

CMSC335

Web Application Development with JavaScript



JS Objects, Fetch

Department of Computer Science

University of MD, College Park

Slides material developed by Ilchul Yoon, Nelson Padua-Perez

Objects

- **Object** - Collection of properties
- **Property** - association between a name and a value
 - A property can be seen as a variable associated with a value (**dot-syntax**)
 - » **obj.propertyName = "Mary";**
 - A property can be accessed using square brackets (**key-value syntax**)
 - » **obj["propertyName"] = "Mary";**
 - When the value is a function, the property is referred to as a **method**
 - The value in the **key-value** syntax approach can be any valid JavaScript string or anything that can be converted to a String (that includes an empty string)
 - » Any invalid property name can only be accessed using square bracket notation
 - A property that does not exist has a value of undefined

How to Create Objects

- **Using Object constructor (e.g., new Object())**
 - Object constructor creates an object wrapper for the given value
 - » **Example:** let x = new Object(true);
 - If the provided value is **null** or **undefined** an empty object will be created
- **Using object initializer/literal notation**
 - An initializer is a list of zero or more property names/values in { }
 - **Example:** let x = {};, y = { radius: 20 };
- **Using Object.create()**
 - Creates a new object, using an existing object as the prototype of the newly created object
- **Example:** Objects.html
- Using the [] operator can provide an excellent alternative to add properties to an object dynamically (when the program is executing)
- **Example:** AddingProperties.html

Destructuring Assignment

- **Destructuring**
 - A destructuring assignment allows us to unpack values from arrays, or properties from objects, into distinct variables
- **Example:** Destructuring.html

JSON

- JSON - JavaScriptObjectNotation
- Text data format used to store and send/receive data
- **Example:**

```
{"firstName":"Mary", "lastName":"Smith", "age" : 30}
```
- Popular format used by APIs to return results
- JSON syntax is derived from JavaScript, but code for generating and reading JSON can be done in any language
- JSON objects are written using { }
- JSON data is written as name/value pairs where the name must be in quotes (that is not the case for JavaScript objects). The value can be a string, number, boolean, array, object, etc.
- Arrays are written using square brackets ([])
- **Reference:** https://www.w3schools.com/whatis/whatis_json.asp
- **Example:** JSONExample.html
- See JSON resources (e.g., formatters) at
 - <https://www.cs.umd.edu/~nelson/classes/resources/web/>

Promises

- **Promise** - an object that represents the eventual completion (or failure) of an **asynchronous operation**
 - We attach callbacks to the promise object
 - Allows **promise chaining**
 - » Execution of two or more asynchronous operations back to back where results of one step are used by the next
- First, we will explore how to use promises by using the Fetch API
- Later we will see how we can define our own promises
- Reference
 - https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Using_promises

Fetch API

- Provides an interface for fetching resources (including across the network)
 - Takes one argument: the path to the resource
 - **Returns a promise** that resolves the response to that request, whether it is successful or not
- By default (by just providing a URL), we are generating a GET request
 - A second **options** parameter allows you to issue a POST request
- URL has to be an absolute URL
- Response object has methods/information such as:
 - **json()** - parses the body of the response into a JSON object and generates an error if the parsing fails
 - **text()** - returns the body of the response as text
 - **status** and **statusText** – information about HTTP status code
 - **ok** - true if the status is a 2xx status code
 - **Headers**
 - Reference:
 - <https://stackabuse.com/making-http-requests-in-node-js-with-node-fetch/>

Fetch API

- Example where we display json

```
fetch(url)
```

```
  .then(response => response.json())
```

```
  .then(json => console.log(json));
```

- **Example:** PromisesFetch*.html

async/await

- Standardized in ES8
- `async` and `await` **sequentializes** asynchronous code
- Makes the use of **Promises** more comfortable (easy to write and read) – (e.g., can avoid using `.then()` chain)
- **`async`** - put before a function
 - Means that a function returns a promise
 - Returned values are wrapped in a resolved promise
- **`await`** - put before using a returned promise
 - Makes JavaScript wait until that promise is resolved or rejected
 - Can't use `await` in regular functions (Syntax error)
 - » `SyntaxError: await` is only valid in **`async`** function
 - We use `await` to retrieve the result associated with a promise
- Example: `asyncAwait.html`

Additional Fetch Examples

- **Example:** FetchingImage.html
- **Example:** DisplayingCats.html

Default Parameters

- Standardized in **ES6**
- Allow named parameters to be initialized with default values
 - Literal or computed by a function
- **Example:** DefaultParameters.html

Rest Operator (...variable)

- Rest Operator
 - (...variable) appears at the end of the parameters list
 - It will receive all remaining arguments
 - Stores the remaining arguments as an array
- **Example:** RestOperator.html

Spread Operator

- Spreads (extracts) elements from an iterable (e.g., array, string) generating a sequence
 - Uses triple dot (exactly like rest operator)
 - let p = ["C", "Java"]
 - » ...p corresponds to "C", "Java"
 - Can be used inside array literals
- **Example:** SpreadOperator.html
 - The value **arguments** in a function provides information about arguments