Computer Science Principles

Web Programming

JavaScript Programming Essentials

CHAPTER 8: FUNCTIONS

DR. ERIC CHOU

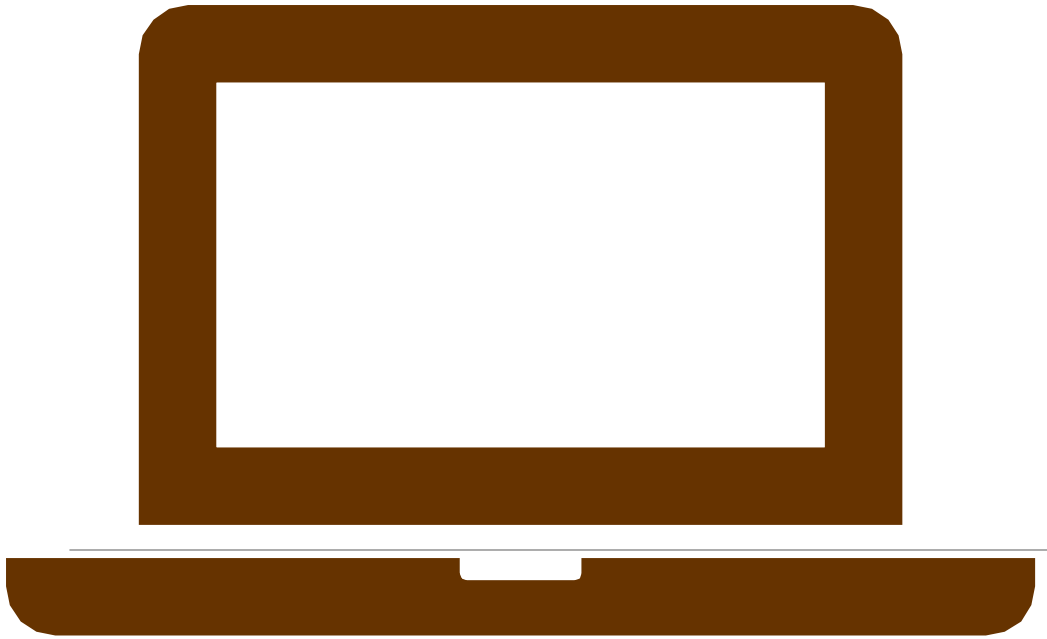
IEEE SENIOR MEMBER

Introduction

LECTURE 1

The Basic Anatomy of a Function

- Figure below shows how a function is built. The code between the curly brackets is called the *function body*, just as the code between the curly brackets in a loop is called the *loop body*.



In-Class Demonstration Program

- Function with return value
- Function sweeping a range
- data.csv

FUNCTION1.HTML

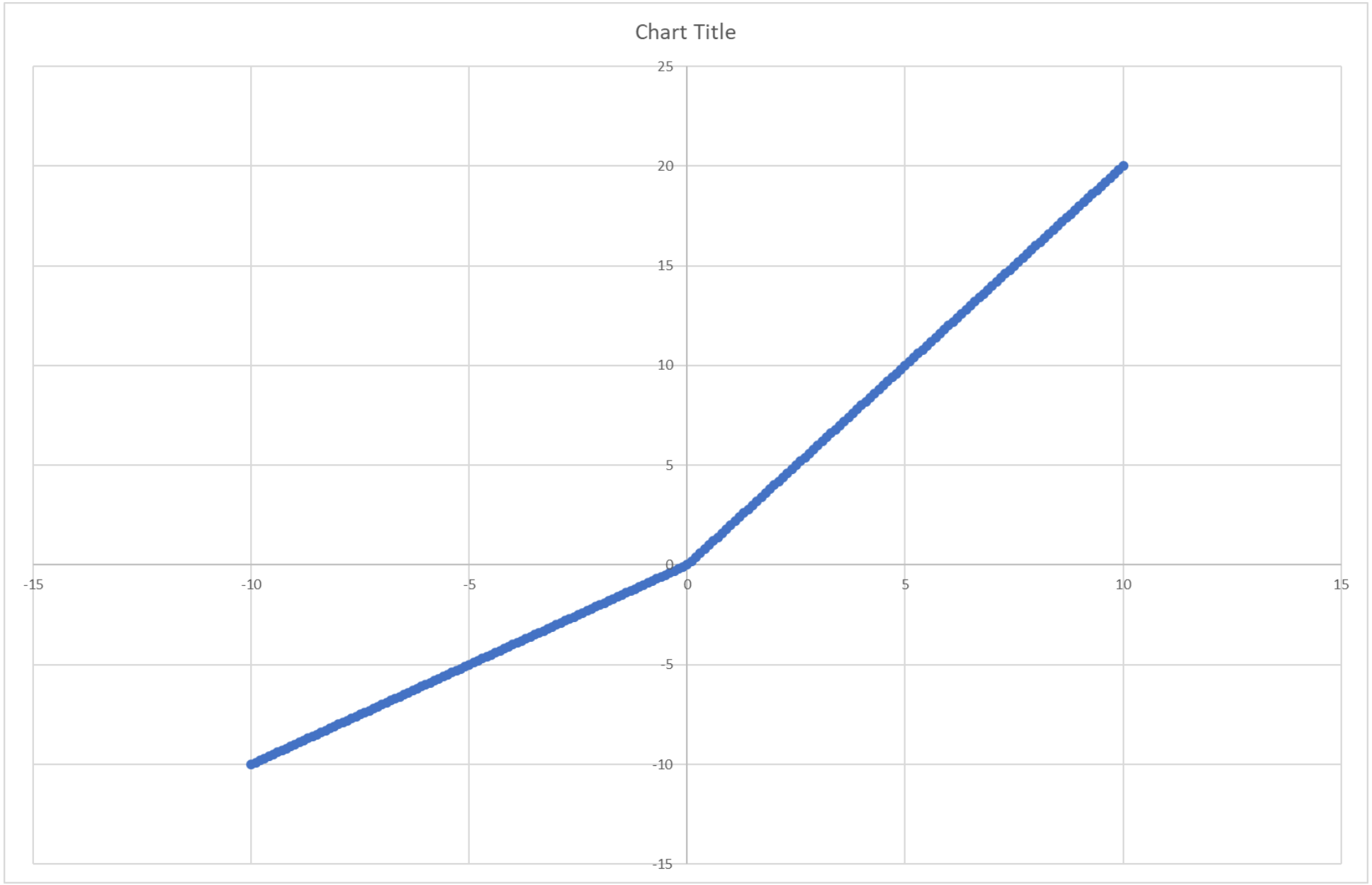


Chart Title
25
20
15
10
5
0
-5
-10
-15
-15
-10
-5
0
5
10
15

Creating a Simple Function

- Let's create a simple function that prints Hello world!. Enter the following code in the browser
- console. Use SHIFT-ENTER to start each new line without executing the code.

```
var ourFirstFunction = function () {
console.log("Hello world!");
};
```

- This code creates a new function and saves it in the variable **ourFirstFunction**.

Calling a Function

- To run the code inside a function (the function body), we need to call the function. To call a function, you enter its name followed by a pair of opening and closing parentheses, as shown here.

```
> ourFirstFunction();
Hello world!
```

- Calling **ourFirstFunction** executes the body of the function, which is **console.log("Hello world!");**, and the text we asked to be printed is displayed on the next line: **`Hello world!`**.

- But if you call this function in your browser, you'll notice that there's a third line, with a little leftfacing arrow, as shown in Figure 8-2. This is the return value of the function.

Functional Call

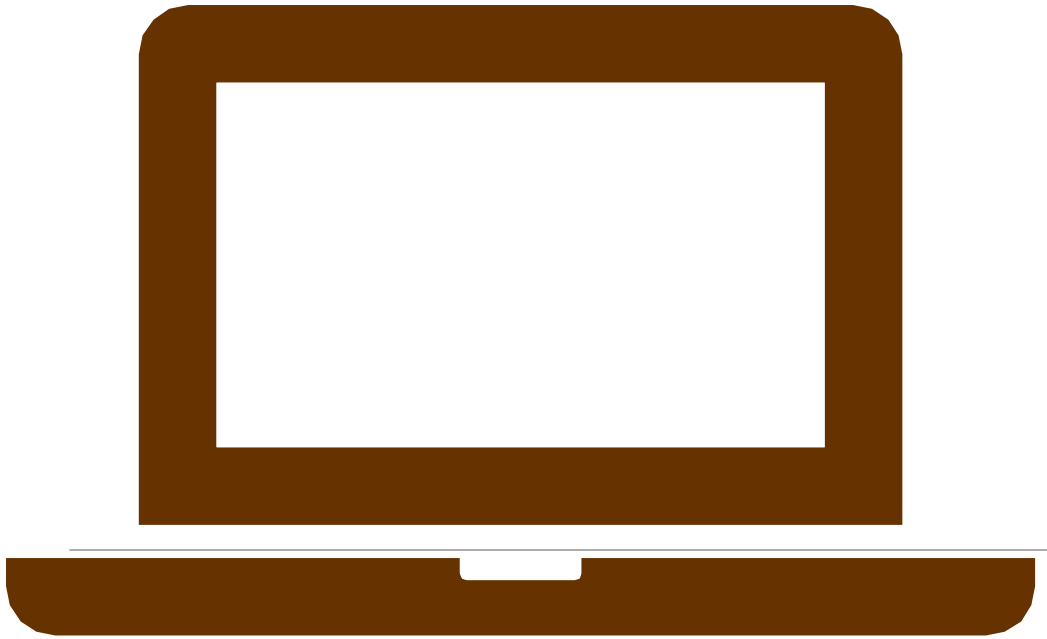
Figure 8-2. Calling a function with an undefined return value

Functional Call with undefined Return Value

- A **return** value is the value that a function outputs, which can then be used elsewhere in your code.
- In this case, the return value is undefined because we didn't tell the function to return any particular value in the body of the function.

Functional Call with undefined Return Value

- All we did was ask it to print a message to the console, which is not the same as returning a value.
- A function always returns **undefined** unless there is something in the function body that tells it to return a different value. (We'll look at how to specify a return value in Returning Values from Functions.)



In-Class Demonstration Program

- Function returning undefined

FUNCTION2.HTML

Parameters

LECTURE 2

Passing Arguments into Functions

- **ourFirstFunction** just prints the same line of text every time you call it, but you'll probably want your functions to be more flexible than that.
- Function *arguments* allow us to pass values into a function in order to change the function's behavior when it's called.
- Arguments always go between the function parentheses, both when you create the function and when you call it.

Passing Arguments into Functions

- The following **`sayHelloTo`** function uses an argument (name) to say hello to someone you specify.

```
var sayHelloTo = function (name) {
  console.log("Hello " + name +
"!");
};
```

Passing Arguments into Functions

We create the function in the first line and assign it to the variable **sayHelloTo**. When the function is called, it logs the string “Hello ” + name + “!”, replacing name with whatever value you pass to the function as an argument.

Figure 8-3 shows the syntax for a function with one argument.

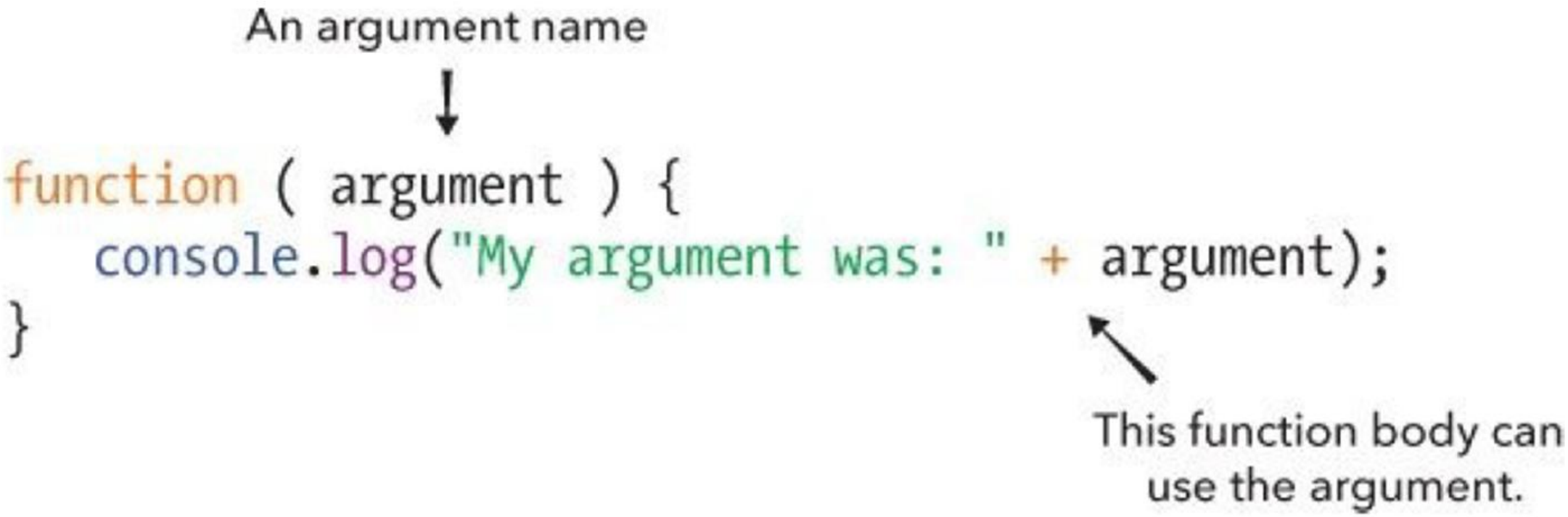


Figure 8-3. The syntax for creating a function with one argument

Passing Arguments into Functions

- To call a function that takes an argument, place the value you'd like to use for the argument between the parentheses following the function name. For example, to say hello to Nick, you would write:

```
sayHelloTo("Nick");
Hello Nick!
```

Passing Arguments into Functions

- Or, to say hello to Lyra, write:

```
sayHelloTo("Lyra");
Hello Lyra!
```

- Each time we call the function, the argument we pass in for name is included in the string printed by the function. So when we pass in “Nick”, the console prints “Hello Nick!”, and when we pass in “Lyra”, it prints "Hello Lyra!".

Demo Program: sayHello.html

```
1  <html>
2  <body>
3  <script>
4    var sayHelloTo = function (name) {
5     document.write("Hello " + name + "!<br>");
6     };
7    sayHelloTo("Eric");
8    sayHelloTo("Tom");
9    sayHelloTo("Johnny");
10 </script>
11 </body>
12 </html>
```

Printing Cat Faces!

```
var drawCats = function (howManyTimes){
for (var i = 0; i < howManyTimes; i++){
      console.log(i + " =^.^=");
  }
};
```

Printing Cat Faces!

Demo Program: printCats.html

```
drawCats(5);
```

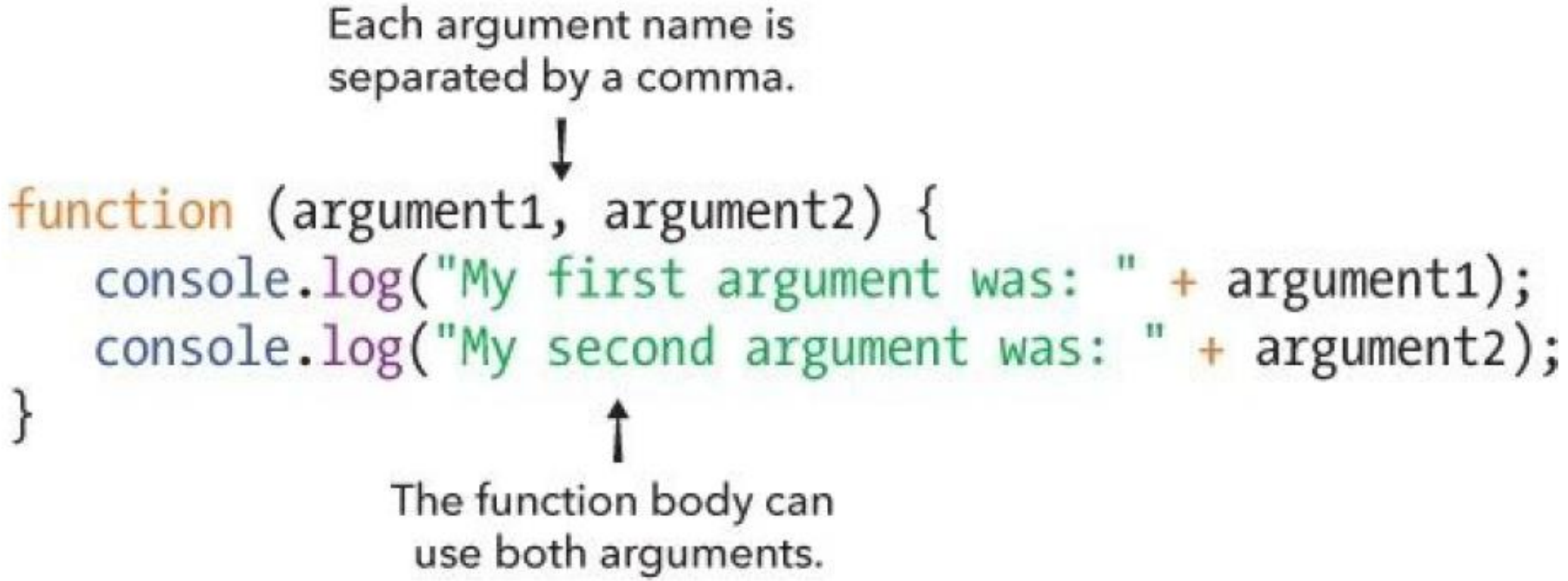
```
0 =^.^=
1 =^.^=
2 =^.^=
3 =^.^=
4 =^.^=
```

```
1  <html>
2  <body>
3  <script>
4  var drawCats = function (howManyTimes){
5  for (var i = 0; i < howManyTimes; i++){
6        document.write(i + " =^.^=<br>");
7    }
8  };
9   drawCats(5);
10 </script>
11 </body>
12 </html>
```

Multiple Arguments

LECTURE 3

Passing Multiple Arguments to a Function



Passing Multiple Arguments to a Function

- The following function, **printMultipleTimes**, is like drawCats except that it has a second argument called **whatToDraw**.

```
var printMultipleTimes = function (howManyTimes, whatToDraw){
for (var i = 0; i < howManyTimes; i++){
      console.log(i+" "+whatToDraw);
  }
};
```

Multiple Parameters

Demo Program: printManyTimes.html

```
> printMultipleTimes(5, "=^.^=");
0 =^.^=
1 =^.^=
2 =^.^=
3 =^.^=
4 =^.^=
```

```
printMultipleTimes(4, "^_^");
0 ^_^
1 ^_^
2 ^_^
3 ^_^
```

```
1 ▼ <html>
2 ▼ <body>
3 ▼ <script>
4 ▼ var printMultipleTimes = function (howManyTimes, whatToDraw){
5 ▼ for (var i = 0; i < howManyTimes; i++){
6         document.write(i+" "+whatToDraw+"<br>");
7     }
8   };
9    printMultipleTimes(5, "=^.^=");
10   printMultipleTimes(4, "^_^");
11  </script>
12  </body>
13  </html>
```

Return Value

LECTURE 4

Returning Values from Functions

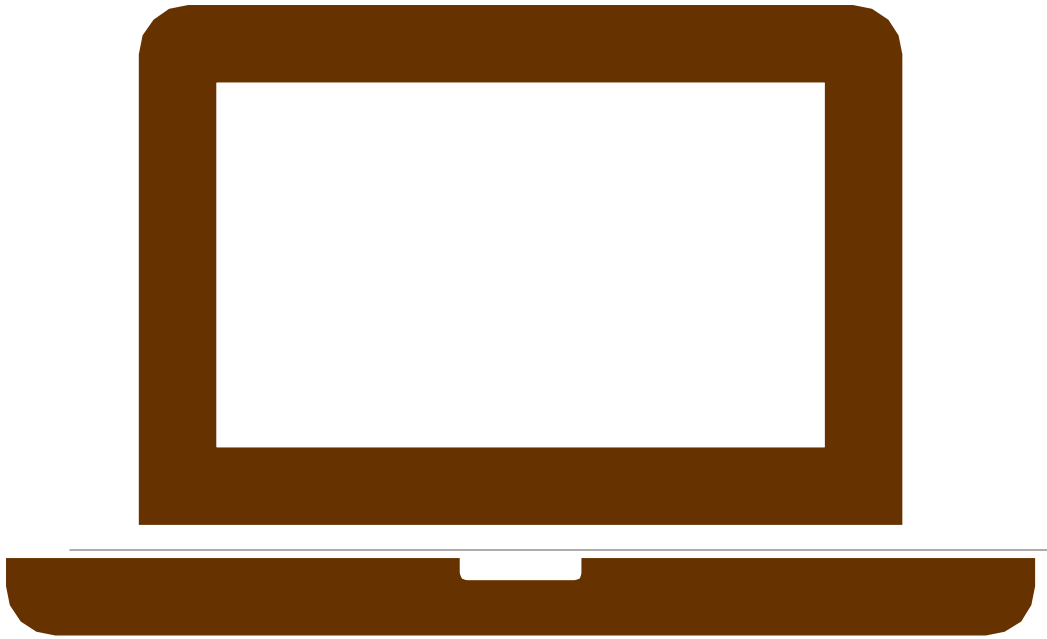
- The output of a function is called the return value. When you call a function that returns a value, you can use that value in the rest of your code (you could save a return value in a variable, pass it to another function, or simply combine it with other code).
- For example, the following line of code adds 5 to the return value of the call to **`Math.floor(1.2345)`**:

```
> 5 + Math.floor(1.2345);
6
```

Define a Function with Return Value

- Let's create a function that returns a value. The function double takes the argument number and returns the result of number * 2. In other words, the value returned by this function is twice the number supplied as its argument.

```
var double = function (number) {
    return number * 2;
};
> double(3);
6
```



In-Class Demonstration Program

- Dice (6 faces)
- Dice (multiple faces, base)

DICE.HTML

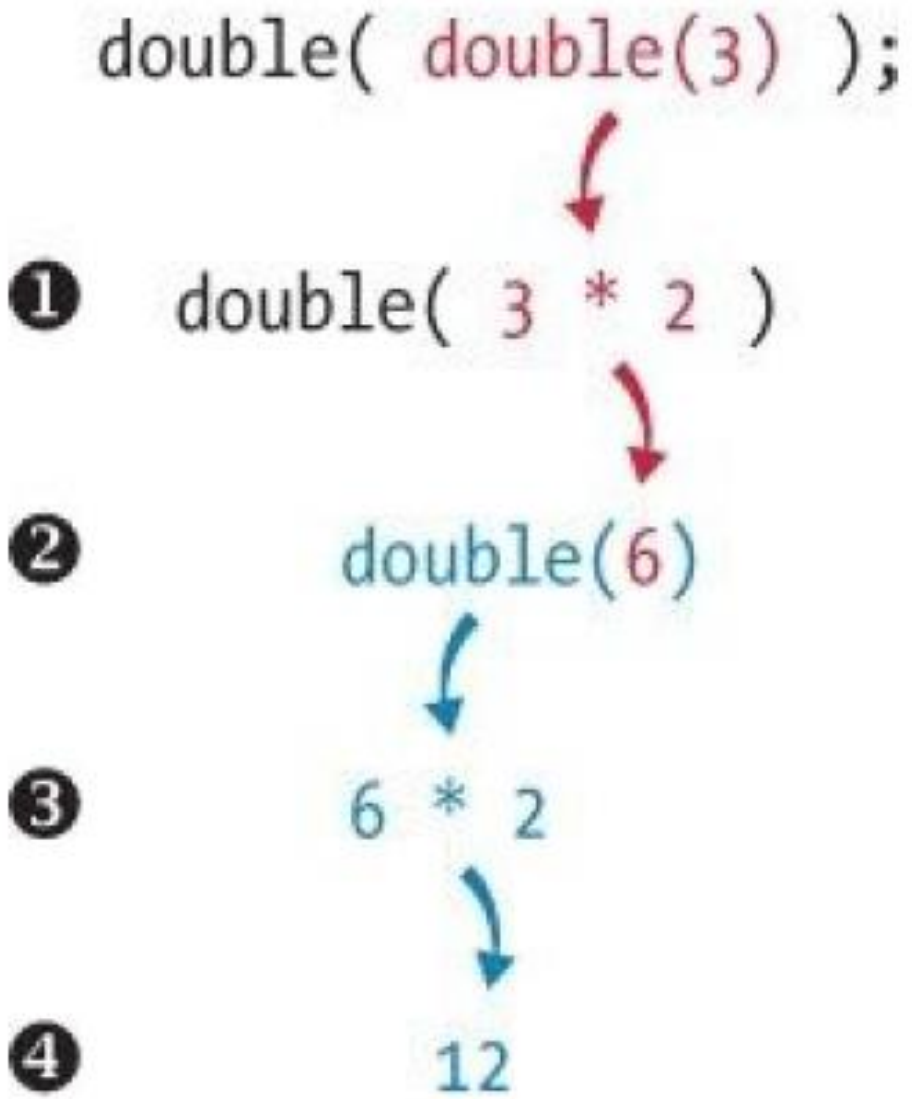
Function Call as Value

LECTURE 5

Using Function Calls as Values

- You can also pass a function call into another function as an argument, and the function call will be substituted with its return value.
- In this next example we call double, passing the result of calling `double` with 3 as an argument. We replace double(3) with 6 so that `double(double(3))` simplifies to `double(6)`, which then simplifies to 12.

```
> double(double(3));
12
```


double(double(3));
❶ double(3 * 2)
❷ double(6)
❸ 6 * 2
❹ 12

Procedural Abstraction

LECTURE 6

Procedural Abstraction

- Procedural abstraction is the idea that each method should have a coherent conceptual description that separates its implementation from its users.
- You can encapsulate behavior in methods that are internal to an object or methods that are widely usable.

Using Functions to Simplify Code

In Chapter 3, we used the methods **`Math.random`** and **`Math.floor`** to pick random words from arrays and generate random insults. In this section, we'll re-create our insult generator and simplify it by creating functions.

A Function to Pick a Random Word

PART 1

A Function to Pick a Random Word

Demo Program: randomWords.html

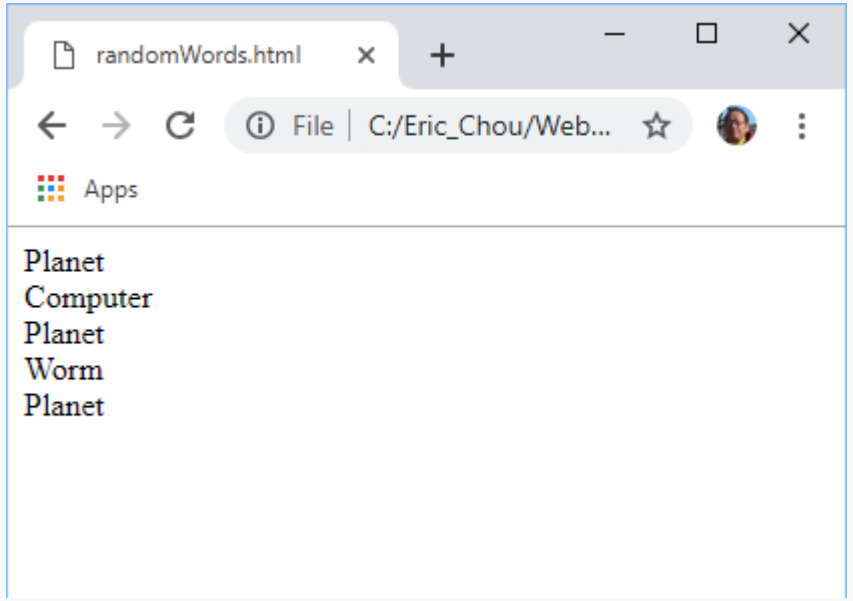
```
var randomWords = ["Planet", "Worm", "Flower", "Computer"];
var pickRandomWord = function (words) {
    return words[Math.floor(Math.random() * words.length)];
};

> pickRandomWord(randomWords);
"Flower"
```

A Function to Pick a Random Word

- We can use this same function on any array. For example, here's how we would pick a random name from an array of names:

```
> pickRandomWord(["Charlie", "Raj", "Nicole", "Kate", "Sandy"]);
"Raj"
```

```
1  <html>
2  <body>
3  <script>
4     var randomWords = ["Planet", "Worm", "Flower", "Computer"];
5  var pickRandomWord = function (words) {
6        return words[Math.floor(Math.random() * words.length)];
7  };
8  for (var i=0; i<5; i++){
9        var w = pickRandomWord(randomWords);
10       document.write(w+"<br>");
11 }
12 </script>
13 </body>
14 </html>
```



A Random Insult Generator

PART 2

A Random Insult Generator

Demo Program: insult1.html

- Now let's try re-creating our random insult generator, using our function that picks random words. First, here's a reminder of what the code from Chapter 3 looked like:

```
var randomBodyParts = ["Face", "Nose", "Hair"];
var randomAdjectives = ["Smelly", "Boring", "Stupid"];
var randomWords = ["Fly", "Marmot", "Stick", "Monkey", "Rat"];

// Pick a random body part from the randomBodyParts array:
var randomBodyPart = randomBodyParts[Math.floor(Math.random() * 3)];
// Pick a random adjective from the randomAdjectives array:
var randomAdjective = randomAdjectives[Math.floor(Math.random() * 3)];
// Pick a random word from the randomWords array:
var randomWord = randomWords[Math.floor(Math.random() * 5)];
// Join all the random strings into a sentence:
var randomString = "Your " + randomBodyPart + " is like a " + ↵
randomAdjective + " " + randomWord + "!!!";
randomString;
"Your Nose is like a Stupid Marmot!!!"
```

A Random Insult Generator

- Notice that we end up repeating ***words*[Math.floor(Math.random() * *length*)]** quite a few times in this code. Using our **pickRandomWord** function, we could rewrite the program like this:

```
var randomBodyParts = ["Face", "Nose", "Hair"];
var randomAdjectives = ["Smelly", "Boring", "Stupid"];
var randomWords = ["Fly", "Marmot", "Stick", "Monkey", "Rat"];

// Join all the random strings into a sentence:
var randomString = "Your " + pickRandomWord(randomBodyParts) + ↵
" is like a " + pickRandomWord(randomAdjectives) + ↵
" " + pickRandomWord(randomWords) + "!!!";

randomString;
"Your Nose is like a Smelly Marmot!!!"
```

A Random Insult Generator

- There are two changes here. First, we use the **`pickRandomWord`** function when we need a random word from an array, instead of using ***`words`*`[Math.floor(Math.random()*`*`length`*`)]`** each time. Also, instead of saving each random word in a variable before adding it to the final string, we're adding the return values from the function calls directly together to form the string. A call to a function can be treated as the value that the function returns. So really, all we're doing here is adding together strings.
- As you can see, this version of the program is a lot easier to read, and it was easier to write too, since we reused some code by using a function.

```
<html>
<body>
<script>
var pickRandomWord = function (words) {
    return words[Math.floor(Math.random() * words.length)];
};
var randomBodyParts = ["Face", "Nose", "Hair"];
var randomAdjectives = ["Smelly", "Boring", "Stupid"];
var randomWords = ["Fly", "Marmot", "Stick", "Monkey", "Rat"];
// Pick a random body part from the randomBodyParts array:
var randomBodyPart = randomBodyParts[Math.floor(Math.random() * 3)];
// Pick a random adjective from the randomAdjectives array:
var randomAdjective = randomAdjectives[Math.floor(Math.random() * 3)];
// Pick a random word from the randomWords array:
var randomWord = randomWords[Math.floor(Math.random() * 5)];
// Join all the random strings into a sentence:
var randomString = "Your " + randomBodyPart + " is like a " +
randomAdjective + " " + randomWord + "!!!";
document.write(randomString+"<br>");
</script>
</body>
</html>
```

Making the Random Insult Generator into a Function

PART 3

Making the Random Insult Generator into a Function

- We can take our random insult generator one step further by creating a larger function that produces random insults. Let's take a look:

```
generateRandomInsult = function () {
  var randomBodyParts = ["Face", "Nose", "Hair"];
  var randomAdjectives = ["Smelly", "Boring", "Stupid"];
  var randomWords = ["Fly", "Marmot", "Stick", "Monkey", "Rat"];
  // Join all the random strings into a sentence:
  var randomString = "Your " + pickRandomWord(randomBodyParts) + ↵
  " is like a " + pickRandomWord(randomAdjectives) + ↵
  " " + pickRandomWord(randomWords) + "!!!";

❶   return randomString;
};
```

Making the Random Insult Generator into a Function

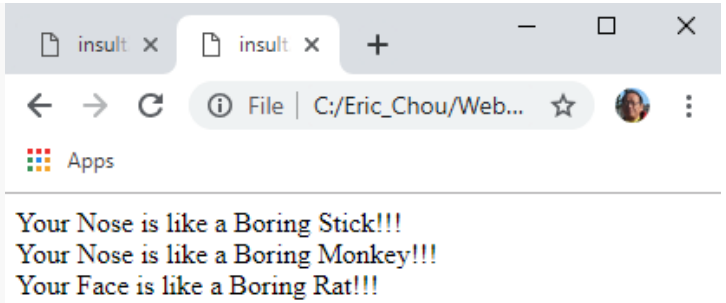
```
generateRandomInsult();
"Your Face is like a Smelly Stick!!!"
generateRandomInsult();
"Your Hair is like a Boring Stick!!!"
generateRandomInsult();
"Your Face is like a Stupid Fly!!!"
```

Making the Random Insult Generator into a Function

- Our new **`generateRandomInsult`** function is just the code from before placed inside a function with no arguments. The only addition is at ❶, where we have the function return **`randomString`** at the end. You can see a few sample runs of the preceding function, and it returns a new insult string each time.
- Having the code in one function means we can keep calling that function to get a random insult, instead of having to copy and paste the same code every time we want a new insult.

Demo Program: insult2.html

```
1  <html>
2  <body>
3  <script>
4  var pickRandomWord = function (words) {
5      return words[Math.floor(Math.random() * words.length)];
6  };
7
8  var generateRandomInsult = function () {
9      var randomBodyParts = ["Face", "Nose", "Hair"];
10     var randomAdjectives = ["Smelly", "Boring", "Stupid"];
11     var randomWords = ["Fly", "Marmot", "Stick", "Monkey", "Rat"];
12     // Join all the random strings into a sentence:
13     var randomString = "Your " + pickRandomWord(randomBodyParts) +
14                        " is like a " + pickRandomWord(randomAdjectives) +
15                        " " + pickRandomWord(randomWords) + "!!!";
16     return randomString;
17     };
18 for (var i=0; i<3; i++){
19     var randomString = generateRandomInsult();
20     document.write(randomString+"<br>");
21   }
22 </script>
23 </body>
24 </html>
```



Return as A Break level

LECTURE 7

Leaving a Function Early with return

- One common way to use return is to leave a function early if any of the arguments to the function are *invalid*; that is, if they're not the kind of arguments the function needs in order to run properly.
- For example, the following function returns a string telling you the fifth character of your name. If the name passed to the function has fewer than five characters, the function uses return to leave the function immediately.
- This means the return statement at the end, which tells you the fifth letter of your name, is never executed.

Return as a break level

```
  var fifthLetter = function (name) {
❶   if (name.length < 5) {
❷     return;
    }

    return "The fifth letter of your name is " + name[4] + ".";
  };
```

- At ❶ we check to see whether the length of the input name is less than five. If it is, we use return at ❷ to exit the function early.

Return as a break level

- Let's try calling this function.

```
> fifthLetter("Nicholas");
"The fifth letter of your name is o."
```

- The name *Nicholas* is longer than five characters, so **fifthLetter** completes and returns the fifth letter in the name *Nicholas*, which is the letter *o*. Let's try calling it again on a shorter name:

```
> fifthLetter("Nick");
undefined
```

- When we call **fifthLetter** with the name *Nick*, the function knows that the name isn't long enough, so it exits early with the first return statement at ❷. Because there is no value specified after the return at ❷, the function returns **undefined**.

Replacement of Else by Return

LECTURE 8

Replacement of Else by Return

- We can use multiple return keywords inside different if statements in a function body to have a function return a different value depending on the input. For example, say you're writing a game that awards players medals based on their score. A score of 3 or below is a bronze medal, scores between 3 and 7 are silver, and anything above 7 is gold. You could use a function like **medalForScore** to evaluate a score and return the right kind of medal, as shown here:

```
var medalForScore = function (score) {
  if (score < 3) {
❶    return "Bronze";
  }

❷  if (score < 7) {
     return "Silver";
  }

❸  return "Gold";
};
```

Function as HTML Wrapper

LECTURE 9

HTML Wrapper

Demo Program: wrapper.html

- In order to write an object to the document, functions can be used as HTML wrappers.
- Using function calls can make repetition more efficient.
- The template can be pre-defined and then, brought into JavaScript section.
- Template literals can be used (but not demonstrated in here.)
- The pre-defined template is in the field.html file.

```
<html>
<body>
<script>
 function print(student){
    return '<fieldset><ul>'+
           '<li>Name: '+student.name+'</li>'+
           '<li>Age: '+student.age+'</li>'+
           '<li>Address: '+student.address+', '+student.city+', '+student.state+', '+
           student.zipcode+'</li>'+
           '</ul></fieldset>';
  }
var s1 = {
  name: "Tommy Jones",
  age: 15,
  address: "1 A Street",
  city: "Los Angeles",
  state: "CA",
  zipcode: "90007"
 };
var s2 = {
  name: "Nancy Adams",
  age: 16,
  address: "100 B Street",
  city: "Las Vegas",
  state: "NV",
  zipcode: "80109"
 };
 document.write(print(s1));
 document.write(print(s2));
</script>
</body>
</html>
```

- Name: Tommy Jones
- Age: 15
- Address: 1 A Street, Los Angeles, CA, 90007

- Name: Nancy Adams
- Age: 16
- Address: 100 B Street, Las Vegas, NV, 80109

Advanced Topics I: Function as Object

LECTURE 10

Functions as Objects

- Everything in JavaScript happens in functions.
- A **function** is a block of code, self contained, that can be defined once and run any times you want.
- A function can optionally accept parameters, and returns one **value**.

Functions as Objects

- **Functions** in JavaScript are **objects**, a special kind of objects: **function objects**. Their superpower lies in the fact that they can be invoked.
- In addition, functions are said to be first class functions because they can be assigned to a value, and they can be passed as arguments and used as a return value.

Syntax

- **Function Declaration:**

```
function dosomething(foo) {
  // do something
}
```

(now, in post ES6/ES2015 world, referred as a **regular function**)

- Functions can be assigned to variables (this is called a **function expression**):

```
const dosomething = function(foo) {
  // do something
}
```

Named Function Expressions (Optional)

Named **function expressions** are similar, but play nicer with the stack call trace, which is useful when an error occurs - it holds the name of the function:

```
const dosomething = function dosomething(foo) {
  // do something
}
```

Lambda Function: (optional)

Arrow Functions

- ES6/ES2015 introduced **arrow functions**, which are especially nice to use when working with inline functions, as **parameters** or **callbacks**:

```
const dosomething = foo => {
  //do something
}
```

- Arrow functions have an important difference from the other function definitions above, we'll see which one later as it's an advanced topic.

Advanced Topics II:

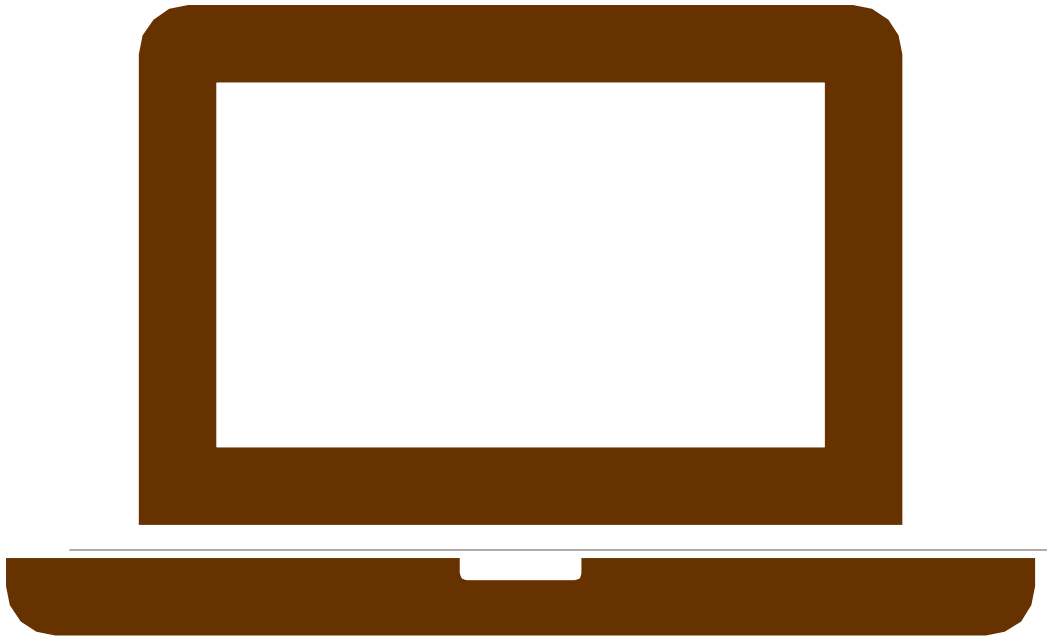
Parameters

LECTURE 11

Parameters

A function can have one or more parameters.

```
const dosomething = () => {
  //do something
}
const dosomethingElse = foo => {
  //do something
}
const dosomethingElseAgain = (foo, bar) => {
  //do something
}
```



In-Class Demonstration Program

LAMBDA.HTML

Parameters with Default Values

- Starting with ES6/ES2015, functions can have default values for the parameters:

```
const dosomething = (foo = 1, bar = 'hey') => {
  //do something
}
```

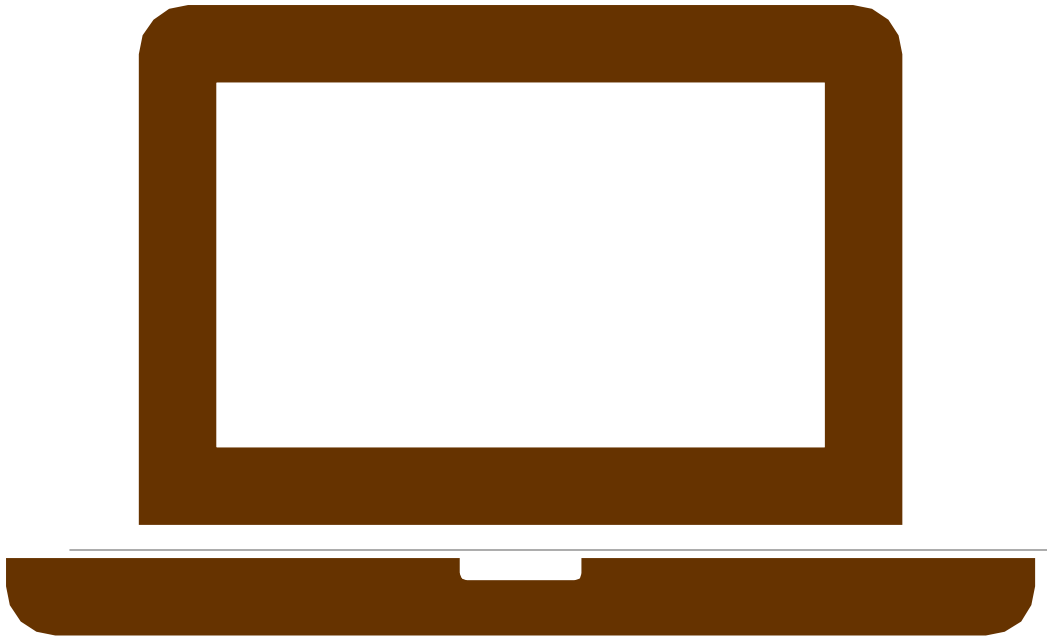
- This allows you to call a function without filling all the parameters:

```
dosomething(3)

dosomething()
```

Parameters with Default Values

- ES2018 introduced trailing commas for parameters, a feature that helps reducing bugs due to missing commas when moving around parameters (e.g. moving the last in the middle):

```
const dosomething = (foo = 1, bar = 'hey') => {
  //do something
}

dosomething(2, 'ho!')
```



In-Class Demonstration Program

LAMBDA2.HTML

Parameters with Default Values

- You can wrap all your arguments in an array, and use the spread operator when calling the function:

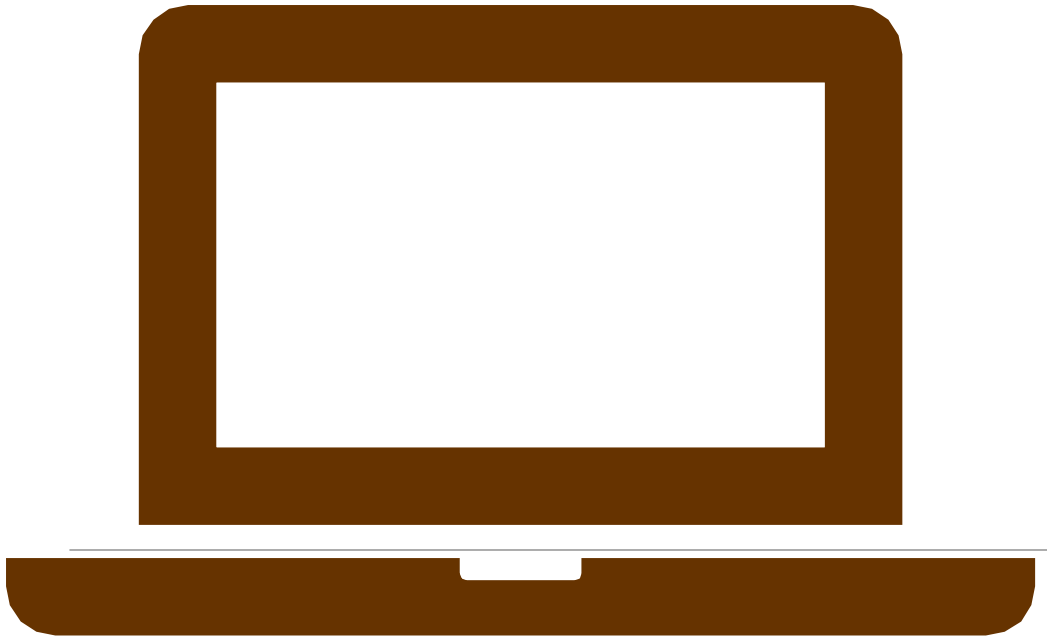
```
const dosomething = (foo = 1, bar = 'hey') => {
  //do something
}

const args = [2, 'ho!']

dosomething(...args)
```

Out of Order Parameters (Using Object)

- With many parameters, remembering the order can be difficult.

Using objects, destructuring allows to keep the parameter names:

```
const dosomething = ({ foo = 1, bar = 'hey' }) => {
  //do something
  console.log(foo) // 2
  console.log(bar) // 'ho!'
}
const args = { foo: 2, bar: 'ho!' }
dosomething(args)
```



In-Class Demonstration Program

LAMBDA3.HTML

Advanced Topics III:

Use Array as Tuple for Return Value

LECTURE 12

Returning Multiple Values

Arrays

- To simulate returning multiple values, you can return an object literal, or an array, and use a destructuring assignment when calling the function.
- Using arrays:

```
> const dosomething = () => {
    return ['Roger', 6]
  }
  const [ name, age ] = dosomething()
<· undefined
> name
<· "Roger"
> age
<· 6
```

Returning Multiple Values

Objects

- Using objects:

```
> const dosomething = () => {
    return { name: 'Roger', age: 6 }
  }
  const { name, age } = dosomething()
<· undefined
> name
<· "Roger"
> age
<· 6
```

Advanced Topics IV:

Object Methods

LECTURE 13

Object Methods

- When used as object properties, functions are called methods:

```
const car = {
  brand: 'Ford',
  model: 'Fiesta',
  start: function() {
    console.log(`Started`)
  }
}
> car.start()
```

this in Arrow Functions

- There's an important behavior of Arrow Functions vs regular Functions when used as object methods. Consider this example:

```
const car = {
  brand: 'Ford',
  model: 'Fiesta',
  start: function() {
    console.log(`Started ${this.brand} ${this.model}`)
  },
  stop: () => {
    console.log(`Stopped ${this.brand} ${this.model}`)
  }
}
```

this in Arrow Functions

```
> const car = {
    brand: 'Ford',
    model: 'Fiesta',
    start: function() {
      console.log(`Started ${this.brand} ${this.model}`)
    },
    stop: () => {
      console.log(`Stopped ${this.brand} ${this.model}`)
    }
  }

  car.start()
  car.stop()
```

```
Started Ford Fiesta
```

```
Stopped undefined undefined
```

The stop() method does not work as you would expect. This is because the handling of **this** is different in the two functions declarations style. **this** in the arrow function refers to the enclosing function context, which in this case is the **window** object.

```
const car = {
  brand: 'Ford',
  model: 'Fiesta',
  start: function() {
    console.log(this)
    console.log(`Started ${this.brand} ${this.model}`)
  },
  stop: () => {
    console.log(this)
    console.log(`Stopped ${this.brand} ${this.model}`)
  }
}

car.start()
car.stop()
```

```
▶ {brand: "Ford", model: "Fiesta", start: ƒ, stop: ƒ}
Started Ford Fiesta

▶ Window {postMessage: ƒ, blur: ƒ, focus: ƒ, close: ƒ,
  , …}
Stopped undefined undefined
```

this in Arrow Functions

Dynamic Binding

this, which refers to the **host** object using function()

This implies that arrow functions are not suitable to be used for object methods and constructors (arrow function constructors will actually raise a **TypeError** when called).

Note:

this is static binding to the object car.