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.

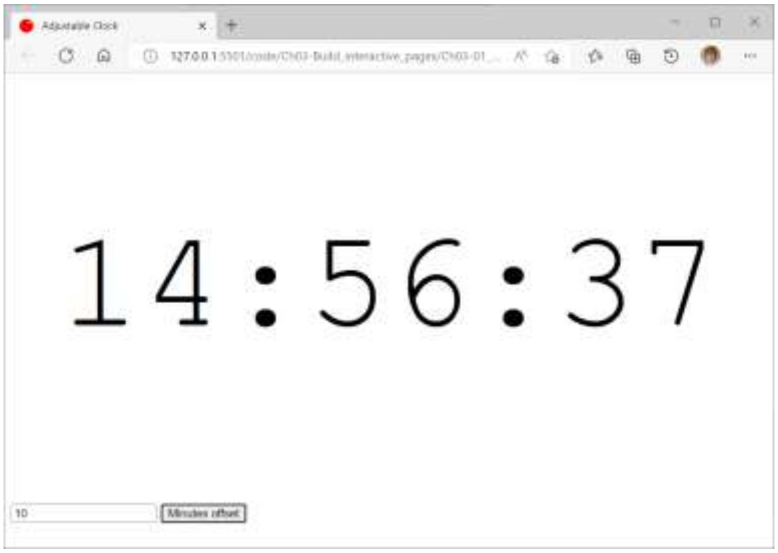


Figure 3.1 Ch-03_Fig_01 Adjustable Clock

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");[1]
  let minutesOffsetValue = minutesOffsetElement.value;[2]
  minutesOffset = Number(minutesOffsetValue);[3]
}
```

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.

Make Something Happen

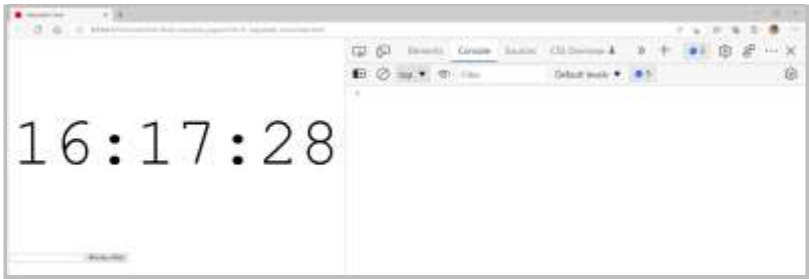
Adjustable time travel clock

[1] *Get the input element*

[2] *Get the value out of the input element*

[3] *Convert the string into a number*

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.

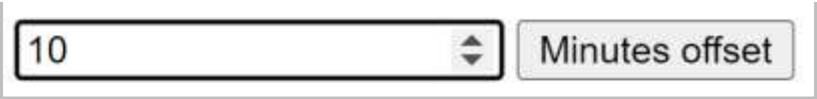


Ch03_inset01_01 adjustable clock

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:

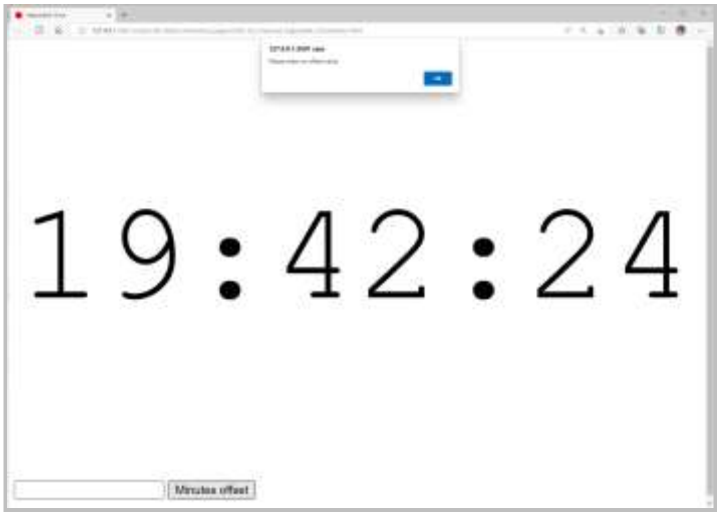
```
> Number("")
0
```

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;[4]
  if (inputString.length==0){[5]
    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.

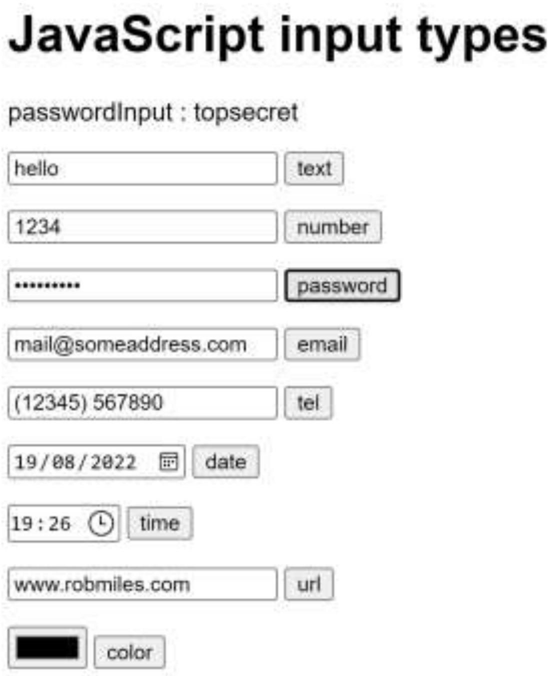


Ch03_inset01_04 Empty offset alert

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**

[4] *Get the value from the input element*

[5] *Check for an empty string*


JavaScript input types

passwordInput : topsecret

Figure 3.2 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);[6]
  let outputElement = document.getElementById("outputPar");[7]
  let message = itemName + " : " + inputElement.value[8]
  outputElement.textContent = message;[9]
}
```

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

[6] *Get the source element*

[7] *Get the destination element*

[8] *Build the message*

[9] *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");[10]
  if (offsetString==null){[11]
```

[10] Get the stored value

[11] Check for a missing value

```
    offsetString = "0";[12]
  }
  return Number(offsetString);[13]
}
```

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

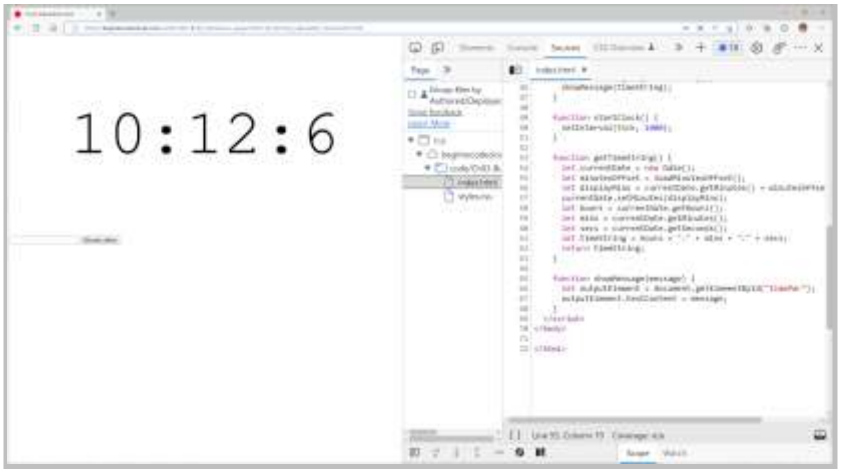
The storing adjustable clock works fine. But there is no way that the user can see the value of the minutes offset when they use the clock. It is not displayed on the page. But does this mean that it is impossible for anyone to discover this value? Let's see if we can use the debugger to get that value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

https://begintocodecloud.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.

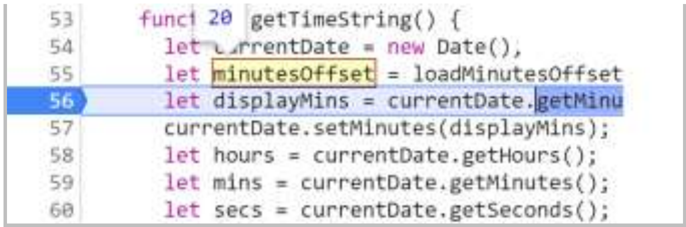
[12] *Set the offset to 0 if nothing is stored*

[13] *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.

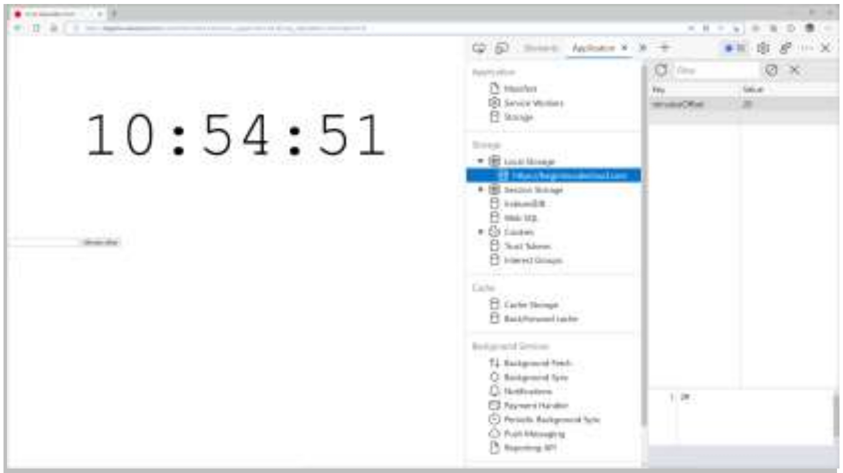


```
53      funci 20 getTimeString() {
54        let currentDate = new Date();
55        let minutesOffset = loadMinutesOffset
56        let displayMins = currentDate.getMinu
57        currentDate.setMinutes(displayMins);
58        let hours = currentDate.getHours();
59        let mins = currentDate.getMinutes();
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.



Ch03_inset02_03 Viewing local storage

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

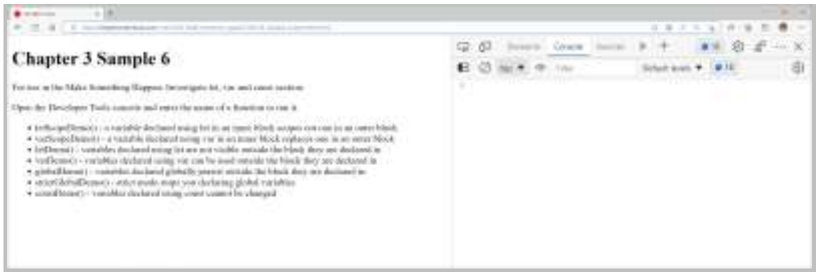
Some features of a programming language are provided so you can use the language to make a working program. For example, a program needs to be able to calculate answers and make decisions, so JavaScript provides assignments and if constructions. However, let, var and const are not provided to make programs 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.


Chapter 3 Sample 6

For use in the Make Something Happen Investigate let, var and const section

Open the Developer Tools console and enter the name of a function to run it

- letScopeDemo() - a variable declared using let in an inner block scopes out one in an outer block
- varScopeDemo() - a variable declared using var in an inner block replaces one in an outer block
- letDemo() - variables declared using let are not visible outside the block they are declared in
- varDemo() - variables declared using var can be used outside the block they are declared in
- globalDemo() - variables declared globally persist outside the block they are declared in
- strictGlobalDemo() - strict mode stops you declaring global variables
- constDemo() - variables declared using const cannot be changed

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:

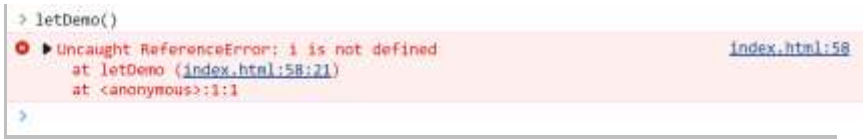
```
> varScopeDemo()
var inner i: 100
var outer i: 100
```

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:



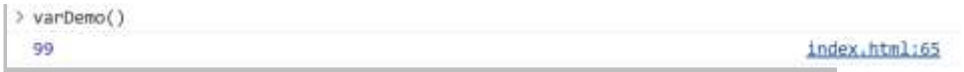
```
> letDemo()
● ▶Uncaught ReferenceError: i is not defined                    index.html:58
     at letDemo (index.html:58:21)
     at <anonymous>:1:1
>
```

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);
}
```

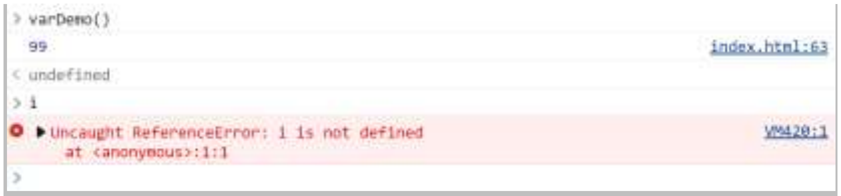
The function varDemo is very similar to letDemo, but this time i is declared using var. Let's see what happens when we run this function.



```
> varDemo()
  99                                                              index.html:65
```

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 varDeno has finished.



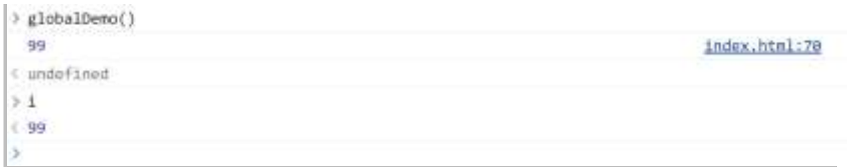
```
> varDemo()
  99                                                              index.html:63
< undefined
> i
● ▶Uncaught ReferenceError: i is not defined                          VM420:1
     at <anonymous>:1:1
>
```

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                                                    index.html:78
< undefined
> i
< 99
>
```

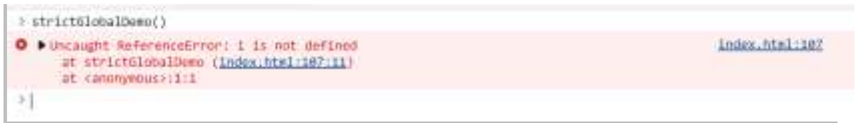
Ch03_inset03_05 global i

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            index.html:107
     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

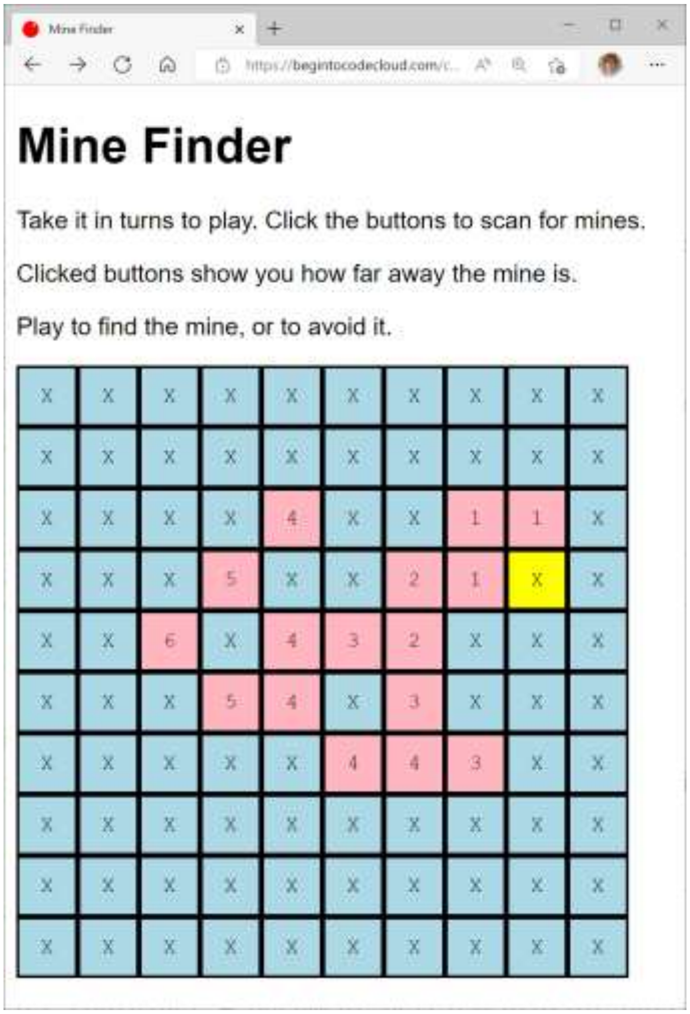
The let, var, const and strict features of JavaScript are not there to allow you to do things, they are there to help you make programs safer. When I create a new variable, I consider how visible it needs to be. If I need to make the value widely available I'll try to find ways to do it without creating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



Mine Finder

Take it in turns to play. Click the buttons to scan for mines.

Clicked buttons show you how far away the mine is.

Play to find the mine, or to avoid it.

X	X	X	X	X	X	X	X	X	X
X	X	X	X	X	X	X	X	X	X
X	X	X	X	4	X	X	1	1	X
X	X	X	5	X	X	2	1	X	X
X	X	6	X	4	3	2	X	X	X
X	X	X	5	4	X	3	X	X	X
X	X	X	X	X	4	4	3	X	X
X	X	X	X	X	X	X	X	X	X
X	X	X	X	X	X	X	X	X	X
X	X	X	X	X	X	X	X	X	X

Figure 3.3 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://begintocodecloud.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");[14]

  for (let y = 0; y < height; y++) {[15]
    for (let x = 0; x < width; x++) {[16]
      let newButton = document.createElement("button");[17]
      newButton.className = "upButton";[18]
      newButton.setAttribute("x", x);[19]
      newButton.setAttribute("y", y);[20]
      newButton.textContent = "X";[21]
      newButton.setAttribute("onClick", "doButtonClicked(this);");[22]
      container.appendChild(newButton);[23]
```

[14] *Find the destination paragraph*

[15] *Work through each row*

[16] *Work through each column in a row*

[17] *Make a button*

[18] *Set the style to "upButton"*

[19] *Store the x position in the button*

[20] *Store the y position in the button*

[21] *Draw an X in the button*

[22] *Add an event handler*

[23] *Append the button to the destination*

```
    }
    let lineBreak = document.createElement("br");[24]
    container.appendChild(lineBreak);[25]
  }

  mineX = getRandom(0, width);[26]
  mineY = getRandom(0, height);[27]

}
```

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.

[24] Create a line break
[25] Add the line break to the paragraph
[26] Set the X position for the mine
[27] 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 createEl-ement 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 with 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.

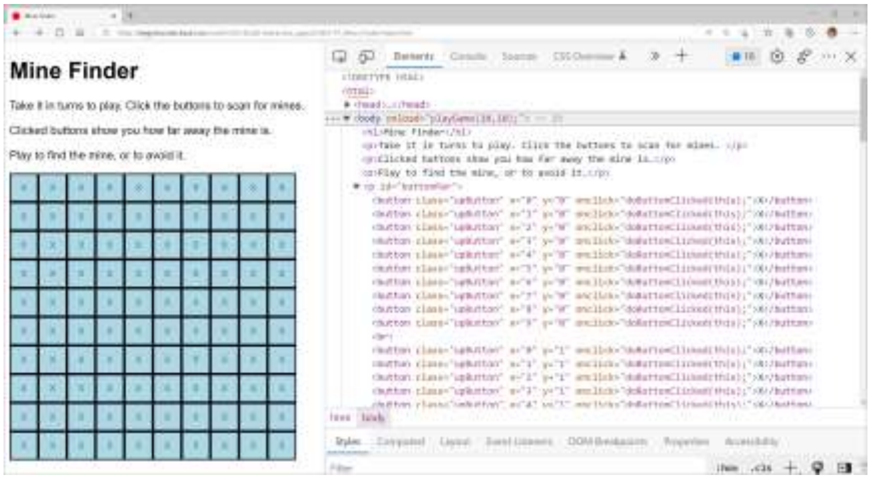
This will make more sense when we look at what the doButtonClicked function does. And we have a whole JavaScript Hero description for the this keyword coming up. For now, if you are happy with the value of this delivering a reference to the button that was pressed, we can move on to the next part of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



Mine Finder

Take it in turns to play. Click the buttons to scan for mines.

Clicked buttons show you how far away the mine is.

Play to find the mine, or to avoid it.

```
<!DOCTYPE html>
<html>
 ▶ <head>…</head>
 ▼ <body onload="playGame(10,10);"> == $0
    <h1>Mine Finder</h1>
    <p>Take it in turns to play. Click the buttons to scan for mines. </p>
    <p>Clicked buttons show you how far away the mine is.</p>
    <p>Play to find the mine, or to avoid it.</p>
  ▼ <p id="buttonPar">
      <button class="upButton" x="0" y="0" onclick="doButtonClicked(this);">X</button>
      <button class="upButton" x="1" y="0" onclick="doButtonClicked(this);">X</button>
      <button class="upButton" x="2" y="0" onclick="doButtonClicked(this);">X</button>
      <button class="upButton" x="3" y="0" onclick="doButtonClicked(this);">X</button>
      <button class="upButton" x="4" y="0" onclick="doButtonClicked(this);">X</button>
      <button class="upButton" x="5" y="0" onclick="doButtonClicked(this);">X</button>
      <button class="upButton" x="6" y="0" onclick="doButtonClicked(this);">X</button>
      <button class="upButton" x="7" y="0" onclick="doButtonClicked(this);">X</button>
      <button class="upButton" x="8" y="0" onclick="doButtonClicked(this);">X</button>
      <button class="upButton" x="9" y="0" onclick="doButtonClicked(this);">X</button>
      <br>
      <button class="upButton" x="0" y="1" onclick="doButtonClicked(this);">X</button>
      <button class="upButton" x="1" y="1" onclick="doButtonClicked(this);">X</button>
      <button class="upButton" x="2" y="1" onclick="doButtonClicked(this);">X</button>
      <button class="upButton" x="3" y="1" onclick="doButtonClicked(this);">X</button>
      <button class="upButton" x="4" y="1" onclick="doButtonClicked(this);">X</button>
html body
```

Figure 3.4 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");[28]
  let y = button.getAttribute("y");[29]
  if (x == mineX && y == mineY) {[30]
    button.className = "explodeButton";[31]
    alert("Booom! Reload the page to play again");[32]
  }
  else {[33]
    let dx = x - mineX;[34]
    let dy = y - mineY;[35]
    let distance = Math.round(Math.sqrt((dx * dx) + (dy * dy)));[36]
    button.textContent = distance;[37]
    button.className = "downButton";[38]
  }
}
```

The doButtonClicked function has a single parameter, which is a reference to the button that has been clicked. This is obtained from the this reference which is added when the event is bound in the element definition.

[28] *Get the x position of the button*

[29] *Get the y position of the button*

[30] *Check to see if this is the mine button*

[31] *Set the button style to "explode"*

[32] *Tell the player they have found the mine*

[33] *Do this part if the mine was not found*

[34] *Get the x distance to the mine*

[35] *Get the y distance to the mine*

[36] *Work out the distance*

[37] *Put the distance value into the button*

[38] *Set the button to the "down" style*

```
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;[39]
  . . .
  }
```

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.

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

[39] *Get the button reference from the event description*

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) an 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.