Lecture 2

Web Layouts

Introduction to JavaScript (JS)

List of topics

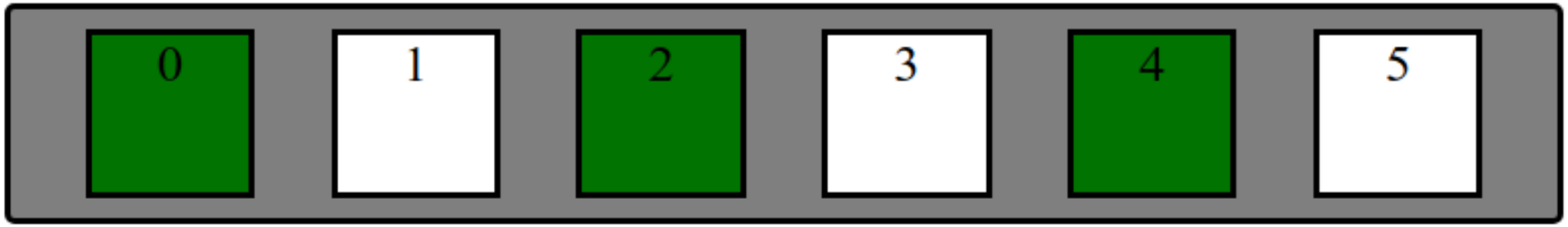
- Layout with Flex
- Responsive Web Design
- What is JavaScript?
- JavaScript in HTML
- Language Basics
- Event-driven programming

Layout with Flex

Distributing Boxes Evenly in a Container

How could we distribute boxes across box container evenly (equal space between each box)?.

... what should we do about the margins of the boxes?
... what value do we put?
... how many screen sizes do we need to design for?



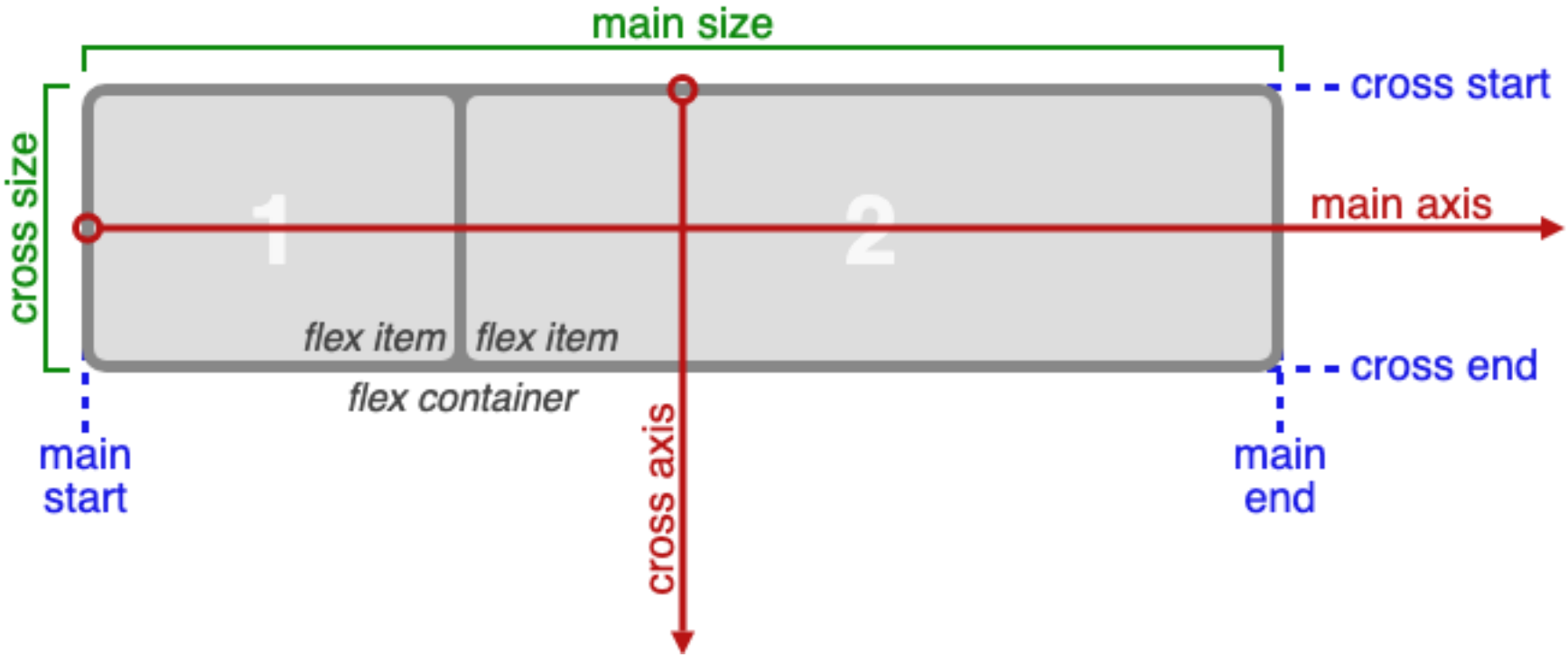
`display` Property

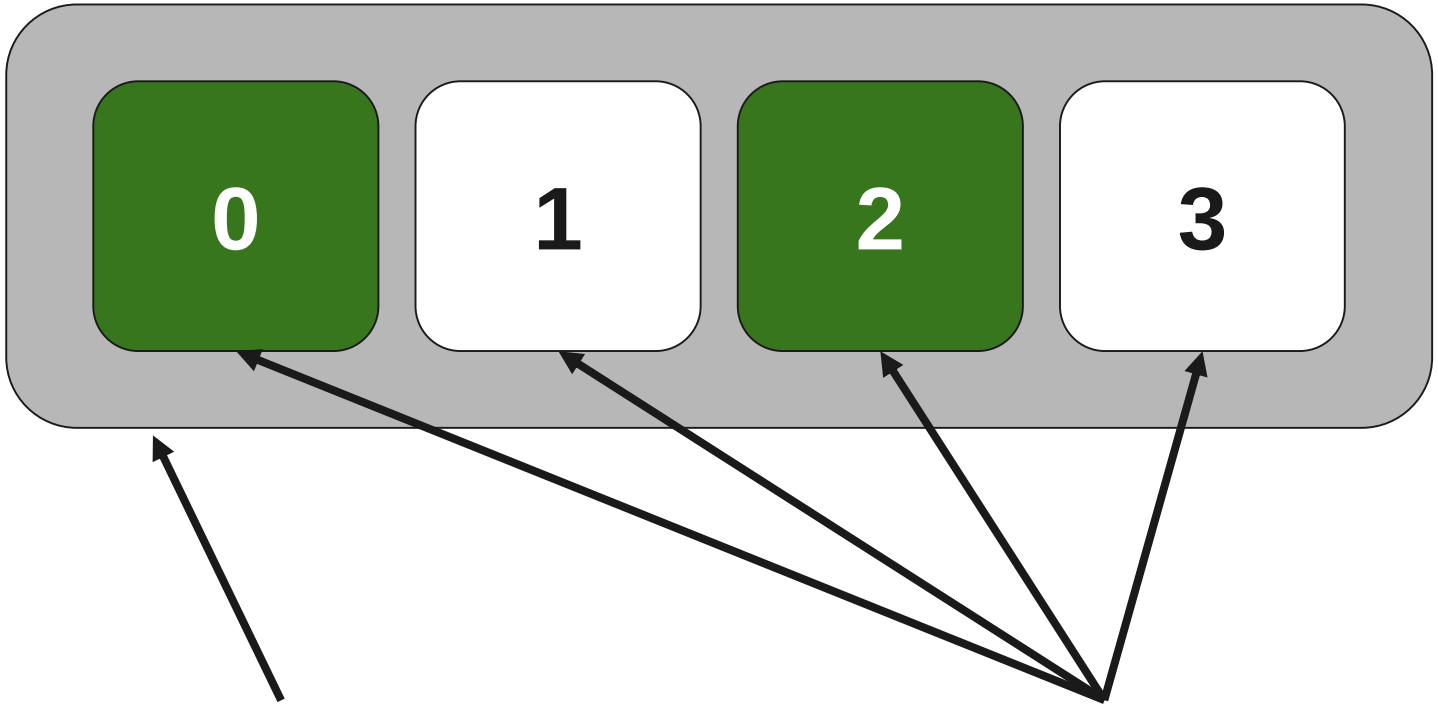
The display property specifies the display behavior (the type of rendering box) of an element. The four values you most often will see are:

- `inline`: Displays an element as an inline element, spanning the width/height of its content. Any height and width properties will have no effect.
- `block`: Displays an element as a block element. It starts on a new line, and takes up the width of the parent container.
- `initial`: Initial or default display value.
- `none`: The element is completely removed.
- `flex`: Displays an element as a block-level flex container.
- `grid`: Displays elements in a 2-dimensional grid.

Flexbox

- Flexbox is a set of CSS properties for aligning block level content.
- Flexbox defines two types of content - "containers" and "items".
- Anything directly nested inside of a flex container becomes a flex item.
- Various properties on the container can be set to determine how its items are laid out.





Flex **container**
(Defined with the CSS property `display: flex`)

Flex **items**
(Defined implicitly because their parent is a flex container)

1. What is this container's "*flex direction*"?
2. What CSS do we need to add to make the 0,1,2,3 boxes "*flex items*"?
3. What is the container's "*main axis*"? "*cross axis*"?
4. What is the container's size?
5. What are the *items'* sizes?

Basic properties for the flex container

`display: flex;`

- makes an element a "flex container", items inside automatically become "items" - by default, starts as a row

`justify-content: flex-end; (flex-start, space-around,...)`

- indicates how to space the items inside the container along the main axis

`align-items: flex-end; (flex-start, center, baseline,...)`

- indicates how to space the items inside the container along the cross axis

`flex-direction: row; (column)`

- indicates whether the container flows horizontally or vertically (default row)

`flex-wrap: wrap; (no-wrap, ...)`

- indicates whether the container's children should wrap on new lines

Basic properties for flex items

There are also cases when you will need to add flex properties to flex *items* rather than the flex *container*

`flex-grow: <number>`

- Defines a proportional value to determine whether a flex items can grow (what amount of the available space inside the container it should take up).

`flex-basis: 20%; (3em, 50px,...)`

- indicates the default size of an element before the extra space is distributed among the items

`align-self: flex-end; (flex-start, center, stretch,...)`

- indicates where to place this specific item along the cross axis

positioning Elements

`position: static`

- Default value. Elements render in order, as they appear in the document flow

`position: fixed`

- Puts an element at an exact position within the browser window

`position: absolute`

- Puts an element at an absolute position based on the location of the element's parent container

`position: relative`

- Makes children positioned relative to the parent container
- Handy for sticking a footer to the bottom of a page, for example

`position: sticky`

- A "hybrid" - toggles between relative and fixed depending on scroll position

Another good explanation is here

Responsive Web Design

What is Responsive Web Design?

Responsive Web Design (RWD) is an approach to web design that ensures a website looks good and functions well on all devices, from desktops to smartphones.

Fluid Layouts

Layouts that use relative units (like percentages) rather than fixed units (like pixels) to ensure content adapts to different screen sizes.

Flexible Images

Images, videos, and other media are designed to scale proportionally, maintaining their visual integrity as the layout adapts to varying screen dimensions.

Media Queries

CSS media queries allow designers to apply specific styles based on device characteristics, such as screen size, resolution, and orientation, ensuring the website's visual presentation is optimized for each user's environment.

Benefits of Responsive Design

Improved User Experience

Ensures a consistent and user-friendly experience across all devices.

Cost-Effective

A single responsive website is more cost-effective than maintaining separate versions for desktop and mobile.

SEO Advantages

Google prefers responsive designs as they ensure all content is accessible from a single URL.

Responsive with Media Queries

Media queries are a CSS technique used to apply styles based on the characteristics of the device or viewport.

To create responsive designs that adapt to different screen sizes and orientations, improving user experience on various devices.

Basic Syntax

```
@media (condition) {
      /* CSS rules */
}
```

```
@media (max-width: 600px) {
    body {
        background-color: lightblue;
    }
}
```

Types of Media Queries

Media Features:

Width and Height: `min-width, max-width, min-height, max-height`
Orientation: `orientation: portrait, orientation: landscape`
Resolution: `min-resolution, max-resolution`
Aspect Ratio: `min-aspect-ratio, max-aspect-ratio`

Combining Media Queries:

Logical Operators: `and, not, only`
Comma Separated List: Apply styles for multiple conditions.

Example

```
/* Combine conditions using 'and' */
@media (min-width: 600px) and (max-width: 1200px) {
    body {
        background-color: lightgreen;
    }
}

/* Using 'not' to exclude certain devices */
@media not screen and (min-width: 600px) {
    body {
        background-color: lightcoral;
    }
}

/* Applying styles for multiple conditions */
@media (max-width: 600px), (orientation: landscape) {
    body {
        background-color: lightyellow;
    }
}
```

JavaScript (JS)

What we've learned so far

- How to write content for a webpage using HTML5
- How to add styles to a webpage using CSS and linking a CSS file to an HTML file
- How to inspect the HTML and CSS of web pages in the browser

What we’re going to discuss...

1. Web page structure and appearance with HTML5 and CSS.
2. **Client-side interactivity with JS DOM and events.**

JavaScript is to Java as ...

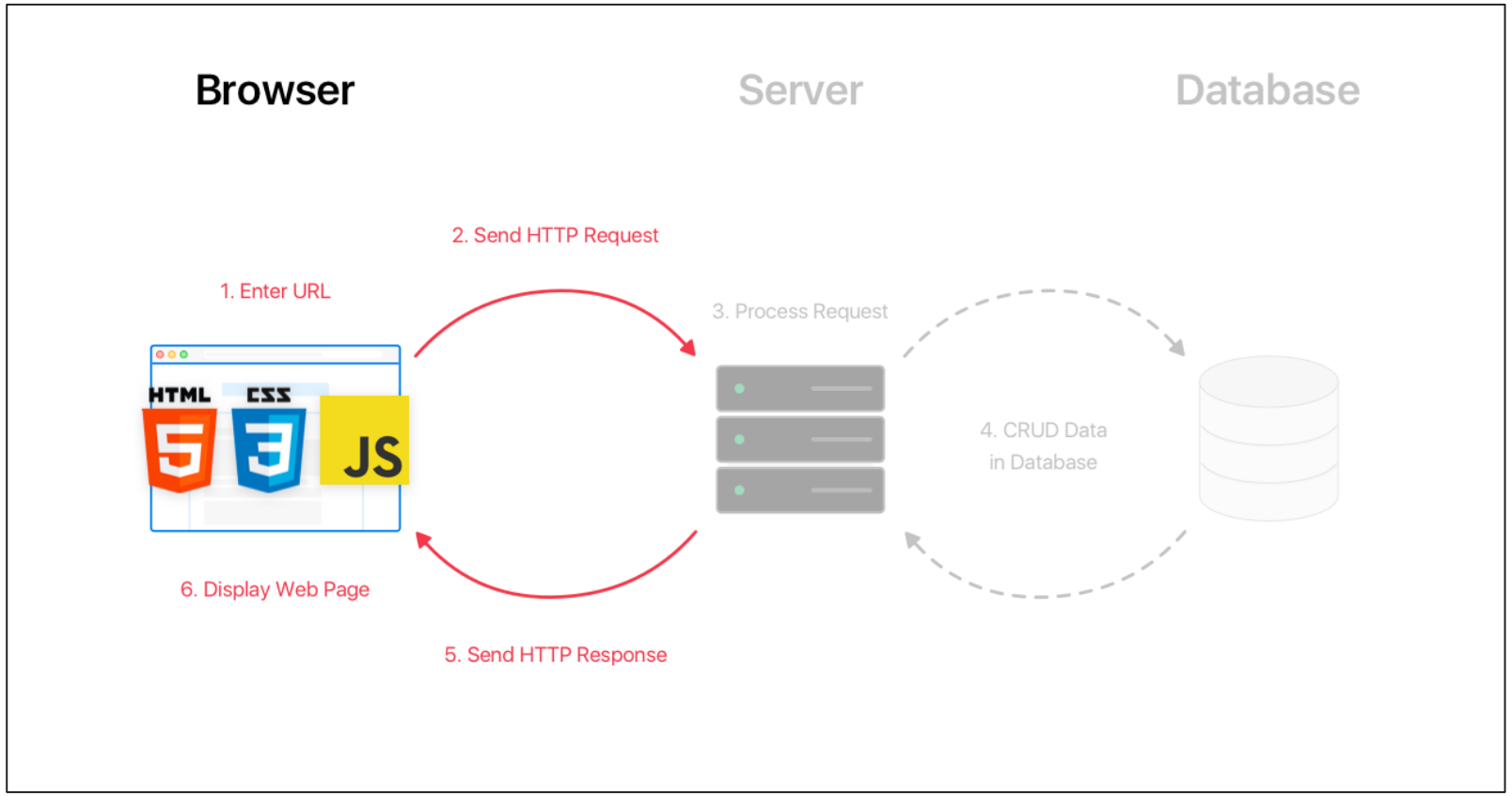
Grapefruit → Grape
Carpet → Car
Hamster → Ham
Catfish → Cat

In short: there's no relationship.

What is JavaScript?

- A lightweight "scripting" programming language
- Created in 1995 by Brendan Eich (original prototype was created in 10 days and called LiveScript)
- NOT related to Java other than name and some syntactic similarities...
- Used to define interactivity for web pages.

Terminology: Client-Side Scripting



Client-side script: Code that runs on the user's computer and does not need a server to run (just a web browser!).

Client-side JavaScript runs as part of the browser's process to load HTML and CSS (e.g., from a server response). This JavaScript usually manipulates the page or responds to user actions through "event handlers."

Why JavaScript and not another language?

Popularity.

The early web browsers supported it as a lightweight and flexible way to add interactivity to pages.

Microsoft created their own version, called JScript, but the open source browsers (notably Firefox and Chromium) and Adobe (via Flash and ActionScript) put all their effort into JavaScript and standardized it as ECMAScript.

Note: If you want to run anything other than JavaScript in the browser… it's Very Hard™. There's a long list of languages that people have "transpilers" for -- basically converting one language to JavaScript. These are often error-prone, and have potential performance problems.

JS: Adding Behavior to HTML/CSS

We can use JavaScript functions to...

- Insert dynamic text into HTML (e.g., username)
- React to events (e.g., page load, user's mouse click)
- Get information about a user's computer (e.g., what browser they are using)
- Request additional data needed for the page (e.g., from an API; more on this in a couple weeks)

JS: Code in web pages

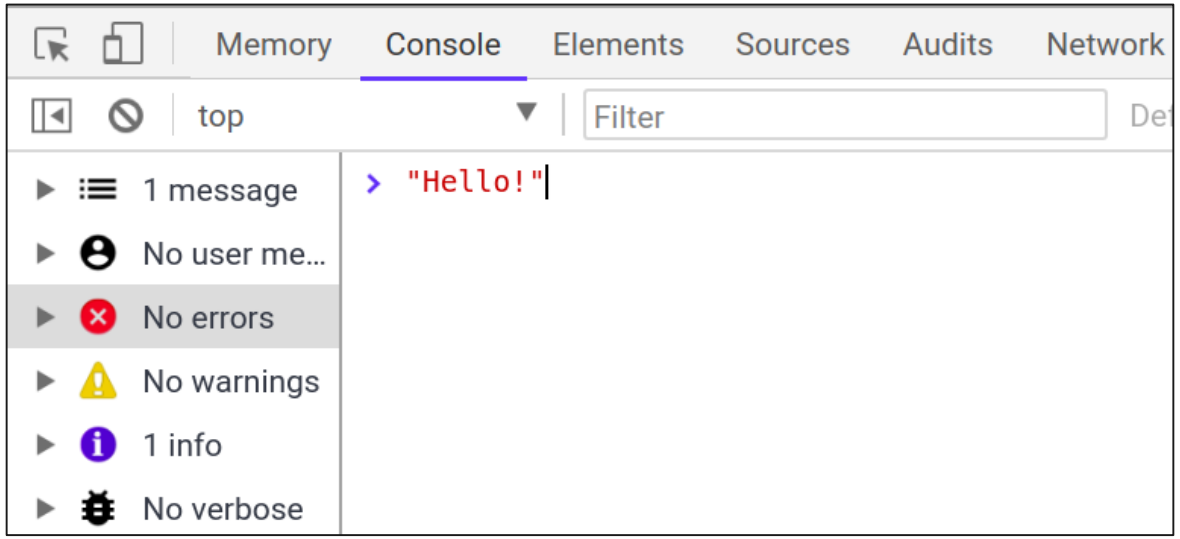
HTML can embed JavaScript files into the web page via the `<script>` tag.

```
<!DOCTYPE html>
<html>
<head>
    <title>WPR</title>
    <link rel="stylesheet" href="style.css" />
    <script src="filename.js"></script>
</head>
<body>
    ... contents of the page...
</body>
</html>
```

Today: Following Along

As an interpreted programming language, JS is great to interact with a line at a time (similar to Python, but very different from Java). Where do you start?

The easiest way to dive in is with the Chrome browser's Console tab in the same inspector tool you've used to inspect your HTML/CSS.



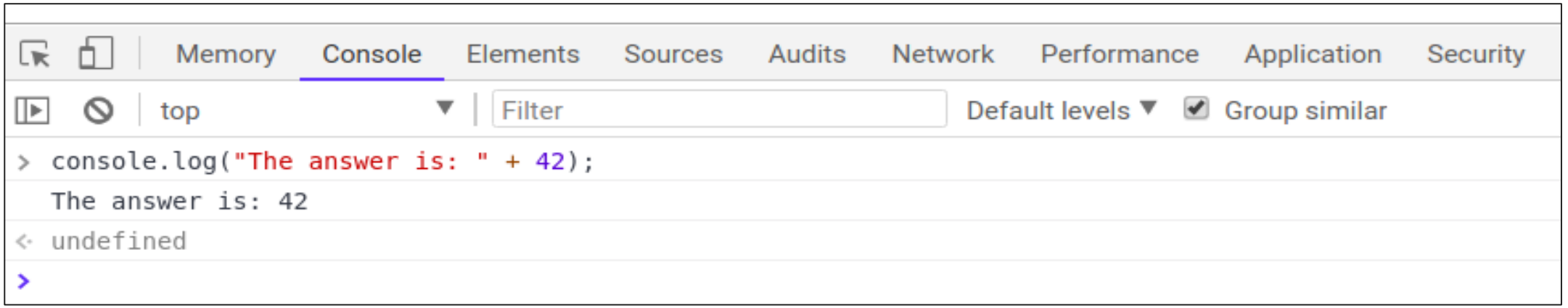
Until we learn how to interact with the HTML DOM with JS, we recommend experimenting with the following code examples using this console to get comfortable with the basic syntax and behavior.

Our First JavaScript Statement: `console.log`

Used to output values to the browser console, most often used to debug JS programs. You can think of this as `System.out.println` in Java or `print` in Python.

```
console.log("message");
```

```
console.log("The answer is: " + 42);
```

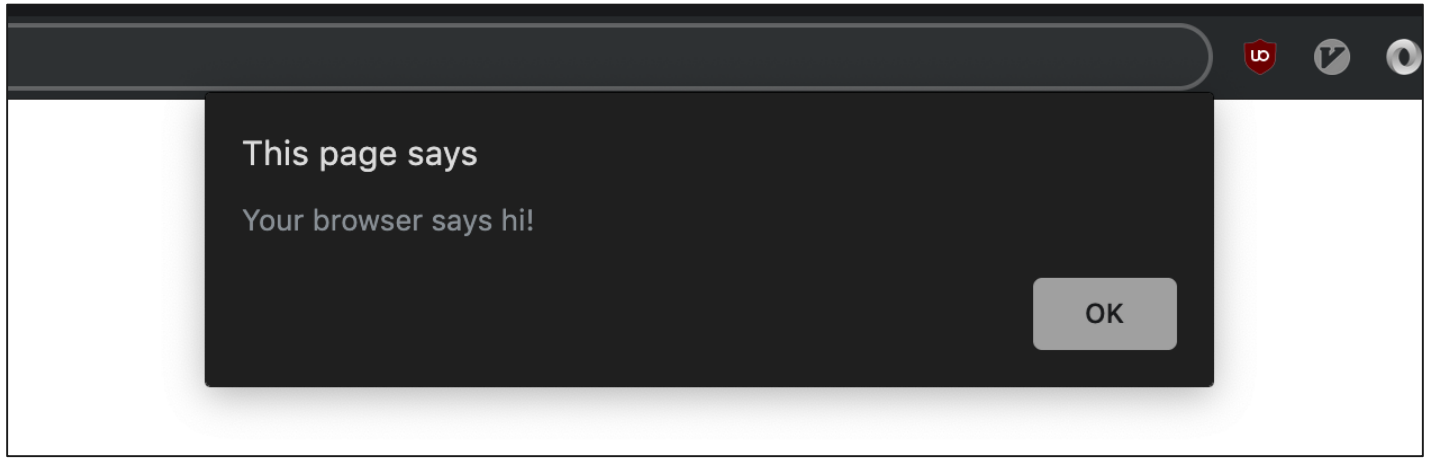


The `alert` function

A JS function that pops up a dialog box with a message - **not ideal in practice**, but sometimes a recommended debugging tool when first learning JS.

```
alert("message");
```

```
alert("Your browser says hi!");
```



Comments (similar to Java)

```
// single-line comment

/**
 * multi-line
 * comment
 */
```

Identical to Java's comment syntax

Recall: 3 comment syntaxes

- HTML: <!-- comment -->
- CSS/Java/JS: /* comment */
- Java/JS: // single comment
    - /* multi-line */

Variables

```
// template
let name = expression;

// examples
let level = 23;
let accuracyRate = 0.99;
let name = "Pikachu";
```

Variables are declared with the `let` keyword (case-sensitive). You may also see `var` used instead of `let` - this is an older convention with weaker scope -> **DO NOT USE** `var` anywhere

Code Quality Tips: Use camelCasing for variable (and function) names

“Types” in JavaScript

```
let level = 23; // Number
let accuracyRate = 0.99; // Number
let name = "Pikachu"; // String
let temps = [55, 60, 57.5]; // Array
```

Types are not specified, but JS does have types ("loosely-typed")

- `Number, Boolean, String, Array, Object, Function, Null, Undefined`
- Can find out a variable's type by calling typeof, but usually this is poor practice (why?)

A Note about Declaring Types in JavaScript

If you've programmed in a statically-typed language like Java, you will recall that when declaring variables, you must specify their type which **must** always stay the same.

```
boolean isValid = "hello!";
// error in JavaScript. boolean keyword doesn't exist
```

```
let isValid = true; // no error
isValid = "hello!";
isValid = 1;
isValid = true;
```

In a dynamically-typed language like JavaScript, you don't need to specify the type (just use `let` or `const`) and you may change the type the variable refers to later in execution.

Number Type

```
let enrollment = 99;
let medianGrade = 2.8;
let credits = 5 + 4 + (2 * 3);
```

- Integers and real numbers are the same type (no `int` vs. `double`). All numbers in JS are floating point numbers.
- Same operators: `+ - * / % ++ -- = += -= *= /= %=` and similar precedence to Java.
- Many operators auto-convert types: `"2" * 3` is `6`
- `NaN` ("Not a Number") is a return value from operations that have an undefined numerical result (e.g. dividing a String by a Number).

String type

```
let nickName = "Sparky O'Sparkz";                    // "Sparky O'Sparks"
let fName = nickName.substring(0, s.indexOf(" ")); // "Sparky"
let len = nickName.length;                           // 15
let name = 'Pikachu';                                // can use "" or ''
```

Methods: charAt, charCodeAt, fromCharCode, indexOf, lastIndexOf, replace, split, substring, toLowerCase, toUpperCase

More about Strings

Escape sequences behave as in Java: `\' \" \& \n \t \\`

To convert between Numbers and Strings:

```
let count = 10;                                  // 10
let stringedCount = "" + count;                  // "10"
let puppyCount = count + " puppies, yay!";       // "10 puppies, yay!"
let magicNum = parseInt("42 is the answer"); // 42
let mystery = parseFloat("Am I a number?");  // NaN
```

To access characters of a String `s`, use `s[index]` or `s.charAt(index)`:

```
let firstLetter  = puppyCount[0];                              // "1"
let fourthLetter = puppyCount.charAt(3);                       // "p"
let lastLetter   = puppyCount.charAt(puppyCount.length - 1); // "!"
```

Common Bugs when Using Strings

While Strings in JS are fairly similar to those you'd use in Java, there are a few special cases that you should be aware of.

- Remember that `length` is a property (not a method, as it is in Java)
- Concatenation:

    `1 + 1` is `2`, but `"1" + 1` and `1 + "1"` are both `"11"`!

Practice: Write a function named `repeat` that accepts a string and a number of repetitions as parameters and returns the String concatenated that many times. For example, the call of `repeat("echo...", 3)` returns `"echo...echo...echo..."`. If the number of repetitions is 0 or less, return an empty string.

Special Values: `null` and `undefined`.

```
let foo = null;
let bar = 9;
let baz;

/* At this point in the code,
 * foo is null
 * bar is 9
 * baz is undefined
 */
```

`undefined`: declared but has not yet been assigned a value

`null`: exists, but was specifically assigned an empty value or `null`.

A good motivating overview of `null` vs. `undefined`

Note: This takes some time to get used to, and remember this slide if you get confused later.

Arrays

```
let name = [];                              // empty array
let names = [value, value, ..., value]; // pre-filled
names[index] = value;                       // store element
```

```
let types = ["Electric", "Water", "Fire"];
let pokemon = [];          // []
pokemon[0] = "Pikachu";  // ["Pikachu"]
pokemon[1] = "Squirtle"; // ["Pikachu", "Sqiurtle"]
pokemon[3] = "Magikarp"; // ["Pikachu", "Sqiurtle", undefined, "Magikarp"]
pokemon[3] = "Gyarados"; // ["Pikachu", "Sqiurtle", undefined, "Gyarados"]
```

- Two ways to initialize an array
- `length` property (grows as needed when elements are added)

Some Notes on Typing

As you write JS programs, you may run into some silent bugs resulting from odd typing behavior in JS. Automatic type conversion, or coercion, is a common, often perplexing, source of JS bugs (even for experienced JS programmers).

Why is this important to be aware of? You'll be writing programs which use variables and conditional logic. Knowing what is considered truthy/false and how types are evaluated (at a high level) can make you a much happier JS developer.

Examples of some "not-so-obvious" evaluations:

```
2 < 1 < 2;                  // true
0 + "1" + 2;                // "012"
[] + [];                    // ""
"1" / null;                 // Infinity
[+!![]]+[[![]+!![]]];       // "11"
```

This is worth 3 minutes of your viewing pleasure. (starting at 1:20)

Equality

JavaScript's == and != are basically broken: they do an implicit type conversion before the comparison.

```
'' == '0'           // false
'' == 0             // true
0 == '0'            // true
NaN == NaN          // false
[''] == ''          // true
false == undefined  // false
false == null       // false
null == undefined   // true
```

Equality

Instead of fixing == and != , the ECMAScript standard kept the existing behavior but added === and !==

```
'' === '0'           // false
'' === 0             // false
0 === '0'            // false
NaN === NaN          // still weirdly false
[''] === ''          // false
false === undefined  // false
false === null       // false
null === undefined   // false
```

→ Always use === and !== and don't use == or !=

Defining Functions

```
// template
function name(params) {
  statement;
  statement;
  ...
  statement;
}

// example
function myFunction() {
  console.log("Hello!");
  alert("Your browser says hi!");
}
```

The above could be the contents of `basics.js` linked to our HTML page

Statements placed into functions can be evaluated in response to user events

JS Function vs. Java Method

```
function repeat(str, n) {
  let result = str;
  for (let i = 1; i < n; i++) {
    result += str;
  }
  return result;
}
let repeatedStr = repeat("echo...", 3); // "echo...echo...echo..."
```

```
public static String repeat(String str, int n) {
  String result = str;
  for (int i = 1; i < n; i++) {
    result += str;
  }
  return result;
}
String repeatedStr = repeat("echo...", 3); // "echo...echo...echo..."
```

Why do we talk about these things separately?

- *Separation of Concerns*: a concept from Software Engineering that says every part of a program should address a separate "concern".
- In web programming, we define those concerns as:
  - Content (words, images)
  - Structure/Meaning (HTML)
  - Style/Appearance (CSS)
  - Behavior (JS)

General model for HTML & JS

In practice, to add interactivity to our HTML/CSS websites we need to

1. Link a JS program to our HTML (in `<head>` or at the bottom of `<body>`)
2. Identify the elements we want to “listen” to user/page events
3. Identify the events we want to respond to
4. Identify what each response function is
5. Assign the functions to the listening elements when the event(s) occurs

Linking to a JS file: `<script>`

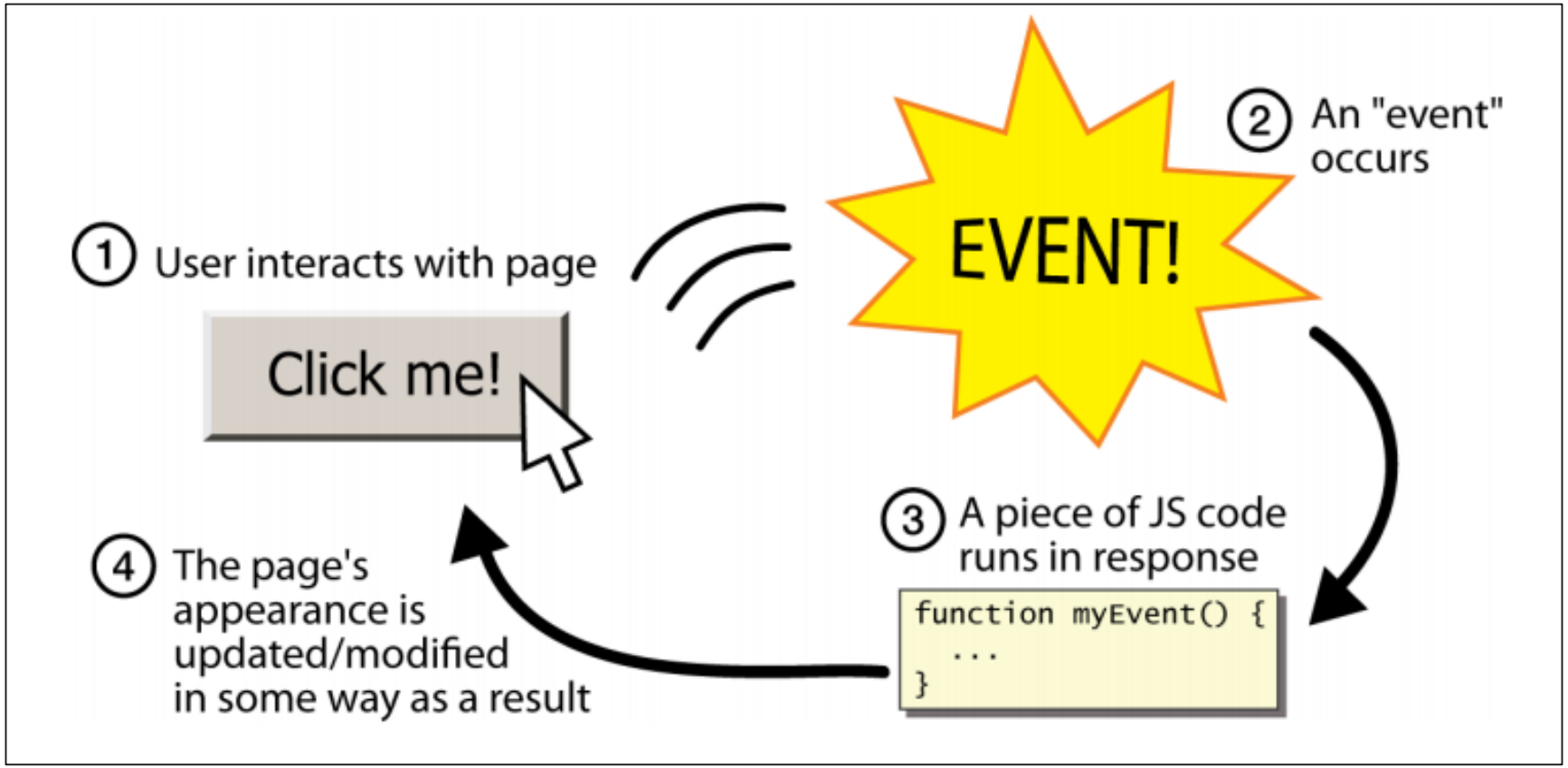
```
// template
<script src="filename"></script>

// example
<script src="example.js"></script>
```

JavaScript code used in the page should be stored in a separate `.js` file

JS code can be placed directly in the `HTML` file's `body` or `head` (like `CSS`), ***but this is only for small projects. In more serious projects, you should separate content, presentation, and behavior ("separation of concerns" rule).***

Event-driven programming

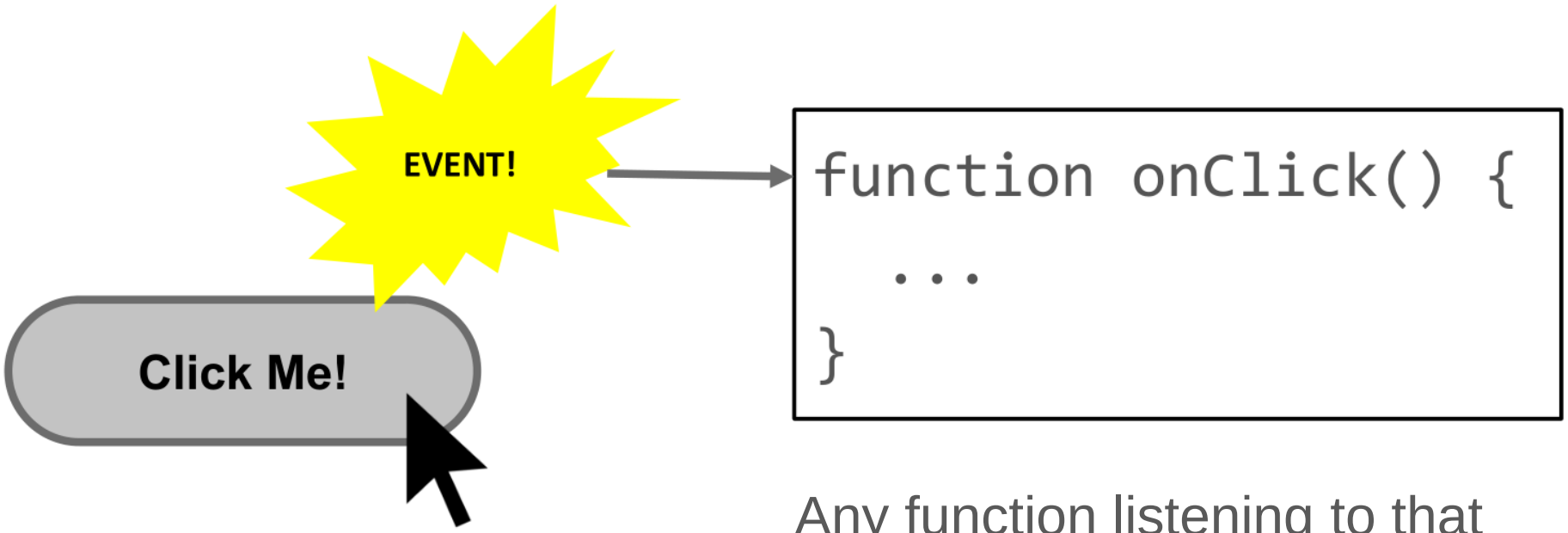


Unlike Java programs, JS programs have no `main`; they respond to user actions called **events**

Event-Driven Programming: writing programs driven by user events

Event-driven programming

- Most JavaScript written in the browser is event-driven.



Any function listening to that event now executes. This function is called an "event handler."

Accessing an element by `id`

```
let elem = document.getElementById("id");
```

- `document.getElementById` returns the DOM object for an element with a given `id` (note that you omit the `#` when giving an `id`)
- Other ways to get DOM elements include for example `document.querySelector` to get elements by class name

Using CSS Selectors in JS

Name	Description
`querySelector(selector)`	returns the first element that would be matched by the given CSS selector string
`querySelectorAll(selector)`	returns an array of all elements that would be matched by the given CSS selector string

What’s inside a DOM object?

```
<img src="images/puppy.png" alt="A fantastic puppy photo">
```

For starters, the HTML attributes. This HTML above is a DOM object (let's call it `puppyImg`) with these two properties:

- `puppyImg.src` -- set by the browser to `images/puppy.png`
- `puppyImg.alt` -- set by the browser to `"A fantastic puppy photo"`

When are elements accessible?

```
window.onload = function() {
    // access DOM elements here
}

// or (BETTER)
window.addEventListener('load', (event) => {
    // access DOM elements here
});
```

DOM elements are available when page is loaded. Listen to the `load` event of the `window` object.

Another possible way is to put JS code at the end of `<body>` but it's not the best practice. (still being done by many programmers)

The <button>

```
<button id="my-btn">Go!</button>
```

Text inside of the `button` tag renders as the button text

To make a responsive button (or other UI controls):

1. Choose the control (e.g., button) and event (e.g., mouse click) of interest
2. Write a JavaScript function to run when the event occurs
3. Attach the function to the event on the control

Listen & respond to events

```
// attaching a named function
element.addEventListener("click", handleFunction);


function handleFunction() {
    // event handler code
}
```

- JavaScript functions can be set as event handlers (also known as “callbacks”)
- When you interact with the element and trigger the event, the callback function will execute
- click is just one of many event types we'll use

Example: `click` event handling

```
<img id="mystery-box" src="question-block.png" alt="A Mystery Box">
<button id="box-btn">Click me!</button>
```

```
let boxBtn = document.getElementById("box-btn");
boxBtn.addEventListener("click", openBox);

function openBox() {
  // 1. Get the box image
  let box = document.getElementById("mystery-box");
  // 2. Change the box image's src attribute
  box.src = "star.png";
}
```

Common Types of JavaScript Events

Name	Description
load	Webpage has finished loading the document
scroll	User has scrolled up or down the page
click	A pointing device (e.g. mouse) has been pressed and released on an element
dblclick	A pointing device button is clicked twice on the same element
keydown	Any key is pressed
keyup	Any key is released

You can find an exhaustive list here

Event handler syntax

What's the difference between these two?

```
addEventListener("click", openBox);
```

```
addEventListener("click", openBox());
```

Event handler syntax

What's the difference between these two?

```
addEventListener("click", openBox);
```

```
addEventListener("click", openBox());
```

Answer: `openBox()` will execute right away (not waiting for the event to fire)