COMP 1073

Client-Side Scripting

Lesson 4 – Part 1

Variables, Functions, and Loops

Georgian

Objectives

In this Lesson we will learn about:

1. **Defining Variables**
2. **Functions**
3. **Scope**
4. **Loops**
5. **Conditionals**

Variables, Functions and Loops

Introduction – Variables, Functions and Loops

- Using **variables**, **functions**, and **loops** are often the only thing a person knows how to do in **JavaScript**, and they usually get along just fine.
- You're already past that part and on your way to becoming a JavaScript developer.
- Up to this point in the course, the examples have been pretty specific, but also a little abstract.
- You've been manipulating content and data, then alerting or observing the result.
- In this lesson we expand on what you've learned already and begin building a simple **JavaScript application** that will get more robust as you step through the subsequent lessons.
- As you progress though this lesson, you notice that an **address book application** should be starting to form.

Defining Variables

Georgian

Defining Variables

- For the most part, you learned about variables within the context of **data storage**, but they also have an integral part in your application when it comes to **functionality**.
- When considering variable and function naming, it's best to make them **meaningful** and speak to their contents or purpose.
- For example, using a variable name of **`myBestFriend`** would be much more helpful than something like, **`firstVariableName`**.
- Something else to consider when naming variables is that they can't start with a number.
- They can contain numbers, such as **`dogs3`** or **`catsStink4Eva`**, but they can't begin with a number, such as **`3dogs`**.

Grouping Variables

- When you're writing an application, it's best to try to group all variables at the top of your **JavaScript** file or function (when possible) so they can all be **immediately cached** for later reference.
- Some people find this method a little unnatural because functions are defined throughout the document, and it's a little easier to maintain when variables are right there with the function they belong to; but grouping variables at the top is one of those small performance boosts you can give to your application.
- It helps to think of it as one large file containing JavaScript for an application versus thinking of the file as a collection of one-off actions that get executed.
- When thinking of it as a single unit, it's better grouping all variables together at the top.

Grouping Variables (cont’d)

- You can **group variables** in your document in two ways.
- Up to this point we have been using a new **var** declaration for each variable; a lot of people prefer this method, and it’s perfectly fine to use.
- An alternative method is to use a single var declaration, using **commas** to separate the individual variables and a **semicolon** at the very end.
- The following Listing shows an example of grouping variables with a single **var** declaration. **Note** the commas at the end of each line.

```
var highSchool = "Hill",
    college = "Paul",
    gradSchool = "Vishaal";
```

- There’s **no difference** in the way you access these variables compared to how you access variables declared with individual **var** declarations.
- At the variable level, it’s purely a way to **group**.
- It isn’t good or bad at this point—it’s only personal preference.
- You’ll see both methods in looking through JavaScript others have written, so it’s good to know what’s going on.

Reserved Terms

- JavaScript contains a lot of **core functionality**.
- We've been over quite a bit of it so far. Beyond that core functionality you will be defining a lot of your own **custom code**.
- If the names of your custom JavaScript match up with anything built into the language, it can cause **collisions** and **throw errors**.
- It's the same as if you're writing a large JavaScript file—you want to make sure all the **function** and variable names are as **unique** as possible to prevent problems and confusion while parsing the information.
- If you have two functions with the **same name**, it's difficult to tell the browser which one to use, so it's just not allowed.

Reserved Terms (Cont'd)

- To prevent these issues with native JavaScript, there are some **reserved words (keywords)** that you can't use when defining **variables**, **functions**, **methods**, or **identifiers** within your code.
- Following is a list of the reserved words:

`break`	`finally`	`new`	`this`
`case`	`for`	`package`	`throw`
`catch`	`function`	`private`	`try`
`continue`	`if`	`protected`	`typeof`
`debugger`	`implements`	`public`	`var`
`default`	`in`	`return`	`void`
`delete`	`instanceof`	`static`	`while`
`do`	`interface`	`switch`	`with`
`else`			

Reserved Terms (Cont’d)

- Most of these are no-brainers, like function and **`var`**, and under normal circumstances you probably would never come across a situation where something like “ implements ” would be a reasonable name for a **variable** or **function.**
- If you end up using any of these terms in your code, the console will throw an error and let you know that you’re using a reserved word.
- With that in mind, I think the value in this list is not so much memorizing it, but rather recognizing that these words map to native actions in the language.
- It will help you write better code and also aid in learning more advanced JavaScript down the road if you choose to research some of those terms that are beyond the scope of this book, such as **public**, **private**, and **protected**.

Functions

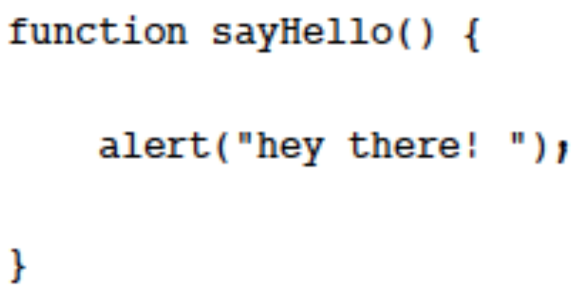
Georgian

Functions

- **Functions** in any programming language are ways to write code that can be used later.
- At its most basic form, this is also true for JavaScript.
- You can write a chunk of **custom code** and not only execute it at will, but you can also execute it over and over, which can help streamline your application by increasing its maintainability (declaring a chunk of code one time and referencing it, rather than rewriting what it does).
- It's like keeping all your CSS in the same file or why you keep all JavaScript in the same file—you know exactly where it is when you need to change or add something.
- You've been using functions already in earlier chapters when you pass data into an `alert()`.
- **Alert** is technically called a **method** but for all intents and purposes, it's the same as a **function**.

Basic Functions

- The chance of creating a JavaScript application without having to write your own functions is pretty low.
- It’s something that you’ll be doing on every project, and it’s very easy to do using the **`function`** keyword (remember the reserved words list? This is what function is for).
- Using the function **`keyword`** is like saying, “Hey, I’m building something over here that should be treated as a function.”
- The following Listing shows a basic function declaration.



```
function sayHello() {

    alert("hey there! ");

}
```

Calling a Function

- Calling a **function** is very simple. You type out the **name**, and then **parentheses** and the function will be executed.
- The parentheses tell the browser that you want to execute the function and to use any data (arguments) contained within the parentheses within the function.
- The following Listing shows how to call the function we declared in the previous Listing. It should alert the text, “hey there!”

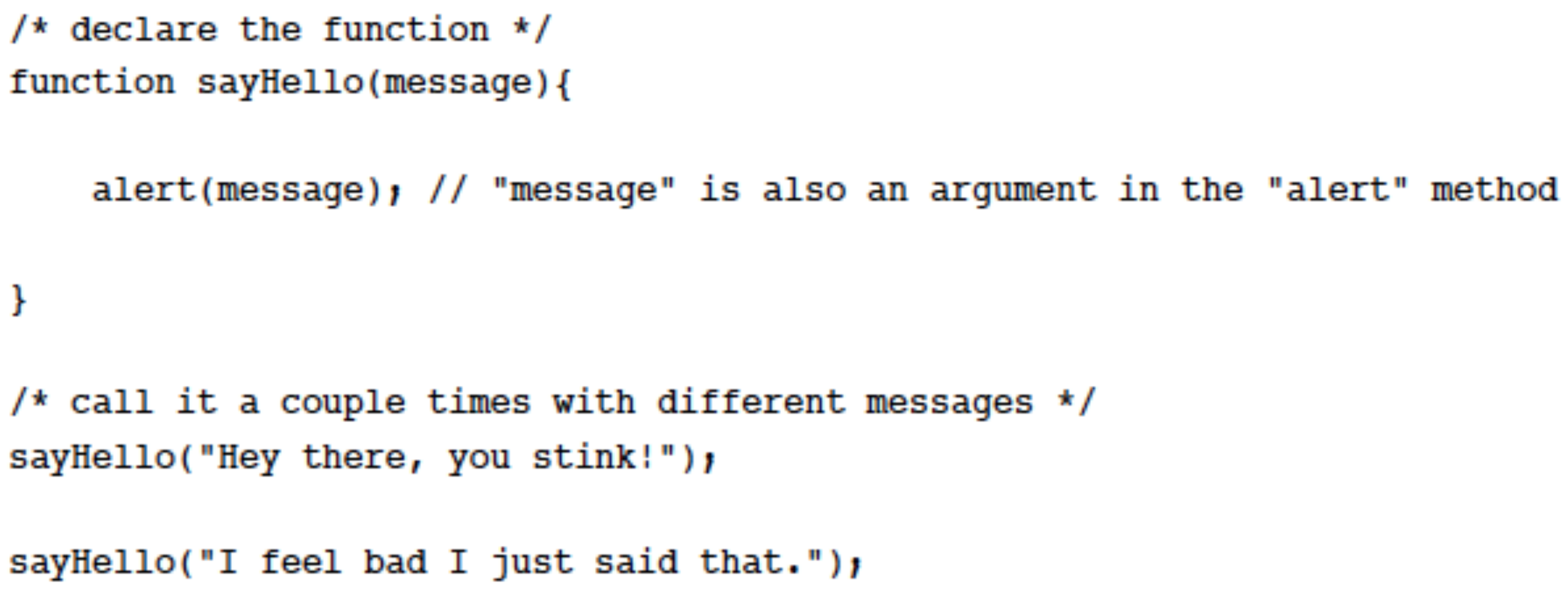
```
sayHello(); // hey there
```

Arguments

- **Arguments** are a way to pass information or data **into a function**.
- As previously mentioned, up to this point you've been using the `alert()` method.
- We've also been passing it arguments.
- The **alert** method is designed in native JavaScript to take arguments and display them in the form of a pop-up box in the browser.
- Functions can take **any number of arguments**. They can be any type of information; **strings**, **variables**, large data sets, and anything else you can think of can be passed into a function through an argument.
- As you're defining your functions, you will be **assigning names** to the arguments, sort of like the way you assign names to a variable.
- After that argument is named in the **function**, it becomes a variable you'll be using inside that function.

Arguments (cont’d)

- In the following Listing you can see that the **`sayHello()`** function now has a single argument called “message.”
- Inside, the function “message” is used as a variable that gets passed into the JavaScript **`alert()`** method.



```
/* declare the function */
function sayHello(message){

    alert(message); // "message" is also an argument in the "alert" method

}

/* call it a couple times with different messages */
sayHello("Hey there, you stink!");

sayHello("I feel bad I just said that.");
```

Arguments (cont'd)

- When this **function** is called, we're setting the string argument to "Hey there, you stink!" and then quickly apologizing with another alert, because frankly it was kind of rude.
- This is a very real-life way arguments are used in functions.
- The **`string`** can either be declared upon calling the function (like we're doing in the previous Listing) or it can be declared immediately in the function declaration.

Anonymous Functions

- **Anonymous functions** are functions that have **no name.**
- They execute immediately and can contain any number of **other functions**.
- The syntax for declaring an anonymous function is a little different.
- They are **dynamic** in nature because they are executed at **runtime** rather than waiting to be called.
- Anonymous functions **perform very well** in the browser because there is no reference to them in another part of the document.
- This comes with pluses and minuses. So as you write your JavaScript, it is always good to note that if you have to rewrite an anonymous function over and over, it's probably best to pull it out into a **normal function** to cut down on maintenance and repetitive code.

Anonymous Functions (Cont'd)

- There is often a little confusion as to the purpose of anonymous functions. If you want something to **execute at runtime**, why wouldn't you just dump the code right into your JavaScript file?
- Why even bother wrapping it in an anonymous function? Well, this is a good place to bring up a term you may hear a lot: **scope.**

Scope

Georgian

Scope

- **Scope** is a programming concept that exists to reduce the amount of variable and function collisions in your code.
- It controls how far information can travel throughout your JavaScript document.
- Earlier on, we briefly mentioned **global variables.**
- "Global" is a type of scope; the global scope for a variable means that the variable can be accessed and used **anywhere** in the document.
- Global variables are generally **a bad thing**, especially in larger files where naming collisions are more likely.
- Try to keep things out of the **global scope** if possible.

Scope (Cont’d)

❖ The following Listing shows how to declare a basic **anonymous function** and keep variables out of the **global scope**.

```
/* set up your anonymous function */
(function () {

    /* define a variable inside the function */
     var greeting = "Hello Tim";

     /* access the variable inside the function */
     alert("in scope: " + greeting);

})(); // end anonymous function
```

Scope (Cont'd)

- For the most part, you will be dealing in **function-level scope**.
- This means that any variable defined inside a **function** cannot be used outside that function.
- This is a great benefit of using **anonymous functions**.
- If you wrap a code block in an **anonymous function**, the contents of that function, which would normally default to the **global scope**, will now be contained within the scope of that **anonymous function**.

Scope (Cont'd)

- The following Listing defines a **variable** inside an anonymous function, alerts the variable, and then tries to alert the variable again, outside the function.

```
/* set up your anonymous function */
(function () {

    /* define a variable inside the function */
     var greeting = "Hello Tim";

     /* access the variable inside the function */
     alert("in scope: " + greeting);

})(); // end anonymous function

/* try and access that variable outside the function scope */
alert("out of scope: " + typeof(greeting)); // alerts "undefined"
```

Calling a Function with a Function

- When you have a **function** that calls another **function**, the second function is referred to as a **callback**.
- The **callback function** is defined as a normal function with all the others but is executed inside another function.
- Callback functions are a great way to separate out the levels of functionality in your code and make parts more **reusable**.
- Often you will see callback functions passed as **arguments** to other functions.

Calling a Function with a Function

❖ The following Listing shows our **`sayHello()`** function being defined and then called inside the anonymous function. In this case, **`sayHello()`** is a callback function (calling it twice).

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

(function (){

    var greeting = "Welcome",
        exitStatement = "ok, please leave.";

    sayHello(greeting);
    sayHello(exitStatement);

})();
```

Returning Data

- Every **function** you create will not result in a direct output.
- Up to this point you've been creating functions that do something tangible, usually alerting a piece of data into the browser.
- You won't always want to do that, though; from time to time you will want to create a function that **returns information** for another function to use.
- This will make your functions a little smaller, and if the function that gathers information is general enough, you can reuse it to pass the same (or different) information into multiple functions.
- Being able to **return data** and pass it into another function is a powerful feature of JavaScript.

Returning a Single Value

- Going back to the **`sayHello()`** function that was defined in a previous Listing, we're going to remove the **`alert()`** action that was previously being executed when the **function** was called, and we'll replace it with a **return** statement.

```
function sayHello(message){
    return message + "!"; // add some emotion too
}
```

- You'll probably notice that the **`sayHello()`** function doesn't do anything in the browser anymore.
- That's a good thing (unless you're getting an error—that's a bad thing).
- It means the **function** is now returning the data but it's just sitting there waiting to be used by another function.

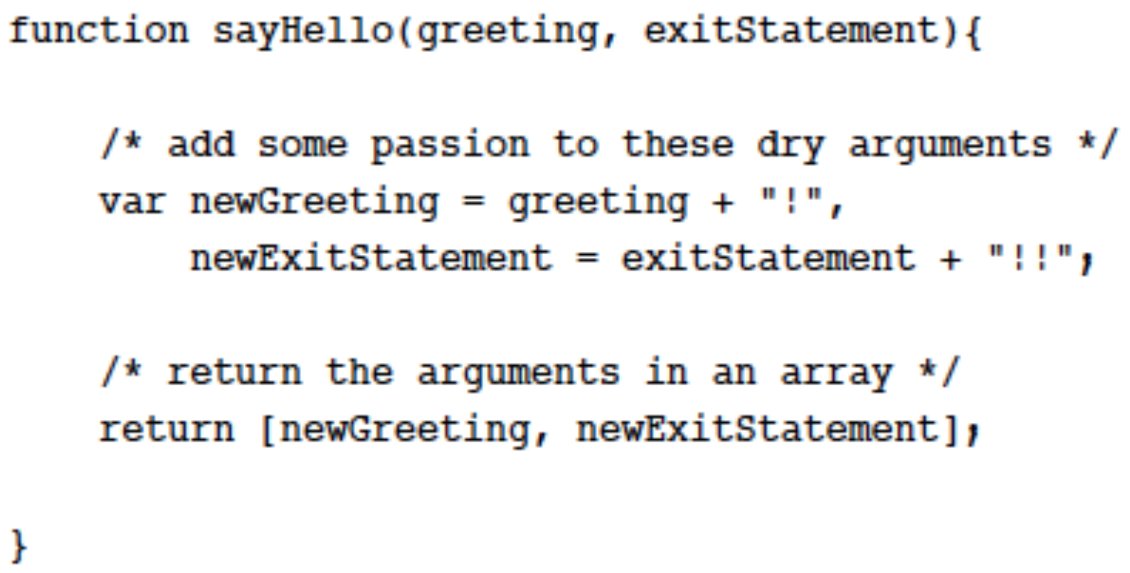
Returning Multiple Values

- Sometimes returning a **single value** isn't enough for what you're trying to accomplish.
- In that case you can return **multiple values** and pass them in an **array format** to other functions.
- In the following Listing you can see the **`sayHello()`** function taking two arguments.
- Those arguments get changed slightly and are resaved to variables; then they are returned in an array format to be accessed later.

```
function sayHello(greeting, exitStatement){

    /* add some passion to these dry arguments */
    var newGreeting = greeting + "!",
        newExitStatement = exitStatement + "!!";

    /* return the arguments in an array */
    return [newGreeting, newExitStatement];

}
```

Passing Returned Values to Another Function

- Now that you're **returning variables**, the next step is to pass those variables into another function so they can actually be used.
- The following Listing shows the **`sayHello()`** function returning an **array** of information and a new function called **`startle()`**, taking two arguments, passing them through the original **`sayHello()`** function, and alerting the results.



```
function sayHello(greeting, exitStatement){

    /* add some passion to these dry arguments */
    var newGreeting = greeting + "!",
        newExitStatement = exitStatement + "!!";

    /* return the arguments in an array */
    return [newGreeting, newExitStatement];

}
```

Passing Returned Values to Another Function (Cont'd)

```
function startle(polite, rude){

     /* call the sayHello function, with arguments and same each response to a
➥variable */
    var greeting = sayHello(polite, rude)[0],
        exit = sayHello(polite, rude)[1];

    /* alert the variables that have been passed through each function */
    alert(greeting + " -- " + exit);

}

/* call the function with our arguments defined */
startle("thank you", "you stink");
```

A Function as a Method

- Just as you can **group variables** and data into objects, you can also do it with functions.
- When you **group functions** into objects, they're not called functions anymore; they're called **methods**.
- Also, when you're dealing with JavaScript libraries everything is a **method** and nothing is a function, even though they all look and act the same.
- In the previous lesson, you learned about **storing information** in objects.
- I mentioned that you could also **store functions** in objects.
- When you do that, they're called **methods** instead of functions, but they work the same way.
- It's weird, I know, and it's not even an important distinction while you're coding.
- It's more about **organizing your functions in groups** to make them easier to maintain.
- The **`alert()`** method lives inside a global object, which is why it's called a **method**.

A Function as a Method (Cont’d)

- The following Listing should look a little familiar; it shows how to organize our two functions (**`sayHello`** and **`startle`**) inside an object called **`addressBookMethods`**.

```
var addressBookMethods = {

    sayHello: function(message){

        return message;

    },
    startle: function(){

        alert(addressBookMethods.sayHello("hey there, called from a method"));

    }

}

/* call the function */
addressBookMethods.startle();
```

Performance Considerations

- **Nesting functions** in objects has the same performance implications that we spoke of when nesting variables in objects.
- The deeper a function is nested inside an object (addressBookMethods), the more resources it takes to extract.
- This is another place in your code where you will have to balance **performance** with **maintainability**.
- We're not talking a ton of time here—maybe a few milliseconds difference—but it can add up.
- Most of the time it won't matter, but if you find yourself needing a performance boost, **function objects** would be a place to look for a **bottleneck**.

Loops

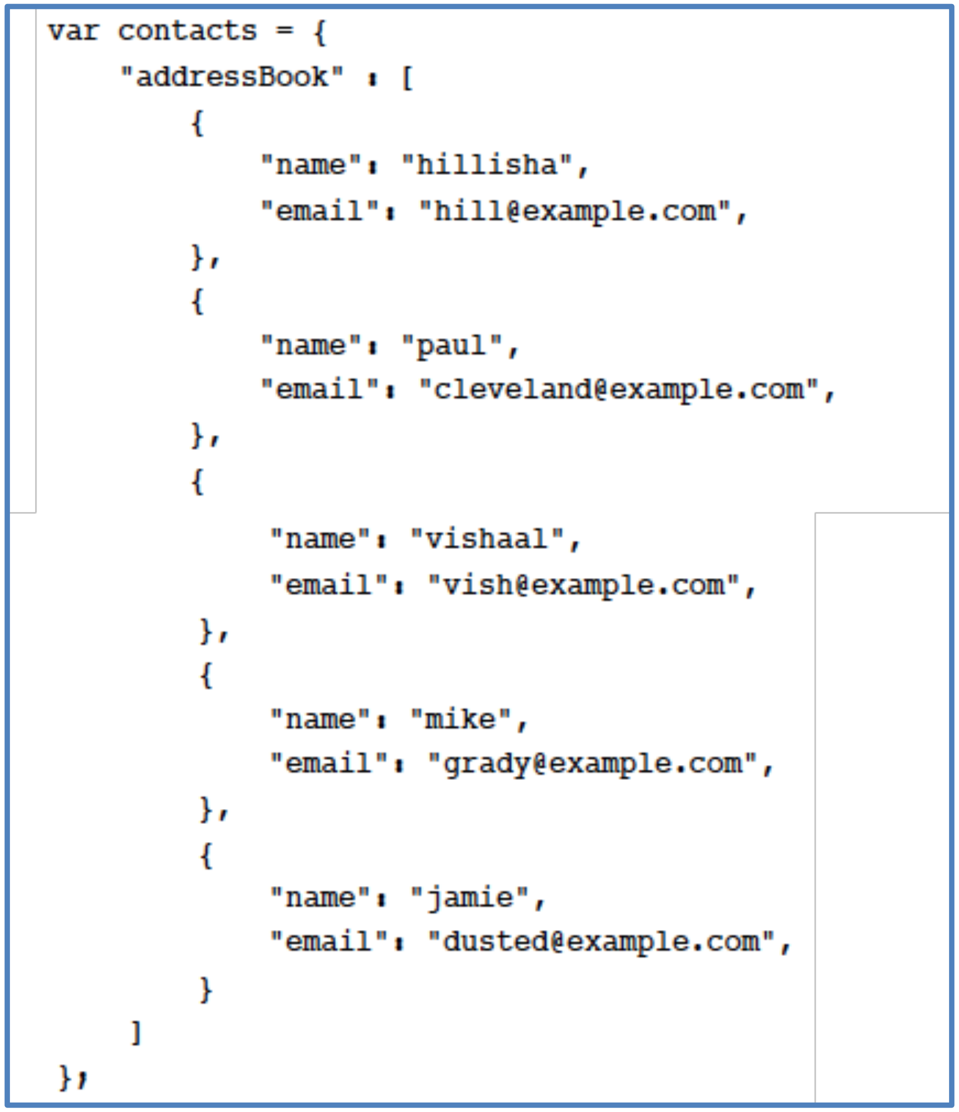
Georgian

Loops

- A **loop** will execute a block of code over and over until you tell it to stop.
- It can **iterate** through data or HTML. For our purposes we'll mostly be **looping through data**.
- Much the way a **function** is a chunk of JavaScript code, a loop can make that function execute over and over.

Loops (Cont'd)

- The following Listing shows a small sample of the contact information we'll be looping through.



```
var contacts = {
    "addressBook" : [
        {
            "name": "hillisha",
            "email": "hill@example.com",
        },
        {
            "name": "paul",
            "email": "cleveland@example.com",
        },
        {
            "name": "vishaal",
            "email": "vish@example.com",
        },
        {
            "name": "mike",
            "email": "grady@example.com",
        },
        {
            "name": "jamie",
            "email": "dusted@example.com",
        }
    ]
};
```

`for` Loop

- There are few different types of loops in JavaScript: a **`while` loop**, a **`do-while` loop**, and a **`for` loop**.
- Most of them are perfectly fine; You should avoid the **foreach** loop because it’s known to be a performance hog, but the others are fine to use.
- A **`while` loop** and a for loop are basically the same thing, but the **`for` loop** is a little more direct, to the point, and it’s the most common kind of loop you’re going to find in the wild.

for Loop (Cont’d)

❖ The following Listing will show you a basic **for loop**, and then we’ll go over what’s happening.

```
/* cache some initial variables */
var object = contacts.addressBook,
    contactsCount = object.length,
    i;

/* loop through each JSON object item until you hit #5, then stop */
for (i = 0; i < contactsCount; i = i + 1) {

     // code you want to execute over and over again

} // end for loop
```

`for` Loop (Cont’d)

- Right away, you can see that we’re saving some information to variables.
- The **first** variable **object** is saving the JSON object we create to a variable so it’s a little easier to work with.
- The **second** variable, **`contactsCount`**, looks through the JSON object and counts the number of items in there.
- This will let us know how many times to loop through the data.
- The **third** variable, **`i`**, is just a way to declare the counting variable for our **loop**. Later on we’ll be setting the value.
- Inside the **`for`** you can see **three** statements.
  - The **first** statement is setting the counter variable **`i`** to its initial value of **`0`**.
  - The **second** statement is the **condition** in which you run the loop. As long as the “**`i`**” value is less than the overall count of items in the data, it should execute the code contained inside the loop brackets **`{ }`**.
  - The **last** statement takes the **`i`** value and adds **`1`** to it each time the loop executes until it’s no longer less than the overall count. In our case, this loop will execute 5 times because there are five people in the address book.

`for` Loop (Cont'd)

- The following Listing will show the actual loop to **iterate** through the address book data saved to the **`JSON object`**, and then, using the innerHTML DOM method, output the result into the document's **`<body>`** element.
- Besides the output, a main difference to note in the following Listing is that we're now running a check on the **contactsCount** variable to make sure it's greater than **`0`** before continuing onto the loop.

for Loop (Cont'd)

```
/* cache some initial variables */
var object = contacts.addressBook, // save the data object
    contactsCount = object.length, // how many items in the JSON object? "5"
    target = document.getElementsByTagName("body")[0], // where you're outputting the
➡data
    i; // declare the "i" variable for later use in the loop

/* before doing anything make sure there are contacts to loop through */
if(contactsCount > 0) {

    /* loop through each JSON object item until you hit #5, then stop */
    for (i = 0; i < contactsCount; i = i + 1) {

        /* inside the loop "i" is the array index */
        var item = object[i],
            name = item.name,
            email = item.email;

        /* insert each person's name & mailto link in the HTML */
       target.innerHTML += '<p><a href="mailto:'+ email +'">' + name + '</a></p>';

    }
}
```

Performance Considerations

- As mentioned in an earlier Lesson, JavaScript, by nature, is **blocking**.
- That means it will stop the download of other objects on the page **until it is finished** with its business.
- This can be very evident when dealing with loops.
- The data we're dealing with here is only five items in length, so there isn't a problem executing this block of code 5 times.
- However, as the number of elements you're looping through increases, so will the time it takes to **iterate** over them.
- This is important to note when you're **looping** through a lot of items because it can really bog down the loading time of a page.

Conditionals

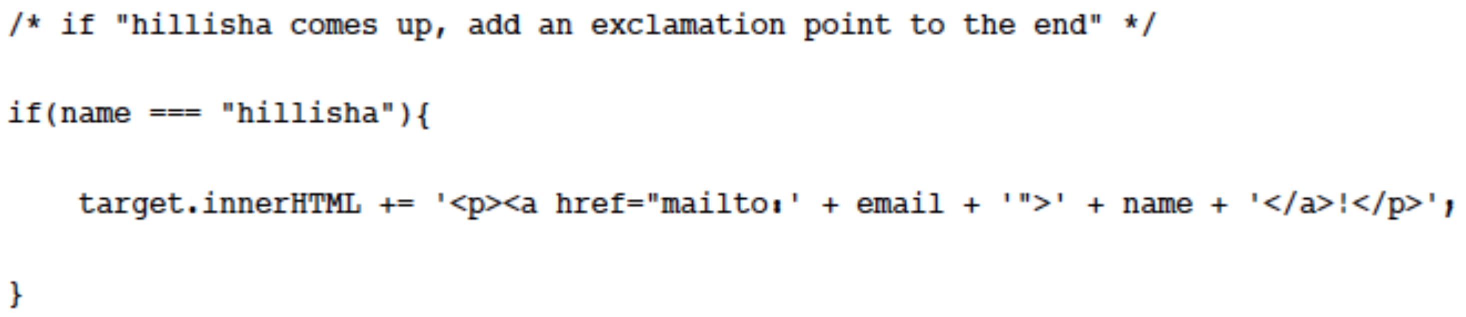
Georgian

Conditionals

- **Conditionals** are how you let your program make decisions for you.
- Decisions can be based on the data presented (decisions you make) or based on user input, like one of those choose-your-own adventure books. It’s a way to **inject some logic** into your JavaScript.
- Conditionals can be used for everything from outputting different information into the DOM to loading a completely different JavaScript file.

`if` Statement

- By far, the most common type of **conditional** is the **`if` statement**.
- An **`if` statement** checks a certain condition, and if true, executes a block of code.
- The **`if` statement** is contained within two curly brackets **`{ }`**, just like the loops we were talking about earlier and the functions before that.
- This is best described through a coding sample so let's move right to it.
- In the following Listing you can see a basic **`if` statement** that is being applied inside the loop of our **`JSON object`** in the following Listing.
- Inside the loop, if the person's name is "hillisha" the name and mailto link with an exclamation point at the end will be outputting into the document.
- This output should only be Hillisha's **mailto link** without any other names.



```
/* if "hillisha comes up, add an exclamation point to the end" */

if(name === "hillisha"){

    target.innerHTML += '<p><a href="mailto:' + email + '">' + name + '</a>!</p>';

}
```

`if / else` Statement

- In the following Listing the output was only a single person's name because the condition was set to handle only that one instance of name `===` " hillisha". Normally you will want do something for the rest of the people in your address book as they are outputted.
- The **`if / else`** statement is for just that purpose.
- The **`if / else`** statement gives you the capability to create multiple conditions and then a fallback condition for any items that don't meet the conditions' criteria.
- In the following Listing you can see that we are still looping through the address book JSON object, but this time we're setting three conditions:
  - if name is hillisha
  - if name is paul
  - everyone else

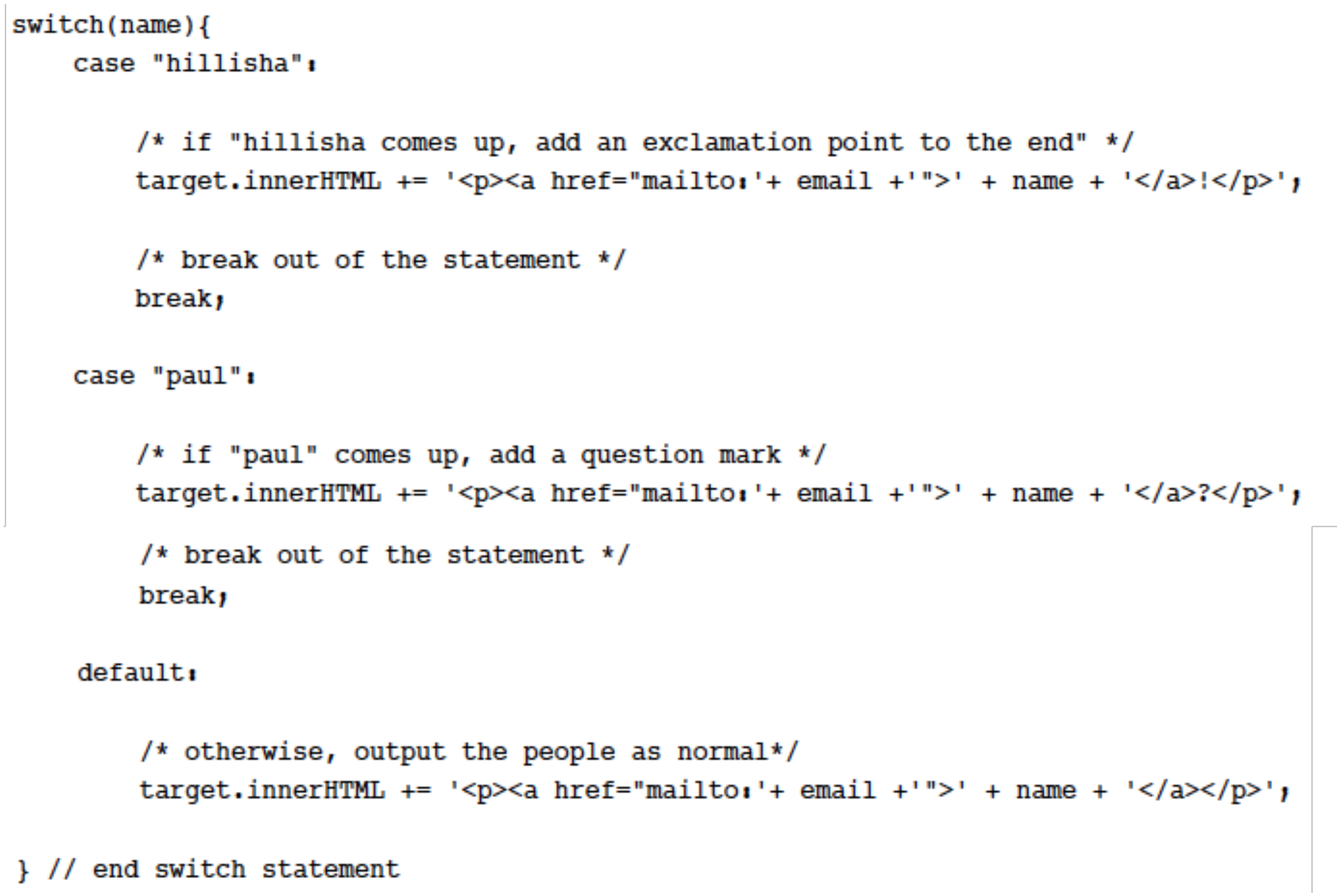
`if / else` Statement (Cont'd)

```
if(name === "hillisha"){

    /* if "hillisha comes up, add an exclamation point to the end" */
    target.innerHTML += '<p><a href="mailto:'+ email +'">' + name + '</a>!</p>';

} else if (name === "paul") { // line 5

    /* if "paul" comes up, add a question mark */
    target.innerHTML += '<p><a href="mailto:' + email + '">' + name + '</a>?</p>';

/* otherwise, output the person as normal*/
} else {

     target.innerHTML += '<p><a href="mailto:'+ email +'">' + name + '</a></p>';

}
```

`switch` Statement

- A **`switch`** statement, on the surface, functions almost exactly like an **`if / else`** statement. In a switch statement, you first have to set a **`switch`** value (the thing you're going to check for); in this example, we have been checking for name , so that's the switch value.
- You then set up cases to **test against**. We checked for "hillisha" once and also "paul"; those would be the cases used.
- Last, there is a **default state** if none of the cases return as **`true`**.
- The **`switch`** statement in the following Listing creates the same output as the **`if / else`** statement in the previous Listing, but under the hood and in syntax they are pretty different.
- Let's take a look at this **`switch`** statement.

switch Statement (Cont'd)



```
switch(name){
    case "hillisha":

        /* if "hillisha comes up, add an exclamation point to the end" */
        target.innerHTML += '<p><a href="mailto:'+ email +'">' + name + '</a>!</p>';

        /* break out of the statement */
        break;

    case "paul":

        /* if "paul" comes up, add a question mark */
        target.innerHTML += '<p><a href="mailto:'+ email +'">' + name + '</a>?</p>';

        /* break out of the statement */
        break;

    default:

        /* otherwise, output the people as normal*/
        target.innerHTML += '<p><a href="mailto:'+ email +'">' + name + '</a></p>';

} // end switch statement
```