Chapter 6

The Lifetime of a Variable

It's easy to create a variable and use it right away. But what if you want to create a variable, put something in it, and make sure it's still around the *next* time someone clicks a button?

Keeping variables alive is an important technique in programming. Imagine trying to build a game if every variable resets itself each time you click a button! But before you can take control of your variables and store information for the long term, you need to understand a bit more about how they live… and when they die.

Local variables

When you define a variable inside a function, it's called a *local variable*. It lives until the function ends. Here's an example with a string called `welcomeMessage`:

```
function sayHello() {
  let welcomeMessage = "Hi, fancy pants";

  alert(welcomeMessage);
}
```

Once the code runs and the `sayHello()` function ends, the `welcomeMessage` is thrown away, along with whatever information is inside the variable. There's no way to get that information back.

In a program, the same piece of code can run many times. For example, maybe the `sayHello()` function is attached to the `onclick` event of a button. In that case, every time you click the button this happens:

1. The `sayHello()` function starts.
2. It creates a new, blank copy of the `welcomeMessage` variable.
3. It puts the "Hi, fancy pants text" inside the `welcomeMessage` variable.
4. It uses `alert()` to show the message.
5. The `sayHello()` function ends, and the `welcomeMessage` function variable disappears again, taking its contents with it.

Clearly, you can't use `welcomeMessage` to store information for a longer time, because the variable only lives for a few seconds. This isn't a problem in this example, because `welcomeMessage` never changes. But there is *another way*.

Global variables

Sometimes you have a piece of information that you want to keep around for longer, maybe even as long as your program is running.

> **NOTE** In a web page, "as long as the program is running" means as long as the web page is open in the browser. As soon as you close the browser tab or go to a different page, your program is over. (In CodePen, there's one more rule—when you edit your example, the program restarts.)

To store data for a longer period of time, you need to use a *global variable*. Local variables stay in one *location* (the function where you create them). Global variables are available everywhere in your code.

To define a global variable you use the same `let` keyword, but you put it in a different place. Instead of creating your variable inside a function, you declare it *outside* of your function.

Here's an example:

```
let welcomeMessage = "Hi, fancy pants";

function sayHello() {
  alert(welcomeMessage);
}
```

You might be suspicious of this code if you remember the rule from Chapter 1 (don't put code outside a function). But code that creates variables is acceptable. It runs right away, and sets everything up for your functions to use.

> **TIP** You can put your variables before the functions that use them—or anywhere you want, really. But usually, the simplest way to organize your code is to put all your global variables at the beginning of the code file (at the top of the JavaScript box), above all your functions.

This code in this example (with the global `welcomeMessage` variable) still works the same as the earlier version (with the local `welcomeMessage` variable). But that quickly changes in slightly more complicated examples, like the ones you'll see next.

Sharing the same variable with different functions

Here's a quick recap of the two differences between global variables and a local variables:

- Local variables live until the function ends. Global variables live until the program ends—say, when you close the browser (or edit your code in CodePen).
- Local variables can only be used in one function. But you can use global variables in *every* function.

The second concept is called *scope*, and it's the one we'll explore next.

So far, you've been looking at a lot of examples with just one button and one function. But most JavaScript programs will have a bunch of functions that run when different things happen. For example, imagine you have a three-button program like this:

Become Nice | Become Mean | Say Hello

When you click **Become Nice**, it sets `welcomeMessage` to something nice:

```
function beNice() {
  welcomeMessage = "I'm so delighted you could join me for tea.";
}
```

When you click **Become Mean**, it sets `welcomeMessage` to something else:

```
function beMean() {
  welcomeMessage = "I plan to eat you and your pet goldfish.";
}
```

When you click **Say Hello**, you get whatever greeting you last chose:

```
function sayHello() {
  alert(welcomeMessage);
}
```

In the next exercise, you can try creating this example.

Try it yourself!

▶ The Friendly Angry Ogre (Challenge Level: Medium)

https://codepen.io/prosetech/pen/mdVzpNg

In this example you start out with three buttons (Become Nice, Become Mean, and Say Hello). There's also a tiny bit of code that changes the picture to match the ogre's mood.

The part that's missing is the message you're going to show. To make this example work, you need to set the right message, depending on the ogre's mood. Then you need to show this message at the right time.

Here's what you need to do:

1. Create the `welcomeMessage` variable. Remember, it needs to be a global variable, and you only need to define it once.
2. Add the code that sets `welcomeMessage` when **Become Nice** or **Become Mean** is clicked. You need two different messages, depending on what mood you've put the ogre in.
3. Add the code to `sayHello()` that shows the message.

What happens if you click **Say Hello** without clicking **Become Mean** or **Become Nice** first? Can you give `welcomeMessage` a starting value so this doesn't happen? (A good starting value for an ogre with no mood is a message like "I just don't know how to behave.")

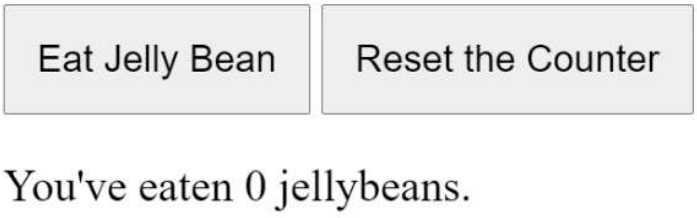
Try it yourself!

▶ The Broken Jellybean Counter (Challenge Level: Medium)

https://codepen.io/prosetech/pen/PoZyQPo

If there's one thing programmers like to do, it's counting things. (If there are two things programmers like to do, it's counting things and eating jellybeans.) That makes this next example the perfect bit of practice.

The Broken Jellybean Counter lets you eat as many jellybeans as you want—just click a button each time you want to eat one. It counts the number of jellybeans you've eaten so far.



Unfortunately, there are two disastrous mistakes in the program. Can you fix them?

> **HINT** Right now, the bean counter uses local variables. But you need to turn `beanCount` into a global variable, so you can use it everywhere. Make that change, and you're well on your way to solving the problem.

The secret of the braces

I have a confession to make. Earlier, I said that local variables live until the function ends. That's true, but it's not the whole story.

Technically, local variables live in the block that contains them. As you might remember from Chapter 1, a block is a common structure in JavaScript code. It's any section of code that fits between two braces `{ }`.

So far, all the braces you've seen have been for functions. However, in the upcoming chapters you're going to learn that you also use braces to group together code for different reasons. For example, you use braces to repeat code in a loop, or make it conditional. If you create a variable inside one of these sections, it doesn't live for the whole function. It only lives until that *block* is finished.

This will all make more sense when you see examples of the other ways we use braces. But here's a cheat rule that always works—when you create a local variable, it will disappear from existence when you reach the next `}` closing brace.

Wise advice about global variables

The more global variables you use, the more difficult it is to figure out what's going on in your code. Imagine you see a new JavaScript program for the first time, and it has a bunch of global variables. How do you know which functions use which variables? There's no way to be sure without reading all the code, and that means there's plenty of room to make a mistake.

To make your code as simple as possible, and to reduce the chance of making mistakes in the future, you should use global variables only when you need them to share information.

Sometimes, you might notice in your program that different functions are using the same local variables for the same tasks. For example, maybe you have three

functions and each one creates its own local `welcomeMessage` variable. It might occur to you that you could replace these three local variables with one global `welcomeMessage` variable. Don't do it! It's always better to make your functions as independent as possible.