




Welcome to the CoGrammar Lecture: Objects

The session will start shortly...

Questions? Drop them in the chat. We'll have dedicated moderators answering questions.



Full Stack Web Development Session Housekeeping

- The use of disrespectful language is prohibited in the questions, this is a supportive, learning environment for all - please engage accordingly.
(Fundamental British Values: Mutual Respect and Tolerance)
- No question is daft or silly - **ask them!**
- There are **Q&A sessions** midway and at the end of the session, should you wish to ask any follow-up questions. Moderators are going to be answering questions as the session progresses as well.
- If you have any questions outside of this lecture, or that are not answered during this lecture, please do submit these for upcoming Academic Sessions. You can submit these questions here: [Questions](#)

Full Stack Web Development Session Housekeeping cont.

- For all **non-academic questions**, please submit a query:
www.hyperiondev.com/support
- Report a **safeguarding** incident:
www.hyperiondev.com/safeguardreporting
- We would love your **feedback** on lectures: [Feedback on Lectures](#)

Skills Bootcamp

8-Week Progression Overview

Fulfil 4 Criteria to Graduation

✓ Criterion 1: Initial Requirements

Timeframe: First 2 Weeks

Guided Learning Hours (GLH):

Minimum of 15 hours

Task Completion: First four tasks

Due Date: 24 March 2024

✓ Criterion 2: Mid-Course Progress

60 Guided Learning Hours

Data Science - **13 tasks**

Software Engineering - **13 tasks**

Web Development - **13 tasks**

Due Date: 28 April 2024

Skills Bootcamp Progression Overview

✓ Criterion 3: Course Progress

Completion: All mandatory tasks,
including Build Your Brand and
resubmissions by study period end
Interview Invitation: Within 4 weeks
post-course
Guided Learning Hours: Minimum of
112 hours by support end date
(10.5 hours average, each week)

✓ Criterion 4: Demonstrating Employability

Final Job or Apprenticeship
Outcome: Document within 12
weeks post-graduation
Relevance: Progression to
employment or related
opportunity

**SKILLS
FOR LIFE**

SKILLS BOOTCAMPS



Department
for Education

CoGrammar Objects

March 2024

Lecture Overview

- Objects
- Prototypes
- Classes
- Constructor Functions
- JSON



What are Objects in JavaScript?

- ❖ **Objects** in JavaScript are fundamental data structures consisting of **key-value** pairs.
- ❖ **Keys** are strings (or symbols), and **values** can be any data type, including other objects.
- ❖ Objects provide a powerful way to represent complex **data structures** and **entities** in JavaScript.

```
// Creating an object
let person = {
  name: "John Doe",
  age: 30,
  city: "New York"
};
```


Prototypes and Prototype Chains

- ❖ **Prototypes** are the mechanism through which JavaScript implements **inheritance**.
- ❖ Each **object** in JavaScript has a **prototype**, which it inherits **properties** and **methods** from.
- ❖ Objects can delegate **property** and **method** lookup to their prototype, forming a **prototype chain**.

```
// Creating objects and setting prototypes
let parent = {
  greet: function() {
    return "Hello";
  }
};

let child = Object.create(parent);
console.log(child.greet()); // Output: "Hello"
```

Classes in JavaScript

- ❖ **ES6** introduced **class** syntax to JavaScript, offering a more familiar and structured way to create objects and manage inheritance.
- ❖ Classes provide syntactic sugar over prototype-based inheritance, making object-oriented programming in JavaScript more intuitive.

```
class Person {  
  constructor(name, age) {  
    this.name = name;  
    this.age = age;  
  }  
  greet() {  
    return `Hello, my name is ${this.name} and I'm ${this.age} years old.`;  
  }  
}  
  
// Creating an instance of the class  
let person1 = new Person("Alice", 25);  
console.log(person1.greet());
```

Constructor Functions

- ❖ **Constructor functions** are traditional JavaScript functions used for **creating** objects before the introduction of classes.
- ❖ They are invoked using the **new** keyword and initialize object properties using **this**.

```
// Constructor function
function Car(make, model) {
  this.make = make;
  this.model = model;
}

// Creating an instance using the constructor function
let car1 = new Car("Toyota", "Camry");
console.log(car1.make);
```

Object Serialization with JSON

- ❖ **JSON** is a lightweight data interchange format inspired by JavaScript object literal syntax.
- ❖ It's commonly used for **serializing** and **transmitting** structured **data** over a **network** connection.
- ❖ JSON is **language-independent**, making it easy to work with in various programming languages.

```
// Converting JavaScript object to JSON
let person = {
  name: "John Doe",
  age: 30,
  city: "New York"
};

let jsonStr = JSON.stringify(person);
console.log(jsonStr);
```

Parsing JSON

- ❖ JSON can be parsed back into JavaScript objects using the **JSON.parse()** method.
- ❖ This allows us to work with **JSON** data received from external sources.

```
// Parsing JSON back to JavaScript object
let jsonStr = '{"name":"Jane Doe","age":25,"city":"Los Angeles"}';
let person = JSON.parse(jsonStr);
console.log(person.name);
```

Working with Object Methods

- ❖ Objects in JavaScript can have **methods**, which are functions associated with the **object**.
- ❖ These methods can **access** and manipulate the object's properties.

```
// Object methods
let person = {
  name: "John Doe",
  greet: function() {
    return `Hello, my name is ${this.name}.`;
  }
};
```

```
console.log(person.greet()); // Output: "Hello, my name is John Doe."
```


Let's Breathe!

Let's take a small break
before moving on to
the next topic.



Understanding 'this' in JavaScript

- ❖ In JavaScript, the **this** keyword refers to the current execution **context**.
- ❖ Its **value** is determined by how a function is called.

```
let person = {  
  name: "John Doe",  
  greet: function() {  
    return `Hello, my name is ${this.name}.`;  
  }  
};  
  
let anotherGreet = person.greet;  
console.log(anotherGreet()); // Output: "Hello, my name is undefined."
```

Deep Dive into Prototypal Inheritance

- ❖ Prototypal inheritance is a core concept in JavaScript, enabling **objects** to inherit **properties** and **methods** from other **objects**.

```
// Prototypal inheritance
let parent = {
  greet: function() {
    return "Hello";
  }
};

let child = Object.create(parent);
console.log(child.greet()); // Output: "Hello"
```

Exploring Object Property Descriptors

- ❖ In JavaScript, each object property has associated property **descriptors**, which define its **behavior**.
- ❖ Property descriptors can **control** whether a property is writable, enumerable, and configurable.

```
let obj = {  
  name: "John"  
};  
  
let descriptor = Object.getOwnPropertyDescriptor(obj, "name");  
console.log(descriptor); // Output: {value: "John", writable: true}
```

Accessor Properties in JavaScript Objects

- ❖ Accessor properties in JavaScript objects are defined using **getters** and **setters**.
- ❖ They allow for **controlled access** and **manipulation** of object **properties**.

```
let obj = {  
  _name: "John",  
  get name() {  
    return this._name.toUpperCase();  
  },  
  set name(value) {  
    this._name = value;  
  }  
};
```

```
obj.name = "Alice";  
console.log(obj.name); // Output: "ALICE"
```


Iterating Over Object Properties

- ❖ JavaScript provides **various** methods for **iterating** over object **properties**, including **for...in loop** and **Object.keys()**, **Object.values()**, and **Object.entries()** methods.

```
let obj = {  
  name: "John",  
  age: 30,  
  city: "New York"  
};  
  
// Using for...in loop  
for (let key in obj) {  
  console.log(`${key}: ${obj[key]}`);  
}  
  
// Using Object.keys()  
let keys = Object.keys(obj);  
console.log(keys); // Output: ["name", "age", "city"]
```


Combining Objects with Spread Syntax

- ❖ **Spread** syntax in JavaScript allows an **iterable** to be **expanded** in places where zero or more **arguments** are expected.
- ❖ It's commonly used for **combining** objects or arrays.

```
let obj1 = { name: "John" };  
let obj2 = { age: 30 };  
let combinedObj = { ...obj1, ...obj2 };  
  
console.log(combinedObj); // Output: { name: "John", age: 30 }
```

Object Destructuring in JavaScript

- ❖ Object **destructuring** is a convenient way to **extract** multiple properties from an object and assign them to **variables**.

```
let person = {  
  nickname: "John",  
  age: 30,  
  city: "New York"  
};  
  
let { nickname, age } = person;  
console.log(nickname); // Output: "John"  
console.log(age); // Output: 30
```

Cloning Objects in JavaScript

- ❖ **Cloning** objects in JavaScript can be done using various techniques, including **spread syntax**, **Object.assign()**, and **JSON.parse()** and **JSON.stringify()**.

```
// Cloning objects
let obj = { name: "John", age: 30 };

// Using spread syntax
let clone1 = { ...obj };

// Using Object.assign()
let clone2 = Object.assign({}, obj);

// Using JSON
let clone3 = JSON.parse(JSON.stringify(obj));

console.log(clone1); // Output: { name: "John", age: 30 }
```

Preventing Object Modification

- ❖ JavaScript provides methods to **prevent** modification of object **properties**, like **Object.freeze()**.

```
// Preventing object modification
let obj = { name: "John" };

Object.freeze(obj); // Prevents any changes to the object
obj.name = "Alice"; // Change will not take effect
console.log(obj.name); // Output: "John"
```


Questions and Answers



Thank you for attending



Department
for Education

CoGrammar

