

Begin to Code: Building Apps and Games in the Cloud

Rob Miles

Initial Draft

Introduction

The fundamentals of what a program does have not changed since the invention of the first computer over eighty years ago. Programs still takes data in, do something with it, and then send data out. However, the way that programs are created, deployed, and consumed has changed massively over the years, from central mainframe to personal computer to the cloud.

The cloud takes your programs and give them wings. The cloud enables you to turn your ideas into solutions that anyone in the world can use. This book gives you a handle on cloud development. It explains the evolution of the cloud, identifies challenges that it poses, and sets you on the road to becoming an accomplished cloud developer. You will learn how to code for the cloud, how to use cloud technology on your local machine, where code and data can be hosted and how applications are built from co-operating software components.

It won't always be an easy journey. Things worth doing tend to involve effort and learning how to code for the cloud is one of them. Not everything will make sense when you first see it. Cloud solutions may contain multiple moving parts which must all fit together to work correctly. In addition, there are people out there who will make it their business to try and undermine, overload, break or steal your work so you need to be prepared for this. You will have to learn how the cloud enables bad behavior as well as good. However, if you stay the course, you'll be rewarded with skills which you can use to use to take your ideas and bring them to life across the world.

How this book fits together

I've organized this book in three parts. Each part builds on the previous one with the aim of turning you into a successful cloud developer.

Part 1: The Cloud

In the first part we start by considering where the cloud came from and the drivers behind its development. Then we move on to get you started developing cloud applications. The final two chapters focus on safety and security, starting with a look at how to stay safe in the cloud and finishing with an examination of the TypeScript extension to JavaScript as a way of enforcing greater code security.

Part 2: Make a Cloud based application

In part 2 we discover how applications can be built and deployed using the cloud; starting with a simple (but useful) browser-based application and then moving on to discover how to use JavaScript to host web pages in the cloud. We'll also take a detailed look at hosting options that put your code into the cloud for anyone to access and how to connect browser and server based using JSON messages.

Part 3: Building with Cloud Technologies

In the final part we explore detailed examples of cloud applications, starting with a database powered customer facing application, moving into a cloud connected "Internet of Things" application and finishing with a connected multi-player game.

How to use this text

A good way to use the text is to read through a section away from the computer, perhaps on the bus (unless you are driving it), and then go back and work through the examples and exercises when you are sat next to a machine. This way you can pick up on the theory and context without feeling forced to do anything with it, and then you can reinforce your understanding by applying it later. Each chapter starts by setting out what you will learn and finishes with a set of questions that help you validate your understanding and give you thoughts to ponder on.

Everything will be described in a strong context. You might not initially understand how something works, but you should understand the problem it is being used to solve. Eventually you will start to see other contexts in which the tool or technique is used, at which point you can call yourself a proper developer.

You learn programming techniques and technologies best when you see them applied in a strong problem-solving context. The text is sprinkled with code examples that you can try along with suggestions of how the examples can be applied and extended.

Like learning to ride a bicycle, you'll learn by *doing*. You must put in the time and practice to learn how to do it. But this book will give you the knowledge and confidence to try your hand at programming, and it will also be around to help you if your programming doesn't turn out as you expected. Here are some elements in the book that will help you learn by doing:

Make Something Happen

Yes, the best way to learn things is by doing, so you'll find "Make Something Happen" elements throughout the text. These elements offer ways for you to practice your programming skills. Each starts with an example and then introduces some steps you can try on your own.

Everything you create will run on Windows, macOS, or Linux.

Code Analysis

A great way to learn how to program is by looking at code written by others and working out what it does (and sometimes why it doesn't do what it should). The book contains over 150 sample programs for you to look at. In this book's "Code Analysis" challenges, you'll use your deductive skills to figure out the behavior of a program, fix bugs, and suggest improvements.

What Could Go Wrong?

If you don't already know that programs can fail, you'll learn this hard lesson soon after you begin writing your first program. To help you deal with this in advance there are "What Could Go Wrong?" elements, which anticipate problems you might have and provide solutions to those problems. When we encounter something new, we might spend some time considering how it can fail and what we need to worry about when we use it.

JavaScript Heroes

There are some features of the JavaScript language that work especially well in particular situations, just like some superheroes are fast, other strong and other able to see through walls. When we hit these situations, we'll take a time out and explore in detail how the language feature works and give plenty of examples of where it can be used.

JavaScript Zeroes

JavaScript is a programming language created by fallible human beings. And like everything human-built it has inherent flaws. This does not make JavaScript a bad language, but it does mean that there are things about the language that you really should know when you start building large applications. There are not many "JavaScript Zeroes" in the book, but when you come across one it is worth taking a good look, knowing about it might save you your job one day.

Programmer's Points

I've spent a lot of time teaching programming. But I've also written many programs and sold a few to paying customers. I've learned some things the hard way that I really wish I'd known at the start. The aim of "Programmer's Points" is to give you this information up front so that you can start taking a professional view of software development as you learn how to do it.

"Programmer's Points" cover a wide range of issues, from programming to people to philosophy. I strongly advise you to read and absorb these points carefully—they can save you a lot of time in the future!

What you will need

You'll need a computer and some software to work with the programs in this book. I'm afraid I can't provide you with a computer, but in the first chapter you'll find out how you can get started with nothing more than a computer and a web browser. Later you'll discover how to use the Visual Studio Code editor to create JavaScript programs.

As you work through the book you will be creating and using cloud services which will be hosted on systems in the cloud. You might think that this would be expensive, but all the example applications are all based on technologies that are free to access on a personal level. You will have to register for some of them, but they are not going to cost you any money to use for this book.

Using a PC or laptop

You can use Windows, macOS, or Linux to create and run the programs in the text. Your PC doesn't have to be particularly powerful, but these are the minimum specifications I'd recommend:

- A 1 GHz or faster processor, preferably an Intel i5 or better.
- At least 4 gigabytes (GB) of memory (RAM), but preferably 8 GB or more.
- 256 GB hard drive space. (The JavaScript frameworks and Visual Studio Code installations take about 1 GB of hard drive space.)

There are no specific requirements for the graphics display on your machine, although a higher-resolution screen will enable you to see more when writing your programs.

Programming experience

This book will not tell you what programs do nor the fundamentals of program creation. You need to know a bit about programming, ideally with the JavaScript language. What the book will do is put lots of programming techniques into a cloud context. There are lots of examples you can use as jumping-off points for your own ideas, and we will use the cloud in lots of different scenarios, from useful applications, to turning lights on and off in your house to with "Internet of Things" devices to creating compelling multi-player gameplay.

You can write cloud applications in any programming language. But the JavaScript language has been associated with the cloud ever since the language was first built into early web browsers. JavaScript lends itself very well to cloud development, not least because of the huge number of libraries that have been built around it and the ease with which these can be used to develop solutions.

If you have a lots of JavaScript experience that is wonderful, you'll be able to get the most from

the content straight away. However, if you've programmed in any language, you should be able to get the hang of what the sample code is doing. Programming is a universal skill; the programming language is just how you present your program instructions to the computer. So don't be afraid to have a go just because your background is C, C++, Java or Python (to name a few).

Book resources

In every chapter in this book, I'll demonstrate and explain programs that teach you how create cloud applications. You can download this book's sample code from GitHub by following the link here:

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

The resources pages also include how-to videos and instructions to help you get started, along with

Acknowledgments

Thanks to everyone for giving me another chance to do this kind of thing.

We know you're busy, so we've kept it short with just a few questions. Your answers go directly to the editors at Microsoft Press. (No personal information will be requested.) Thanks in advance for your input!

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.

JavaScript and the Cloud

JavaScript grew out of the **world wide web**. When you use the web, you run a **web browser** application which interacts with a **web server** using the **internet** to connect them together. The very first browser application was text only and was used for document sharing. Later versions added graphics and animation. 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 a page more interactive without increasing the load on the web server. The program could animate the display or check user input to make sure it was correct before sending it to the server.

JavaScript and the browser

JavaScript was created to run inside a browser. There have been several different versions. 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.

JavaScript has become extremely popular. Whenever you visit a website, you will almost certainly be interacting with some 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 the server that can respond to requests from browsers. You can run a node.js server on your own computer to allow your machine to run JavaScript outside the web browser. You can also create and distribute applications that contain JavaScript code but don’t need a browser to run inside. We’ll discover how to do these things later, for now we are going to focus on running JavaScript in your web browser. You can use Microsoft Edge, Google Chrome or the Chromium browser for this chapter.

The Cloud and the server

You may be wondering where the “cloud” fits in with the world wide web. When the web was first developed the web server program accepting requests from browsers ran on a physical machine. If that machine was switched off or disconnected from the internet the web sites that it was hosting would become unavailable. If a web site became very popular the machine hosting the site would struggle and users would have to wait for pages to arrive.

The cloud was developed to address these issues. A cloud hosted resource can use multiple servers. Browser requests are directed to the nearest or the least loaded one. This works because each request that a browser makes to a server is “atomic”, it does not rely on the server remembering anything about any previous requests. When you request a resource, you are not aware of precisely which server you are using. The server is just “out there” in “the cloud”.

This explanation is a bit of a simplification. It is quite tricky to make sure that users of a cloud-

based resource all get the same experience and that data received by one server is propagated to all the others. However, the cloud is a great way to create services that can be used by many millions of people. We can put our own applications into the cloud and make them instantly available. We can also use cloud technology to “scale up” a service and add more underlying servers if it becomes popular.

In the early days of the world wide web you needed your own server machine to host your site. Nowadays there are service providers who will do this for you. Many provide service plans that are free to use for low volumes of traffic. This means you can put your web applications in the cloud for free. We are going to use free services provided by GitHub and Microsoft Azure in this text.

Our starting point for making our applications is the JavaScript programming language. You may have already written programs in JavaScript. However, you might not. If you’re not a JavaScript developer I’d advise you to make good use of the glossary each time you come across a term you are not familiar with.

This text is not intended to teach you how to program. I’m going assume that you know terms such as **variable**, **condition**, **array** and **loop**. Instead, we are going to take a very detailed look at JavaScript with a particular focus on features of interest when making applications to run in and interact with the cloud. Then we are going to look at the features provided by the browser and by frameworks such as node.js and how these are used from JavaScript applications. We are going to start by looking at JavaScript functions and discover how to use the JavaScript application programming interface. On the way we will touch on lots of other JavaScript topics.

JavaScript functions

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.

We already have experience of functions from real life. When I drive my car, I’m surrounded by buttons which I can press to make things happen. Each button triggers a function. In my first car most of the function buttons were connected to mechanical systems. When I pressed the windscreen wash button it moved a piston in a pump that sprayed water on the screen. In my current car the windscreen wash button is monitored by the car computer system. When the windscreen wash button is pressed the car computer calls a function (probably called “washScreen”) to turn on the pumps and motors that do the job.

When your application runs inside the browser it has a huge range of “buttons” it can press. Each of them is provided in the form of a function that is part of 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 some buttons in my car that I have never pressed. This is because I don't know what they do. This doesn't bother me too much. I know enough to drive safely, and I can always use the car handbook to find out more if I need to.

It is the same with programming languages. 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.

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.

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

We call the 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 1.1 Alert box

Figure 1.1 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 beginnocodecloud.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.

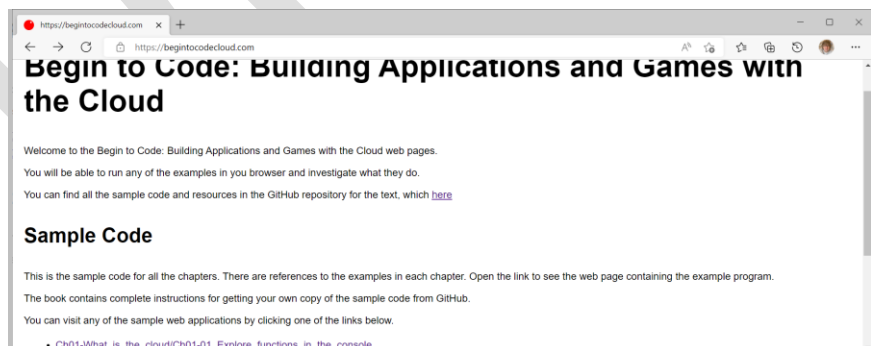
Lifting the lid on JavaScript

Wouldn't it be nice if we could watch the `doAddition` function run? It turns out that we can. The Edge, Chrome and Chromium browsers contain a very powerful "Developer Tools Console" which lets you type in JavaScript statements and view the results. Let's do our first "Make Something Happen" and explore what these functions do 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 beginnocodecloud.com. Open this web site and scroll down to the Samples section on the page.



Ch01_inset01_01 Web Site Samples

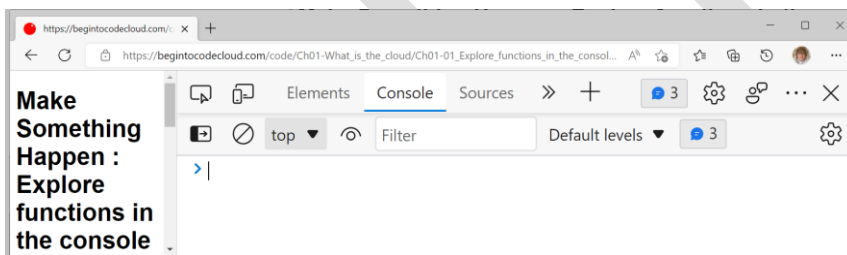
The sample code is presented as a list of links. There is a link for each sample page. We are

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

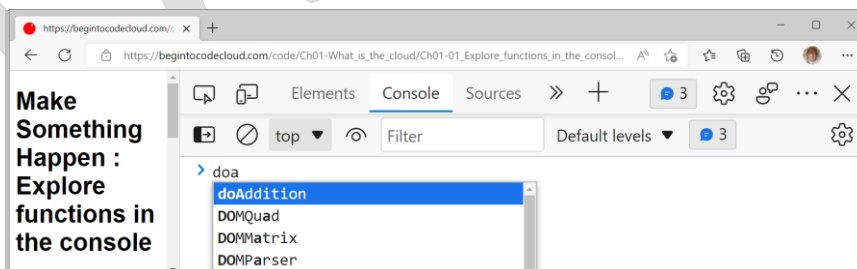
This is the web page for this exercise. Press F12 (or CTRL+SHIFT+J) to open the Developer Tools. You might be asked to confirm this action the first time you do it.



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.

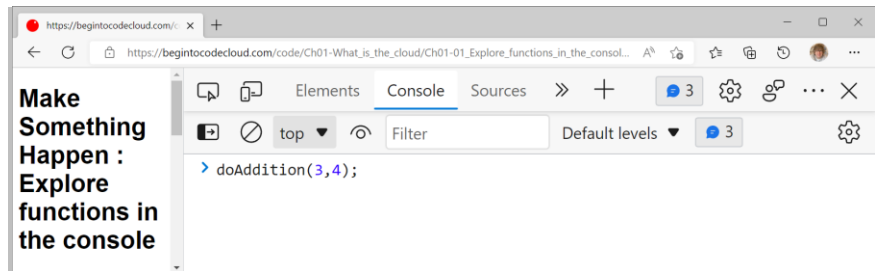
We can type JavaScript statements into the console, and they will be performed in the browser and the results displayed. The console provides a '>' prompt which is where you type commands. Click the console and start typing "doAddition" and watch what happens.



Ch01_inset01_04 Typing doAddition

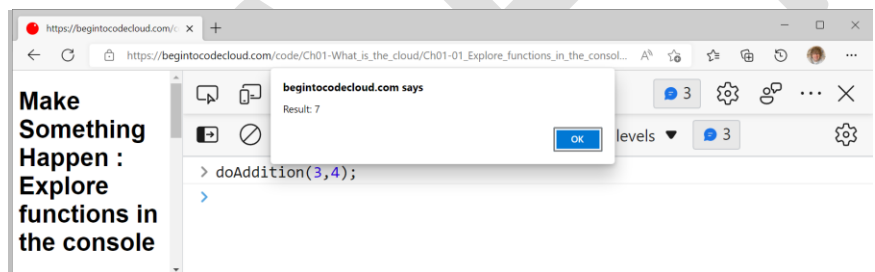
As you type the text the console presents you with a menu of things you could type in. This is

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:



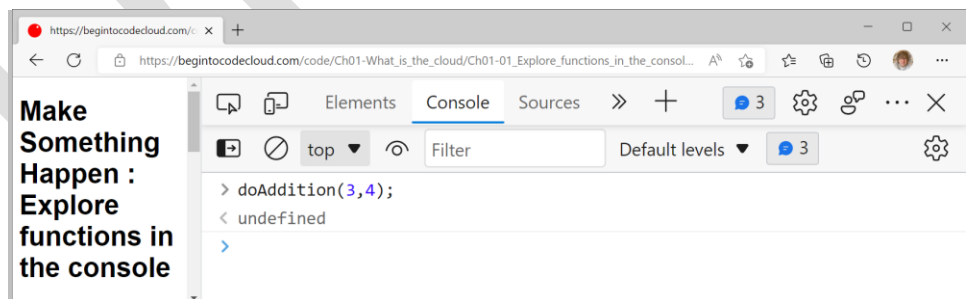
Ch01_inset01_05 Calling doAddition

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



Ch01_inset01_06 doAddition 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 doAddition 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 **doAdditon**?

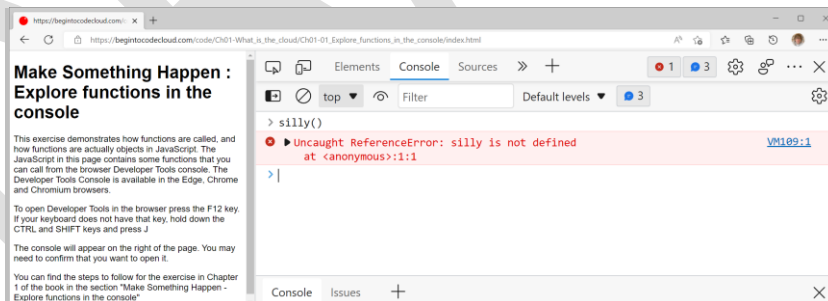
The console takes a JavaScript statement, executes it, and displays the value generated by the statement. If the statement calculates a result the statement will have the value of that result. This means you can use the console as a calculator. If you type in `2+2` the console will display 4.

```
> 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
```

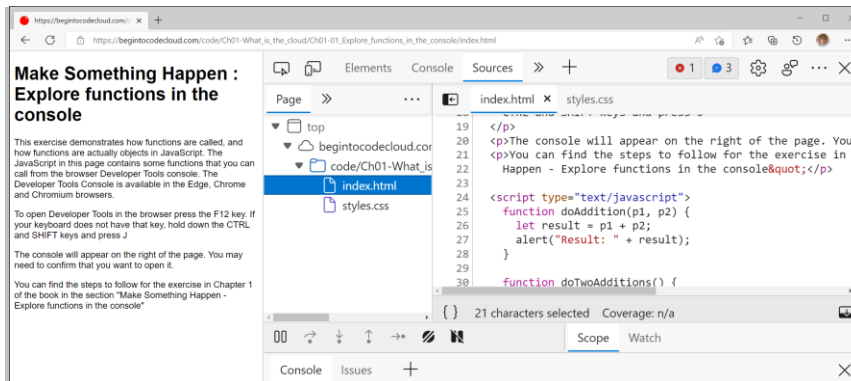
The value “Nan” means “not a number”. There is only one version of the `-` operator, the numeric one. However, this can’t produce a number as a result because it is meaningless to subtract one string from another. So the result of the operation is to set the value of result to a “special” value **NaN** 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 `begincodecloud.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.

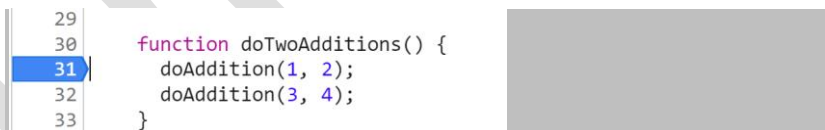


Ch01_inset02_02 viewing the web page source

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?

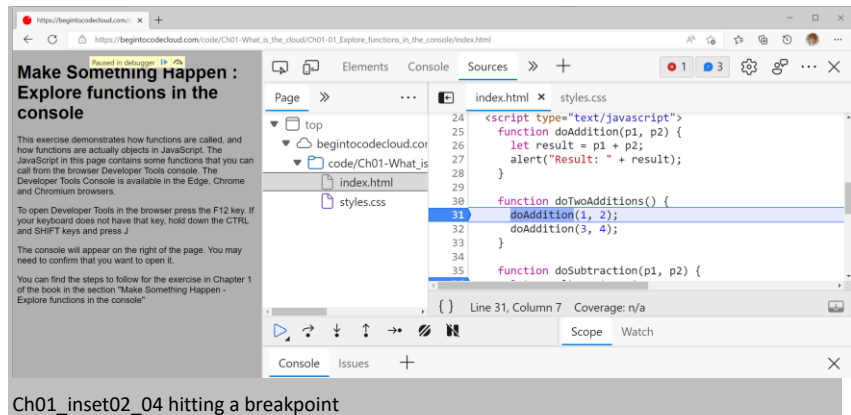
Yes. We can set a "breakpoint" at a statement and when that statement is reached the program will pause and we can go through it one step at a time. This is a wonderful way to see what a program is doing. Put a breakpoint at the first statement of `doTwoAdditions` by clicking in the left margin to the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



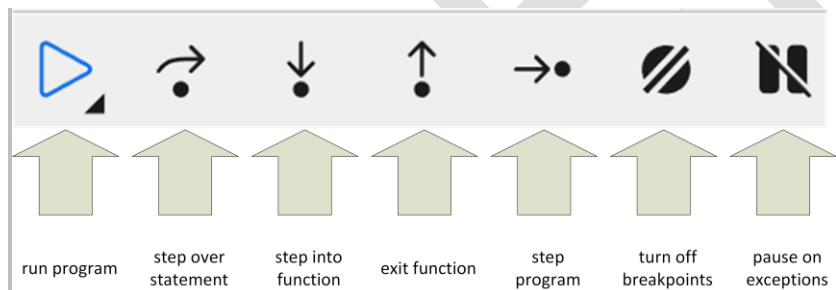
Ch01_inset02_04 setting a breakpoint

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();
```



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:



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&quot;</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.

JavaScript functions as objects

A JavaScript **object** is a type of variable which contains a set of properties. Objects are managed by **reference**. Our programs can contain reference variables that can be made to refer to objects. JavaScript functions are stored in objects.

```
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 an 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 a function 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 is 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 begin-to-codecloud.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'
```

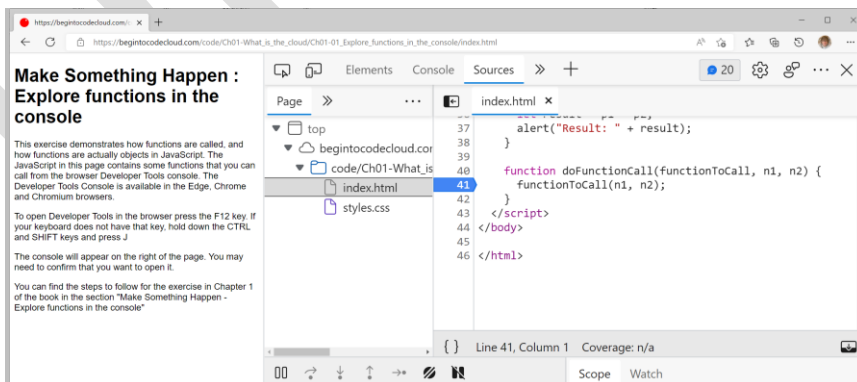
The console displays the value returned by the statement. In this case the statement is accessing the `name` property of the variable `x` which is the string `doAddition`.

```
> 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.

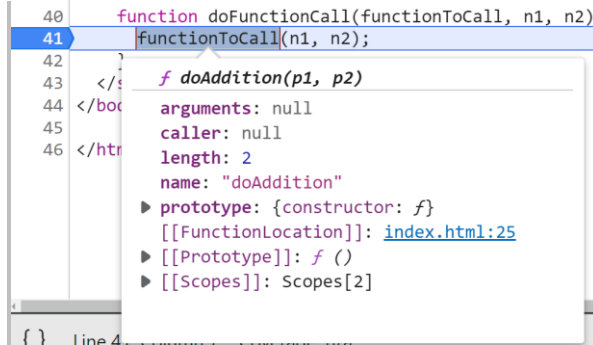


Ch01_inset03_01 breakpoint in doFunctionCall

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.



```
40 function doFunctionCall(functionToCall, n1, n2)
41 functionToCall(n1, n2);
42 }
43 </script>
44 </body>
45 </html>
```

doAddition(p1, p2)

- arguments: null
- caller: null
- length: 2
- name: "doAddition"
- ▶ prototype: {constructor: f}
- [[FunctionLocation]]: [index.html:25](#)
- ▶ [[Prototype]]: f ()
- ▶ [[Scopes]]: Scopes[2]

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:


```
40 function doFunctionCall(functionToCall, n1, n2)
41   functionToCall(n1, n2);
42 }
43 </script>
44 </body>
45 </html>
46 </html>
```

f (p1,p2)

arguments: null
caller: null
length: 2
name: ""

- ▶ prototype: {constructor: f}
- ▶ [[FunctionLocation]]: VM980:1
- ▶ [[Prototype]]: f ()
- ▶ [[Scopes]]: Scopes[2]

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.

```
index.html VM980 x
1 doFunctionCall(function (p1,p2){ let result=p1+p2; alert("Result: "+result);},1,7);
```

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

```
if(reply[0] != "") {  
    alert(reply[0]);  
}
```

The above code tests the element at the start of the reply array (JavaScript arrays are indexed starting at 0). If this element is not an empty string the code displays an alert containing the

content of this element. 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 string contains an error message. In JavaScript arrays are indexed starting with an index of 0.

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 returning an array is a very good idea. Let's look at a better one.

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

```
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.

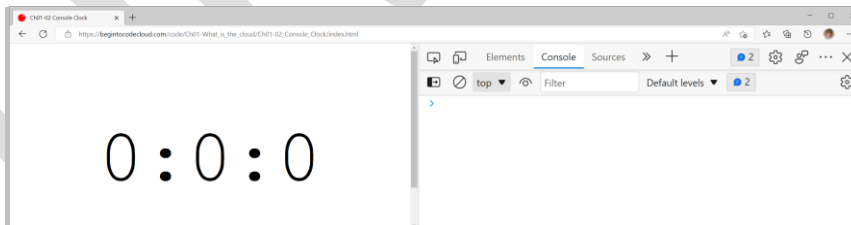
```
function getTimeString() {  
  let currentDate = new Date();  
  let hours = currentDate.getHours();  
  let mins = currentDate.getMinutes();  
  let secs = currentDate.getSeconds();  
  let timeString = hours + ":" + mins + ":" + secs;  
  return timeString;  
}
```

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 and scroll down to the Samples section. Click **Ch01-What_is_the_cloud/Ch01-02_Console_Clock** to open the sample page. Then press function key F12 (or CTRL+SHIFT J) to 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
```

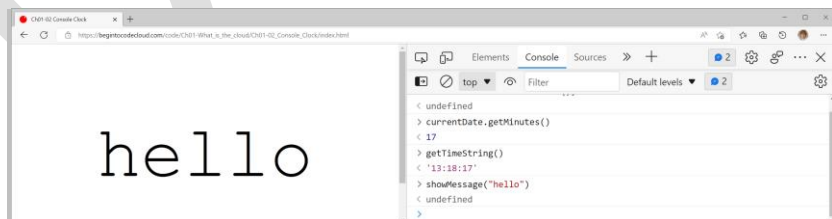
The code above was run at seventeen minutes past the hour, so the value returned by `getMinutes` will be 17. Note that `currentDate` holds a "snapshot" of the time. To get an updated date you will have to make a new `Date` object. You can also call methods to set values in the date too. The date contents 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");
```



Ch01_inset04_02 test message

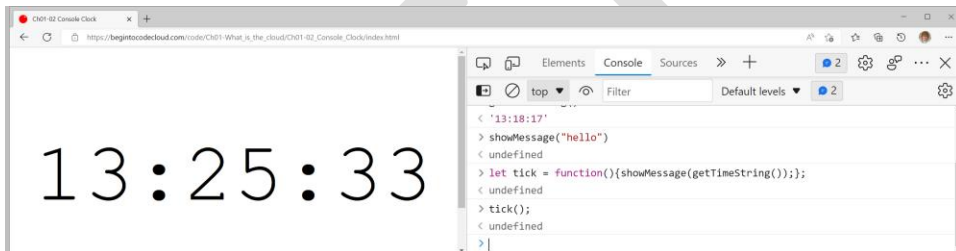
The web page now displays the string that was entered. Now we need a function that will display a clock tick. We can define this in the console window. Enter the following statement

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 thousandths 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.

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. Web hosting is now performed by large numbers of systems. Page requests from a browser are routed to the most appropriate server at any given time. Resources accessed in this way are said to be “in the cloud”.
- 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.
- 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.
- 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 state after the function call. A function can return a value using the return keyword.
- 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 JavaScript object. JavaScript manages objects 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.

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 did we do before we had the cloud?

There are two answers to this question. The first one is that in many cases we didn't do anything. Modern applications such as Facebook and Amazon would be impossible to create without a cloud to underpin them. The second answer is that before we had the cloud we used people. Lots of people. If you wanted to know how much money you had in the bank you would have to go to the bank and ask someone to find out for you. Banks employed thousands of people just to keep track of account balances.

Why is JavaScript such a popular language for cloud development?

JavaScript is popular because it is universal. All browsers can run JavaScript.

Is a browser a piece of application software or a piece of system software?

Most operating systems come with a browser built in. A Windows PC will have the Edge browser, a Mac will have Safari and a Chromebook will have Chrome. However, software makers also provide versions of their browsers for use on different platforms. There are also other browsers, for example Firefox, which are not affiliated to any particular machine. All

of these have their advantages and disadvantages. However, only the Edge and Chrome based browsers have the Developer Tools that we are using in this text.

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(function(){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.

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.

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 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.