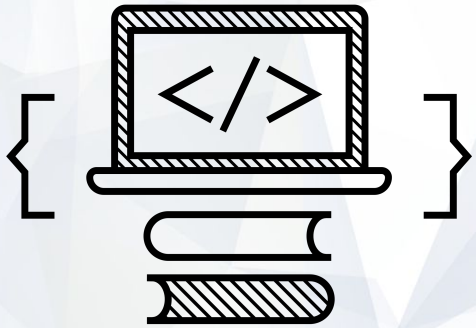
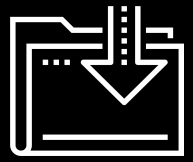


ES6 Classes

Skills Bootcamp in Front-End Web Development

Lesson 12.3




WELCOME

Learning Objectives

By the end of class, you will be able to:

Implement ES6 class syntax to instantiate multiple instances of a single type of object.

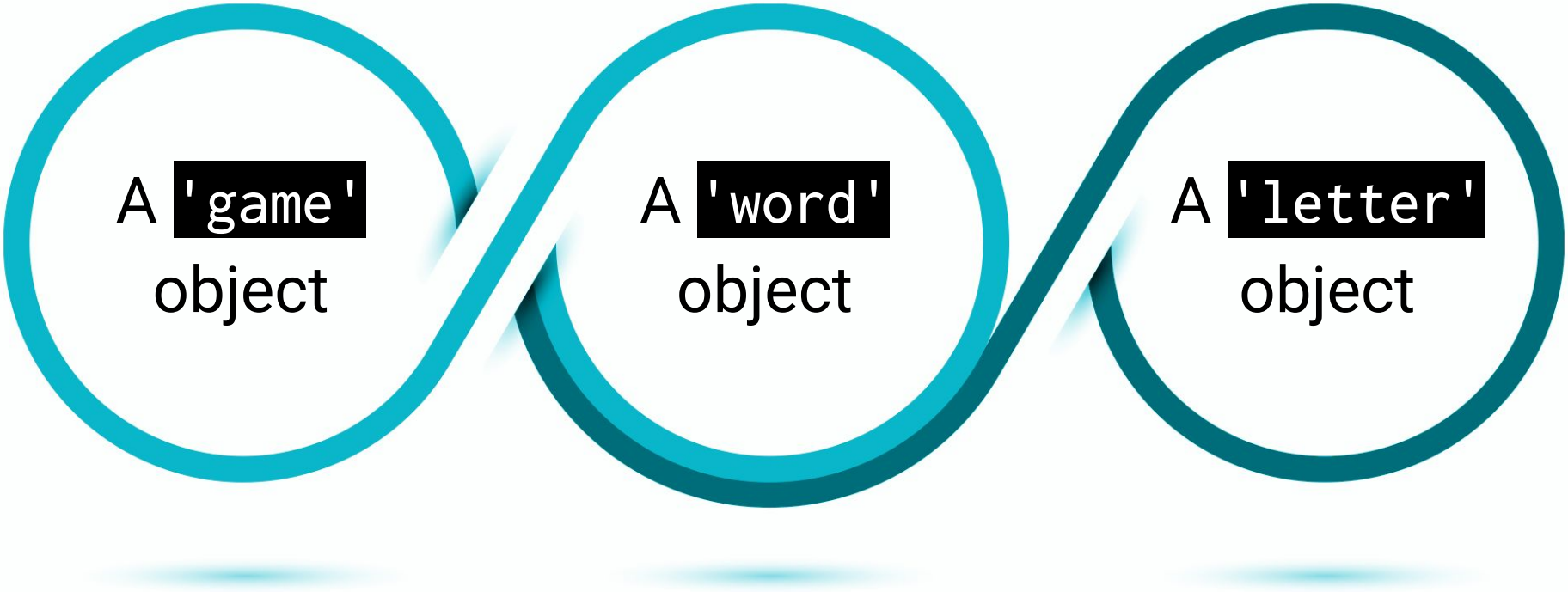
Construct subclasses that inherit features from a common ancestor class.

Demonstrate “thinking in OOP” by using objects to control the flow of action in an application.

Word Guess Game

From an Object Oriented perspective, what are three objects we could create to represent this application?

Word Guess Game



What kinds of methods could we have inside the `game` object?

Word Guess Game

Operational things like...

- The initialization of the game
- Prompting the user for letters
- Ending the game, etc.

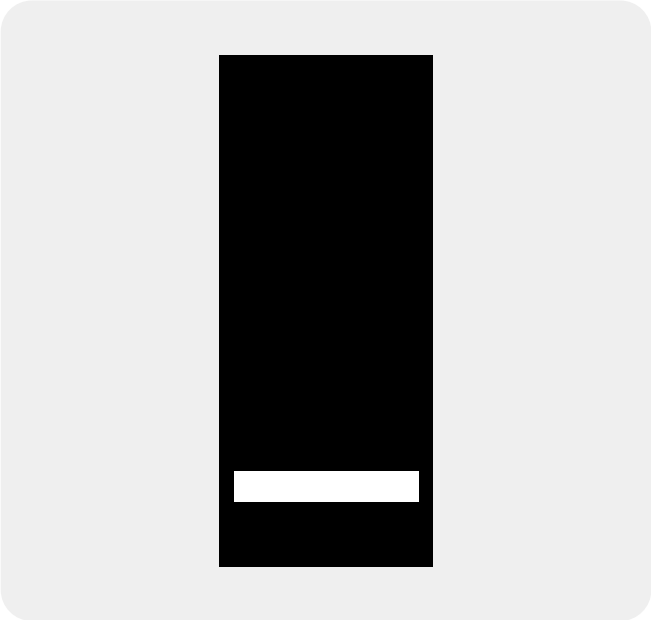
What kinds of methods could we have inside the `word` object?

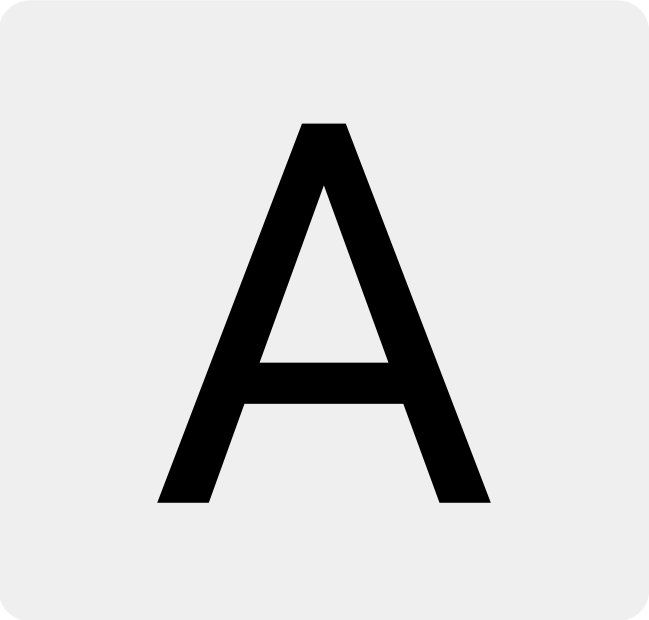
A function that checks to see if the word matches the solution.

What kinds of methods could we have inside the `letter` object?

Word Guess Game

A function that determines whether to display each letter as a _ or a letter.





Word Guess Game

The purpose of pseudocoding here is to practice organizing our code in a way that models real life objects.

Instructor Demonstration

ES6 Classes

What is the value of `this` within the `printInfo` method?

`this` is equal to the instance of the object that is created with the `new` keyword.

Activity: ES6 Classes

In this activity, you will use classes to make two RPG characters match against each other.

Suggested Time:

Time's Up! Let's Review.

Review: ES6 Classes

- Our Character class takes 3 arguments.
- As part of our bonus, we check to make sure the 3 arguments are provided.
- Here, we check each argument individually.
- If we wanted to, we could also validate the **type** of data being passed in to `Character`.

```
class Character {
  constructor(name, strength, hitPoints) {
    // Bonus
    if (!name) {
      throw new Error("You are missing the name.");
    }
    if (!strength) {
      throw new Error("You are missing the strength.");
    }
    if (!hitPoints) {
      throw new Error("You are missing the hitPoints.");
    }
    this.name = name;
    this.strength = strength;
    this.hitPoints = hitPoints;
  }
```

Review: ES6 Classes

`isAlive` checks if a character has 0 or less hitPoints.

```
isAlive() {
  if (this.hitPoints <= 0) {
    console.log(`${this.name} has been defeated!`);
    return false;
  }
  return true;
}
```

`Attack` deals damage to the provided opponent equal to the characters strength.

```
attack(opponent) {
  console.log(`${this.name} hit ${opponent.name} for ${this.strength}`);
  opponent.hitPoints -= this.strength;
}
```

Review: ES6 Classes

After we create two characters, we initialize the game to start with Grace.

```
const grace = new Character("Grace", 30, 75);
const dijkstra = new Character("Dijkstra", 20, 105);

let graceTurn = true;
```

After creating an interval, we toggle the turn.

Next, we check to see if either Grace or Dijkstra has been defeated.

If so, we clear the interval and end the game.

If not, we have Dijkstra or Grace attack, depending on whose turn it is.

Review: ES6 Classes

```
const turnInterval = setInterval(() => {
  graceTurn = !graceTurn;

  if(!grace.isAlive() || !dijkstra.isAlive()) {
    clearInterval(turnInterval);
    console.log("Game over!");
  } else if(graceTurn) {
    grace.attack(dijkstra);
    dijkstra.printStats();
  } else {
    dijkstra.attack(grace);
    grace.printStats();
  }
}, 2000);
```

Instructor Demonstration

Class Inheritance

Activity: Class Inheritance

In this activity, you will extend basic vehicle classes with additional functionality.

Suggested Time:

Time's Up! Let's Review.

Review: Class Inheritance

The Vehicle class is a good spot for us to keep a lot of the information that is shared between different types of vehicles.

```
class Vehicle {
  constructor(id, numberOfWheels, sound) {
    this.id = id;
    this.numberOfWheels = numberOfWheels;
    this.sound = sound;
  }

  printInfo() {
    console.log(`vehicle has ${this.numberOfWheels} wheels`);
    console.log(`vehicle has an id of ${this.id}`);
  }
}
module.exports = Vehicle;
```

Review: Class Inheritance

- We `require` the Vehicle so that we can use the `extends` keyword.
- `super()` is called so that **every** Vehicle that is a Car has 4 wheels and makes the `beep` sound.
- Even though we did not define `this.sound` in our Car class, we can still access it since our Car extends Vehicle and we called `super` as a method.

```
const Vehicle = require("./vehicle");

class Car extends Vehicle {
  constructor(id, color, passengers) {
    super(id, 4, "beep");
    this.color = color;
    this.passengers = passengers;
  }

  useHorn() {
    console.log(this.sound)
  }
```

Review: Class Inheritance

`checkPassengerLength` checks to make sure there are 4 or fewer passengers.

If not we can use a template literal to perform subtraction and get the number of seats remaining.

```
    checkPassengerLength() {
      if(this.passengers.length > 4) {
        console.log("Cars only seat 4 people. You have too many passengers!")
      }
      else {
        console.log(`You have room for ${4-this.passengers.length} people.`)
      }
    }
}
```

Review: Class Inheritance

- Just like our Car class, we need to require Vehicle so that we can extend the Boat class.
- This time we're automatically setting boat-type vehicles to have 0 wheels and make the "bwom" sound.
- We use `crewSoundOff` to perform a boat-specfic behaviour.

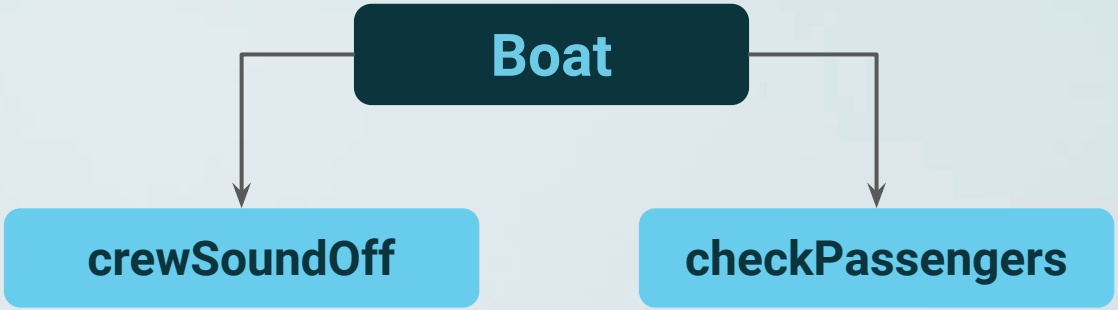
```
const Vehicle = require("./vehicle");

class Boat extends Vehicle {
  constructor(id, type, crew) {

    super(id, 0, "bwom");
    this.type = type;
    this.crew = crew;
  }

  useHorn() {
    console.log(this.sound);
  }

  crewSoundOff() {
    this.crew.forEach((member) => {
      console.log(`${member.name} reporting for duty!`)
    })
  }
}
```

Why did we create our two passenger methods, `crewSoundOff` and `checkPassengers`, in their own classes instead of handling it in the Vehicle class?

Review: Class Inheritance

It all comes down to organization.

Even though we could choose to handle it in Vehicle, that would not take advantage of our use of subclasses. Since each method is specific to the type of subclass, it makes sense, organizationally, to keep that logic in the subclass.



Instructor Demonstration

Multiple Classes

Why did we only call `prepareOrders` once?

Review: Multiple Classes

`prepareOrders` continues to prepare orders until `restaurant.orders.length === 0`.

Why do we have all of our initialization and method execution in the `restaurant.js` file and not inside `item.js` or `order.js`?

Review: Multiple Classes

Separation of concerns.

Our code is easier to navigate if we give each class a clear responsibility.

`Restaurant` is in charge of all operational things within the restaurant, whereas `Order` and `Item` are lightweight constructors.

```
const restaurant = new Restaurant("McJared's");

const items = [
  new Item("Burger", 5.99),
  new Item("Soda", 3.5),
  new Item("Chips", 2.0)
];

const orders = items.map(item => new Order(item));

orders.forEach(order =>
restaurant.takeOrder(order));

restaurant.prepareOrders();
```

Activity: Multiple Classes

In this activity, you will create a store class that allows you to handle different interactions within the store. You will use multiple classes with differing purposes to practice the OOP paradigm.

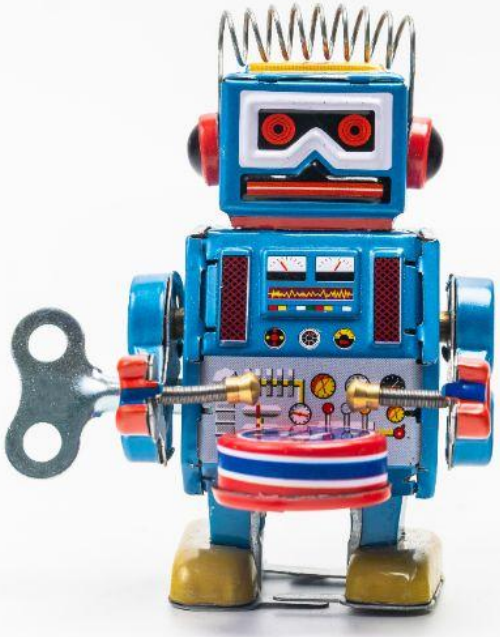
Suggested Time:

Time's Up! Let's Review.

Review: Multiple Classes

We purposely purchase enough Rare Toys to make sure that we get a message upon running out of stock.

```
store.welcome()
store.processProductSale("Action Figure");
store.processProductSale("Action Figure");
store.processProductSale("Rare Toy");
store.processprocessProductSalesProductSale("Action Figure");

store.processProductSale("Rare Toy");

store.replenishStock("Rare Toy", 2);
store.processProductSale("Rare Toy");

store.printRevenue();
```



Review: Multiple Classes

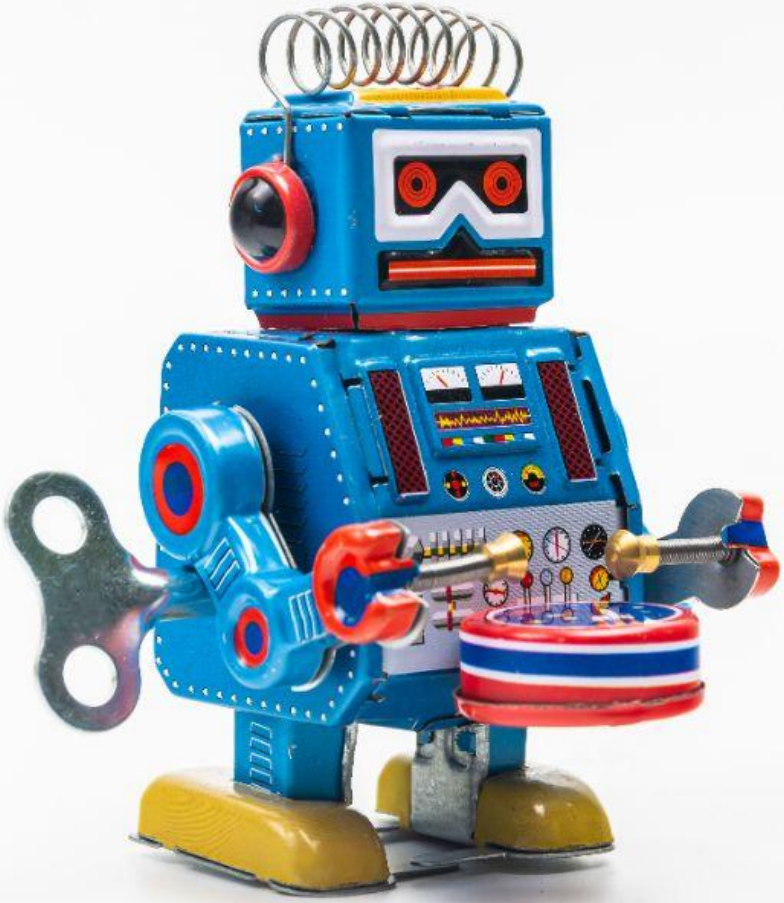
We use the name and stock arguments passed in through `new Store` and set the revenue to start at 0.

In `processProductSale`, we loop through each stock item in our Store. Once we've found one with a name that matches the name of the product we want to process, we decrease its count by one and increase the store's revenue by the price of the item.

In `replenishStock` we find the matching item by name and increase its count by the specified number.

```
class Store {
  constructor(name, stock) {
    this.name = name;
    this.stock = stock;
    this.revenue = 0;
  }

  processProductSale(name) {
    this.stock.forEach((item) => {
      if(item.name === name) {
        if(item.count > 0) {
          item.count--;
          this.revenue += item.price;
          console.log(`Purchased ${item.name} for ${item.price + item.calculateTax()}`);
        } else {
          console.log(`Sorry, ${item.name} is out of stock!`);
        }
      }
    })
  }

  replenishStock(name, count) {
    this.stock.forEach((item) => {
      if(item.name === name) {
        item.count += count;
        console.log(`Replenished ${item.name} by ${item.count}`)
      }
    })
  }
}
```

Why did we write our store initialization and product processing in a separate file called `index.js` instead of including it inside `store.js`?



We break away from `store.js` in this activity so that we can run isolated unit tests.

If we were to include all of our method calls in `store.js`, then trying to test individual methods in `store.js` would also cause all of the method calls to run.

Mini Project

Mini Project: Word Guess Game

In order to properly demonstrate a won game, it is recommended that you use `word.js` to guide your guesses.

We get different responses depending on whether or not our guess was successful.

If we guess correctly or run out of guesses, the game is over.

Lastly, we can choose to play another game or quit.

Activity: Mini Project

In this activity, you will create a Word Guess command-line game using OOP.

Suggested Time:

Time's Up! Let's Review.

Review: Mini Game

The Letter Class is responsible for displaying either an underscore or the underlying character for each letter in the word.

In the constructor, we determine if a character is not a number or a letter, and if not, make it visible right away.

```
class Letter {
  constructor(char) {

    this.visible = !/[a-z1-9]/i.test(char);
    this.char = char;
  }
//… continued on the next slide
```

Review: Mini Game

If the character should not be visible, return an underscore.

```
    toString() {
      if (this.visible === true) {
        return this.char;
      }
      return "_";
    }

    getSolution() {
      return this.char;
    }
  //… continued on the next slide
```

Review: Mini Game

We transform the character and the guess to uppercase so that the guess is case insensitive.

```
    guess(charGuess) {
      if (charGuess.toUpperCase() === this.char.toUpperCase()) {
        this.visible = true;
        return true;
      }

      return false;
    }
  }

  module.exports = Letter;
```

Review: Mini Game

In the constructor, we create a new `Letter` object for each character in the word string. `getSolution` returns a string of all of the solved letters.

```
class Word {
  constructor(word) {
    this.letters = word.split("").map(function(char) {
      return new Letter(char);
    });
  }

  getSolution() {
    return this.letters
      .map(function(letter) {
        return letter.getSolution();
      })
      .join("");
  }

  toString() {
    return this.letters.join(" ");
  }
```

Review: Mini Game

`guessLetter` checks to see if the user guessed correctly, then prints the word guessed so far.

```
guessLetter(char) {
  let foundLetter = false;
  this.letters.forEach(function(letter) {
    if (letter.guess(char)) {
      foundLetter = true;
    }
  });

  console.log("\n" + this + "\n");
  return foundLetter;
}
```

Review: Mini Game

`guessedCorrectly` uses `.every` to only return true if `letter.visible` is true for every letter.

```
    guessedCorrectly() {
      return this.letters.every(function(letter) {
        return letter.visible;
      });
    }
  }

  module.exports = Word;
```


The End