

Above is an implementation of a `readPerson` function that sets up some return values but doesn’t return anything. What we want now is a way the function can return these values to a caller.

## Returning an array from a function call

```
function readPersonArray() {  
  let name = "Rob Miles";  
  let address = "House of Rob in the city of Hull";  
  let status = "";  
  return [status, name, address];  
}
```

The above code creates a function called `readPersonArray` that returns an array containing the status, name and address values. We create an array in JavaScript by enclosing a list of values in brackets.

```
let reply = readPersonArray();
```

The statement above shows how we would create a [reply](#) variable that holds the result of a call to [readPersonArray](#). We can now work with the values in the array by using an index value to specify which element we want to use in our program.

```
let status = reply[0];
if(status != "") {
    alert("Person read failed:" + status);
}
```

The above code puts the element at the start of the reply array (JavaScript arrays are indexed starting at 0) into a variable called [status](#). This should be the status value of the call that has just been made. If this element is not an empty string the code displays an alert containing the status so that the user can see that something has gone wrong. If you look at the code for [readPersonArray](#) you'll see that the element at the start of the array is the status value, so this code would display an alert if the [status](#) variable contains an error message.

This code works, but it has an obvious disadvantage. The person making the call of [readPersonArray](#) needs to know the order of the values returned by the function. For this reason, I don't think using an array here is a very good idea. Let's look at a better one.

#### Programmer's Point

#### Use extra variables to make code clearer

You might look at the code above and decide that I've used a variable when I don't need to. I've created a variable called [status](#) that contains a copy of the value in [reply\[0\]](#). I've done this because it makes the code that follows much clearer. The test of the status and the display in the alert make a lot more sense to the reader than they would if the code contained the variable [reply\[0\]](#). This won't slow the program down or make it larger because the JavaScript engine is very good at optimizing statements like these.

## Returning an object from a function call

```
function readPersonObject() {
    let name = "Rob Miles";
    let address = "House of Rob in the city of Hull";
    let status = "";
    return {status:status, name:name, address:address};
}
```

The above code creates a function called `readPersonObject` that returns an object containing status, name and address properties.

```
let reply = readPersonObject();
if(reply.status != "") {
    alert(reply.status);
}
```

The code above shows how the `readPersonObject` function would be called and the status property tested and displayed. Note that this time we can specify the parts of the reply that we want by using their name. If you want to experiment with these functions you can find them in the example web page we have been using for this chapter: **Ch01-What\_is\_the\_cloud/Ch01-01\_Explore\_functions\_in\_the\_console**

#### Programmer's Point

#### Make good use of object literals

This way of creating objects in JavaScript programs is called an object literal. I'm a big fan of them. They are a great way to create data structures which are easy to use and understand, right at the point you want to use them. You can also make an object literal to supply as an argument to a function call. If I want to supply name and address values to a function, I can do this because a function can have multiple arguments.

```
function displayPersonDetails(name, address) {
    // do something with the name and address here
}
```

The function `displayPersonDetails` has two parameters so that it can accept the incoming information. However, I'd have to be careful when calling the function because I don't want to get the arguments the wrong way round:

```
displayPersonDetails("House of Rob", "Rob Miles");
```

This would display the details of someone called "House of Rob" living at "Rob Miles". A much better way would be to have the function accept an object which contains name and address properties:

```
function displayPersonDetails(person) {
    // do something with the person.name and person.address here
}
```

When I call the function I create an object literal to deliver the parameters.

```
displayPersonDetails({ address:"House of Rob", name:"Rob Miles"});
```

This call of the function creates an argument which is an object literal containing the name and address information for the person. It is now impossible to get the properties the wrong way round in the function.

# Make a Console Clock

We are now going to apply what we have learned to create a clock that we can start from the console. The clock will display the hours, minutes and seconds on a web page. At the moment we don't know how to display things on web pages (that is a topic for Chapter 2) so I've provided a helper function that we can use. You can use the Developer Tools to see how it works.

## Getting the date and time

Our clock is going to need to know the date and time so that it can display it. The JavaScript environment provides a [Date](#) object we can use to do this. When a program makes a new [Date](#) object the object is set to the current date and time.

```
let currentDate = new Date();
```

The statement above creates a new [Date](#) object and sets the variable [currentDate](#) to refer to it. The keyword **new** tells JavaScript to find the definition of the [Date](#) object and then construct one. We will look at objects in detail later in the text. Once we have our date object we can call **methods** on the object to make it do things for us.

```
function getTimeString() {  
  let currentDate = new Date();  
  let hours = currentDate.getHours();  
  let mins = currentDate.getMinutes();  
  let secs = currentDate.getSeconds();  
  let timeString = hours + ":" + mins + ":" + secs;  
  return timeString;  
}
```

The function [getTimeString](#) above creates a [Date](#) object and then uses the methods called

[getHours](#), [getMinutes](#) and [getSeconds](#) to get those values out of it. The values are then assembled into a string which the function returns. We can use this function to get a time string for display.

## Make Something Happen

### Console Clock

Open [beginnertocloud.com](https://beginnertocloud.com) and scroll down to the Samples section. Click **Ch01-What\_is\_the\_cloud/Ch01-02\_Console\_Clock** to open the sample page. Then open the Developer Tools. Select the console tab.



Ch01\_inset04\_01 console clock page

The page shows an “empty” clock display. Let’s start by investigating the [Date](#) object. Type in the following:

```
> let currentDate = new Date();
```

Now press enter to create the new [Date](#) object.

```
> let currentDate = new Date();  
< undefined
```

This statement creates a new [Date](#) and sets the variable [currentDate](#) to refer to it. As we have seen before, the [let](#) statement doesn’t return a value, so the console will display “undefined”. Now we can call methods to extract values from the object. Type in the following statement:

```
> currentDate.getMinutes();
```

When you press Enter the [getMinutes](#) method will be called. It returns the minutes value of the current time and the console will display it.

```
> currentDate.getMinutes();  
< 17
```

The code above was run at seventeen minutes past the hour, so the value returned by [getMinutes](#) will be 17. Note that [currentDate](#) holds a “snapshot” of the time. To get an

updated date you will have to make a new [Date](#) object. You can also call methods to set values in the date too. The date contents will automatically update. You could use [setMinutes](#) to add 1000 to the minutes value and discover what the date would be 1000 minutes in the future.

The web page for the clock has the [getTimeString](#) function built in, so we can use this to get the current time as a string. Try this by entering a call of the function and pressing enter:

```
> getTimeString();  
< '13:18:17'
```

Above you can see a call of the function and the exact time returned by it. Now that we have our time we need a way of displaying it. The page contains a function called [showMessage](#) which displays a string of text. Let's test it out by displaying a string. Type the statement below and press enter

```
> showMessage("hello");
```



The web page now displays the string that was entered. Now we need a function that will display a clock tick. We can define this in the console window. Enter the following statement and press Enter:

```
> let tick = function(){showMessage(getTimeString());};  
< undefined
```

Take a careful look at the contents of the function and see if you can work out what they do. If you're not clear about this, remember that we want to get a time string and display it. The [getTimeString](#) function delivers a time string, and the [showMessage](#) function displays a string. If we have typed it in correctly, we should be able to display the time by calling the function [tick](#). Type the following statement and press Enter

```
> tick();
```



Ch01\_inset04\_03 time display

The time is displayed. The final thing we need for our ticking clock is a way of calling the tick function at regular intervals so that the clock keeps time. It turns out that JavaScript provides a function called `setInterval` that will do this for us. The first parameter to `setInterval` is a reference to the function to call. The second parameter is the interval between calls in thousands of a second. We want to call the function `tick` every second so type in the statement below and press Enter.

```
> setInterval(tick,1000);
```

This should start the clock ticking. The `setInterval` function returns a value as you can see below:

```
> setInterval(tick,1000);  
< 1
```

The return from `setInterval` is a value which identifies this timer. You can use multiple calls of `setInterval` to set up several timers if you wish. You can use the `clearInterval` function to stop a particular timer:

```
> clearInterval(1);
```

If you perform the statement above you will stop the clock ticking. You can make another call of `setInterval` to start the clock again. At the moment we have to enter commands into the console to make the clock. In the next chapter we'll discover how to run JavaScript programs when a page is loaded so that we can make the clock start automatically.

## Arrow functions

If you look at lots of popular cartoon figures you will notice that some only have three fingers on each hand, not five. You might be wondering why this is. It is to reduce the “pencil miles” for the animators. The first cartoons were made from frames that were all hand-drawn by animators. They discovered that they could make their lives easier by reducing the number of fingers they had to draw. Fewer fingers meant fewer “pencil miles”.



The JavaScript arrow function is a way of reducing the “keyboard miles” of a developer. The character sequence `=>` provides a way to create a function without using the word `function` in the definition.

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

The JavaScript above creates a function called `doAdditionArrow` which is exactly the same as the original `doAdditon`. However, it is much quicker to type in. If the body of the arrow function only contains one statement you can leave off the braces that mark the start and end of the function body. And a single statement arrow function returns the value of the statement, so you can leave off the return keyword too. The code below creates a function called `doSum` which returns the sum of the two arguments.

```
doSum = (p1,p2) => p1 + p2;
```

We could call the `doSum` function as we would any other:

```
let result = doSum(5,6);
```

This would set the value of `result` to 11. You see the true power of the arrow notation when you start using it to create functions that are to be used as arguments to function calls.

```
setInterval(()=>showMessage(getTimeString()),1000);
```

This innocent looking statement repays careful study. It makes our clock tick. In the “Make Something Happen: Console clock” above we used the `setInterval` function to make the clock tick. The `setInterval` function accepts two arguments, a function to call and the interval between each function call. I’ve implemented the function to call as an arrow function containing a single statement which is a call to `showMessage`. The `showMessage` function has a single argument which is the message to be displayed. This is provided by a call to the `getTimeString` function.

## Arrow functions can be confusing

JavaScript is not the first programming language that I've learned. It might not be your first language either. When I was learning JavaScript I found the arrow function one of the hardest things to understand. If you have seen C, C++, C#, or even Python programs before you will know about functions, arguments, parameters and return values already. This means that "traditional" JavaScript functions will be easy to grasp.

But you will not have seen arrow functions before because they are unique to JavaScript. And you can't easily work out what they do from seeing them in code. If you don't know what an arrow function does you will find quite tricky to understand what the creation of `doSum` above is doing. One way to deal with this would be to just regard arrow functions as a little extra provided by the language to make your life easier. If you don't mind doing the extra typing you can create all your programs without using the arrow notation. However, I think you should spend extra time learning how they work so that you can understand JavaScript written by other people.

There is one other aspect of arrow functions that can be confusing. There is a specific behavior that they have involving the `this` reference which it is important to know about. We will cover this in the section "The meaning of this" in Chapter 3.

When we started talking about JavaScript functions we noted that they are used to attach JavaScript code to events. The arrow function makes this very easy to do.

## Make Something Happen

### Arrow function Console Clock

You should already have the sample `Ch01-What_is_the_cloud/Ch01-02_Console_Clock` open. If not, open it. Then open the console tab and use the arrow function above to get the clock ticking. If the clock is already ticking you can reset everything by reloading the example page in the browser.

# What you have learned

At the end of each chapter, we have a "what you have learned" section which sets out the major points covered in the text and poses some questions that you can use to reinforce your understanding.

- A web browser is an application that requests data from a web server in

the form of web pages. The connection between the two applications is provided by the internet.

- Web servers were originally single machines connected to the internet which were owned and operated by the owner of the site. The cloud made computing power into a resource that can be bought and sold. We can pay to have our web sites hosted in the internet. Page requests sent to the address of our site will be processed on machines at our service provider.
- The JavaScript programming language was invented to allow a browser to run program code that has been download from a website. It has since developed into a language that can be used to create web servers and free-standing applications.
- A JavaScript function is a block of code and a header which specifies the name of the function and any parameters that it accepts. Within the function the parameters are replaced by values which were supplied as arguments to the function call.
- When a running program calls a function the statements in the function are performed and then the running program continues from the statement after the function call. A function can return a value using the return keyword.
- When a JavaScript program runs inside the browser it uses an Application Programming Interface (API) to interact with the services the browser provides. The API is provided by many JavaScript functions.
- Operators in JavaScript statements perform an action according to the context established by the operands they are working on. As an example, adding two numbers together will result in an arithmetic addition, but adding two strings together will result in the strings being concatenated. If a numeric operation is attempted with operands that are not compatible the result will be set to the value “not a number”.
- Variables in JavaScript can hold values that indicate specific variable state. A variable that has not been assigned anything has the value “undefined”. A calculated value that is not a number (for example the result of adding a number to a string of text) will have the value NaN – short for Not A Number.
- Modern browsers provide a Developer Tools component that contains a console that can be used to execute JavaScript statements. The Developer Tools interface also lets you view the JavaScript being run inside the page and add breakpoints to stop the code. You can step through individual statements and view the contents of variables.

- A JavaScript function is represented by a function object. JavaScript manages these by reference so variables can contain references to functions. Function references can be assigned between variables, used as arguments to a function call and returned by functions.
- Function expressions allow a function to be created and assigned to a reference at any point in a program. A function expression used as an argument to a function call is called an *anonymous function* because it is not associated with any name.
- The JavaScript API provides a [Date](#) object which can be used to determine the current date and time and also allows date and time manipulation. The API also provides the functions [setInterval](#) and [clearInterval](#) which can be used to trigger functions at regular intervals.
- Functions can be defined using “arrow notation” which is shorter than the normal function definition. This is especially useful when creating functions to be used as arguments to function calls.

To reinforce your understanding of this chapter, you might want to consider the following "profound questions" about the cloud and what we do with it.

What is the difference between the internet and the web?

The internet is the technology that allows computers to communicate. The web is a service that uses the internet to link web browsers and web servers.

What is the difference between the cloud and the internet?

The internet is the networking technology that allows a program running on one computer to exchange data with a program running on another computer. We don't need the cloud to make an internet-based application. We just need two computers with internet connections. The cloud lets you replace a computer connected to the internet with a service that you have purchased from a cloud service provider. The cloud based server will have a network address which is used by the cloud service provider to locate the required service.

How does the cloud work?

A server hosted by a cloud service provider runs an operating system that allows it to switch between processes running services for different clients. At the front of the cloud service is a component which accepts requests and routes them to the process that provides the required service. The computing resources used by each process are monitored so that the services can be billed for the services they have provided.

What is the difference between a function and a method?

A function is declared outside of any objects. We have created lots of functions in this chapter. A method is a function which is part of an object. A [Date](#) object provides a [GetMinutes](#) method that returns the minutes value for a given date. This is called a method because it is part of the [Date](#) object. Methods themselves look like functions. They have parameters and can return values.

What is the difference between a function and a procedure?

A function returns a value. A procedure does not.

Can you store functions in arrays and objects?

Yes you can. A function is an object and is manipulated by reference. You can create arrays of references and objects can contain references.

What makes a function anonymous?

An anonymous function is one which is created in a context where it is not given a name. Let's take a look at the tick function we created for the clock:

```
let tick = function(){showMessage(getTimeString());};
```

You might think that this function is anonymous. However JavaScript can work out that the function is called "tick". If you look at the name property of the function you will find that it has been set to tick. But instead of creating a tick function we might use a function expression as an argument to the call of [setInterval](#):

```
setInterval(()=>showMessage(getTimeString()),1000);
```

The statement above feeds a function expression into [setInterval](#). The function expression does the same job as [tick](#), but now it is an anonymous function. There is no name attached to the function. Note that this statement uses the arrow notation to define the function.

Why do we make functions anonymous?

We don't have to use anonymous functions. We could create every function with a name and then use the name of that function. However, anonymous functions make life a lot easier. We can bind behaviors very tightly to the place they are needed. Also, if we are only ever going to perform a behavior once it is rather tedious to have to invent a function name for it.

Can an anonymous function accept parameters and return a result?

Yes it can.

Are arrow functions always anonymous?

An arrow function is simply a quick way of creating a function definition. Arrow functions can have names.

What happens if I forget to return a value from a function?

A function that returns a value should contain a return statement is followed by the value to be returned. However, if the function contains multiple return statements you might forget to return a value from one of them. The program will still run, but the value returned by the function at that point would be set to “undefined”.

What happens if I don't use the value returned by a function.

There is no need for a program to use the value returned by a function.

What does the let keyword do?

The let keyword creates a variable which is local to the block of code in which it is declared. When execution leaves the block the variable is discarded.

Do JavaScript programs crash?

This is an interesting question. With some languages the program text is carefully checked for consistency before it runs to make sure that it contains valid statements. With JavaScript, not so much. Using the wrong type of value in an expression, giving the wrong numbers of arguments to a function call or forgetting to return a value from function are all examples of program mistakes which JavaScript does not check for. In each of the above situations the program would not fail when it ran, instead the errors would cause variables to set to values like undefined or Not a Number. This means that you need to be careful to check the results of operations before using them in case a program error has made them invalid.

So the answer is that your JavaScript program probably won't crash, but it might display the wrong results.

2

# Begin to Code: Chapter 2 Get Building Apps Into the Cloud and Games in

What you will learn

In the last chapter we discovered the origins of the cloud and ran some JavaScript code in a web page using the browser Developer Tools. We also learned a lot about JavaScript functions how to connect them to events

## the Cloud

In this chapter we are going to take our JavaScript code and put it into the cloud for anyone to access. We are going start by getting the tools we are going to use and then move on to look at the format of the documents that underpin web pages. Then we are going to use JavaScript to add programmed behaviors to pages and discover how we can put our active pages into the cloud.

Rob Miles

Don't forget that you can use the Glossary to look up any terms that seem unfamiliar. Words defined in the glossary will be formatted like **this** in the text. Note that both **this** and **in** are in the glossary.

## Working in the cloud

Before you start to build your applications, you need to find a nice place to work. There is a physical aspect to this. You work best if you are comfortable so a decent screen, nice keyboard and a responsive computer are all great things to have. However, there is also a “logical” element too. You need to find a place to store all the materials that you’re going to generate. You could just store all the files on your computer, but you might lose them if your machine fails. What’s more, if you make the wrong modifications to the only copy of a crucial file you can lose a lot of work very quickly (I have done this many times). So, let’s look at how we can manage our stuff. Starting with git.





The sample repository is exposed by GitHub as a web page, so it contains an index.html file. This is so you can view any of the sample code in your browser without downloading any samples onto your machine. Not all GitHub repositories are web pages, but it is very useful to be able to host a web site in this way. We will be doing this at the end of this chapter.

If we are going to work with repositories, we need to install the git software. This is a free download and there are versions for all types of computers.

## Make Something Happen

### Install Git

First you need to open your browser and visit the web page:

<https://git-scm.com>



Ch02\_inset01\_01 Git Install page

Now follow the installation process selecting all the default options.

## Storing git repositories

You can use the git program on a single computer to manage your work. In this case you will put each repository in a folder somewhere. If you want to store your files in a central location, you can also set up a computer as a “git server”. This is a bit like a “web server for software developers”. You use the git program to send copies of your repositories over the network to the server and have them stored there. You can also “clone” a repository from the git server (this is called “checking out” a repository), work on it and then synch your changes with the original (this is called “checking in”). Git provides user management so individuals can have their own login names and be formed into teams that have access to particular repositories.

A git server makes it easy for programmers in a company to work together. But what if you want anyone in the world to be able to work on your project? Lots of important software used today, including operating systems, network tools and even games are now developed as **open-source** projects. All the software code that comprises these applications is stored openly, frequently in a **git repository** which is accessible to people who want to help with the project. Anyone can check out the repository, make some changes and then submit a “pull request” to the project owners. The project owners can “pull” in a copy of the changes. The changes could be a fix to a problem, a new feature, or even just improved error messages. If the changes check out these are then incorporated into the application which then gets that bit better.

**Figure 2.8** Ch02\_Fig\_02 Sample Repository on GitHub

Figure 2.2 above shows the sample repository as it appears on GitHub. GitHub also provides **organizations** as a way of managing projects. An organization can be created by a user and can contain multiple repositories. I've created one called "Building-Apps-and-Games-in-the-Cloud" for this book. The clock repository is one of several that are held in this organization. If you are storing repositories for a group or a large project which should not be directly associated with a particular GitHub user, you can create an organization to hold those repositories.

The clock repository is public, so anyone can look inside the files. They can even clone the

repository onto their computer, make some changes and send me a pull request. I have also added a license file which sets out what other people can do with the code, so you could regard the clock as a mini Open-Source Project. You can make private repositories which will not be visible to other GitHub users if you need them.

## Programmer's Point

### Open-Source projects are a great place to start your career

You might think that you must become a great programmer before you can start to make a name for yourself in software development. This is not true. You can add value to an open-source project well before you can create complete programs. You can learn a huge amount just by looking at code written by other people. And working out how the internal pieces of a system fit together is very satisfying, even if you don't know how the whole thing works.

An important part of being a successful developer is being able to work well with other people, so being part of an open-source project prepares you well for this. Many developers can also remember what it was like to start out and will be happy to help you improve, as long as you keep your input constructive and focused.

In addition, there are lots of things a project needs that have nothing to do with programming. A large project will need people to test things, write documentation, make artwork, create different language versions and so on. If you've got any of those skills, you could find yourself in high demand just for those. And that alone could open a totally different career path for you.

Many open-source projects (and lots of commercial ones) are hosted by GitHub. Companies rent space on GitHub rather than set up their own git server, but GitHub also offers comprehensive free services for open-source developers. It also offers a good service to individual developers; you can host your own private repositories on GitHub for your personal projects.

I strongly advise you to set up an account on GitHub and start using it for your projects. It will not cost you anything to do this. Whenever I start a new project one of my first thoughts is "what happens if I lose everything?". GitHub is a good answer to that question. If I put things into a GitHub repository they will be as safe as they can be. And if I make good use of the ability to commit changes at regular intervals I can save myself from losing work. Furthermore, once the repository is on GitHub I can invite other people into my project to work on it with them, or make it public so anyone can use it.

You don't need a GitHub account to make use of everything in this book, but if you want to get the most out of the contents you really should make one. It can completely change the way you work. If I want to do anything these days, from organize a party to write a book, I start by creating a GitHub repository for the project.

GitHub is a great place to store your data. However, it is also a great place to network with others. Each repository has a Wiki. This is a space for collaboratively creating documentation. A repository also has issue tracking discussions that people can use to report bugs and request features and you can create and manage projects within the repository.

GitHub also has a final ace up its sleeve. It can host web sites. You can take your web pages, put them into a GitHub repository and then make them available for anyone in the world to see. We will be doing this at the end of this chapter.

## Make Something Happen

### Join GitHub

If you don't want to join GitHub you can skip this section. You can still grab the sample repositories and work on the code but you won't be able to create your own repositories or use GitHub to host websites.

You will need an email address to create a GitHub account. Start by opening your browser and visit the web page: <https://github.com/join>



Ch02\_inset02\_01 GitHub join page

Enter your email address and click "Sign up for GitHub". It takes a few steps, and you have to wait for a message to validate your email address, but eventually you will end up at the dashboard for your GitHub account.

You are now logged in to the GitHub website and can work with your repositories and clone those of other GitHub users. If you want to use GitHub from the browser from another machine you will have to log in to the service on that machine.

You can work with your repositories using the web page interface. You can also enter git commands into your Windows terminal or MacOS console. However, we are going to use the Visual Studio Code program to edit and debug our programs and it can also talk to git and manage our repositories for us.

## Programmer's Point

### GitHub is also a social network.

You can regard GitHub as a social network. If you make something that you think would be useful to other people, you can share it by creating a public GitHub repository. Other GitHub members can download your repository and use it. They can also make changes and send you “pull requests” to signal that they have made a new version you might like.

Members can award repositories “stars” and comment on their contents. There are also lots of discussions going on. This means that GitHub is another place you can start to make a name for yourself. However, just as you should be cautious when you post details on other social network sites, you should also be aware of the potential for misuse that social networking frameworks provide.

Make sure not to give out too much personal information and ensure that anything you put into a public repository does not contain personal data. Don't put personal details, usernames, or passwords into program code that you put on GitHub. Later in the text we will discover how to remove personal information from projects that we want to make public.

## Get Visual Studio Code

Now that we have git installed and working, we are going to install Visual Studio Code which we are going to use to create all our programs. It is free to download and versions are available for Windows, Macintosh and Linux based computers, including the Raspberry Pi. It also supports lots of extensions that can be used to extend its capabilities.

### Make Something Happen

### Install Visual Studio Code and clone a repository

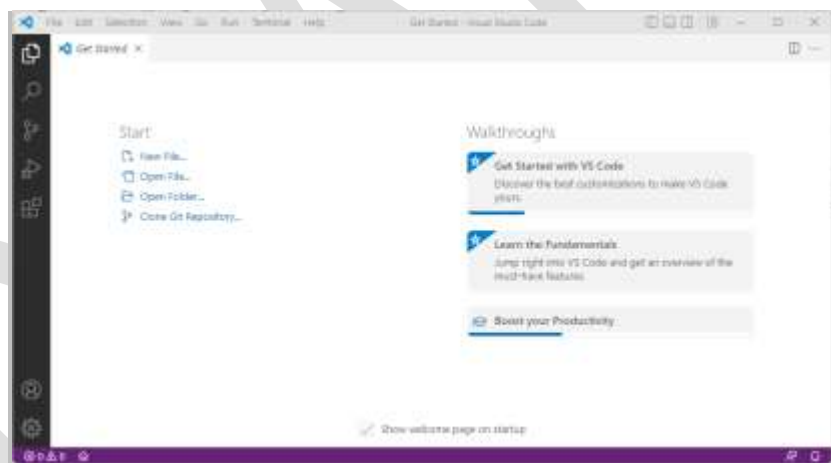
I'm going to give you instructions for Windows. The instructions for macOS are very similar. First you need to open your browser and visit the web page:

<https://code.visualstudio.com/Download>



Ch02\_inset03\_01 Visual Studio Install page

Click the version of Visual Studio Code that you want and follow the instructions to install it. Once it is installed you will see the start page.



Ch02\_inset03\_02 Visual Studio install complete

Now that you have Visual Studio installed the next thing you need to do is fetch the sample files to work on. To do this click the Clone Git Repository link which is about halfway up the left-hand side of the page.

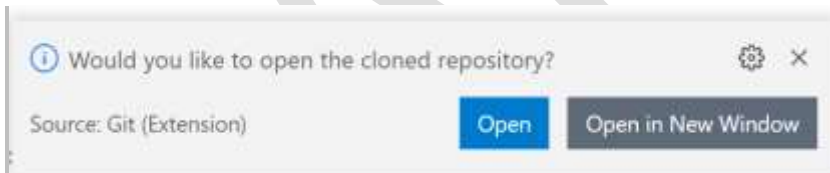


Ch02\_inset03\_03 Clone examples repository

<https://github.com/Building-Apps-and-Games-in-the-Cloud/Building-Apps-and-Games-in-the-Cloud.github.io>

Enter the address of the repository into the dialog box as shown above. Then click the “Clone from URL” button underneath the address. If you click the “Clone from GitHub” button underneath “Clone from URL” you will be prompted to log in to GitHub. This can be useful if you want send local repositories from your machine into GitHub, but you don’t need to do this to just fetch files.

The repository will be copied into a folder on your machine. The next thing that Visual Studio needs to know is the location of that folder. I have a special GitHub folder where I store my repositories. This is not in my OneDrive folder. Since I’m using GitHub to keep my files safe, I don’t need to use OneDrive to synchronize things.



Ch02\_inset03\_04 Clone complete

Once the files have been cloned Visual Studio Code offers you the chance to open the new repository. Click **Open** to do this. You will be asked to confirm that you trust the author. It is best to say yes here. Then Visual Studio will open the repository and show you the contents. On the far left are the tools you can use to work on the repository. Click the top one to select Explorer. The examples are in the code folder, organized by chapter. Click on **Ch02-Get\_into\_the\_cloud** to open the folder and then open the “**Ch02-01\_Simple HTML**” folder and then click the **index.html** file to open it in the editor.



Ch02\_inset03\_05 editing code

This is the first example file in this chapter. We will be looking at it later. Leave Visual Studio Code running, we will be using it again in a moment. The next thing to do is install our first Visual Studio Code extension.

## Install the Live Server Extension

Before we can work with our web pages there is something we can do to make our job much easier. If you think about it, what we are going to do is edit a web page with Visual Studio Code, view the page in our web browser, edit it again and so on. We could do this by repeatedly saving the web page to a file and opening it by hand each time. However, this would be hard work and programmers hate hard work. What programmers do when faced with a problem like this is create a tool that will do the work for them.

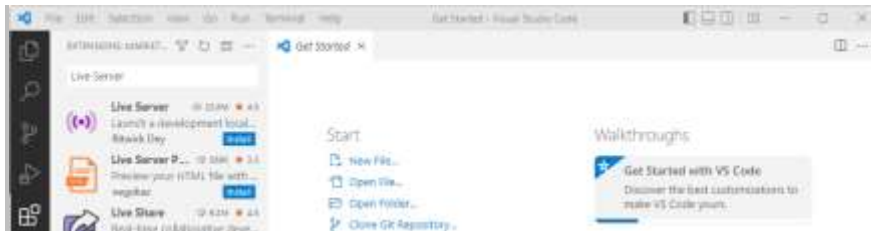
Ritwick Dey is a programmer who solved this problem by creating the Live Server extension for Visual Studio Code. An extension is something you can add to Visual Studio Code to add a new feature. There are thousands of extensions available for download, we are going to use the Live Server extension. Then, when we want to view HTML in our browser we can just press a button to do this.

### Make Something Happen

## Install the Live Server extension

To begin, open Visual Studio Code. Then click the extensions button on the left-hand tool-strip (it is the fifth one down as highlighted below. The Extensions Marketplace opens. Type Live Server into the search box.





#### Ch02\_inset04\_01 Installing Live Server

The marketplace will show all the extensions with this name. Click the Install button on the one written by Ritwick Dey.



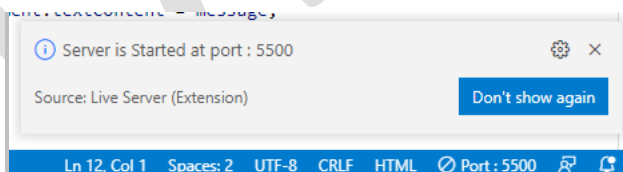
#### Ch02\_inset04\_02 Live Server installed

The extension will be installed and will start up each time Visual Studio Code is loaded. We can test it by using the clock page. If you have not left Visual Studio Code open from earlier, you should open the **"Ch02-01\_Simple HTML"** folder in the **clock** repository and then click the **index.html** file to open it in the editor.



#### Ch02\_inset04\_03 VSC with live server ready

You can see the Go Live button on the bottom left of the screen in the blue border. Click the button and Go Live will now open the index file in your browser. You might see some messages from the firewall on your machine the first time you do this. You should allow Visual Studio Code access to the ports that it needs.



#### Ch02\_inset04\_05 VSC live server starting

You will also see a message from Live Studio telling you the server has started and specifying the network port that it is using. Then the browser opens and displays the page.



Ch02\_inset04\_06 VSC live server web page

Above you can see the page as displayed by the browser. Note that the address of the page starts with 127.0.0.1 which indicates that the page is being hosted by your computer. If you make changes to index.html and then save them the browser will automatically reload the updated page. This is a very useful extension. It has been downloaded more than 23 million times. It turns out that writing extensions for Visual Studio Code is also a great way to make a name for yourself. Next, we need to build an understanding of the contents of the web page.

## How a web page works

We have been using web pages in our browsers for years. In chapter 1 we saw that a web page can contain JavaScript code. Now it is time to dig deeper and consider what makes a web page. We are not going to go into too much detail, there are books specifically about these topics that are much thicker than this one. However, you may think you already know what a web page is, but I'd be most grateful if you would read this section anyway. You may find a few things you didn't know.

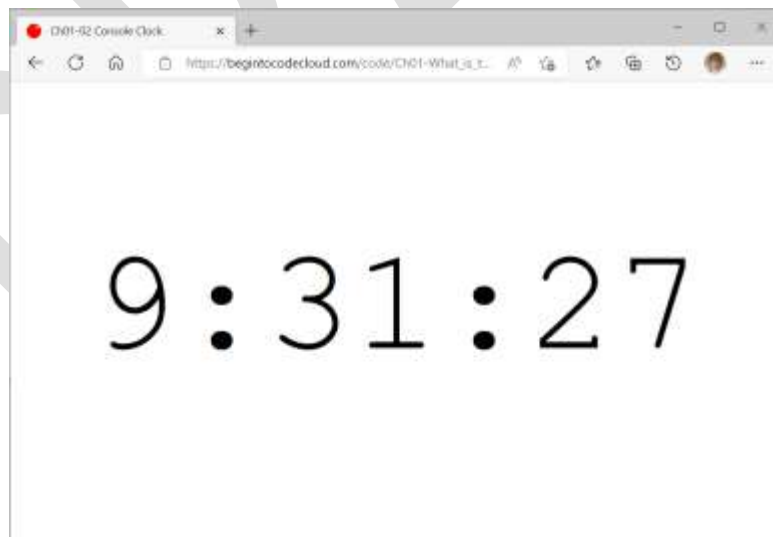


Figure 2.9 Ch02\_Fig\_03 Ticking Clock

A web page is a “logical document”. What do I mean by that? Well, a book is a **physical** document. It exists in the real world. We can read it, add annotations, leave it on the bus and do everything else that you can do with a physical book in the real world. An electronic book (or e-book) is a **virtual** document. It is something created to play the part of a physical book in a virtual environment created by a computer. We can read it, annotate it, and if we delete the file containing the book, we could even manage to lose it. A web page is a **logical** document. It is a thing created by software and hosted on a computer. A web page has no physical counterpart. There is no physical version of a book that contains a ticking clock, as you can see in Figure 2.3 the best we can do is a stationary image.

## Loading a page and displaying it

The starting point for a web page is a file of text which is hosted on a server. When you visit a website, for example [www.robmiles.com](http://www.robmiles.com), the browser looks for a file called `index.html` at that location which contains the start page of the site. The text in this file describes the logical document that makes up the web page. The browser reads the file of text from the server and uses it to build the logical document that is the web page. At this point you might say “Aha! The file of text on the server describes the web page so a web page is just a file of text.” Well, no. A web page could contain JavaScript code that runs when the document is loaded into the browser. That code can change the contents of the logical document and even add new things. The logical document that the HTML describes is held by the browser in a structure called the **Document Object Model (DOM)**. The DOM is a very important part of web programming. Our JavaScript programs will interact with the DOM to display output.

Once the browser has built the DOM it draws it on the display. The browser program then repeatedly checks the DOM for changes and redraws it if any changes are found. This is how the ticking clock we created in chapter 1 works. There is a JavaScript function running which changes something in the document and the browser redraws the page to reflect the changes. We don’t have to do anything to trigger a redraw, our code can change the contents of the logical document and the updated version of the page is displayed automatically.

A logical document can contain images, sounds and videos which are all updated automatically. The initial contents of the document objects are expressed using a language called Hypertext Markup Language or HTML. Let’s look at that next.

## Hypertext Markup Language (HTML)

We can discover what Hypertext Markup Language really is by unpicking its name. Let’s start with Hypertext. Remember that we can call things **logical** to indicate that they have no counterpart in real life. **Hypertext** is a logical version of text. Normal text, whether it is printed or displayed on a screen, is something you read from beginning to end. Hypertext can only be displayed by a computer. This is because hypertext can contain **hyperlinks** which refer to other documents. You can start reading one document, open a hyperlink and be moved to a completely different one, perhaps served by a completely different computer in a different country. When hypertext and

hyperlinks were invented, it was thought cool to put the word hyper in front of them to make them sound impressive. So that is where the name came from. At least, that's what I think.

So, the word hyperlink in HTML means that the aim of the language is to express a page that can contain hyperlinks. HTML was invented to make it easier to navigate reports. Before hyperlinks, if a report contained a reference to another report you would have to go and find that report and open it to read it. After hyperlinks you could just follow a link in the original report. Hypertext was designed to be extensible, so that it would be easy to add new features. The features provided by modern web pages are way beyond any foreseen by Tim Berners-Lee, the inventor of HTML, but the fundamental content of a page remains the same.

The word **markup** refers to the way that an HTML document separates the intent of the author from the content in the page. Let's look at a tiny web page to discover how this works.

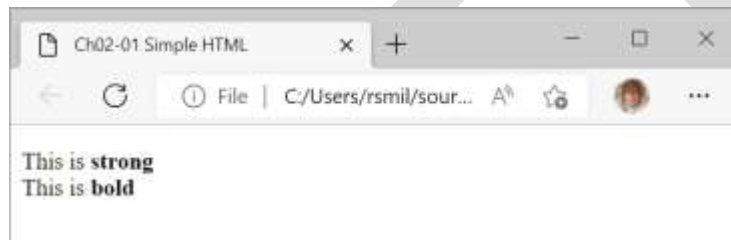


Figure 2.10 Ch02\_Fig\_04 Simple HTML

Figure 2.4 shows a tiny web page. It only contains six words. Let's look at the HTML file behind it and discover the role of markup in creating the page contents:

```
<!DOCTYPE html>
<html lang="en">
<head>
<title>Ch02-01 Simple HTML</title>
</head>
<body>
  <p>
    This is <strong>strong</strong><br>
    This is <b>bold</b><br>
  </p>
</body>
</html>
```

This is the html file that describes the page in Figure 2.4. The most important characters in the file are the < and > characters that mark the start and end of HTML element names. The < and >

are called **delimiters**. They *define the limits* of something. An **element** is a thing in the document that the browser knows how to work with. Elements can have attribute values. These are name-value pairs included in the definition of the element. The `<html lang="en">` element above has an attribute called `lang` which gives the language of the html page. The language code `en` means English.

Elements can be containers. A container starts with the name of the element and ends with the name preceded by a forward slash (/). You can see that the `<title>` element contains the text that is to be used as the title of the web page. You can see this title on the top of the web page in Figure 2.4. The title information is part of the header for the page, which is why it is enclosed in the `<head>` element.

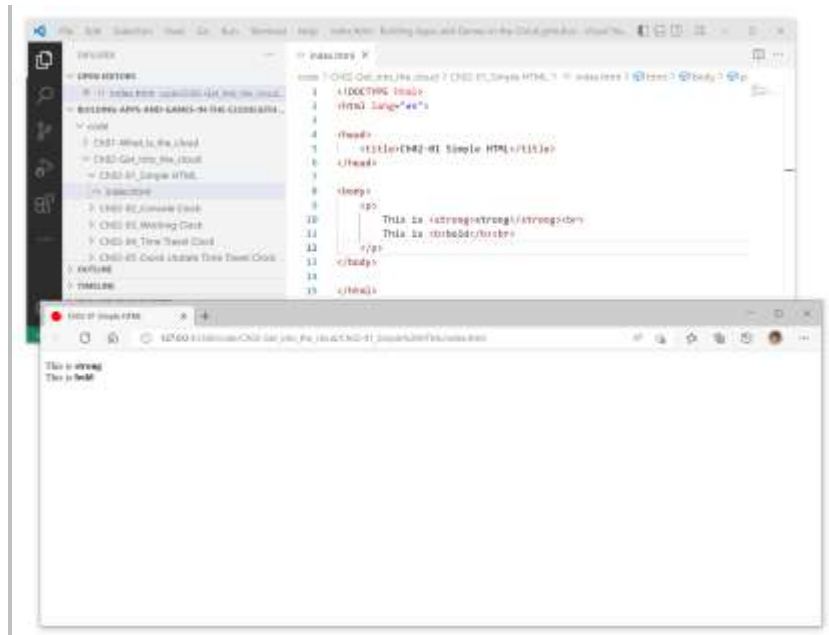
The `<body>` element contains all the elements to be drawn by the browser. The `<p>` element (short for paragraph) groups text into a paragraph. The `<bold>` and `<strong>` elements give formatting information to the browser. However, an element can exist as the starting element name, without another to mark the end of the element. The `<br>` element tells the browser to add a line break in the text. You don't need to add a `</br>` element to a page to mark the end of a line break.

Now we can look at the last word in the phrase "Hypertext Markup Language". The word language means that we are going to use HTML to express things. HTML is very specific (it is being used to tell a browser how to build a logical document) but it is a language.

## Make Something Happen

### Web page editing

If you have been following this exercise you will already have the first sample file open in your browser. If not, use Visual Studio Code to open the "Ch02-01\_Simple HTML" in the examples and then click the **index.html** file to open it in the editor. Then click **Go Live** to open the page in the browser.



Ch02\_inset05\_01 Visual Studio and Browser

Your desktop should now have both Visual Studio and your browser on it, as shown above. Now go back to Visual Studio Code and add the following element into the body element of the page:

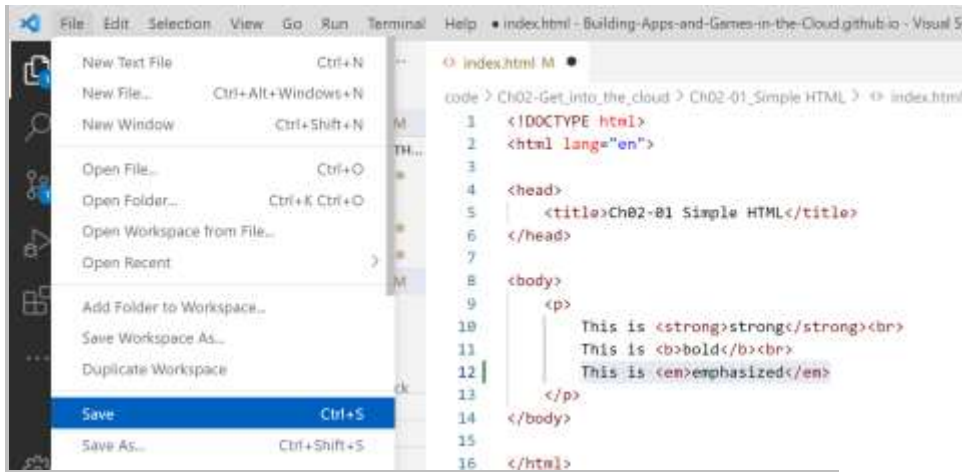
This is <em>emphasized</em>

Visual Studio Code should now look like this:



Ch02\_inset05\_02 web page edits

Now select **File>Save** from the Visual Studio Code menu bar to save the updated file



Ch02\_inset05\_03 edit save

Because you are using Go Live you should see the page in the browser update automatically and display the new text on the page.



Ch02\_inset05\_04 updated page

This is a very nice way to work on web pages. Each time you save your file the web page updates automatically.

## CODE ANALYSIS

### Investigating HTML

If this is your first brush with HTML you may have some questions.

Question: What is the difference between bold and strong?

The HTML file marks one piece of text as bold and another as strong. But in Figure 2.2 they both look the same. You might think that this means that bold and strong are the same thing. However, they are intended to be used for different kinds of text. Text that needs to stand out should be formatted bold. Text that is more important than the text around it should be formatted strong. I would print my name in bold (because I would like it to stand

out). I would print “Do not put your head out of the window when the train is moving” in strong because that is probably more important than the text around it.

Remember that HTML just contains instructions to the browser to display things in certain ways. The browser has a default behavior (to display bold and strong as bold text) but you can change this for your web pages by adding styles, of which more later.

Question: How do I enter a < or a > into the text?

Good question. HTML uses another character to mark the start of a **symbol** entity. The & character marks the start of a symbol. Symbols can be identified by their name. Some useful ones are:

```
&lt; &gt; &amp;
```

You can find a list of all the symbols here: <https://html.spec.whatwg.org/multipage/named-characters.html> You can also use symbols to add emoticons to your pages. Take a look here for the codes to use <https://emojiguide.org/>

Question: What happens if I mis-spell the name of an element?

If the browser sees an element it doesn’t know it will just ignore it.

Question: What happens if I get the nesting of the elements wrong?

If you look at the sample code, you will see that some elements are inside others. The `<p>` element is inside the `<body>` for example. If you get the nesting wrong (for example put the `</body>` element inside the `<p>` the browser will not complain and the page will display, but it might not look how you were expecting.

Question: What does the `<!DOCTYPE html>` element mean?

Good question. The very first line of a resource loaded from a web server should describe what it contains. The resource could contain an image, a sound file, or any number of other kinds of data. The `!DOCTYPE` element is used to deliver this information. The browser doesn’t always use this. A browser will try to display any file of text it is given, but it is very useful to add the information.

Question: How do I put JavaScript into a web page?

You use the `<script>` element to embed JavaScript into a page.

Question: Are there other kinds of markup language?

Yes there are. There is XML (eXtensible Markup Language) which is used for expressing structures of data. There are also lots of others which have been developed for particular applications.

Question: How to I stop the Go Live server?



You might want to stop the Go Live server and open a web page from a different index file. You can do this by restarting Visual Studio Code, but you can also press the close button next to the port number on the bottom right of the Visual Studio Code window.

# Make an active web page

We have reached a pretty powerful position. We have tools that we can use to create and store web resources and we are building an understanding of the way that web pages are structured. Now we are going to take another big step forwards. We are going to discover how a JavaScript program can modify the contents of the document object and change the appearance of the web page. This is how JavaScript programs running in the browser communicate with the user.

## Interact with the document object

### Make Something Happen

### Interact with a web page

If you have been following this exercise you will already have the console clock open in your browser. If Go Live is displaying the page, stop it from running by pressing the close button next to the port number at the bottom of the Visual Studio Code window. If not, use Visual Studio Code to open the “**Ch02-02\_Console Clock**” folder in the examples and then click the **index.html** file to open it in the editor. Then click **Go Live** to open the page in the browser. Now open the Developer Tools and then select the **Elements** tab. This shows us the elements in the document:



Ch02\_inset06\_01 Clock page elements

In the figure above I've resized the Developer Tools part of the screen to give me more space. On the left of the window, you can see the page as displayed by the browser. On the right you can see a display of the contents of the web page. One item has been selected in

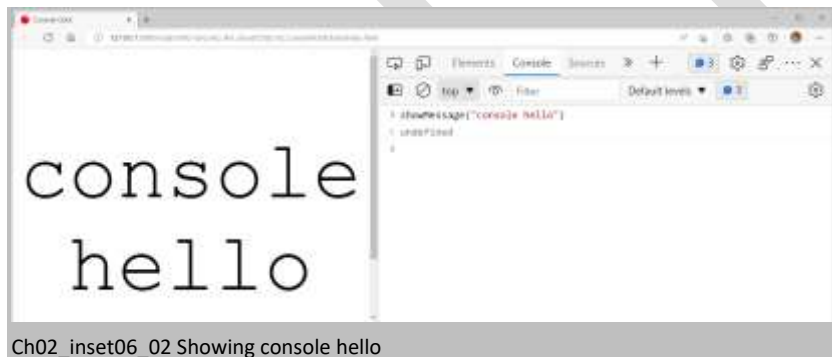
the view. This is a paragraph which has an attribute called `id` which is set to the value `timePar`. This page looks a lot like the source code of the page, but it is actually a view of the elements in the **Document Object Model** (DOM) that was created from the html file.

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

Above you can see the element that is highlighted in the image. The clock program changes the content of this element to display the time. The program looks for the element with an `id` attribute of `timepar` and then displays the time in this element. In the last chapter we used the `showMessage` function to display a message on the page. Lets look at how it works.

```
showMessage("console hello");
```

This is how we call `showMessage`. We give it an argument which is the message to be shown on the screen.



Ch02\_inset06\_02 Showing console hello

You can see the effect of the call of `showMessage` above. The message is displayed because the browser redrew the document and the data in the document object has changed. Now let's take a look at the `showMessage` function itself.

```
function showMessage(message) {  
  let outputElement = document.getElementById("timePar");  
  outputElement.textContent = message;  
}
```

You can see the function above. It contains just two statements. The first statement finds an HTML element in the document and the second statement sets the `textContent` property of this element to the message that was supplied as a parameter. That sounds simple enough, especially if we say it very quickly. Or not. Let's break it down into a series of steps and enter them into the console. Select the Console tab in the Developer Tools window and enter the following statement:

```
let outputElement = document.getElementById("timePar");
```

This statement finds the element in the document that displays the output (that's what

`getElementById` does). The `getElementById` method is provided by the DOM to search for elements by name. The element we have asked it to look for is a paragraph that has an `id` property set to `timePar`. The value returned by `getElementById` is assigned to a variable called `outputElement`. Press enter to perform the statement.

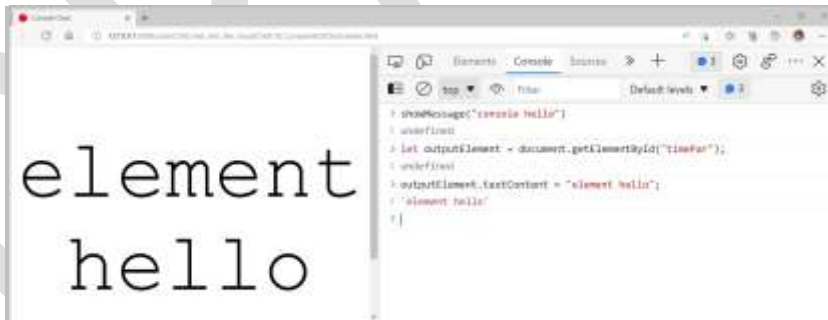


Ch02\_inset06\_03 Finding the output element

We know that `let` does not return a value, so the console displays "undefined". The display has not changed because all we have done is obtain a reference to a paragraph in the document. We need to change the text in that paragraph. Enter the following statement:

```
outputElement.textContent = "element hello";
```

This statement uses the `outputElement` reference that we have just created. It puts the string "hello" into the `textContent` property of the element that `outputElement` refers to. Press enter to see what that does.



Ch02\_inset06\_04 element hello

The page now shows the message "element hello" because the `textContent` property of the `outputElement` is now "element hello". The `showMessage` function performs these two steps and sets the `textContent` property to a parameter it has been supplied with, so we can use `showMessage` to show any message we like.

## CODE ANALYSIS

# Documents objects and JavaScript

The way that JavaScript programs can interact with document is very powerful. But it can also be very confusing. You might have some questions.

Question: Why is the text that we are displaying so large?

This is a good question. All the other text on the web pages we have made has been tiny. But the time message in our clock is huge, and in a different font. How does this work? It works because we are using a feature called a **stylesheet**. Let's take a look at the HTML that defines the paragraph containing the clock output:

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

This is a paragraph that initially contains "0:0:0". It has the id `timePar` (this is how our JavaScript finds the paragraph). It also contains another attribute. It has a `class` attribute which is set to the value of `clock`. The class attribute tells the browser how to display an item when it is drawn. Alongside the HTML file containing the clock web page is a field which defines the styles for this page. There is a link element in the `head` element for the clock web page that tells the browser which stylesheet file to use.

```
<link rel="stylesheet" href="styles.css">
```

This document is using the `styles.css` file. Inside this file you can find the following style definition:

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

This style information defines the clock **class**, which can then be assigned to elements in the web page. It sets the size, the font, and the alignment. When a web page is loaded the browser also fetches the stylesheet file that goes with it. You can use stylesheet files to separate the formatting of a page from the code that creates it. If the designer and the programmer agree on the names the classes to use the designer can create the styles and the programmer can create the software.

It is possible for a running program to change the style class of an element. You see this happen when invalid items on a web form are highlighted in red. What has happened is that the JavaScript processing the form has changed the style class of an element to one which has a color setting of red.

Question: What if we try to find a document element that is not there?

```
let outputElement = document.getElementById("timeParx");
```

The statement above is valid JavaScript and is intended to get a reference to the element with the id `timePar`. However, it will not do what we want because we have mis-typed the id and there is no element in the web page with the id `timeParx`. When the statement runs the `getElementById` function returns the value of `null`. This means that `outputElement` is set to `null`. The value `null` is another JavaScript special value. We've seen ones called `nan` (not a number) and `undefined` (I have not been given a value). The value of `null` is how `getElementById` can say "I looked for it, but I couldn't find it – therefore I'm giving you a reference back that explicitly means that it was not found". If a program tries to set a property on a `null` reference we get an error that stops the program:

```
> let outputElement = document.getElementById("timeParx");
< undefined
> outputElement
< null
> outputElement.textContent = "hello";
Uncaught TypeError: Cannot set properties of null (setting 'textContent')
    at <anonymous>:1:27
```

Ch02\_inset07\_01 null reference error

Above you can see the effect of trying to use a `null` reference. The message in red means that the program would stop at this point. In other words, any statements after the one that tries to set the `textContent` property on a `null` reference would not be performed. Later we will discover how we can detect `null` references and deal with statements that fail like this.

Question: What if we try to use an element property that is not there?

```
outputElement.textContentx = "hello";
```

The statement above is also legal JavaScript. But it won't display the message hello. This is because it sets a property called `textContentx` rather than the correct `textContent` one. What happens next is really very interesting. You don't get hello displayed because you've written to the wrong property. Instead, JavaScript creates a new property on the `outputElement` object which is called `textContentx`. It then sets the content of this property to "hello". This is a powerful feature of the JavaScript language. It means that code running in the web page can attach its own data to elements on the page. We will explore this feature later in the book.

Leave this page open because we are going to be working on it in the next "Make Something Happen".

## Web pages and events

We are going to make a web page that contains active content which runs when the page is

loaded. The previous pages have contained JavaScript functions, but we have had to run them from the Developer Tools console. Now we are going to give them control to make truly interactive web sites. To do this we are going to bind a function to the event that is fired when a web page is loaded by the browser. In chapter 1 in the section “JavaScript Heroes: Functions” we saw how we can pass an event generator a function reference so that the function is called when the event occurs. We used it to make a ticking clock. The clock contained a function `tick` which could get the time and display it.

```
function tick(){
    let timeString = getTimeString();1
    showMessage(timeString);2
}
```

You can see the `tick` function above. I’ve expanded it slightly from the version in chapter 1 to make it clear how it works. The first statement in the function gets the time into a string called `timeString` and the second statement shows it on the screen. Each time `tick` is called it will show the latest time value. To make the clock display update continuously we need a way of calling the tick function at regular intervals. We can use the `setInterval` function to do this:

```
setInterval(tick,1000);
```

The `setInterval` function is called with two arguments. The first argument is a reference to `tick`, and the second argument is the interval between calls, in this case 1000 milliseconds. This will make our clock tick every second. The `setInterval` function causes an event to be created every second, so the clock will update every second. What we need next is a way of calling this function to start the clock each time the clock web page is loaded.

```
function startClock(){
    setInterval(tick,1000);
}
```

---

<sup>1</sup> *Get the time*

<sup>2</sup> *Display the time*

Above you can see a function called [startClock](#). If we can find a way of calling this function when the clock web page is loaded we can make a clock that starts when the page is loaded. When we began learning JavaScript in chapter 1 we saw the importance of learning the Application Programming Interface (API) that provides functions you can perform. Now we are starting to see the importance of another kind of interface; the one provided by properties of elements in the web page itself. We've seen how we can add properties to elements in an HTML document. Each element supports a set of properties. Some of the properties can be bound to snippets of JavaScript.

```
<body onload="startClock();">
```

The body element above now has an attribute called [onload](#) which is the string "startClock();". Properties with names that start with the word "on" are event properties that will be triggered when the event occurs. The [onload](#) event is triggered when the body of a page is loaded. When the page is loaded the string of JavaScript is performed by the browser in just the same way as the browser performs commands that we have typed into the console.

## Make Something Happen

### Make a ticking clock

If you have been following this exercise you will already have the console clock open in your browser. If not, use Visual Studio Code to open the "**Eg 01 Console Clock**" folder in the **clock** repository and then click the **index.html** file to open it in the editor. You are going to add some functions to the JavaScript code in the web page, so scroll to that part of the document.

```
function tick(){
    let timeString = getTimeString();
    showMessage(timeString);
}

function startClock(){
    setInterval(tick,1000);
}
```

These are the two functions that make the clock work. Type them into the [index.html](#) page inside the `<script></script>` part of the page, near the two existing functions.

Now we need to modify the [body](#) element to add the [onload](#) attribute that will call the [startClock](#) function. Navigate to line 10 in the file and modify the statement as follows:

```
<body onload="startClock();">
```

Now the body element has an `onload` attribute which will call the `startClock()` function when the page is loaded. Now press the Go Live button to open the page in the browser. You should see the clock start ticking.

## CODE ANALYSIS

### Events and web pages

Events are great fun, but you might have some questions about them:

Question: My clock is not ticking. Why is this?

There are a number of reasons why your clock might not tick. If you mis-spell the name of any of the functions they might not be called correctly. For example, if you call the starting function `StartClock` (with an upper-case S at the beginning) then this will not work, because the onload event is expecting to call a function called `startClock`. If you get completely stuck, head over to the **Eg 02 Working clock** folder where there is a working version of the clock.

Question: What is the difference between an attribute and a property?

Good question. A property is associated with a software object. We have seen how objects can have properties that we can access when our programs run. For example, we looked at the name property of a function object in chapter 1 in the section Make Something Happen - Fun with Function Objects. An attribute is associated with an element in a web page. The `body` element can have an `onload` attribute.

Question: Can you run more than one function when a page loads?

You can do this, but you wouldn't do it by adding multiple onload attributes. Instead you would write a single function that runs all the functions in turn, and connect that to the onload event.

## Making a time machine clock

Now that we know how to make a clock, we are going to make a clock that lets us travel through time. Sort of. In chapter 1 in the section Make Something Happen - Console Clock we noticed that the `Date` object provides methods that could be used to set values in a date as well as read them back. If we use this to add 1000 minutes to the minute value, the `Date` object will work out the date and time 1000 minutes into the future.



```
let d = new Date();3  
let mins = d.getMinutes();4  
let mins = mins + 1000;5  
d.setMinutes(mins);6
```

The code above shows how this would work. The first statement creates a variable called `d` that refers to an object containing the current date. The second statement creates a variable called `mins` that holds the number of minutes in the date stored in `d`. The third statement adds 1000 to the value of `mins`. The fourth statement sets the minutes value of `d` to the value in `mins`. This moves the date in `d` 1000 minutes into the future. The `Date` object sorts out the date value, updating the hours and even the day, month and year if required.

We can use this to make a time travel clock which is fast or slow by an amount that we can nominate. At some times of the day, perhaps the morning, we can make the clock go fast so we are not late for anything. At other times, perhaps when it is time to go to bed, we can make the clock go slow, so we can go to bed a little later.



**Figure 2.11** Ch02\_Fig\_05 Time Travel Clock

Figure 2.5 above shows how it would be used. The user clicks the buttons to select a fast clock, a

---

<sup>3</sup> Make *d* refer to a new *Date* object

<sup>4</sup> Extract the minutes from the date

<sup>5</sup> Add 1000 to the minutes value

<sup>6</sup> Set the minutes in the future

slow clock, or a normal clock.

## Add buttons to a page

The first thing we need to do is add some buttons to the page for the user to press. We do this with the `button` element.

```
<p>
  <button onClick="selectFastClock();">Fast Clock</button>
</p>
```

Above you can see a paragraph that contains a `button`. The button encloses the text that will appear on the button on the page. The button element can have an `onclick` attribute which contains a string of JavaScript to be performed when the button is clicked. When this button is clicked a function called `selectFastClock` is called. Now we need to create some code to put inside this function that will make the clock five minutes fast when it runs. We can do this by creating a **global** variable.

## Share values with global variables

Up until now every variable we have created has been used inside a function body. We have used `let` to create the variable. A variable declared with `let` ceases to exist when the program execution exits the block in which the variable was declared. So when the function completes the variable is discarded. Most of the time this is just what you want. It is best if variables don't "hang around" after you have finished with them. I'm very partial to using a variable with the identifier `i` for counting. This is because I am a very old programmer. However, I don't want an `i` used in one part of the program to be confused with one used somewhere else. I like the idea of a variable disappearing as soon as the program leaves the block where it was declared.

However, in the case of the minutes offset value we want to share the value between functions. The `minutesOffset` mustn't disappear when the program exits a function where it is used. We can't declare `minutesOffset` in just one function, we must declare it so that it can be shared between all functions. We must make it **global**.

```
var minutesOffset = 0;
```

The variable `minutesOffset` above is not declared inside any code block. It is declared outside all

the functions. It is also declared using `var` rather than `let`. This means that the variable can be used in any of the functions that follow it. The value in the variable will be shared by all the functions, which is exactly what we want. If we change the value in `minutesOffset` the next time the clock is updated by the tick `function` it will draw the new time.

#### Programmer's Point

### Global variables are a necessary evil

When you make a variable global by declaring it outside any function, you lose control of it. What do I mean by this? Well, suppose I'm working with a bunch of programmers, each of them writing some of the functions in my JavaScript application. If I declare all the variables in my functions by using `let` I can be sure that those variables can only be changed by me. I can also be sure that I won't change any values in other functions.

However, if a variable is made global it is possible for code in any of the functions view it and change it, which might lead to mistakes and makes the program less secure.

Some things must be global. It would be very hard to make the clock work without creating a global variable called `minutesOffset`. But when you write code you should start by making the variables as local as possible (by declaring them using `let`) and then make things global if you have a need for this.

JavaScript give you control of variable visibility. We will discover how to do this in the next chapter in the section "JavaScript Heroes: `let`, `var` and `const`".

```
function selectFastClock() {  
  minutesOffset = 5;7  
}
```

Above you can see the code for the `selectFastClock` function. When the function runs it sets the variable `minutesOffset` to 5. The contents of the `minutesOffset` variable are added to the minutes value when the time is displayed.

---

<sup>7</sup> *Set the minutes offset to 5*

```
function getTimeString() {
    let currentDate = new Date();8
    let displayMins = currentDate.getMinutes()
                        + minutesOffset;9
    currentDate.setMinutes(displayMins);10
    let hours = currentDate.getHours();
    let mins = currentDate.getMinutes();
    let secs = currentDate.getSeconds();

    let timeString = hours + ":" + mins + ":" + secs;11
    return timeString;12
}
```

Above is a modified version of the `getTimeString` function. This adds the value of `minutesOffset` onto the minutes in the time string. This means that when the fast button is clicked the clock will display the time five minutes into the future. The page also has button handlers for `selectSlowClock` and `selectNormalClock` which set the value of `minutesOffset` to the appropriate values. The complete HTML file for the Time Travel Clock is shown below. You can view the code running in the example **Eg 03 Time Travel Clock**.

```
<!DOCTYPE html>
<html>

<head>
    <title>Time Travel Clock</title>
    <link rel="shortcut icon" type="image/x-icon" href="favicon.ico">
    <link rel="stylesheet" href="styles.css">
</head>

<body onload="startClock();">
    <p id="timePar" class="clock">0:0:0</p>

<p>
```

---

<sup>8</sup> *Get the date*

<sup>9</sup> *Calculate the new minutes*

<sup>10</sup> *Set the new minutes value*

<sup>11</sup> *Build the time string*

<sup>12</sup> *Return the time string*

```
<button onclick="selectFastClock();">Fast Clock</button>
</p>
<p>
  <button onclick="selectSlowClock();">Slow Clock</button>
</p>
<p>
  <button onclick="selectNormalClock();">Normal Clock</button>
</p>

<script type="text/javascript">

  var minutesOffset = 0;

  function selectFastClock() {
    minutesOffset = 5;
  }

  function selectSlowClock() {
    minutesOffset = -5;
  }

  function selectNormalClock() {
    minutesOffset = 0;
  }

  function tick() {
    let timeString = getTimeString();
    showMessage(timeString);
  }

  function startClock() {
    setInterval(tick, 1000);
  }

  function getTimeString() {
    let currentDate = new Date();
    let displayMins = currentDate.getMinutes() + minutesOffset;
    currentDate.setMinutes(displayMins);
    let hours = currentDate.getHours();
    let mins = currentDate.getMinutes();
    let secs = currentDate.getSeconds();
    let timeString = hours + ":" + mins + ":" + secs;
    return timeString;
  }
}
```

```

function showMessage(message) {
  let outputElement = document.getElementById("timePar");
  outputElement.textContent = message;
}
</script>
</body>

</html>

```

## CODE ANALYSIS

### Time Travel Clock

You may have some questions about the time travel clock:

Question: Can we make the display update immediately after a button is pressed?

There is a problem with the time travel clock. It takes a while to “catch up” when you click a button. You have to wait up to a second before you see the time change to reflect a new offset value. We can fix this by making the [selectFastClock](#), [selectSlowClock](#) and [selectNormalClock](#) functions call the [Tick](#) function once they have updated the offset.

```

function selectFastClock() {
  minutesOffset = 5;
  tick();
}

```

Now, when the user clicks the button the clock updates instantly. You can find this version in the example **Eg 04 Quick Update Time Travel Clock**

Question: Can we make the clock display change color to indicate whether the clock is fast or slow?

Yes we can. We could have a different style class of the display paragraph for each of the different clock options.

```

.normalClock, .fastClock, .slowClock {
  font-size: 10em;
  font-family: 'Courier New', Courier, monospace;
  text-align: center;
}

.normalClock{

```

```

    color: black;
}

.fastClock {
    color: red;
}

.slowClock {
    color: green;
}

```

Above you can see a stylesheet that creates three styles, [normalClock](#), [fastClock](#) and [slowClock](#). You can see which settings are shared by all the styles and which just set the specific colors. The fast clock is displayed in red and the slow one in green. We can now set the style class to the appropriate style when the selection button is pressed.

```

function selectFastClock() {
    let outputElement = document.getElementById("timePar");
    outputElement.className = "fastClock";
    minutesOffset = 5;
    tick();
}

```

An HTML element has a [className](#) property which is set to the name of the class. We can change the style class by changing this name. The [selectFastClock](#) function sets the [className](#) for the [outputElement](#) to "fastClock" so that the [fastClock](#) style class is used to display it. So the text now turns red when the clock is fast.

Question: Is there a way to write this program without using a global variable?

At the moment the time travel clock uses a global variable called [minutesOffset](#) to determine whether the clock is fast or slow. Global variables are something to be avoided if possible, but how can we do this? It turns out that we can.

We've just made a modification that changes the [className](#) property of the time paragraph to select a different display style depending on whether the clock is fast, slow or normal. We can use the value of the [className](#) property on the time paragraph to set the time offset as well.

```

function getMinutesOffset(){
    let minutesOffset = 0;
    let outputElement = document.getElementById("timePar");
    switch(outputElement.className) {
        case "normalClock": minutesOffset = 0;
        break;
        case "fastClock": minutesOffset = 5;
        break;
        case "slowClock": minutesOffset = -5;
        break;
    }
}

```

```
    return minutesOffset;
}
```

The function `getMinutesOffset` above uses the JavaScript **switch** construction to return an offset value which is 0 if the `className` property of the `timePar` element is `normalClock`, 5 if the `className` is `fastClock` and -5 if the `className` is `slowClock`. This can be used in `getTimeString` to calculate the time to be displayed.

```
function getTimeString() {
    let currentDate = new Date();
    let minutesOffset = getMinutesOffset();
    let displayMins = currentDate.getMinutes() + minutesOffset;
    currentDate.setMinutes(displayMins);
    let hours = currentDate.getHours();
    let mins = currentDate.getMinutes();
    let secs = currentDate.getSeconds();
    let timeString = hours + ":" + mins + ":" + secs;
    return timeString;
}
```

This version gets the value of the minutes offset and then uses it to create a time string with the required offset. There is now no need for a global `minutesOffset` variable.

This is a very good way of solving the problem. The setting is held directly on the element that will be affected by it. There is also no chance that the color of the display can get out of step with the `minutesOffset` value. This version is in the example folder **Eg 06 No Globals Clock**

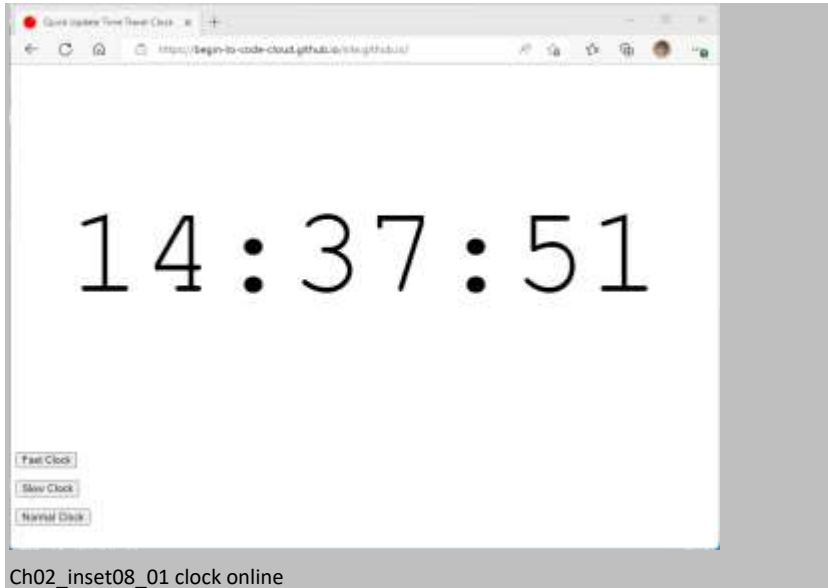
## Host a website on GitHub

You now have something you might like to show off to the world. What better way to do this than putting it on a website for everyone to see? Then anyone who wants a time travelling clock can just go to your site and start it running. One way to do this is to create a website repository on GitHub. To do this you must have a GitHub account. You can only host one website on your account, but the site can contain multiple pages with links between them. We are going to start with a really simple web site which just contains the time travel clock we have just made. If you want to experiment with styles you can change the color, size and font of the text in the clock. In the next chapter we'll discover how to add images to pages too.

**Make Something Happen**

**Host a web page on GitHub**





Above you can see the endpoint of this exercise. This shows the clock program we have just created running in a website hosted by GitHub. Covering the steps to get this would take more pages than would fit in this chapter. You need to create an empty repository, use Visual Studio Code to clone the repository onto your PC, add the clock files to the repository, check in the changes and then synchronize the changes from the PC up to the repository. Then you just need to configure GitHub to tell it which part of the repository to share on the web and you have your new web page. The good news is that you won't be doing this very often. The better news is that I've made a step-by-step video that you can watch that takes you through the process.

Authors note: I tried to make a guided section for this, but it got much too large and convoluted. My thinking is that if they want to do this they'd be happy to watch a video.

## What you have learned

This has been a very busy chapter. We've covered a lot of ground. Here is a recap plus some points to ponder.

- Git is a tool that was created to make it easier to work on large projects. It organizes units of work into repositories. A repository is a folder full of files plus a special folder managed by the git tool to track changes by making copies of changed files. You "commit" changes to the repository at which a snapshot is taken of their contents. You can return to the snapshot at any time. You can also compare the snapshot with the current

files.

- Git can be used on one machine by one person, or a git server can be set up for network access to repositories by several people. Git also provides a means of resolving changes to the same file by multiple people.
- GitHub is a cloud-based service that hosts git repositories. Users can take copies of (clone) repositories, work on them and then check them back over the network. A repository can be private to a particular user or public. Public repositories are the basis of open-source projects which have managers to accept contributions and then commit them after testing. A GitHub repository can be exposed as a web page, making GitHub a good way to host a simple web site.
- Git (and by extension GitHub) support can be added to software tools which can then make use of repository storage and management. The Visual Studio Code integrated development environment (IDE) works in this way. We can check repositories in and out as we work on them.
- Visual Studio Code also provides an extension mechanism which can be used to add extra features. The “Live Server” extension allows you to deploy websites on your PC so that you can test them.
- A web page is expressed by a file of text containing Hypertext Markup Language (HTML). The contains elements on the page that will be drawn by the browser when the page is displayed. The names of the elements are distinguished from text to be displayed by the use of < and > characters to delimit the element names, for example <head> is how the start of the header element would be expressed. The end of the header is expressed by an element containing the name preceded by a forward slash: </head>. The <br> element (line break) does not need a corresponding </br> element.
- HTML elements can be nested. The <body> element contains all the elements which are to be displayed in the body of a web page.
- Html elements can have attributes which give information about the element. Attributes are added as named values in the definition of the element. The HTML <p id="timePar"> marks the start of a paragraph. This paragraph has an id attribute which is set to the value timePar.
- Elements in an html document can be given a class attribute which maps back to a style definition in a stylesheet file which is loaded by the browser. The source HTML file specifies the stylesheet file location using a link element to the head of the document.

- The browser uses the HTML file to create a Document Object Model (DOM). The DOM is a software object that describes the web page structure. The DOM contains references to objects that represent the elements described in the HTML Page.
- Element objects in the DOM contain property values which are mapped onto the attributes assigned in the HTML. For example, the `class` attribute on an element in an HTML document is mapped onto the `className` property in the element object in the DOM. This makes it possible for JavaScript programs to change the values of properties and their appearance on the page. We used this ability to change the color of the text in the time travel clock.
- You can use the Elements tab in the Developer Tools for the browser to look at the elements in the DOM.
- An element in an HTML document can be given an id attribute that allows a JavaScript program to find it in the DOM. The method provided by a document that will get an element by its id is called, not surprisingly, `getElementById`. If the method can't find an element with the requested id it will return a null reference.
- You can change the `textContent` property of a paragraph element by assigning a string to it. The element will then display this new text on the web page.
- Some HTML elements can generate events. An event is identified by a name (usually starting with the word on) and contains a string of JavaScript code to be performed when the event occurs. The `body` element can have an `onload` attribute that specifies JavaScript to run when a web page is loaded by the browser.
- A web page can contain button elements that generate events when the user clicks on them. When the user of the web page clicks the button a JavaScript function can be called to respond to this event.
- GitHub can be made to host web pages as well as store repositories.

To reinforce your understanding of this chapter, you might want to consider the following "profound questions".

What is the difference between Git and GitHub?

Git is the program that you run to manage your repositories. GitHub is a cloud based service that hosts repositories and is accessed using the git program.

Can I use Git on my own machine?

Yes you can. You can use it to manage a repository stored on your machine. You can also set up your own Git server and use it to store private projects on your home network.

Will I ever use the git program directly?

I try to avoid doing this too much. Most of the time Visual Studio Code will hide a lot of the complexities of git, but every now and then, particularly if you work in group projects, you might find that you have to type in a git command to fix something.

Are there different kinds of Open-Source projects?

Yes there are. This is controlled by the terms of the license agreement assigned to a particular project. It is worth reading through these to find out what permissions you can give people on the projects that you make open source. You should also check when you use an open-source project that you are not breaking any of their conditions.

Why is HTML called a markup language?

HTML is used to express the design of a view of a web page. In the days before computers a printer would write instructions on the original copy of a document they were printing. The instructions would specify the fonts and type sizes for the text to be printed. This was called “marking up”. HTML can be used to express how text is to be formatted and so it was given the name markup.

What would happen if I just gave a browser a file of text?

The browser tries very hard to display something, even if it doesn't look like a proper HTML document. It would display a file of text as a single line however, as it ignores line feeds and reduces multiple spaces to 1. If you want to display separate lines and paragraphs you have to add the appropriate formatting elements.

Where does the browser store the Document Object Model?

The Document Object Model is stored in the memory of the computer by the browser when a web page is loaded. A software object provides data storage and methods that are used to interact with the data stored in the object. The HTML file sets the initial elements in the object and their initial values.

What is the difference between HTML and HTTP?

HTML is a way of expressing the content of a web page. HTTP (Hypertext Transport Protocol) is all about getting that page from the server to the browser. The browser uses the HTTP protocol to get resources and the server delivers them. The browser sends an HTTP get request (which actually includes the word GET). The server finds the resource and sends it

back. A get request always returns a status value. A status of 200 means “all is well, here is the data” whereas 404 is the infamous “file not found” error that has become so notorious that it now appears on T-shirts.

Is HTML a programming language?

No. A programming language expresses a solution to a problem. Solving a problem may involve making decisions and repeating behaviors. HTML does not have constructions for doing either of these things. HTML is used to express the contents of a logical document, not to solve problems.

What happens if a JavaScript program changes a visible property on an element very quickly?

We have seen that a JavaScript program can display messages in the browser by updating a property on an element in the document object. We set the `textContent` property on a paragraph to update the clock. If we do this very quickly the browser will not be able to keep up. The browser updates at a particular rate, usually 60 times a second. There is a way that code in a web page can get an event each time that the browser wants to update the page. We will be using this in the last chapter of this text when we look at games.

Does every element on a web page need to have an id attribute?

We added an `id` attribute to the time paragraph on the web page so that the clock program could find that paragraph and change the text on it when the clock ticks. We only need to add an `id` to the elements that the JavaScript code needs to find.

What happens if an event function gets stuck?

Events can be assigned to JavaScript functions by attributes in web pages. We have seen how the `onload` attribute of the `body` element lets us call a JavaScript function when a page loads. But what happens if the function never returns? We’ve all had that experience where you visit a web site and your browser stops. This can be caused by a stuck JavaScript function which is triggered by an event in the web page. If you write a function that never returns (perhaps because it gets stuck in a loop) and then call this from an `onload` attribute you will find that the web page will never complete loading. If a function called from `setInterval` never returns the web page will not stop immediately, but it will progressively slow down as the computer memory fills up with more and more untermiated processes. If a JavaScript function assigned to a button gets stuck the user will not be able to press any other buttons until that function returns.

What is the web address of a page hosted by GitHub web server?

It is an address made from the GitHub username and the name of the repository that holds the site files. However, if you register your own domain name you can configure GitHub to use it. In other words, you can set the address of the page to any domain that you can get hold of.

# 3

## Chapter 3 Make an active site

### What you will learn

We now know how to create JavaScript applications and deploy them into the cloud as part of HTML formatted web pages. We have seen how a JavaScript code can communicate with the user by changing the properties of document elements which are held in “document object model” (DOM) which is created by the browser from the original HTML. We did this by making a clock which ticked when our JavaScript code changed the `textContent` property of a paragraph that displays the time. We can also deploy our web pages and their applications into the cloud so that they can be used by anyone with a web browser.

In chapter 2 we also discovered how a program can receive input from the user in the form of button presses, in this chapter we are going to discover how a JavaScript program can accept numbers and text from the user and store their values between browsing sessions. Then we are

going to move on and start writing code that can generate web page content dynamically. And on the way we are going to learn about a bunch of JavaScript heroes.

Don't forget that you can use the Glossary to look up unfamiliar things and find the cloud meaning of things you thought were familiar.

## Get input from a web page

You might find it strange, but people seem to quite like the idea of our time travel clock. However, like most people who like something you've done, they also have suggestions to make it better. In this case they think it would be a good idea to be able to set the amount the clock is fast or slow. We know that web pages can read input from the user. We have been entering numbers, text and passwords into web pages ever since we started using them. Let's see how we can do this to make an "adjustable" clock.



**Figure 3.12** Ch-03\_Fig\_01 Adjustable Clock

Figure 3.1 above shows what we want. The time offset is entered as an input at the bottom left of the page. The entered value is set when the "Minutes offset" button is pressed. The user has entered 10 and pressed the button. Now the clock is now 10 minutes fast.

## The html input element

The first thing we will need is something we can display on the web page for the user to enter text into. HTML provides the `input` element for this:

```
<input type="number" id="minutesOffsetInput" value="">
```

The html statement above creates an `input` element. The element has been given three attributes. The first attribute specifies the type of the input. This has been set to “number” which tells the browser to only accept numeric values in this input. The second attribute is an `id` for the element. This will allow the JavaScript program to find this element and read the number out of it. The third element is the initial value of the input, which we have set as an empty string.

The next thing we need is a button to press to set the new offset value. This will call a function to set the value when the button is pressed:

```
<button onClick="doReadMinutesOffsetFromPage();">Minutes offset</button>
```

We’ve seen buttons before. We used them to set the modes of the clock in chapter 1. A button can have an `onclick` attribute that specifies JavaScript to be run when the button is pressed. This button will call the function `doReadMinutesOffsetFromPage` when it is pressed.

```
function doReadMinutesOffsetFromPage () {  
  let minutesOffsetElement =  
    document.getElementById("minutesOffsetInput");13  
  let minutesOffsetValue = minutesOffsetElement.value;14  
  minutesOffset = Number(minutesOffsetValue);15  
}
```

You can see the `doReadMinutesOffsetFromPage` function above. It does you might expect from its rather long name. It finds the input element into which the user has typed the number, gets the `value` property of that element which contains the text of the number entered, converts the text of the number into a number and then sets a global variable called `minutesOffset` to hold the new value.

---

<sup>13</sup> *Get the input element*

<sup>14</sup> *Get the value out of the input element*

<sup>15</sup> *Convert the string into a number*



## Make Something Happen

### Adjustable time travel clock

The code for this exercise is in the folder **Ch03-01\_Adjustable\_Clock** in the **Ch03-Build\_interactive\_pages** examples. Use Visual Studio to open the **index.html** file for this example, start Go Live to open the page in a browser and open the developer view for the page. Select the console view of the application.



Ch03\_inset01\_01 adjustable clock

On the left you will see the ticking clock. On the right you will have the console. We can check on the current value of `minutesOffset` by using the console. Enter `minutesOffset` and press enter to ask the console to show you the contents of the variable.

```
> minutesOffset  
0
```

The listing above shows what happens. The variable is initially set as zero so the clock will show the current time. Now enter an offset value into the text box and click the Minutes offset button:



Ch03\_inset01\_02 Setting offset

You should see the clock time change to 10 minutes in the future. Now try setting the offset value to an empty string. Clear the contents of the input box and click the Minutes offset button.



Ch03\_inset01\_03 Empty offset input

Watch what happens to the time displayed by the clock. You will notice that it goes back to the correct time. The offset has been set to zero. This is strange. You've not set it to zero, you've set it to an empty string. Some programming languages would give you an error if you tried to convert an empty string into a number. JavaScript doesn't seem to mind. Let's have a look at what is going on. The interesting function here is the one called `Number`. This takes something in and converts it into a number. We can give it a string with a number in it, and `Number` will give you a value back. Let's try a few different inputs, starting with a string that holds a value. Open the Developer Tools, select the console and type the following statement, which will show us the result provided by the `Number` function when it is fed the string "99".

```
> Number("99")
```

Now press enter. Remember that the console always shows us the value returned by the JavaScript that is executed. In this case it will show us the value returned by `Number`. Press enter.

```
> Number("99")
99
```

The result of calling `Number` with the string "99" is the numeric value 99. Now let's try a different string:

```
> Number("Fred")
NaN
```

For brevity I'm showing the call of `Number` and the result it displays from now on. You can check these results yourself in the console if you like. Converting the text "Fred" to a number is not possible. `Number` returns `NaN` (Not a number), which makes sense because "Fred" does not represent a number. Now let's try one last thing:

```
> Number("")
0
```

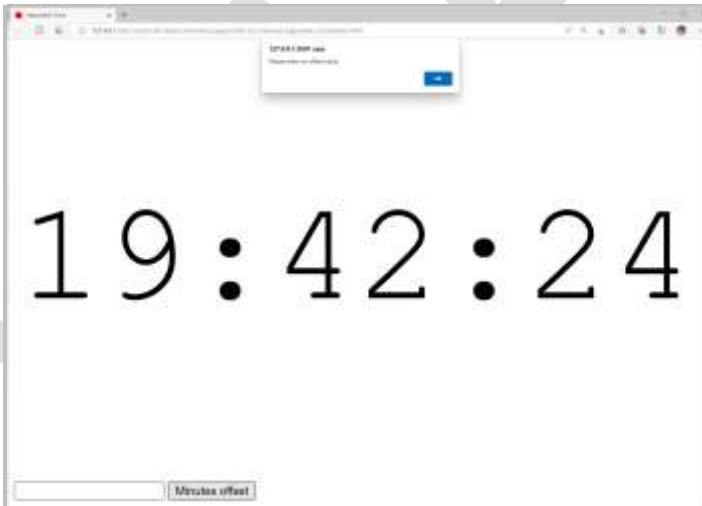
Now the `Number` function is working on an empty string. From with it returns the value 0. You might expect to see `NaN` again here, but you don't. This is one part of the reason that we get a `minutesOffset` of zero when we enter an empty field. The other part has to do with using a `type` of `number` for an input element. As you will have noticed when you tried to set the offset value, a number input tries not to let you enter text. On a device with a touch screen it may even display a numeric keyboard rather than a text one. If you do manage to type in an invalid number (or you leave the text out) the `input` element `value` property is an empty string. This empty string then gets fed into the `Number` function which then produces a result of 0.

This might not be what you want. You might take the view that if the user doesn't enter a number you want the `minutesOffset` to stay the same rather than go back to 0. We can fix

this by adding some extra code:

```
function doReadMinutesOffsetFromPage() {  
  let minutesOffsetElement =  
    document.getElementById("minutesOffsetInput");  
  let inputString = minutesOffsetElement.value;16  
  if (inputString.length==0){17  
    alert("Please enter an offset value");  
  }  
  else {  
    minutesOffset = Number(inputString);  
  }  
}
```

This version of the function `doReadMinutesOffsetFromPage` checks the length of the value received from the input element. If the length is zero (which means that a number wasn't entered or an empty string was entered) the function displays an alert. Otherwise, it sets the `minutesOffset` value.



Ch03\_inset01\_04 Empty offset alert

Above you can see what happens if you press the Minutes offset button with an empty string in the input element. You can find this version of the clock in the sample folder **Ch03-02\_Improved\_Adjustable\_Clock**

---

<sup>16</sup> *Get the value from the input element*

<sup>17</sup> *Check for an empty string*

# JavaScript input types

passwordInput : topsecret

<input type="text" value="hello"/>	text
<input type="number" value="1234"/>	number
<input type="password" value="*****"/>	password
<input type="email" value="mail@someaddress.com"/>	email
<input type="tel" value="(12345) 567890"/>	tel
<input type="date" value="19/08/2022"/>	date
<input type="time" value="19:26"/>	time
<input type="url" value="www.robmiles.com"/>	url
<input type="color" value="#000000"/>	color

Figure 3.13 Ch-03\_Fig\_02 input types

Figure 3.2 above shows some of the input types available and what they look like on a web page when you enter data into them. Each input item in Figure 3.2 is has a paragraph that contains an input field with the appropriate type and a button which is used to call a function that will display the input that the browser will receive. You can enter input data into an input and then press the button next to it to display the value that is produced. The password input element has just been used to enter `topsecret` and the `password` button has been pressed to display the value in the input element. Note that the browser doesn't display the characters in the password as it is entered.

```
<p>
  <input type="password" id="passwordInput">
  <button onclick='showItem("passwordInput");'>password</button>
</p>
```

Above you can see the HTML that accepts the password input. The outer paragraph encloses an input element and a button element. The `onclick` event for the button element calls a function called `showItem` with the argument of `"passwordInput"`. Note that the program uses two kinds of quotes to delimit items in the string that contains the JavaScript to be performed when the

button is pressed. The outer single quotes delimit the entire JavaScript text and the inner double quotes delimit the string “passwordInput” which is the argument to the function call. There are similar paragraphs for each of the different input types. The [showItem](#) function must find the value that was input and then display it on the output element.

```
function showItem(itemName){  
    let inputElement = document.getElementById(itemName);18  
    let outputElement = document.getElementById("outputPar");19  
    let message = itemName + " : " + inputElement.value20  
    outputElement.textContent = message;21  
}
```

The [showItem](#) function uses the [document.getElementById](#) method to get the input and output elements. It then builds the message to be displayed by adding the value in the input element onto the end of the item name. This message is then set as the content of the [outputElement](#), causing it to be displayed.

The input types work differently in different browsers. Some browsers offer to auto-fill email addresses or prop up a calendar when a date is being entered. However, it is important to remember that all these inputs generate a string of text which your program will need to validate before using it. The email input doesn't stop a user from entering an invalid email address. You can investigate the behavior of the of these types by using the page the sample folder **Ch03-03\_Input\_Types**

## Storing data on the local machine

You show everyone your adjustable clock and they are very impressed. For a while. Then they complain that the clock doesn't remember the offset that has been entered. Each time the clock web page is opened the offset is zero and the clock shows the correct time. What they want is a clock that retains the minutes offset value so that each time it is started it has the same offset

---

<sup>18</sup> *Get the source element*

<sup>19</sup> *Get the destination element*

<sup>20</sup> *Build the message*

<sup>21</sup> *Display the message*

that they set last time it was used. They assumed that you would know that was what they wanted, and now you must provide it.

### Programmer's Points

### Engaged users are a great source of inspiration

It is very hard to find out from a user exactly what they want a system to do. Even when you think you have agreed on what needs to be provided you can still encounter problems like these. The solution is to provide a workflow which makes it very easy for the users to let you know what is wrong with your system and then be constructive in situations like these.

If you store your solution in a GitHub repository you get an issue tracker where users can post issues and you can respond to them. If you do this correctly you can build a team of engaged (rather than enraged) users who will help you improve your solution and even serve as evangelists for it.

JavaScript provides a way that a web page can store data on a local machine. It works because a browser has access to the file storage on the host PC. The browser can write small amounts of data into this storage and then recover it when requested.

```
function storeMinutesOffset(offset){
  localStorage.setItem("minutesOffset", String(offset));
}
```

The function `storeMinutesOffset` shows how we use this feature from a JavaScript program. The `storeMinutesOffset` function accepts a parameter called `offset` which holds the offset value to be stored in the browser local storage. The value is stored by the `setItem` method provided by the `localStorage` object. This method accepts two arguments, the name of the storage location and the string to be stored there.

```
function loadMinutesOffset(){
  let offsetString = localStorage.getItem("minutesOffset");22
  if (offsetString==null){23
```

---

<sup>22</sup> *Get the stored value*

<sup>23</sup> *Check for a missing value*

```
    offsetString = "0";24
  }
  return Number(offsetString);25
}
```

The `loadMinutesOffset` function fetches a stored offset value. It uses the `getItem` method which is provided by the `localStorage` object. The `getItem` method is supplied with a string which identifies the local storage item to return. If there is no item in the local storage with the given name `getItem` returns `null`. We've seen `null` before. It is a value which means "I can't find it". The code above tests for a return value of `null` from the `getItem` function and sets the offset to 0 if this happens. This behavior is required because the first time the clock is loaded into the browser there will be nothing in local storage. You can find this code in the example **Ch03-04\_Storing\_Adjustable\_Clock**.

## Make Something Happen

## Software sleuthing

The storing adjustable clock works fine. But there is no way that the user can see the value of the minutes offset when they use the clock. It is not displayed on the page. But does this mean that it is impossible for anyone to discover this value? Let's see if we can use the debugger to get that value out of the browser. We can start by loading the clock page from the web. You can find it at the location:

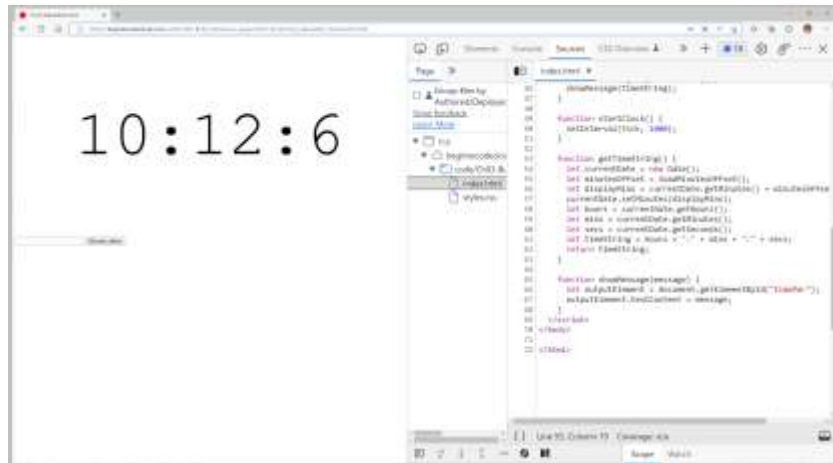
[https://beginintocloud.com/code/Ch03-Build\\_interactive\\_pages/Ch03-04\\_Storing\\_Adjustable\\_Clock/index.html](https://beginintocloud.com/code/Ch03-Build_interactive_pages/Ch03-04_Storing_Adjustable_Clock/index.html)

Once you have loaded the page open the Developer Tools window, select the Sources view and open the file `index.html`. Then scroll down the listing until you find the `getTimeString` function. This function is called every second to display the time.

---

<sup>24</sup> Set the offset to 0 if nothing is stored

<sup>25</sup> Return a Number



Ch03\_inset02\_01 Debugging the adjustable clock

Set a breakpoint at line 56 by clicking on the left margin to the left of the line number. This function is called every second, and so the breakpoint will be hit almost instantly.

```

53  func 20 getTimeString() {
54      let currentDate = new Date();
55      let minutesOffset = loadMinutesOffset
56      let displayMins = currentDate.getMinutes();
57      currentDate.setMinutes(displayMins);
58      let hours = currentDate.getHours();
59      let mins = currentDate.getMinutes();
60      let secs = currentDate.getSeconds();

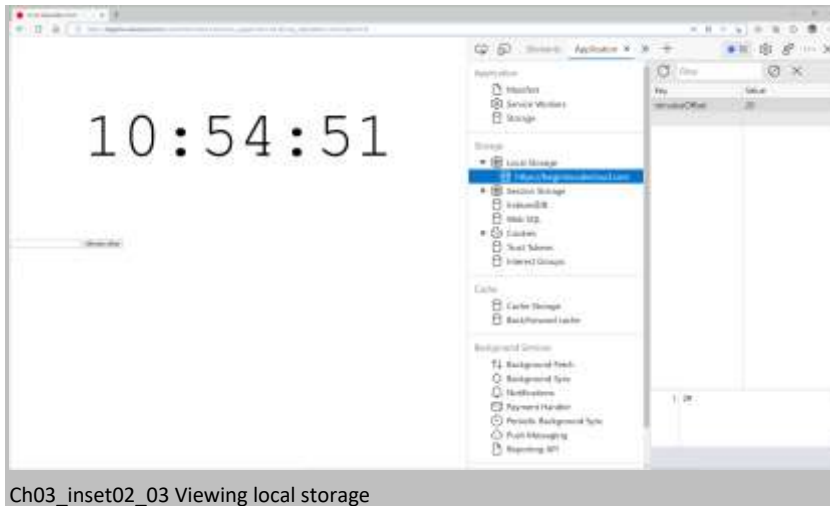
```

Ch03\_inset02\_02 Viewing the minutes offset

You should see a display like the one above in the code window. If you hold the mouse pointer over the `minutesOffset` variable you can see the value 20 has been loaded from local storage. This shows how easy it is to view the values in a program as it runs.

However, if you want to see the values stored in local storage it turns out that there is an even easier way to do this.





If you open the **Application** tab in the Developer Tools you can view the contents of Local Storage. As you can see from the image above, the value `minutesOffset` is stored as 10. Note that this local storage is shared for all the pages underneath the `begintocode.com` domain. In other words, any of our example JavaScript applications can view and change that value. You can use this view to investigate the things that web pages are storing on your computer.

#### Programmer's Points

#### Think hard about security when writing JavaScript

It's not really a problem if someone reads and changes the minutes offset for our clock. But I hope you now appreciate just how open JavaScript is. A badly written application that stores password strings in local storage would be very vulnerable to attack, although the attacker would need to get physical access to the machine. When you write an application you need consider how exposed the application is to attacks like these. Tucking something into a local store might seem a good idea, but you need to consider how useful it would be to a malicious person. And you should make sure that variables are visible only where they are used.

## JavaScript Heroes: let, var and const

Some features of a programming language are provided so you can use the language to make a working program. For example, a program needs to be able to calculate answers and make

decisions, so JavaScript provides assignments and if constructions. However, `let`, `var` and `const` are not provided to make programs work, they are provided to help us more secure code. They help us control the **visibility** of variables we use in our programs. Let's look at why variable visibility is important and how we can use `let`, `var` and `const` to manage it in JavaScript.

Making a variable in a program **global** is rather like writing your name and phone number on the notice board in the office. It makes it easy for your colleagues to contact you, but it also means that anyone seeing the notice board can call you. And someone else could erase the number you wrote and put up a different one if they wanted to redirect your phone calls to the speaking clock. In real life we need to take care of how much data we make public. It is the same in JavaScript programs. Let's look at the how we manage the scope of variables using our new JavaScript heroes.

## Make Something Happen

## Investigate let, var and const

The functions for this make something happen are in the example **Ch03-01\_Variable Scope**. Use Visuals Studio to open the **index.html** file for this example, start Go Live to open the page in a browser and open the Developer Tools for the page. Now open the console tab.



### Ch03\_inset03\_01 Investigate let var and const

The page shows a list of sample functions you can call from the Developer Tools Console to learn more about variables and scope. The scope of a variable is that part of a program where the variable can be accessed. JavaScript has three kinds of scope: global, function and block. A variable with global scope can be accessed anywhere in the program. A variable with function scope can be accessed anywhere in a block. A variable with block scope can be accessed anywhere in a block, except where it is “scoped out”. Let's discover what all this means with some code examples.

```
function letScopeDemo() {
  let i = 99;
  {
    let i = 100;
    console.log("let inner i:" + i);
  }
}
```

```
    console.log("let outer i:" + i);  
}
```

The above function creates two variables, both with the name `i`. The first version of `i` is assigned the value 99. This variable is declared in the body of the function. The second version of `i` is declared in the inner block and set to the value 100. Let's have a look at what happens when we run the function in the console:

```
> letScopeDemo()  
let inner i: 100  
let outer i: 99
```

Above you can see the call of `letScopeDemo` and the results displayed when it ran. You can run the function on your machine. When we run this function the value of each `i` is printed out. Note that within the inner block the outer variable called `i` (the one containing 99) is not accessible. We say that it is "scoped out". When the program exits the inner block the inner `i` (the one containing 100) is discarded and the outer `i` becomes accessible again. You use `let` to create a variable that does not need to exist outside the block in which it was created. These are called global variables.

```
function varScopeDemo() {  
    var i = 99;  
    {  
        var i = 100;  
        console.log("var inner i:" + i);  
    }  
    console.log("var outer i:" + i);  
}
```

The function `varScopeDemo` above is similar to `letScopeDemo` except that `i` is now declared using `var`. When we run it we get a different result:

```
> varScopeDemo()  
var inner i: 100  
var outer i: 100
```

Variables declared using `var` have function scope if declared in a function, or global scope if declared outside all functions. The second declaration of `i` replaces the original one with a new value which persists until the end of the `varScopeDemo` function. So variables declared using `var` within a block exist all the way to the end of the block and can be over written with new ones. Let's build on our understanding of scope by trying some things that might not work.

```
function letDemo() {  
    {  
        let i = 99;  
    }  
    console.log(i);  
}
```

The `letDemo` function body contains a block of code nested inside it. Within this block the `let` keyword is used to declare a variable called `i` and set its value to 99. Then the block ends and the value of `i` is displayed on the console. We can run the program by just typing `letDemo()` on the console:

```
> letDemo()
Uncaught ReferenceError: i is not defined
    at letDemo (index.html:58:21)
    at <anonymous>:1:1
```

Ch03\_inset03\_02 let demo

The `letDemo` function fails because the variable `i` only exists within the inner block in the function. As soon as execution leaves that block the variable is discarded, which means attempts to access that variable will fail as it no longer exists.

```
function varDemo() {
  {
    var i = 99;
  }
  console.log(i);
}
```

The function `varDemo` is very similar to `letDemo`, but this time `i` is declared using `var`. Let's see what happens when we run this function.

```
> varDemo()
99
```

Ch03\_inset03\_03 var demo

This time the function works perfectly. Variables declared using `var` remain in existence from the point of declaration all the end of the enclosing scope, which in this case means the body of function `varDemo`. So, if we try to use the variable `i` from the console we will find that it no longer exists, because the function `varDemo` has finished.

```
> varDemo()
99
< undefined
> i
Uncaught ReferenceError: i is not defined
    at <anonymous>:1:1
```

Ch03\_inset03\_04 undefined i

So far everything makes perfect sense. You use a `let` if you want the variable to disappear when the program leaves the block where the variable is declared. You use a `var` if you want the variable to exist in the entire enclosing scope. So, let's try something weird.

```
function globalDemo() {
  {
    i = 99;
  }
  console.log(i);
}
```

The `globalDemo` function uses neither `let` or `var` to declare the variable `i`. You might think that this would cause an error. But it doesn't. Even stranger, the variable `i` still exists after the function has completed.

```
> globalDemo()
99
< undefined
> i
99
>
```

Ch03\_inset03\_05 global i

This is a “JavaScript Zero”. It is one of the things about JavaScript that I really don't like. If you don't use `let` or `var` to declare a variable you get a variable that has global scope, i.e. it exists everywhere. This is perhaps the worst thing that could happen. It means that if I forget the `var` or the `let` I don't get an error, I get something which exists right through my program and is open to prying and misuse.

This behavior goes back to the very first JavaScript version which was intended to be easy to learn and use. At the time it seemed a good idea to create variables automatically. Nowadays JavaScript is used to create applications which need to be highly secure and resilient, and this behavior is a bad idea. To solve the problem the latest versions of JavaScript have a **strict** mode which you can turn on by adding this statement to your program.

```
function strictGlobalDemo() {
  'use strict';
  i = 99;
}
```

The `strictGlobalDemo` function sets strict mode and then tries to create a global variable.

```
> strictGlobalDemo()
Uncaught ReferenceError: i is not defined
    at strictGlobalDemo (index.html:107:11)
    at <anonymous>:1:1
>|
```

Ch03\_inset03\_06 strict global

This function fails when it tries to automatically create the variable `i`. Note that **strict** mode is only enforced in the body of the function `strictGlobalDemo`. If you want to enforce **strict** mode on all the code in your application you should put the statement at the top of your

program, outside any functions.

Strict mode disallows lots of dangerous JavaScript behaviors, including the automatic declaration of variables. I add it to the start of all the JavaScript programs that I write.

The final hero we're going to meet is `const`. We use this when we don't want our program to change the value in a variable.

```
function constDemo() {  
  {  
    const i = 99;  
    i = i + 1;  
  }  
  console.log(i);  
}
```

In the `constDemo` function above the variable `i` is declared as a `const`. This means that the statement that tries to add 1 to the value of `i` will fail with an error. If you have a value in your program that shouldn't be changed you can declare it as constant. Variables declared using `const` inside a block have the same scope as `let`. Variables declared using `const` outside any function have global scope.

It seems obvious when we need to use `let` or `var`. We use `let` when we want to create a variable that will disappear when a program exits the block where it was declared, and we try hard not to use `var` at all (unless we have something we really want to share over the whole program. But what about `const`? When I write code I try to look out for situations where a bug can happen and then try to remove it. Look at these two statements from the adjustable clock that we created that store the time offset value in the browser:

```
localStorage.setItem("minutesOffset", String(offset));  
...  
offsetString = localStorage.getItem("minutesOffset");
```

The first statement creates a local storage item called "minutesOffset" which contains a string of text specifying the offset value. The second statement gets this value back from local storage. Can you spot anything wrong with this code? The thing I don't like about it is the way that I'm having to type the string "minutesOffset" twice. This string gives the name of the storage location that will be written to and then read back from later.

If I type one of the strings as "MinutesOffset" by mistake (I've made the first letter upper case rather than lower case) the program will either store the value in the wrong place or fail to find it when it looks for it. This means that the code has the potential for a bug. I can solve this problem completely by creating a constant variable that holds the name of the stored item:

```
const minutesOffsetStoreName = "minutesOffset";

localStorage.setItem(minutesOffsetStoreName, String(offset));
...
offsetString = localStorage.getItem(minutesOffsetStoreName);
```

The code above shows how I would do this. This makes it impossible to miss-type the name of the store. The `minutesOffsetStoreName` is declared at global scope outside every function so that it is available over the entire program. I don't mind constant values being global as they are not vulnerable to being changed. You can find this code in the example **Ch03-06\_Variable\_Storage**.

#### Programmer's Point

Use language features to make your code better

The `let`, `var`, `const` and `strict` features of JavaScript are not there to allow you to do things, they are there to help you make programs safer. When I create a new variable, I consider how visible it needs to be. If I need to make the value widely available I'll try to find ways to do it without creating a global variable. I also use the strict mode at all times. You should too.

## Making page elements from JavaScript

We have seen how the Document Object Model (DOM) is built in memory by the browser which uses the contents of the HTML file that defines the web site. The DOM is then rendered by the browser to display contents of the pages for the user. We've also seen how a JavaScript program can interact with the elements in the DOM by changing their properties and how these changes are reflected in what the user sees on the page. We used this to change the time displayed by a paragraph in the clock.

Now we are going to discover how a JavaScript program can create elements when it runs. This is a very important part of JavaScript programming. Some web pages are built from HTML files that are entirely JavaScript code. When the page loads the JavaScript runs and creates all the elements that are used in the display. We are going to show how this works by creating a little game called "Mine Finder". It turns out to be quite compelling.

# Mine Finder

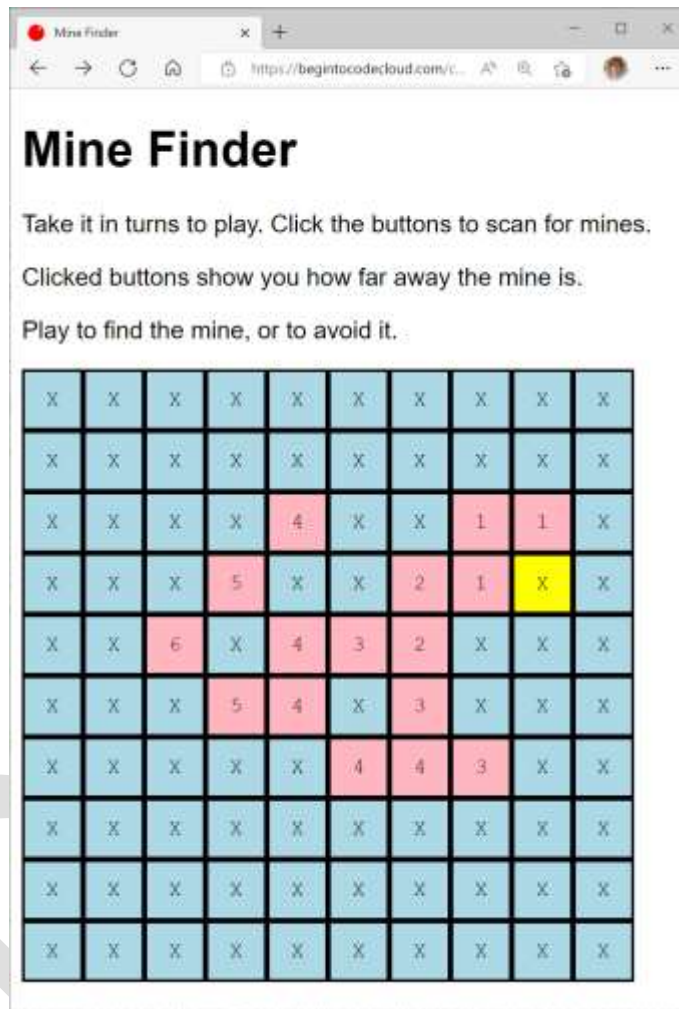


Figure 3.14 Ch-03\_Fig\_03 Mine Finder Game

Figure 3.3 shows the Mine Finder game. It is played on a 10x10 grid of buttons. One of the buttons contains a mine. Before you start you agree whether you are playing to find the mine or avoid it. Then each player in turn presses a button. If the button does not contain the mine it turns pink and displays the distance that square is from the mine. If the button is the mine a message is displayed, the mine button turns yellow and the game is over. Reloading the page creates a brand-new minefield and moves the mine to a new location. You can have a go at the game by visiting the example page at [https://beginintocloud.com/code/Ch03-Build\\_interactive\\_pages/Ch03-07\\_Mine\\_Finder/index.html](https://beginintocloud.com/code/Ch03-Build_interactive_pages/Ch03-07_Mine_Finder/index.html)



## Place the buttons

To make the game work we need a web page that contains 100 buttons. It would be very hard to make all these buttons by hand. Fortunately, we can use loops in a JavaScript program to make the display for us.

```
<p id="buttonPar"> </p>
```

Above you can see the paragraph that will contain the buttons. In the HTML file this paragraph is empty. The buttons will be added by a function which is called when the page is loaded. The paragraph has the id `buttonPar` so that our code can locate it in the document.

```
function playGame(width, height) {  
  
    let container = document.getElementById("buttonPar");26  
  
    for (let y = 0; y < height; y++) {27  
        for (let x = 0; x < width; x++) {28  
            let newButton = document.createElement("button");29  
            newButton.className = "upButton";30  
            newButton.setAttribute("x", x);31  
            newButton.setAttribute("y", y);32  
            newButton.textContent = "X";33  
            newButton.setAttribute("onClick", "doButtonClicked(this);");34
```

---

<sup>26</sup> Find the destination paragraph

<sup>27</sup> Work through each row

<sup>28</sup> Work through each column in a row

<sup>29</sup> Make a button

<sup>30</sup> Set the style to "upButton"

<sup>31</sup> Store the x position in the button

<sup>32</sup> Store the y position in the button

<sup>33</sup> Draw an X in the button

<sup>34</sup> Add an event handler

```

        container.appendChild(newButton);35
    }

    let lineBreak = document.createElement("br");36
    container.appendChild(lineBreak);37
}

mineX = getRandom(0, width);38
mineY = getRandom(0, height);39
}

```

This is the function that creates the buttons and sets the game up. Let's work through what it does.

```
let container = document.getElementById("buttonPar");
```

This statement creates a local variable called `container` which refers to the paragraph that will contain all the buttons on the page. The paragraph has the id `buttonPar`.

```
for (let y = 0; y < height; y++) {
    for (let x = 0; x < width; x++) {
```

These two statements create pair of for loops, one nested inside the other. The outer loop will be performed for each row of the button grid. The inner loop will be performed for each column in each row. The variable `y` keeps track of the row number, and the variable `x` keeps track of the column number.

---

<sup>35</sup> Append the button to the destination

<sup>36</sup> Create a line break

<sup>37</sup> Add the line break to the paragraph

<sup>38</sup> Set the X position for the mine

<sup>39</sup> Set the Y position for the mine

```
let newButton = document.createElement("button");
```

This is something we've not seen before. The document object provides a method called `createElement` that creates a new HTML element. We specify the kind of element we want by using a string. In this case we want a `button`. Note that creating an element does not add it to the DOM, we must do that separately.

```
newButton.className = "upButton";
```

The statement above sets the `className` for the button. This determines the style that will be used to display the button.

```
.upButton,.downButton,.explodeButton {  
  font-family: 'Courier New', Courier, monospace;  
  text-align: center;  
  min-width: 3em;  
  min-height: 3em;  
}  
  
.upButton{  
  background: lightblue;  
}  
.downButton {  
  background: lightpink;  
}  
.explodeButton {  
  background: yellow;  
}
```

These are the styles that are used for the buttons. There are some common style items (the font family, alignment and minimum width and height) along with different colors for each of the states of the button.

```
newButton.textContent = "X";
```

This statement sets the initial text content of the button. This will be replaced by the distance value when the button is clicked.

```
newButton.setAttribute("x", x);  
newButton.setAttribute("y", y);
```

These two statements set up a couple of attributes on the new button that give the location of the button in the grid. We are going to bind a function to the `onclick` event of the button. We don't want to create a different function for each button press. That would mean creating 100 functions. Instead we want to store location values in each button so that a single button function can work the position of a particular button. We have already written code that sets existing attributes on an element (to change the `class` or the `textContent` of a paragraph). The two statements above create attributes called `x` and `y` that contain the x and y positions of the button. This is a very powerful technique. It makes elements in the DOM into an extension of your variable storage.

```
newButton.setAttribute("onClick", "doButtonClicked(this);");
```

Above is the last statement that sets up the button. It binds a method called `doButtonClicked` to the `onClick` event for the button. If the button is clicked this function will run. All the buttons will call the same function when they are clicked. You might be wondering how the `doButtonClicked` function will know which button has been clicked. Let's take a look at the JavaScript statement that is being assigned to `onClick` to find out how this works.

```
doButtonClicked(this);
```

When executed in the context of a JavaScript statement running from HTML the value of the `this` is a reference to element generating the event. So, each time `doButtonClicked` is called it will be given an argument which is a reference to the button that has been clicked. This is terribly useful. It makes it very easy for an event handler to know which element caused the event.

If you are having bother understanding what is happening here, remember the problem that we are trying to solve. We have 100 buttons. Each button can generate an `onClick` event. We don't want to make 100 functions to deal with all these `onClicks`. We would much prefer just to write one function. But if we only have one function; it needs to know which button it has been called from. The `this` reference is an argument to the call of `doButtonClicked` which is fed into the

function when the button is clicked. In this context, the value of `this` refers to the button that has been pressed. So `doButtonClicked` is always told the button that has been clicked.

This will make more sense when we look at what the `doButtonClicked` function does. And we have a whole JavaScript Hero description for the `this` keyword coming up. For now, if you are happy with the value of `this` delivering a reference to the button that was pressed, we can move on to the next part of the button setup.

```
container.appendChild(newButton);
```

Way back at the start of this description we set up a variable called `container` which was a reference to the paragraph which is going to hold all our buttons. The `container` provides a method called `appendChild` which is given a reference to the new element and adds it. This means that the paragraph now contains the newly created button. New elements are appended in order. So the first element will be button (0,0) and the second (0,1) and so on.

```
let lineBreak = document.createElement("br");  
container.appendChild(lineBreak);
```

These two statements are performed after we have added all the buttons in a row. They create a break element (`br`) and then append it to the paragraph container. This is how we separate successive rows in the grid.

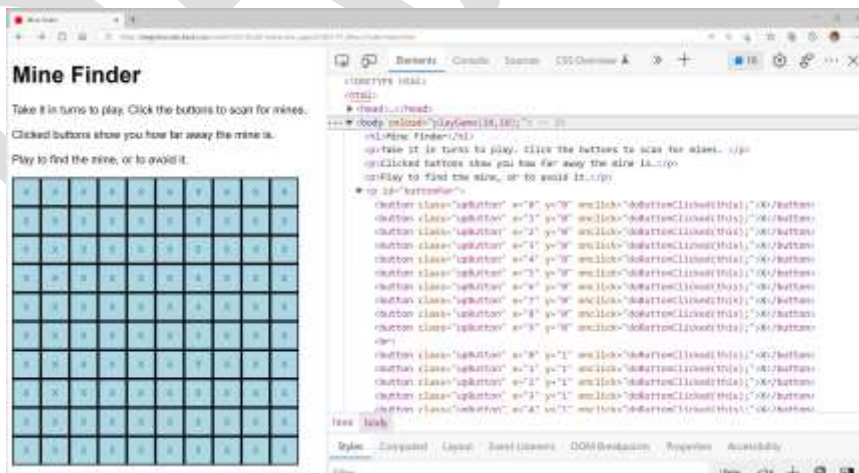


Figure 3.15 Ch\_03\_Fig\_04 Mine Finder buttons

Figure 3.3 above shows how we can use the Elements tab from the Developer Tools to look at all the buttons that have been created by our code. Remember that the original HTML for the page did not contain any buttons. These have all been created by our code. You can see that all the buttons have the properties that you would expect.

## Place the mine

The next thing the game needs to do is place the mine somewhere on the grid. For this we need random numbers. JavaScript has a random number generator which can produce a random value between 0 and 1. It lives in the [Math](#) library and is called [random](#). We can use this in a helper function to generate random integers in a particular range:

```
function getRandom(min, max) {  
  var range = max - min;  
  var result = Math.floor(Math.random() * (range)) + min;  
  return result;  
}
```

The [getRandom](#) function is given the minimum and maximum values of the random number to be produced. It then creates a value which between the two values. The maximum value is an exclusive upper limit. It is never produced. The function uses [Math.random](#) to create a random number between 0 and 1 and [Math.floor](#) to truncate the fractional part of a number and generate the integer value that we need.

```
mineX = getRandom(0, width);  
mineY = getRandom(0, height);
```

These two statements set the variables [mineX](#) and [mineY](#) to the position of the mine. These variables have been made global so that they are shared between all of the functions in the game.

```
var mineX;  
var mineY;
```

Making these values global makes the game a bit less secure, but it also keeps the code simple.

## Respond to button presses

The final behavior that we need is the function that responds to a button press. If you look at the buttons definitions in Figure 3.4 above you will see that the `onClick` attribute of each button makes a call of the `doButtonClicked` function. Let's have a look at this function.

```
function doButtonClicked(button) {  
  let x = button.getAttribute("x");40  
  let y = button.getAttribute("y");41  
  if (x == mineX && y == mineY) {42  
    button.className = "explodeButton";43  
    alert("Booom! Reload the page to play again");44  
  }  
  else {45  
    let dx = x - mineX;46  
    let dy = y - mineY;47  
    let distance = Math.round(Math.sqrt((dx * dx) + (dy * dy)));48  
    button.textContent = distance;49  
    button.className = "downButton";50  
  }  
}
```

The `doButtonClicked` function has a single parameter, which is a reference to the button that has

- 
- <sup>40</sup> *Get the x position of the button*
  - <sup>41</sup> *Get the y position of the button*
  - <sup>42</sup> *Check to see if this is the mine button*
  - <sup>43</sup> *Set the button style to “explode”*
  - <sup>44</sup> *Tell the player they have found the mine*
  - <sup>45</sup> *Do this part if the mine was not found*
  - <sup>46</sup> *Get the x distance to the mine*
  - <sup>47</sup> *Get the y distance to the mine*
  - <sup>48</sup> *Work out the distance*
  - <sup>49</sup> *Put the distance value into the button*
  - <sup>50</sup> *Set the button to the “down” style*

been clicked. This is obtained from the [this](#) reference which is added when the event is bound in the element definition.

```
let x = button.getAttribute("x");  
let y = button.getAttribute("y");
```

The first two statements in the function read the values in the x and y attributes on the button. These give the location of the button in the grid.

```
if (x == mineX && y == mineY) {  
    button.className = "explodeButton";  
    alert("Booom! Reload the page to play again");  
}
```

The next set of statements checks to see if this button is at the location of the mine. If both the x and the y values match the statements set the class style for the button to “explodeButton”. This causes the button to turn yellow. Then an alert is displayed to tell the players the game is over.

```
else {  
    let dx = x - mineX;  
    let dy = y - mineY;  
    let distance = Math.round(Math.sqrt((dx * dx) + (dy * dy)));  
    button.textContent = distance;  
    button.className = "downButton";  
}
```

The final part of this function is the behavior that is performed if the button is not the mine. The first three statements use the laws of Pythagoras (the square of the hypotenuse is equal to the sum of the squares on the other two sides of a right-angled triangle) to work out the distance from this button to the mine. It then sets the text content of the button to this value and changes the style to [downButton](#), which turns the button red.

## Playing the game

The game is quite fun to play, particularly with two or more opponents. If you want to make the



game larger (or smaller) you just change the call of `playGame` which is bound to the `onload` event in the body of the html. This is where the number of rows and columns is set.

```
<body onload="playGame(10,10);">
```

The grid is made up of rows of buttons separated by line breaks. If the user makes the browser window too small the rows of buttons wrap round. We could fix this by positioning the button absolutely or by displaying the buttons in a table construction. We could create the table programmatically as we have created the buttons and then add elements to the table to make the required rows and columns.

## Using events

The present version of Mine Finder works perfectly. But it turns out that there is a neater way of connecting events to JavaScript functions. At the moment we are using this statement to connect an event to an object:

```
newButton.setAttribute("onClick", "doButtonClicked(this);");
```

This works by creating an `onClick` attribute on a new button and then setting it to a string of JavaScript which calls the method that we want (and uses this to provide a pointer to the button being clicked). This works because it is exactly what we would do if we were setting the event handler of an element in the HTML file. However, this not the neatest way of doing it if we are creating an element in software.

The major limitation of this technique is that we can only connect one event handler this way. We might have a situation where we want several events to fire when the button is clicked. This is not possible because an HTML element can only have one of each attribute. However, we can use a different mechanism to connect the button click handler.

```
newButton.addEventListener("click", buttonClickedHandler);
```

The statement above uses the `addEventListener` method provided by the `newButton` object to add an event listener function to a new button. The name of the event is specified by the string, in this case the event we want is "click". The second parameter is the name of the method to be called when the button is clicked. We are using a method called `buttonClickedHandler`. After the

event listener has been added the function `buttonClickedHandler` will be called when the button is clicked.

In the earlier event handler code we used this to deliver a reference to the button that has been pressed. How does the `buttonClickedHandler` function know which button it is responding to? Let's take a look at the code of the function:

```
function buttonClickedHandler(event){  
    let button = event.target;51  
    . . .  
}
```

The `buttonClickedHandler` function is declared with a single parameter called `event` which describes the event that has occurred. One of the properties of the event object is a called `target`. This is a reference to the element that generated the event. The `buttonClickedHandler` function extracts this value from the event and sets the value of `button` to it. The function then works in the same way as the earlier version. You can find this code in the example **Ch03-08\_Mine\_Finder Events**.

## Improve Mine Finder

The game is quite fun, but you might like to make some improvements. Here are some ideas for things that you might like to do:

- You could add a counter that counts the number of squares visited. They you could have a version where the aim is to find the mine in the smallest number of tries.
- You could add a timer which counts down. A player must find the mine in the shortest time (clicking as many squares as they like).
- You could change the way that the distance to the mine is displayed. Rather than putting a number in the square you could use a different color. You would need to create 10 or so new styles (one for each color) and then you could use an array of style names that you index with the distance value to get the style for the square. This might make for some nice-looking displays as the game is played.

---

<sup>51</sup> *Get the button reference from the event description*

# What you have learned

This has been another busy chapter. We've covered a lot of ground. Here is a recap plus some points to ponder.

- A web page can contain input elements that are used to read values from the user. An input tag can have different types, for example text, number, password, date and time. The value of an input tag is always delivered as a string and needs to be checked for validity before it is used. The input tag behaves differently in different browsers.
- The `Number` function is used to convert a string of text into a number. If the text does not contain a valid number the function will return `NaN`. If the text is empty the function returns 0.
- A browser provides local storage where a JavaScript application can store values that persist when the web page is not open. Local storage is provided on a "per site" basis, i.e. each top domain has its own local storage. Local storage is implemented as named strings of text. The Application tab of the browser Developer Tools can be used to view the contents of local storage. Local storage is specific to one browser on one machine.
- JavaScript variables can be declared local to a block of code using the keyword `let`. These variables are discarded when program execution leaves the block where they are declared. Using `let` to declare a variable in an inner block "scopes out" a variable with the same name which was declared in an enclosing block. An attempt to use a variable outside its declared scope will generate an error and stop the program.
- JavaScript variables can be declared global using the keyword `var`. Variables declared using `var` in a function body are global to that function but not visible outside it. Variables declared using `var` outside all functions are global and are visible to all functions in the program.
- Global variables represent risk. A global variable can be viewed by any code in the program (which represents a security risk) and it can be changed by any code in the program, which represents a risk of unintentional change or vulnerability to attack.
- Variables that are not explicitly declared (i.e. not declared using `let` or `var`) are global to the entire program. This dangerous default behavior can be disabled by adding a `"use strict";` statement to the function or at the start of the entire program.
- You can declare a variable using `const`, which prevents the value assigned

to the variable being changed.

- A JavaScript program can add elements to the Document Object Model. This makes it possible for the contents of a web page to be created programmatically, rather than being defined in the HTML file that describes page. You can view the elements in a web page (including those created by code) by using the Elements view in the Developer Tools.
- An element created in a JavaScript program can have additional attributes added to it. We used this to allow a button in the Mine Finder game to hold its x and y position in the grid.
- Creating a new HTML element in a JavaScript program does not automatically add it to the page. The `appendChild` function can be used on the container element (for example a paragraph) to add the new element. When the element is added the page will be re-drawn and the new element will appear on the display.
- The JavaScript `this` keyword can be used in the string of JavaScript bound to an event handler. In this context the `this` keyword provides a reference to the object generating the event. In the case of the Mine Finder program we use this to allow 100 buttons to be connected to the same event handler. By passing the value of `this` into the event handler we can tell the handler which button has been pressed.
- It is also possible to use the `addEventListener` method provided by an element instance (in our case a button in the Mine Finder game) to specify a function to be called when the event occurs. The event handler function is provided with a reference to event details when it is called. These details include a reference to the object that caused the event.
- JavaScript provides a `Math.random` function which produces a random number in the range 0 to 1. We can multiply this value by a range to get a number in that range.

To reinforce your understanding of this chapter, you might want to consider the following "profound questions".

Does the user always have to press a button to trigger the reading of an input?

You can use the `onInput` event to specify a function to be called each time the content of an input box changes. This means that if the user was typing in a number the `onInput` function would be called each time a digit was entered.

Is Number the only way to convert a string to a number?

No. JavaScript provides functions called `parseInt` and `parseFloat` which can be used to parse a string and return a value of the requested type. These behave slightly differently from `Number`. A string that starts with a number would be regarded as that number. For example parsing "123hello" would return the value 123, whereas `Number` would regard this as `NaN`. The parse functions also regard an empty string as `NaN`. It doesn't matter whether you use `Number` or the parse functions, just be mindful of the slight differences in behavior.

How much local storage can I have on a web site?

The limit is around 5Mb for a browser on a PC.

How long are variables stored in local storage?

There is no limit to the time that a value will be stored.

Can I delete something from local storage?

Yes you can. The `removeItem` method will do this. However, once deleted it is impossible to get the data back.

How do I store more complex items in local storage?

Local storage stores a string of data. You can convert JavaScript objects into strings of text encoded using the JavaScript Object Notation (JSON). This allows you to store complex items in a single local storage location. We will be doing this later in the text

How do I protect items stored in local storage?

You can't. They are public. The solution is never to store important data in local storage. You should store such data on the server (which users don't have access to). We will be doing this in the next part of the book.

Can I stop someone looking through the JavaScript code in my web page?

No. There are tools you can use that will take your easy-to-understand code and make it much more difficult to read. These are called obfuscators. However, there are also tools that can unpick obfuscated code. The only way to make a properly secure application is to run all the code in server, not the client. We will be doing this in part 2 of the book.

What is the difference between `let` and `var`?

A variable declared using `let` will cease to exist a program leaves the block in which the variable was declared. This makes `let` very useful for variables that want to use for a short time and then discard. A variable declared using `var` has a longer lifetime. If it is declared in a function body the variable will exist until the function exits. A variable declared as `var` at global level (i.e. outside all functions) is global and will be visible to code in all the functions

in the application. Global variables should be used with caution. They provide convenience (all functions can easily access their content) at the expense of security (all functions can easily access their contents).

#### What does strict do?

Strict mode changes the behavior of the JavaScript engine so that program constructions which might be dangerous are rejected. One of the things that strict does is stop the automatic creation of global variables when a programmer doesn't specify `let` or `var` at the variable creation.

#### When do I use a constant?

You use a constant when you have a particular value which means something in your code. Using a constant makes it easy to change the value for the entire program. It also reduces the chance of you entering the value incorrectly. Finally, it lets you make a program clearer. Having a constant called `maxAge` in a program, rather than the value 70 makes it very clear what a statement is doing.

#### Can document elements created by JavaScript have event handlers?

Yes they can. We have actually done this. The best way is to use the `addEventListener` to specify the function to be called when the event occurs.

#### How does an event handler know which element has triggered an event?

We have done this in two different ways. In the first version of Mine Finder we added a `this` reference to the call of the function that handled the event. This function was specified in the text of a function call bound to the "onClick" attribute added each button element. The function then received the `this` reference (which in this context is set to a reference to the element generating the event) and used it to locate the button that was pressed.

The second way we did this, which is more flexible, used the `addEventListener` method on the new button to add the event handler. When the event handler is called by the browser in response to the event it is passed a reference to an Event object which contains information about the event, including the property `target` which contains a reference to the element that generated the event.

# 4

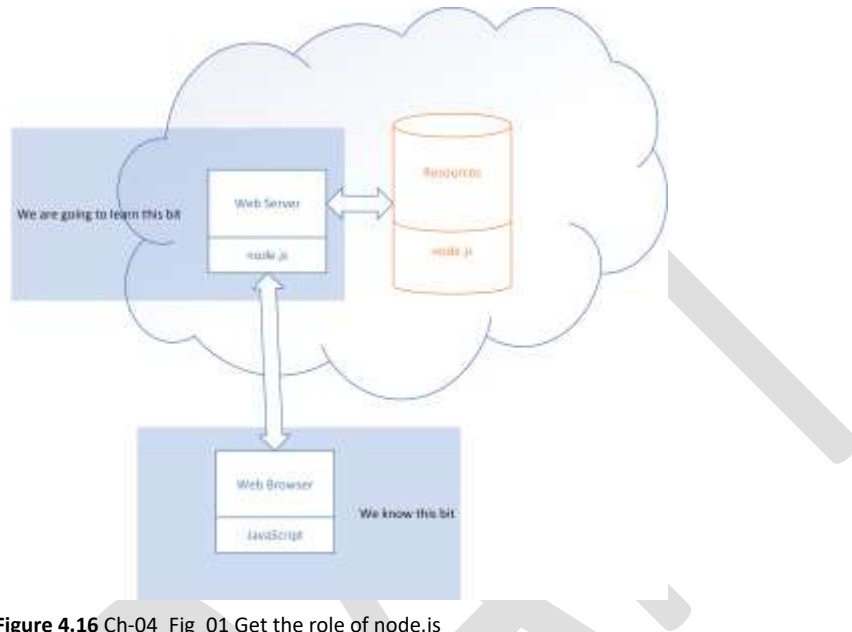
## Host a website

### What you will learn

We now understand how a JavaScript program can interact with the browser and the Document Object Model (DOM) of a web page to create interactive websites. We know that the HTML page that defines a document is only the starting point of the definition of a web site. A JavaScript program can run when a page loads and dynamically create content and connect events to make an active web page. Now we are going to discover how JavaScript can be used to host a web site using node.js. Node.js allows us to run JavaScript programs directly on our computer. In this chapter we are going to discover how a JavaScript program can operate as a web server and generate web sites dynamically. We are also going to take a detailed look at how JavaScript programs can be broken down into modules.

As ever, the Glossary is around to help you with terms you've not seen before.

# node.js



**Figure 4.16** Ch-04\_Fig\_01 Get the role of node.js

Figure 4.1 above shows where we are now in the process of learning how to write applications for the cloud. We can write JavaScript programs that run in the browser and communicate with the user via the elements in the Document Object Model (DOM). Now we are going to learn how to write JavaScript programs that run in the server and respond to requests from the browser. The JavaScript programs that we are going to write will run inside a framework called node.js. Node.js was created by taking the JavaScript component out of a browser and turning it into a free-standing program. I'm going to call it "node" for the rest of this book. The first we need to do is get node running on our computer.

## Make Something Happen

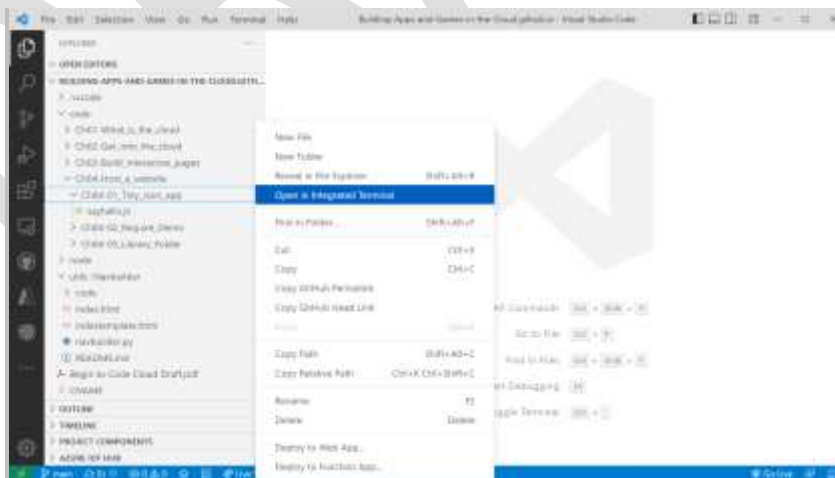
### Install node

Before we can use node we need to install it on our machine. The application is free. There are versions for Windows PC, Mac OS and Linux. Open your browser and go to the <https://nodejs.org/en/download/> page.

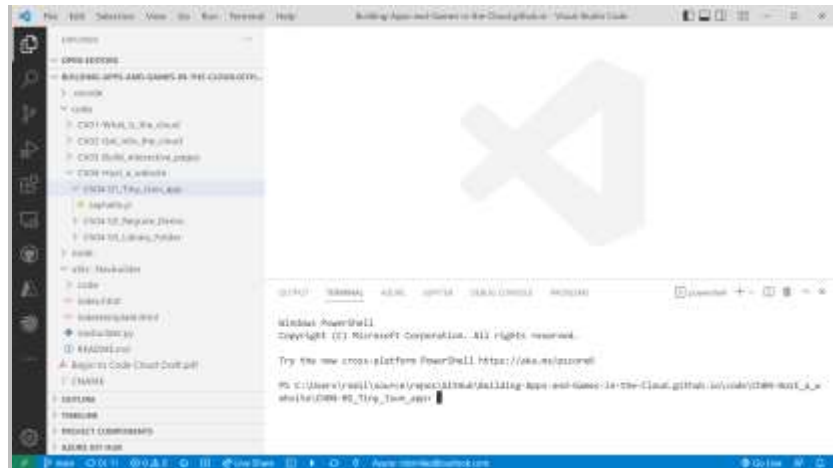




Click on the link for the installer for your machine and go through the installation process. Select all the default options that are suggested. Now, let's have a chat with node using the Visual Studio Code Terminal. Start Visual Studio Code and open the GitHub folder for the sample code. Now we want to start a terminal in the folder that contains a JavaScript program we are going to use to test our node installation. Open the explorer view in Visual Studio by clicking the icon at the top of the column on the left. Now look at the code samples and find the folder called **Ch04-01\_Tiny\_Json\_app**. Right click on this folder to open the context menu as shown below.



Select **Open in Integrated Terminal** from this menu to start a terminal session in this directory. A new terminal window will open at the bottom right of the page.



#### Ch04\_inset01\_03 terminal window

You use this window to send commands to the operating system of your computer. I'm using a Windows PC and so my commands will be performed by a terminal program called Windows PowerShell. If you are using a Mac or a Linux machine you will use the terminal program on that machine.

We tell the terminal what to do by entering text commands. For now, we are going to use the terminal to start node running. Click in the terminal window to make it active and then type the **node** at the terminal command prompt as shown below.



#### Ch04\_inset01\_04 starting node

Next press Enter to start the node program running. The node program is now running inside the terminal. Any commands that you enter will be processed by node, not the terminal.



#### Ch04\_inset01\_05 running node

When node starts it displays a console prompt. If you don't see the output above, make sure that the install process completed correctly. Enter the sum below and watch what happens.

```
> 2+2+2  
6
```

You should see how eager node is to help as you type in the calculation. It will execute the partial statement and show results even before you've typed in the whole line. You can use this console in the same way as you use the console in the browser developer tools. You can type in a JavaScript statement; it will be obeyed and the result it generates is printed.

We are not going to be using the node console very much. We've just run it now to make sure that node works on our machine. We will be using node to run JavaScript programs that we have written. To exit node type in **.exit** (don't forget the period at the beginning) and press enter. This will stop node running and return you the terminal.

The terminal has been opened in a directory which contains a simple JavaScript source file. We can now use node to run this program. To do this we issue the node command and follow it with the name of the JavaScript source file. We want to run a program in the file called **sayhello.js**

```
console.log("We can run this program using node");
```

Above you can see the contents of the program file. It just prints a message on the console. We can run this program by starting the node program and giving it the filename as an argument. Type in "node sayhello" as shown below.

```
Try the new cross-platform PowerShell https://aka.ms/powershell  
PS C:\Users\emil\source\repos\Building-Apps-and-Games-in-the-Cloud\github.io\code\004-01\04-01_Tiny_Team_App\code>  
Welcome to Node.js v16.17.0.  
Type ".help" for more information.  
> 2+2+2  
6  
> .exit  
PS C:\Users\emil\source\repos\Building-Apps-and-Games-in-the-Cloud\github.io\code\004-01\04-01_Tiny_Team_App\code>
```

Ch04\_inset01\_06 running sayhello

Now press enter. The terminal will run the **node** program and pass it the string **sayhello** as an argument to the program. The node program will open the file **sayhello.js** and run the JavaScript code in that file.

```
PS C:\Users\emil\source\repos\Building-Apps-and-Games-in-the-Cloud\github.io\code\004-01\04-01_Tiny_Team_App\code>  
Welcome to Node.js v16.17.0.  
Type ".help" for more information.  
> 2+2+2  
6  
> .exit  
PS C:\Users\emil\source\repos\Building-Apps-and-Games-in-the-Cloud\github.io\code\004-01\04-01_Tiny_Team_App\code>  
PS C:\Users\emil\source\repos\Building-Apps-and-Games-in-the-Cloud\github.io\code\004-01\04-01_Tiny_Team_App\code> node sayhello  
We can run this program using node  
PS C:\Users\emil\source\repos\Building-Apps-and-Games-in-the-Cloud\github.io\code\004-01\04-01_Tiny_Team_App\code>
```

Ch04\_inset01\_07 sayhello output

Above is the result of running the sayhello program. The message has been displayed on the terminal, the program has ended, and the node program has ended. You can now close the

terminal session by using the **exit** command.

## CODE ANALYSIS

### Running node

You might have some questions about terminal and node.

#### Why does mine not work?

There are a few reasons why your program might not run. The most obvious one is that you might have typed the wrong filename. If you give filename of **syHello** the node program will be unable to find such a file and you get an error. The error is not a simple “Hey, you got the filename wrong”. Instead, you get a whole bunch of error reports followed by the message “MODULE\_NOT\_FOUND”. You can also get this error if your filename doesn’t have the language extension “.js” which identifies a file as holding JavaScript code. And, most confusingly, you can get this error on some systems if you give a filename of **sayhello**.

On a Windows PC, whether letters in a file name are CAPITALS or lower case is not significant, so if you enter the name **sayhello** the Windows file system will quite happily match this with **sayHello**. Unfortunately, operating systems based on the Unix – which includes those used in Linux and Apple systems don’t do this matching, which means we can issue commands that work on a Windows device but don’t work on others.

There’s another reason that the command might not work, and that is because you are running the terminal program in the wrong directory. Whenever you are using terminal it keeps track of its “current directory”. Whenever a command or a program specifies a filename, the terminal program will look in the current directory for the file. We started Terminal by using the command “Open in Integrated Terminal” at the directory containing our example code. If we had opened the wrong directory the **node** program would be unable to find the **sayHello** program. The **cd** command lets you tell terminal to move to a different directory. You can find out more about the **cd** command in the glossary entry for the terminal.

#### What happens if the sayhello program contains an infinite loop?

The node program will run a JavaScript program until the JavaScript program ends. If the JavaScript program never ends, node will keep running. This is frequently what we want. If node is hosting web server program written in JavaScript we want the server program to run for ever and so server program will run forever. We can stop a running JavaScript program in node by using keycode CTRL+C (i.e. hold down the control key and press C) in the terminal window.

#### How does a node program communicate with the user?

The node system will run a JavaScript program but it doesn't provide a document object, so we can't create HTML elements and then change their properties to create a display for the user. Node is intended for tasks such as hosting web sites and node programs do not usually interact with users. However, a node program can use the **console.log** function to send messages. These messages can be very helpful when monitoring applications run by node.

#### How does the terminal program find the node program?

This is an interesting question. We've just discovered that the Terminal program looks in the current directory to find files. However, the node program isn't in the current directory. And yet the Terminal program can find the node program and run it. How does this work?

The operating system manages a set of **environment variables** which, as the name implies, describe the environment for programs on that computer. One of these variables is called the **path**. The path is a list of "places to look" for things. If I can't find my keys I'll check the front door, the kitchen door, the key hooks, my pocket and finally my right hand. This list of locations is my "path" for finding keys. In the case of my computer the path is places to look for a program to run.

When you enter the command **node** the terminal program searches through all the directories specified in the **path** variable to see if any of them contain a program called node. When it finds the node program it runs it. The installation process for the node application added the location of the directory containing the node program to the path on the computer. You can use the PowerShell command **\$Env:path** to display the contents of the path on your Windows PC. You might be surprised by the number of different places there are:

```
C:\WINDOWS\system32;C:\WINDOWS;C:\WINDOWS\System32\Wbem;C:\WINDOWS\System32\WindowsPowerShell\v1.0\;C:\WINDOWS\System32\OpenSSH\;C:\Program Files\Microsoft SQL Server\130\Tools\Binn\;C:\Program Files\Microsoft SQL Server\Client SDK\ODBC\170\Tools\Binn\;C:\Program Files\Git\cmd;C:\ProgramData\chocolatey\bin;C:\Program Files\dotnet\;C:\Program Files\CMake\bin;C:\Program Files\nodejs\;C:\Users\rsmil\AppData\Local\Mu\bin;C:\Users\rsmil\AppData\Local\Microsoft\WindowsApps;C:\Users\rsmil\AppData\Local\Programs\Microsoft VS Code\bin;C:\Users\rsmil\AppData\Local\GitHubDesktop\bin;C:\Users\rsmil\.dotnet\tools;C:\Users\rsmil\AppData\Local\Microsoft\WindowsApps;C:\Program Files\heroku\bin;C:\Users\rsmil\.dotnet\tools;C:\Users\rsmil\AppData\Roaming\npm
```

Above you can see some of the directories in the **\$Env:path** variable on my machine. The directory containing node is in bold.

#### Programmer's Point

#### Learn to use the terminal

It is worth spending some time learning how to get the best out of the **terminal**. The tab shortcut is just one of many tricks that you can use. You can find out more about the terminal and how to use it in the Glossary.

## JavaScript Heroes: Modules

Node is much more than just a place you can run JavaScript programs. It also allows you to create programs that are made up of modules. Modules are another JavaScript hero you should know about. A module is a package of JavaScript code that you want to reuse. A module can contain functions, variables, and classes. Some of the elements in a module source file can be “exported” from the module for use in other programs. The first implementation of modules was created as part of the node framework. It uses a function called `require` to import items from a module that has exported them. Let’s look at how it works.

### Create a module and require it

Let’s consider something we might like to turn into a module. In the last chapter we created a function called `getRandom` to generate random numbers. We used it to pick the position of the mine in the game Mine Finder that we built.

```
function getRandom(min, max) {  
  var range = max - min;  
  var result = Math.floor(Math.random() * (range)) + min;  
  return result;  
}
```

Above you can see the code for `getRandom`. When a program needs a random number in a particular range it can use the function to get one:

```
let spots = getRandom(1,7);
```

The above statement creates a variable called `spots` which is set to the result of a call to `getRandom`. The variable `spots` will contain a value in the range 1 to 6. We could use the value of `spots` to replace dice in a board game. Note that the upper limit of our random number generator is *exclusive*. We will never get 7 returned as the number of spots.

We might want to use `getRandom` in another application that also needs random numbers. We

could just copy the text of the function into the new application, but if we ever find a bug in `getRandom` we would then have to find all the applications where `getRandom` has been used and fix the code in each one. If our programs all used a single shared version, we'd just have to fix the fault in one file and then it would be fixed in all the programs. Modules have other advantages too. A module can be developed independently of the rest of the application, perhaps by a different programmer.

```
function getRandom(min, max) {52
  var range = max - min;
  var result = Math.floor(Math.random() * (range)) + min;
  return result;
}
exports.getRandom = getRandom;53
```

We can make a JavaScript file into a module by adding an `exports` statement that specifies what is being exported. You can see it above. The code above exports just one function; `getRandom`. The name of the function as it is exported is also `getRandom`.

This code could be placed in a source file called `randomModule.js`. A program that wants to use the `getRandom` function can use the `require` function to load the module containing it.

```
const randomModule = require("./randomModule");54
let spots = randomModule.getRandom(1,7);55
```

The two statements above show how a node.js application uses `getRandom`. The variable `randomModule` is declared as a constant and then set to refer the result from the `require` function. The `require` function is supplied with a string giving the file path to the source file `randomModule.js`. The character sequence `"./"` in front of the string tells the `require` function to look in the same directory as the one containing the program.

## Make Something Happen

---

<sup>52</sup> Create the function to be exported

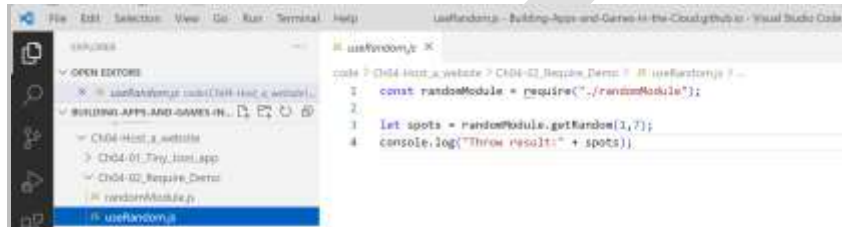
<sup>53</sup> Export the function with the name `getRandom`

<sup>54</sup> Import the function

<sup>55</sup> Use the function in our program.

## Use debug to investigate the require statement

In Chapter 2 in the section **Code Analysis: Calling Functions** we used the debugger in the browser to discover how JavaScript functions are called. Now we are going to use the node.js debugger in Visual Studio Code to investigate how **require** is used to load modules into a program. Start Visual Studio Code and open the GitHub repository for the sample code for this book. Now find the **Ch04-02\_Require\_Demo** directory and open the file **useRandom.js**:



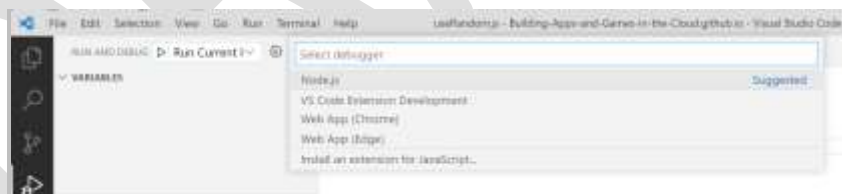
Ch04\_inset03\_01 opening the require demo

This program uses **require** to load the **randomModule.js** file. Then it calls **getRandom** from the module. We can use the Visual Studio Code debugger to step through the code in this program. We do this by starting the debugger. Click on the debug icon in the left-hand column. It looks like a run key button with a little bug sitting on it:



Ch04\_inset03\_02 starting the debugger

The first time you start the debugger it will ask you which debugger to use:



Ch04\_inset03\_03 selecting the node.js debugger

Select Node.js as suggested. The Run and Debug window is now displayed on the left of the Visual Studio Code window.



Ch04\_inset03\_04 starting the debugger



Now we can start the program running in the debugger. Press the **Run and Debug** button at the top of the Run and Debug menu to run the program.



Ch04\_inset03\_05 program completed

You can see above that the program has run and displayed a throw result of 3. You can click the links next to the console log displayed to visit the lines of JavaScript that produced the outputs. This output shows that the program ran correctly, but what we would really like to do is use the debugger to discover how the require process works. We can do this by setting a breakpoint in this program and then stepping through the code.

We've already used breakpoints in the debugger in the browser, they are set in the same way here. Click the left-hand side of the line number. In this debugger a breakpoint is indicated by a red dot in front of the line. Set a breakpoint at the first statement in the program as shown below:



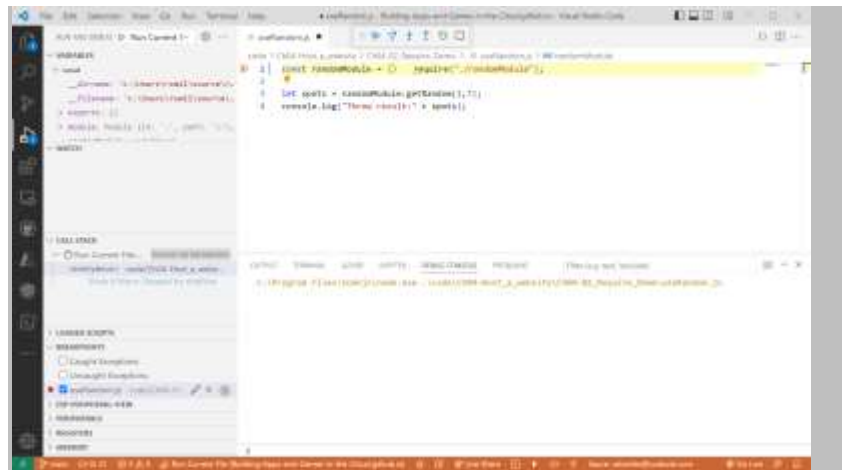
Ch04\_inset03\_06 setting a breakpoint

When we run the program it will stop at this statement and we can take a look at what it is doing. Now we need to restart the program. Click the green triangle next to the Run Current dropdown at the top of the debugger pane to do this.



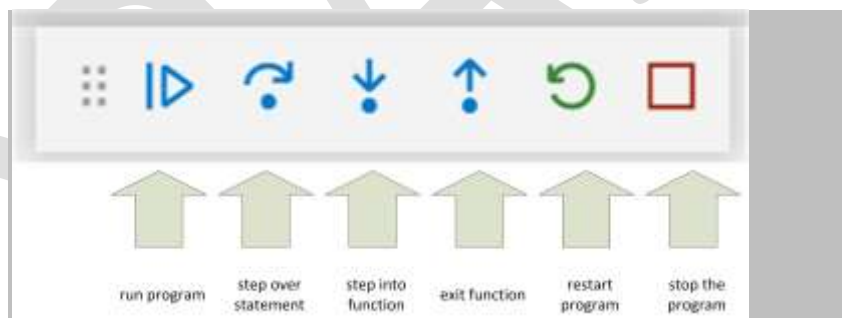
Ch04\_inset03\_07 rerun the debugger

The node environment will now run the JavaScript program until it hits the breakpoint.



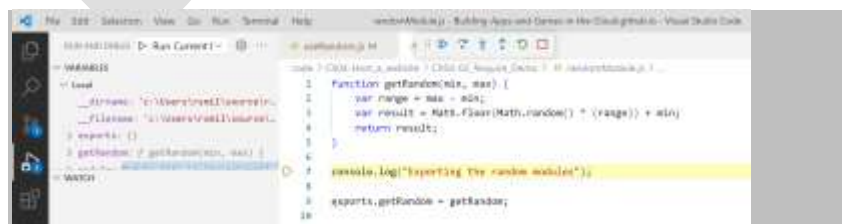
Ch04\_inset03\_08 hitting the breakpoint

Above you can see how the program stopped at the first statement in `useRandom`. At the top of the window you can see a set of debug controls that look rather like those we saw in the Developer Tools for the browser.



Ch04\_inset03\_09 debug controls

Press the “step into function” control (the downward pointing arrow) to run the next statement. This will perform the require statement.



When a JavaScript program performs a [require](#) it executes the JavaScript in the module file that is being required. Above you can see that the module contains a statement that logs a message to the console prior to exporting the [getRandom](#) function. You can repeatedly press the “step into function” button to watch the program run through. When execution reaches the end of the [randomModule](#) source file it returns to [useRandom](#) and calls the [getRandom](#) function.

It makes sense to store modules in a different place from application code. The node environment is designed to allow this. We can create a directory called **node\_modules** and node will search this directory to find required module files. The example **Ch04-03\_Library\_Folder** contains a **node\_modules** directory which contains the module files.

```
const randomModule = require("randomModule");56

let spots = randomModule.getRandom(1,7);57
console.log("Throw result:" + spots);58
```

The code above shows how a module in the **node\_modules** folder is accessed. We don’t need to put the “./” prefix to the path to the module file if we store the module file in **node\_modules** folder. You can open the programs in the debugger and step through them to see how they work.

## Require and Import

The require mechanism works but it does have some disadvantages. A program can use [require](#) to load a module at any point in a program’s execution. You might think that this is a good thing because it is very flexible. However, if you want to manage how external components are used in an application (and this is an important thing to do in large projects) you really should not allow programmers to load modules at any time. It is much more sensible to require all the modules to be loaded at the start a program.

Another issue is that the require mechanism runs *synchronously*. We have seen that when a program uses [require](#) to load a module it runs the entire contents of the JavaScript file containing the module code. This takes place at the time the require is performed, which will cause a program to pause while modules are loaded. If an application contains multiple requires each must be

---

<sup>56</sup> *Load the module*

<sup>57</sup> *Use getRandom from the module*

<sup>58</sup> *Print the result to the console*

completed before the next can be performed. It is also not possible to use [require](#) to only load particular elements from a module. The whole module must be scanned each time it is used.

To address these issues JavaScript was given [import](#) and [export](#) declarations. These provide the same functionality as [require](#) but in a slightly different way.

```
function getRandom(min, max) {  
    var range = max - min;  
    var result = Math.floor(Math.random() * (range)) + min;  
    return result;  
}  
  
export {getRandom} ;59
```

Above you can see the JavaScript code for a module that exports the [getRandom](#) function. Another module can import this by using the [import](#) declaration:

```
import { getRandom } from "./randomModule.mjs";60  
  
let spots = getRandom(1,7);  
console.log("Throw result:" + spots);
```

The code above imports the [getRandom](#) function from a local module file called [randomModule.mjs](#). It then calls the function to generate a [spots](#) value. There are some important things to remember when you are using import.

- The module file and any files that import modules must have the language extension “.mjs” rather than the usual “.js” which means JavaScript program.
- All the imports for a module must be performed at the start of the module.

You can find these example files in the source code folder **Ch04-04\_Import\_Demo**. If you are creating modules for your own projects I think you should use the [import](#) mechanism. It is important

---

<sup>59</sup> *Export the getRandom function*

<sup>60</sup> *Import the getRandom function*

to know about `require` because it is very likely that you'll see it used in older projects that you might work on. It is possible to create modules that can be used with either `require` or `import`. All the libraries that we are going to use in node to create our own web servers can be imported or required, we are going to use `import` in all the examples.

## Using import in the browser

Up until now we have created our JavaScript applications by embedding code directly in a web page. However, this is not a good solution if you're building large projects. We might also like to use code from modules in JavaScript applications that run in the browser. Let's see how we can create a web page that uses a module.

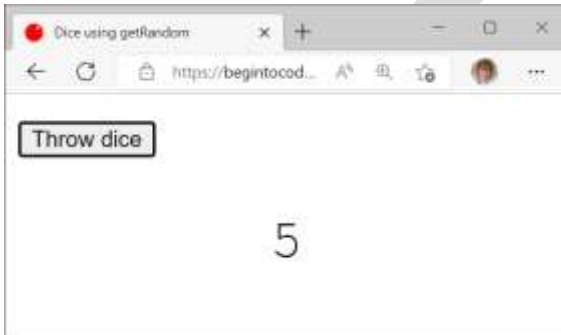


Figure 4.17 Ch-04\_Fig\_02 Browser dice

Figure 4.2 above shows what we are creating. When the Throw Dice button is clicked a new random value between 1 and 6 is displayed on the page. This page uses the same random number module as we have seen earlier. You can find the files behind the site in the examples in the directory **Ch04-05\_Browser\_Import**. You can view the web page here: [https://beginintocodecloud.com/code/Ch04-Host\\_a\\_website/Ch04-05\\_Browser\\_Import/index.html](https://beginintocodecloud.com/code/Ch04-Host_a_website/Ch04-05_Browser_Import/index.html)

```
<!DOCTYPE html>
<html>

<head>
  <title>Dice using getRandom</title>
  <link rel="shortcut icon" type="image/x-icon" href="favicon.ico">
  <link rel="stylesheet" href="styles.css">
</head>

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

```
</p>
<p id="dicePar" class="dice">*</p>
<script type="module">61
  import {doStartPage} from "./pageCode.mjs";62
  doStartPage();63
</script>
</body>
</html>
```

This is the HTML file for the dice web page. It contains a definition for a “Throw dice” button. Our application needs to know when this button is clicked so that it can generate and display new dice throw value. Up until now the definition of a button in the HTML for the web page contained an `onclick` attribute that calls a handler function when the button is clicked:

```
<button onclick="selectFastClock();">Fast Clock</button>
```

This is a button definition from web page for the example **Ch02-04\_Time Travel Clock** that we worked on in Chapter 2. It defines the button that is pressed when we want the clock to be five minutes fast. The `onclick` attribute contains the JavaScript code “`selectFastClock();`” that is performed when the button is clicked. It means that when the button is clicked the `selectFastClock` function runs. This works, but it means that the creator of the web page (which contains the button element) must agree with the creator of the JavaScript (which contains the event handler) on the name of the event handler function and how it is called.

A better way to do this is to make the developer in charge of connecting the event handler to the button. Then they can call the handler function whatever they like. All the developer needs to know is the id of the button element in the page. In the HTML for the dice page the button is given an id of “diceButton”.

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

---

<sup>61</sup> Specify that the script is a module

<sup>62</sup> Import the `doStartPage` function

<sup>63</sup> Call the `doStartPage` function

This is the HTML from the dice page. It defines a dice button which is clicked when the user wants a new dice value to be displayed. The element has the id “diceButton”. The binding of a function to the click event is performed in the `doStart` function which is held in a JavaScript module called `pageCode.mjs`. The HTML file imports the `doStartPage` function from this module and then calls the function to start the page running. This is the `pageCode.mjs` source file:

```
import {getRandom} from '/modules/randomModule.mjs';64

function doThrowDice() {65
  let outputElement = document.getElementById("dicePar");
  let spots = getRandom(1,7);
  outputElement.textContent = spots;
}

function doStartPage(){66
  let diceButton = document.getElementById("diceButton");67
  diceButton.addEventListener("click", doThrowDice);68
}

export{doStartPage};69
```

This file contains two functions. The first one is `doThrowDice`, which is called to display a new dice value. The second function is `doStartPage` which is called to start the page running. This function connects the `doThrowDice` function to the click event on the button. We have seen `addEventListener` before. We used it to add event listeners to the buttons in the Mine Finder game we created in chapter 3. The `doStartPage` function is exported from the module so that it can be imported and used in the web page.

## CODE ANALYSIS

---

<sup>64</sup> *Import the getRandom function*

<sup>65</sup> *Function that throws the dice*

<sup>66</sup> *Starts the page running*

<sup>67</sup> *Find the dice button*

<sup>68</sup> *Bind an event handler to it*

<sup>69</sup> *Export the doStartPage function*

## Modules in the browser

You might have some questions about this code.

Why must the JavaScript files have the language extension **.mjs**?

JavaScript handles module files differently from “ordinary” files. The **import** declaration only works in a module file. Module files also have **strict** mode enabled by default. We first saw the JavaScript **strict** mode in the Make Something Happen: Investigate **let**, **var** and **const** in chapter 3. It asks JavaScript to perform extra checks to make sure that your program is correct. This means that JavaScript needs to know when it is processing a module file. This difference is indicated by a different language extension. A language extension is added at the end of a filename, preceded by a period (**.**). The language extension **.js** means “JavaScript program”. The language extension **.mjs** means “JavaScript module”.

If you want to indicate that JavaScript code in an HTML file is a module you use the **type** attribute of the JavaScript element in the HTML file. For a standard JavaScript program the type is **text/javascript** but for a module the type is **module** so that the JavaScript engine in the browser knows it is module code and can use **import**.

Where is the file **randomModule.mjs** stored?

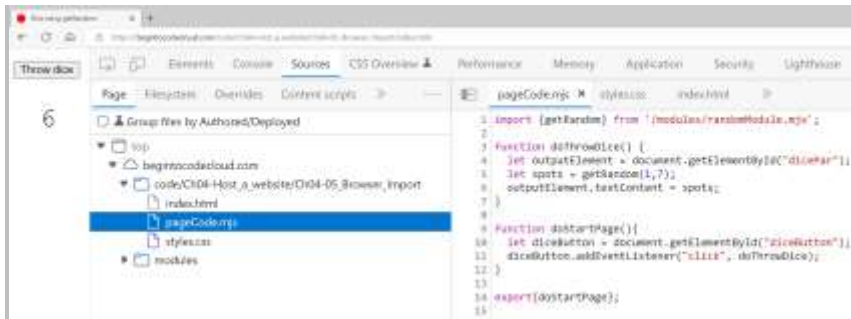
```
import {getRandom} from './randomModule.mjs';
```

Above you can see a statement that imports **getRandom** into a program. The word **from** is followed by the path to the file that contains the JavaScript code to be imported. The path starts with the sequence **“./”** which tells JavaScript to look for the **randomModule.mjs** file in the same folder as the program source. This works, but it means that every application has its own copy of the module file.

```
import {getRandom} from '/modules/randomModule.mjs';
```

This is the statement that imports the **getRandom** module for the dice website. Now the path doesn’t start with a period (**.**). Leaving off the leading period tells JavaScript to look in the top directory of the website rather than the directory containing the JavaScript program. At the top of my website I’ve created a directory called **modules** which stores all my module files. There is a copy of the **randomModule.mjs** file in that directory. This means that all the pages in my website can import from this module.





#### Ch04\_inset04\_01 Modules in the browser

Above you can see how this all fits together. The Sources view in the browser Developer Tools shows where all the files are located and you can browse each one.

Can a node application and a browser share the same module files?

Yes, as long as you are careful where you put the shared module files. If you leave the leading period of a file path given in a node application this means “look at the root of the storage device holding the node program”, which is slightly different from the root of a web site.

Can a module export more than one item?

Yes. You just add the items to the list of things that are exported. You can export variables as well as functions.

Can you declare variables in a module file?

Yes you can. Variables that are declared global to the module (i.e. declared outside any function) will be visible to code in that module file only. To understand how this works (and why we are doing it), suppose that we wanted to make a dice that displayed how many times it had been thrown. We would need a variable to keep track of the number of times the dice has been thrown and then update this and display it after every dice throw:

```
var throwCount = 0;

function doThrowDice() {
  let outputElement = document.getElementById("dicePar");
  let spots = getRandom(1,7);
  throwCount = throwCount + 1;
  outputElement.textContent = spots + " " + throwCount;
}
```

Above you can see how we could do this. This code is in the **pageCode.mjs** source file. It uses a modified version of the **doThrowDice**. The **throwCount** variable is used by the **doThrowDice** function which is called each time the “Throw Dice” button is pressed. The function increments the **throwCount** variable and then displays it after the number of spots. You can find this version of the dice in the example **Ch04-06\_Throw\_Counter**.

This is a neat way of “hiding” variables that you don’t want people to have access to. If the variable is declared inside a module it is only useable by code outside the module if it is explicitly imported.

## The dark side of modules

In the next section we’re going to build a working web server using just a few lines of JavaScript and a lot of code imported from modules. However, before we do that, we should look at the “dark side” of modules and mention something you might like to consider when you use them. Let’s start with a look at a special new version of the `getRandom` function that is supposed to return a random number:

```
function getRandom(minimum, maximum) {  
    var range = maximum - minimum;  
    var result = Math.floor(Math.random() * (range)) + minimum;  
  
    let currentDate = new Date();70  
    if(currentDate.getMinutes() < 10){71  
        if(result > minimum){72  
            result = result - 1;73  
        }  
    }  
    return result;  
}
```

We’ve used the `getRandom` function to generate values for the Mine Finder program and the dice. However, this version of `getRandom` has an extra special feature. For the first ten minutes of every hour the function subtracts 1 from the result. You might be wondering why you might write code like this. But it means that for ten minutes in every hour I can say “I’ll give you a million pounds if this dice rolls a six” and be sure that I won’t lose any money. Because in that time it is impossible to roll a six. If the above version of `getRandom` ended up in a program used by a casino I could use this knowledge to my advantage. You can find this tampered version of the program

---

<sup>70</sup> *Get the date*

<sup>71</sup> *Are we in minute 1?*

<sup>72</sup> *Is the result bigger than the minimum?*

<sup>73</sup> *Subtract 1 from the result*

in the example **Ch04-07\_Tampered\_Random**.

When you are using modules you need to be careful that the code inside them doesn't have any nasty extra features like the function above. It is possible that a module will contain faulty code, but it is also possible for a module to contain malicious code. There have even been reports of people copying GitHub repositories and making tampered versions of libraries for unwary developers to use in their applications. Make sure that you are using the "proper" version of a library by checking the activity level on the GitHub site.

## Make a web server

In chapter 2, we installed the Live Server Extension in Visual Studio Code. Ever since then we have used this extension to view the web pages that we have created in Visual Studio. Live Server provides a tiny web server that runs on our machine. When the browser asks for a web page the Live Server program finds the file containing the page and sends it back to the browser. Now we are going to create a web server which is powered by JavaScript code running inside node.js. This can send files back to a browser, but it can also generate HTML directly from code.

You might think that hosting our own site would mean that we must put something in the cloud, but this is not the case. We can use node.js to run a web server on our local machine and then connect to it with our browser. Our server will listen on a network **port** for incoming requests. When a message comes in the server will generate an HTML formatted response and send it back. We will give the browser a **localhost** network address for the server address so that it connects to our local computer. The server will use the Hyper Text Transfer Protocol (**HTTP**) to interact with the browser. We will connect event handler functions that will respond when requests arrive.

Once the server code is complete it can be moved into the cloud so that the service we have created can be used by anyone around the world. There are cloud hosting services that can take our JavaScript code and run it for us. There is even an extension for Visual Studio code that can take our site and place it in the cloud for us.

## Serving from software

```
import http from 'http';74
```

---

<sup>74</sup> *Load the http library*

```
function handlePageRequest(request, response){75
  response.statusCode = 200; 76
  response.setHeader('Content-Type', 'text/plain'); 77
  response.write('Hello from Simple Server'); 78
  response.end(); 79
}

var server = http.createServer(handlePageRequest);80

console.log("Server running");

server.listen(8080);81
```

The program above hosts a web page that returns a web page containing the text “Hello from Simple Server” when accessed from the web. The web page is hosted on the local machine at port 8080. Let’s use the debugger to step through the code and watch the server build a web response and return it.

## Make Something Happen

## Use debug to investigate the server

We can use the Visual Studio Code debugger to watch our tiny web server in action. Start Visual Studio Code and open the GitHub repository for the sample code for this book. Now use the Explorer to find the **Ch04-08\_Simple\_Web\_Server** directory and open the file **server.mjs**.

This is the most complicated exercise that we have performed so far. In it you will use two programs, Visual Studio Code and your browser. Visual Studio Code will use node to run a

---

<sup>75</sup> Function to deal with a request

<sup>76</sup> Set the status code for the response

<sup>77</sup> Set the content type to text

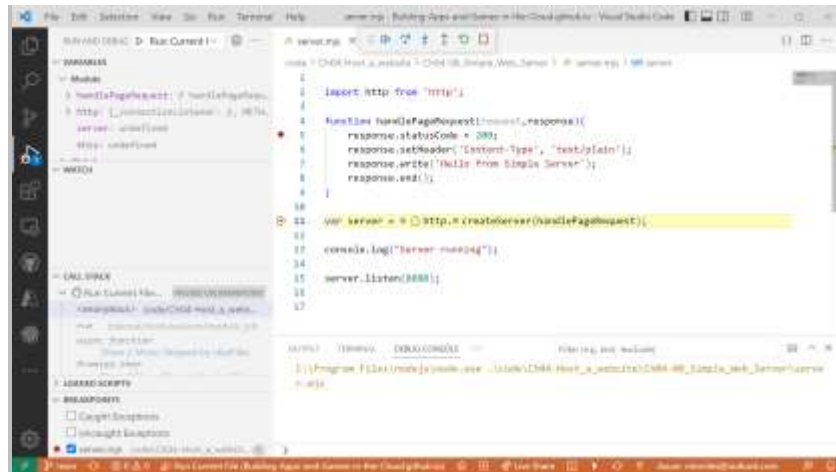
<sup>78</sup> Add the content

<sup>79</sup> Send the response

<sup>80</sup> Create a server

<sup>81</sup> Start the server listening on port 8080





Ch04\_inset05\_03 Start server breakpoint

Above you can see that the program has hit the breakpoint at line 11. This is the statement that starts the server. You might be wondering why the breakpoint at line 5 wasn't hit. This is because that statement is inside the `handlePageRequest` function, which has not been called yet. The `handlePageRequest` function is passed into the `createServer` function so that the server knows which function to call when a page request is received. Click the step into button in the debug controls (or press F11) to execute that statement the program. The program will now move onto a statement that logs "Server running" on the console.



Ch04\_inset05\_04 Console message statement

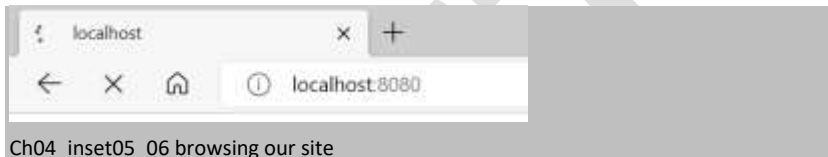
Click the step into button again to perform this statement. You should see the message appear on the console and execution moves on to the next statement, which starts the server listening by calling the `listen` function.



Ch04\_inset05\_05 Server listen statement

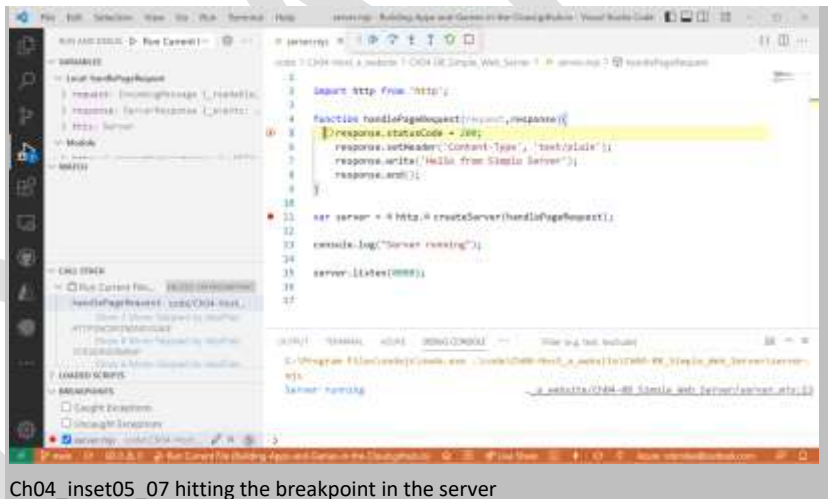
Above you can see the output from the program and the program positioned at line 15 ready to start the server listening for web requests. The call to `listen` is provided with the number of the **port** to listen to. For our demonstration server we will be using port 8080. Press the step button to perform the `listen` function. The listen function is now running and waiting for a web request on port 8080.

We can use our browser to make a web request of the site that our server is hosting. The address we are going to use is **localhost:8080**. The first part of the address is the address of the machine (in our case it is our local host) and the second part of the address is the port number (in our case it is 8080 because that is where our server program is listening). Open the Edge browser, type the address into the address bar and press enter to open the site.



Ch04\_inset05\_06 browsing our site

You will see that the browser will pause, waiting for the site to arrive from the server. Now, go back to the Visual Studio

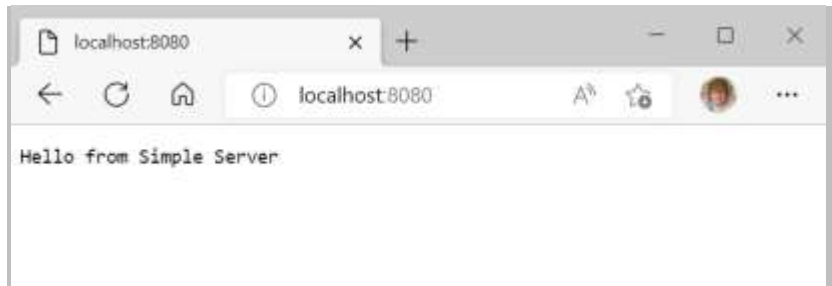


Ch04\_inset05\_07 hitting the breakpoint in the server

The program has hit the breakpoint in the `handleRequest` function. This function is called by the server when a page is requested. The job of the function is to assemble a response and then send it back to the server. We can look at what the function does a little later. For now, it is important that we run the function before the browser times out the web request.

Click the continue button in the debug controls (it's the right pointing blue triangle) to continue the `handleRequest` function. You might be expecting the web page to appear in the browser at this point, but it doesn't. Instead, you will find that the breakpoint in the `handleRequest` is hit a second time. Click the continue button again. Now you can go

back to the browser and see what has happened:



Ch04\_inset05\_08 displaying our web site

Above you can see the page produced by our server. This works because the browser it has been told that the content returned by the server is plain text, so it just displays it. We get to how this works in the next section.

Now stop the server (press the red square in the debug controls), edit the text in the call of `response.write` on line 7, run the program again see that the text that is served has changed. When you have finished with the server you can stop it again.

This is a big moment. You now know how both ends of the world wide web work. You've seen how browsers download web pages and display them, and now you know how a program can serve out a web page.

## CODE ANALYSIS

### Running a server

You might have some questions about what we have just done

What would happen if the server never sent back a response?

The browser sends a request to a web site and then waits for the response. If the response takes too long to arrive the request will time out and the browser tells you that the page is inaccessible.

How does the server build the response to the browser?

```
function handlePageRequest(request,response){
    response.statusCode = 200;
    response.setHeader('Content-Type', 'text/plain');
    response.write('Hello from Simple Server');
    response.end();
}
```



When the `handlePageRequest` function is called it is given two parameters. The first parameter, `request`, is a reference to an object that describes the request from the browser. The second parameter, `response`, is a reference that refers to an object that describes the response to be sent to the browser. At the moment we are not using the `request` parameter at all.

The `handlePageRequest` function builds the same response to any request. The first thing the function does is set the `statusCode` property of the `response` to 200. This value is sent back to the browser at the beginning of the response. The value 200 means “all is well, here is the page”. We could use other values to signal error conditions. The value 404 means “page not found”.

The next thing the function does is use the `setHeader` method in the response to set a value in the header that is to be sent to the browser. It sets the value “Content-Type” to the string “text/plain”. The browser uses the value of Content-Type to decide what to do with the incoming data. If the content type was “text/HTML” the browser would build a document object and display that. However, our server just serves out plain text.

The third statement in the `handlePageRequest` function uses the `write` method in the response to write the actual content of the page. In this case it is just a simple message, but this content could be much longer.

The final statement calls the `end` function on the response. This is the point at which the response is assembled and sent back to the browser.

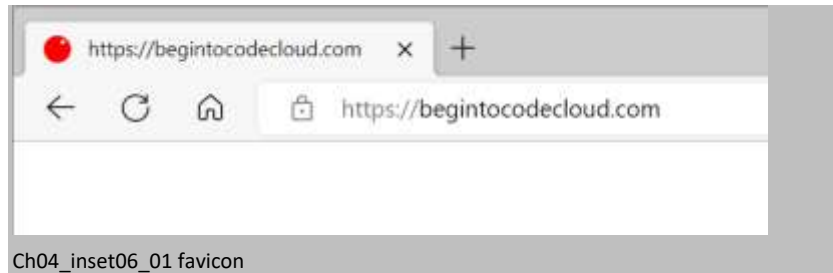
How does the server know which page is being requested by the browser?

The request parameter (which we have not used yet) contains a property called `url` which contains the **URL** (uniform resource locator) for the page that has been requested. If the browser is requesting the index page the url property is the string “/”. We will use the path in our next server, which will serve out files.

Why did the browser make two requests of the server?

In the **Make Something Happen: Use debug to investigate the server** exercise above we set a breakpoint in the `handlePageRequest` function. The breakpoint is hit when a server requests a page. The breakpoint was hit twice when we tried to load the page into the browser, which means that the browser has asked the server for two responses. Why?

This has to do with the way the web works. Many web pages, including the ones for the sample code for the site, have “favicons” on them. A favicon is the little image that is displayed on the top left-hand corner of the page.



Above you can see the favicon for the **[begintocodecloud.com](https://begintocodecloud.com)** site. It is a shiny red ball which I think it is quite artistic. When a browser loads a web site it makes two requests. One request is for the favicon image file. The other is for the actual content of the site. Our server sends back the same response to both requests; the text “Hello from Simple Server”. The browser can’t make a text message into a favicon and so it ignores it. If we want our site to have a working favicon we have to create a bitmap file of the correct type and then serve it out when the file is requested.

```
import http from 'http';
import fs from 'fs';

function handlePageRequest(request, response) {
  let url = request.url;

  console.log ("Page request for:" + url);

  if (url == "/favicon.ico") {
    console.log(" Responding with a favicon");
    response.statusCode = 200;
    response.setHeader('Content-Type', 'image/x-icon');
    fs.createReadStream('./favicon.ico').pipe(response);
  }
  else {
    console.log(" Responding with a message");
    response.statusCode = 200;
    response.setHeader('Content-Type', 'text/plain');
    response.write('Hello from Simple Server');
    response.end();
  }
}
```

This version of [handlePageRequest](#) checks the url of an incoming request. If the request is for a favicon it will open the icon file and send it back to the server. Otherwise, it sends the message “Hello from Simple Server”. You can find this version of the simple server in the folder **Ch04-09\_Simple\_Web\_Server\_with\_favicon** in the sample code for this chapter. If you use this server you should see that the browser displays a shiny red favicon for the page. We will take a close look at how pages are sent back to server in the next section.

# Serving out files

The server we have just created always delivers the same text back to the browser – the string “Hello from Simple Server”. We can turn it into a more useful server by allowing the browser to specify the file that the server is to return. When we ask a browser to show us a particular page the address of the page is expressed as a “universal resource locator” or **url**. This tells the browser where to go and look for a page.



Figure 4.18 Ch-04\_Fig\_03 url structure

An overview of a url is shown in Figure 4.3 above. The protocol and the host elements tell the browser the address of the computer and how to talk to it. The path specifies the file on the server that is to be read. The port value (if present) specifies the network port on the computer to connect to. If the port elements are missing the browser will try to connect to port 80.

The web server uses the value of the path to find the file that has been requested. The path in Figure 4.3 is for the **index.html** file for the **beginintocodecloud.com** website. If you leave the path off the address the server will send the index file automatically. Our server can use the url property of a web request to determine what is to be sent back to the browser.

```
import http from 'http';
import fs from 'fs';
import path from 'path';

function handlePageRequest(request, response) {
  let url = request.url;82

  console.log("Page request for:" + url);
```

---

<sup>82</sup> *Get the url from the response*

```

let filePath = '.' + url;83

if (fs.existsSync(filePath)) {84
  console.log("    found file OK")
  response.statusCode = 200;
  let extension = path.extname(url);85
  switch (extension) {86
    case '.html':
      response.setHeader('Content-Type', 'text/HTML');
      break;
    case '.css':
      response.setHeader('Content-Type', 'text/css');
      break;
    case '.ico':
      response.setHeader('Content-Type', 'image/x-icon');
      break;
    case '.mjs':
      response.setHeader('Content-Type', 'text/javascript');
      break;
  }
  let readStream = fs.createReadStream(filePath);87
  readStream.pipe(response);88
}
else {89
  console.log("    file not found")
  response.statusCode = 404;
  response.setHeader('Content-Type', 'text/plain');
  response.write("Cant find file at: " + filePath);
  response.end();
}
}

```

---

<sup>83</sup> Add make the url into a local path

<sup>84</sup> Check if the file exists

<sup>85</sup> Get the file extension of the url

<sup>86</sup> Select the content type

<sup>87</sup> Create a read stream for the file

<sup>88</sup> Pipe the stream into the response

<sup>89</sup> If the file doesn't exist send file not found

```
let server = http.createServer(handlePageRequest);

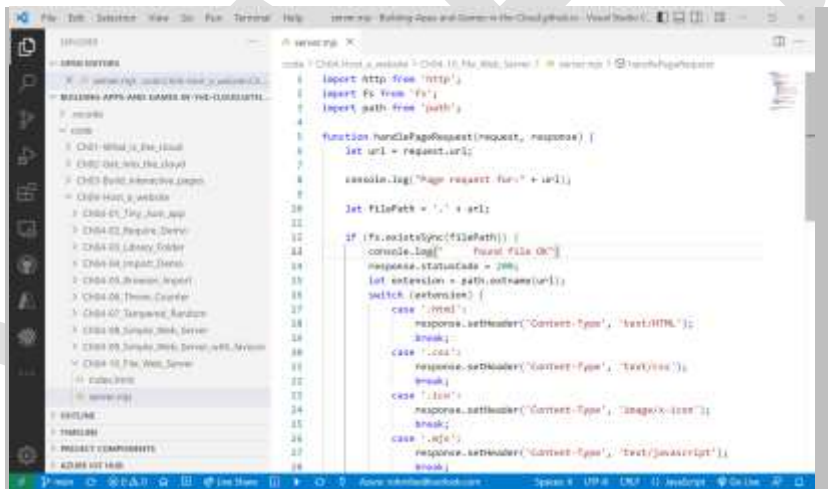
console.log("Server running");

server.listen(8080);
```

## Make Something Happen

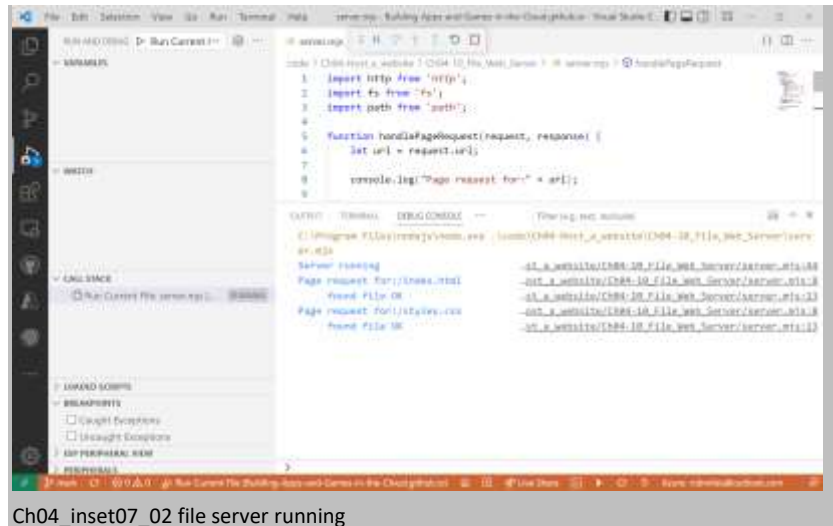
### Using the file server

You can use the above program to view the entire web site for this book. Start Visual Studio Code and open the GitHub repository for the sample code for this book. Now use the Explorer to find the **Ch04-10\_File\_Web\_Server** directory and open the file **server.mjs**.



Ch04\_inset07\_01 file webserver

Now select the debugger and start the program running.



Ch04\_inset07\_02 file server running

Above you can see a debugging session running. I used the browser to open the file <http://localhost:8080/index.html> which is served by this program. The server outputs the name of each file as the browser requests it. The server has sent two files, index.html and styles.css. If you move to other pages on the site you will see these displayed as well. I find it rather impressive that such a tiny program can act as quite a capable web server.

## CODE ANALYSIS

### Simple file server

You might have some questions about the server.

How does the server send a file back to the browser?

A node installation includes a few built-in modules. One of these is the `http` module that we are using to host our website. Another library is the `filesystem` module, `fs`, which node programs use to interact with the local file store.

```
let readStream = fs.createReadStream(filePath);
readStream.pipe(response);
```

These statements send a file back to the browser. The first statement uses the `createReadStream` function from the `fs` module to create a read stream connected to the file. The second statement uses the `pipe` function on the read stream to send the file to the web page response. You might need to think of this a bit. A stream is a bunch of data that you might want to send somewhere, like you might have a tank of water you want to use to fill a wash-basin. In real life you would use a physical pipe to link the two. In our server we have a

response object that wants to be given some data to send (the wash basin), and a file object that wants to supply that data (the water tank). We use the `pipe` method on the stream object to tell it to send the file to the response object. We don't need to worry about precisely how this works, and the response will automatically end the when the file has been received from the stream. If this seems hard to understand, go back to considering what we want to do. We have a thing that has got some data, and a thing that wants to receive some data. The pipe method will perform that transfer using a stream.

What happens if the browser asks for a page that doesn't exist?

The server uses a function from the `fs` module called `existsSync` to check if a requested file exists. This function is part of the file system library. If the file is not found the server responds with a 404 "resource not found" error code.

How does the server know what type of file to send back to the server?

When a server sends a response to a request it must always include Content-Type information so the browser knows what to do with the incoming data. The server works out the type of data to send back by looking at the file extension of the incoming url.

```
let extension = path.extname(url);
```

The statement above uses the `extname` method from the `path` module to get the extension from the url. The extension is the character sequence which is preceded by a period (.) on the end of a file path or url. As an example, the path "index.html" has the extension ".html". The extension specifies the type of data that the file contains. So "index.html" should contain html text that describes a web page. The server uses the extension string to decide what Content-Type to add to the response:

```
switch (extension) {  
  case '.html':  
    response.setHeader('Content-Type', 'text/HTML');  
    break;  
}
```

The case construction selects the response type that matches the extension string.

What happens if the browser asks for a file type that doesn't exist?

The server we have just created can handle html, css, mjs and ico file types. If it is asked for a file of a different type (perhaps a JPEG formatted image with a language extension of .jpg) it will not return a Content-Type value to the browser, resulting in the response being ignored. We can expand the range of content types that our server recognizes by creating a lookup table for the content types:

```
let fileTypeDecode = {  
  html: "text/HTML",  
  css: "text/css",  
  ico: "image/x-icon",
```

```

    mjs: "text/javascript",
    js: "text/javascript",
    jpg: "image/jpeg",
    jpeg: "image/jpeg",
    png: "image/png",
    tiff: "image/tiff"
  }

```

The code above creates a variable called `fileTypeDecode` which can be used to map language extensions onto content type strings. It allows our browser to handle a range of different image file types.

```

let extension = path.extname(url);
extension = extension.slice(1);
extension = extension.toLowerCase();
let contentType = fileTypeDecode[extension];

```

These four statements get the content type from the url. The first statement creates a variable called `extension` from the url specifying the file that has been requested by the browser. This would create `".html"` from a request for `"index.html"`. The second statement removes the leading `"."` from the extension. It would convert `".html"` to `"html"`. The third statement converts the extension to lower case. It would convert `"HTML"` to `"html"`. The fourth statement gets the file type that matches the extension from the `fileTypeDecode` object. This works because JavaScript allows us to specify the property of an object by using a string. In other words, the statements:

```
let x = fileTypeDecode.html;
```

and

```
let x = fileTypeDecode["html"];
```

- would both set the value of `x` to be `"text/HTML"`. If we try to use an extension which is not present in the `fileTypeDecode` object the value `undefined` is returned.

```

let contentType = fileTypeDecode[extension];
if (contentType == undefined) {
  console.log("    invalid content type")
  response.statusCode = 415;
  response.setHeader('Content-Type', 'text/plain');
  response.write("Unspported media type: " + filePath);
  response.end();
}
else {
  response.setHeader('Content-Type', contentType);
  let readStream = fs.createReadStream(filePath);
  readStream.pipe(response);
}

```



If the content type is not recognized the server will respond with error 415 to indicate this. Otherwise the file is piped out to the response as before. You can find this version of the server in the folder **Ch04-11\_Picture\_File\_Web\_Server** in the sample code for this chapter. You can use it to view the picture in the home page for that example. If you wish, you might like to expand the server so that it can deliver audio and video files. You just have to identify the content types for each file type and then add them to the [FileTypeDecode](#) object.

## Active sites

We now know how a JavaScript program running under node.js can serve out both active content (messages from a running program) and file content (the contents of files on the server). Most web applications use a mix of these. Fixed elements of pages will use files and then the program generated content will be inserted as required. It would be useful to have a framework where you could create “templates” which contain the fixed parts of a site and then allow you to inject the program generated content when required. The good news is that you will be learning how to do this in the next chapter. The better news is that you are now well on the way to understanding how the web works in both browser and server.

# What you have learned

This has been another busy chapter. We’ve covered a lot of ground. Here is a recap plus some points to ponder.

- Node.js is a framework that allows JavaScript programs to run outside of the browser. It is a free download for all machines. It doesn’t provide a document object model to communicate with the user. Instead, it is controlled via a terminal interface. It provides a console on which you can enter commands to run JavaScript statements. It can also be given a JavaScript program to load and execute.
- The node framework provides support for modules. A file of JavaScript code can contain statements that export data or code elements that can then be introduced into other programs by using the require statement.
- A module file can contain elements that are not exported which can be used internally by that module.
- When elements are fetched by a call to require the node framework will execute all the code in the module source file before exporting the elements. This execution takes place synchronously, i.e. the program performing the require will be paused until the require completes.
- A module source file can contain elements that are not exported. These

are local to the module and are not visible outside it.

- Node applications can be debugged in Visual Studio Code in a manner similar to JavaScript code running in the browser. You can add breakpoints to the code and view the contents of variables.
- The JavaScript language offers an alternative to the require mechanism using the `import` keyword. Modules containing import statements must have the file extension “.mjs” rather than the “.js”.
- It is not possible to use require in JavaScript code that is held in a web page and executed by the browser. However, JavaScript code running in a browser can use the import statement. JavaScript code in an HTML file that contains import statements must have the type “module”. It is not possible for JavaScript code embedded in HTML element attributes to access elements in a module. Instead code in a module must obtain a reference to a named element in the HTML file and then act on that directly.
- Whenever you use code that you didn’t write (for example importing a module you have downloaded from the internet) you should make sure that the code doesn’t contain any unwanted behaviors.
- A node.js installation contains several built-in modules. One of these is the `http` module that can be used to create JavaScript program that can act a web server.
  - The `http` module contains a `createServer` function that is called to create a web server. The `createServer` function is supplied with a reference to a function that will service incoming page requests from the browser. This function is supplied with two parameters which refer to a request object and a response object. The response object must be populated with page information by the function servicing incoming page requests. The contents of the response object are sent back to the browser making the request.
  - A response to a web request contains a `statusCode` property that gives the status of the response. A `statusCode` of 200 means that the page was found correctly.
  - A response to a web request contains a `Content-Type` property that the browser will use to decide what to do with the page when it arrives. A type of “text/plain” specifies that the file contains plain text.
- You can add plain text to a web request response by using the write method exposed by the request.
- A function servicing incoming requests to an http server also receives a

request parameter describing the request made by the browser. The request parameter contains a `url` property that gives the path to the file on the server which is being requested. The server can map this url onto local filestore to locate the file to be sent back to the server.

- The node framework provides modules called `fs` and `path` that are used to interact with the filesystem on a machine. The `fs` module contains can make stream objects that are connected to local files. A stream contains a `pipe` method that can be used to direct the content of a stream into another object. The response object sent to the server in a web request can receive file streams and send them back to the browser.
- When a browser accesses a web site it will also request a `favicon.ico` file which contains a bitmap that is displayed by the browser.
- A server must ensure that the Content Type element of a reply to a request reflects the content of the file.

To reinforce your understanding of this chapter, you might want to consider the following "profound questions".

What does node.js do?

The node.js framework lets you run JavaScript programs on a computer without using a browser.

What is the difference between synchronous and asynchronous operation?

Synchronous operations are performed "while you wait". In other words, if a program calls a function which performs a task synchronously the program will be paused until the task is complete. Asynchronous operations are performed "in the background". A call to a function will start an operation. The completion of the operation will be signaled by a call to another function. Asynchronous operations are harder to organize but they make an application more responsive as it is not paused waiting for things to happen.

When would you use a module?

You should use a module if are writing code that you want to use in several different applications. You can also use a module to share work around. Once you have decided what each module needs to do it can be developed separately. Another reason to use a module is that it gives your code more privacy. Code and variables in a module that are not exported are not visible outside the module. Modules are also useful when testing. For example, we could test a program that uses our random number generator module by creating a random number generator module to produces a fixed sequence of values. It would be tiresome to have to wait for the random number generator to produce a dice throw of 6 to test a game program. Much better to have a "testing dice" that produces the values that you need.

Why do we have require and import?

It turns out that programming languages are continually evolving as people think of new things that they want them to do. It is also the case that the first attempt to solve a problem might not be the best one. Require was developed specifically for use in node.js applications. Import was developed as a language element that built on what Require does.

Can the same module be both required and imported?

Yes it can. We have been importing modules into our node.js applications. We can also use require in node.js applications to bring in the same modules.

Where would you run a web server program?

A web server program accepts page requests from browsers and responds with content for the browser to display. You can run a web server on any computer. In this chapter we have written programs that act as web servers. Web servers for public use are run on machines that have permanent network connections or as processes in the cloud.

Can two applications on one computer share the same port number?

A port is a numbered connection to a program running on a machine. We have used 8080 as the port number behind which our server is running. Once a program has claimed a port number on a machine it is not possible for any other program running on the machine to use that port for connections.

What is the difference between a port and a path?

A program running on a computer can open a network port that can be used by other programs to connect to. Ports are specified by numbers. Port number 80 is traditionally used by web servers. A path is a string of text that specifies a how to traverse a storage system to get to a particular file or location. For example, the path **“code/ Ch04-Host\_a\_website/ Ch04-10\_File\_Web\_Server/index.html”** tells a program to find the code directory, then look in that for the **Ch04-Host\_a\_website** and so on down to the file **index.html**.

What happens if a server gets the Content-Type wrong?

A browser adds Content-Type to each response that it sends back to the browser. The browser can then work out what to do with that content. If the server makes a mistake – perhaps sending back a jpeg image with the content type “text/plain” the browser will render the content in the wrong way. If an image is marked as text the browser will show a collection of random looking characters rather than a picture. Remember that a computer has no real understanding of data. We must tell it what is in the file so that it can do the right thing with it.

Is hosting a server on your machine a dangerous thing to do?

It might be. The server that we have made can only serve out the contents of the filetypes that we have specified. In other words, it will serve out a jpeg image but not a spreadsheet or a word document. This means that if someone managed browse a path to any of the other directories on my hard disk they would not be able to look at password or system files. However, they could probably learn a lot about me from the other types of file. You should never host a public (i.e. visible to the outside world) facing web site on your own computer. Instead you should move just the files you want to share onto a separate machine or cloud service and host them from that.