INTRO TO OBJECTS

So far, we've only dealt with relatively simple **data types**. For example, we've stored data in **arrays**.

```
let pets = ["cat", "dog", "hamster"]
```

We're going to see that in their simplest form, **objects** are also used to store **data**.

As our code gets more and more complex, objects will provide structure.

In the most basic sense, **objects** are collections of **key-value pairs**. For example, the key 'name' can be paired with the value 'Justin'.

We refer to each **key-value pair** as a **property**. The value can be any **JavaScript value**, e.g. a **function**, **object**, **array**, **boolean**, etc.

Note: This is very different from most other programming languages. This collection would be called a hash or dictionary in other languages. In JavaScript we can be more specific and call it an **object literal**.

With **arrays**, each **element** was **mapped** to an **index**, an **integer**.

With **arrays**, we accessed our **values** by **index**. With **objects**, we access our **values** by **key**.

We'll see with **objects** that our **keys** are like **indexes**, but we have more leeway with them. **Keys** can be any **string**.

We'll use **objects** in two different ways:

- structured data store of **key-value pairs**
- as actual **objects** in **object-oriented programming**, a powerful paradigm we'll use to structure and categorize our code

Creating objects, the following are empty objects:

```
// literal notation
let myArt = {};

//constructor notation
let myArt = new Object();
```

Most of the time we'll use **literal notation**, just like **arrays**:

```
let myArray = [];

let myArray = new Array();
```

Creating **objects** with data: the following **object literal** has multiple **properites** with different **values**.

```
let myMotorcycle = {
        wheels: 2,
        color: "blue",
        maxSpeed: 300,
        owners: ['John', 'Jane']
};
```

Keys 'wheels' and 'maxSpeed' have the values 2 and 300 respectively, which are **numbers**. Key 'color' has the value 'blue', a **string**. Key 'owners' has a value that's an **array**.

Notice how each key is separated by a colon : from the value, and each key-value pair is separated by a comma , .

```
let myMotorcycle = {
        wheels: 2,
        color: "blue",
        maxSpeed: 300,
        owners: ['John', 'Jane']
};
```

"Internally", each of our keys are strings. We *must* include the quotation marks around keys with spaces and other characters.

```
let anotherMotorcycle = {
        wheels: 2,
        "new owners" : ['Bill', 'Todd']
};
```

Setting properties: if our **key** is a valid **JavaScript variable** name, we can use *either* **dot notation** *or* **bracket notation**.

```
let myHouse = {};
myHouse.windows = 6;
myHouse['doors'] = 4;
// the new key is "windows" and the new value is 6
```

If our key is not a valid **JavaScript variable** name, e.g. it has dashes, spaces, we *must* use **bracket notation**:

```
let myCar = {};
myCar["num-of-doors"] = 4;
myCar["new owner"] = "Mike";
```

Overwriting values: we can overwrite the value of a **key** using the previous syntax. It does not have to be a new key.

```
let myHouse = {
        windows: 6
};
// Moving to a bigger house?
myHouse.windows = 20;
```

Getting **properties**: we can get the **value** of a **property** using either **dot notation** or **bracket notation**.

```
let myHouse = {
        windows: 6
}

myHouse["windows"]
// => 6;

myHouse.windows
// => 6;
```

Just like setting **properties**, we might have to use **bracket notation** depending on wether it's a **variable**, **string** with spaces, etc.

Properties using **variables**: just like with **arrays** where we could use a **variable** for our **index**, we can do the same with **objects** and **keys**. The key must **compute** to the type **string**.

We *must* use bracket notation in this case:

```
let myHouse = {
        windows: 6,
        "number of doors": 3
};

let doors = "number of doors";
myHouse[doors] = 6;
```

Here we can have a function that returns our key, or store our key in a variable.

Alright, practice time.

Using repl.it:

- Make a person **object** with *name* and *eyeColor* properties.
- Assign the person **object** another **property**, *favorite food* and give the property a value.
- Print out all three values. If possible, use both **dot and bracket syntax**.
- And which key can you note use dot syntax for?
- BONUS: Assign another **property** to your person **object**, *address*, that has an **object** with 4 properties: *streetNumber*, *streetName*, *city*, *zip*

```
let person = {
        name : "Justin",
        eyeColor : "Hazel"
}
person["favorite-food"] = "pizza";
console.log(person);

person.address = {
        streetNumber : "133",
        streetName : "West 21st",
        city : "New York",
        zip : "10011"
}
console.log(person);
```

METHODS

When the **property** of an **object** is a **function**, we call it a *method*.

We can use **methods** to get information about and manipulate the **properties** of the **object**.

Calling **methods**: whether we use **dot notation** or **bracket notation**, need to add parans () to the end of our statement to **invoke** the **method**.

Without the (), we only get a **reference** to the **function**.

Simple method:

```
let randomUtil = {
        getRandomArbitrary: function(min, max) {
                return Math.random() * (max - min) + min;
        },
        getRandom: function() {
                return Math.random();
        }
};

randomUtil.getRandomAritrary(1,3);
randomUtil.getRandom();
```

Function reference:

```
let randomUtil = {
        getRandomArbitrary: function(min, max) {
                return Math.random() * (max - min) + min;
        },
        getRandom: function() {
                return Math.random();
        }
};

// notice we're not invoking getRandomArbitrary
randomUtil.getRandomArbitrary;
=> function(min, max) {
        return Math.random() * (max - min) + min;
}
```

(what's returned depends on environment)

INTRO TO *THIS* KEYWORD.

We often use ***this*** keyword inside of a **method**. It's an extremely special keyword in **JavaScript**, and refers to the **object** that invoked the **function** it's in.

This is different than **jQuery's $(this)**

this keyword example:

```
let justin = {
        name: "Justin",
        sayHi: function() {
                // this.name refers to the name key
                // of the object invoking the sayHi method
                console.log("Hi, my name is " + this.name);
                // we can access the values
                // for a key using dot notation
        }
};
justin.sayHi();
// => "Hi, my name is Justin"
```

BRIEF INTRO TO OBJECT-ORIENTED PROGRAMMING:

Object-orientated programming (OOP) models data in our code to match how we think of the world.

The nouns describing an object are its properties, and the verbs are its methods.

For example, I am a *person* with a *name* and I can *say* my name.

OOP is often extremely powerful and useful.

However, it may appear convoluted and hard to understand when things get more abstract.

The other programming paradigm available in JavaScript is **functional programming**.

A brief description:
https://www.codenewbie.org/blogs/object-oriented-programming-vs-functional-programming

Practice with **methods**:

- Copy over the person **object** you previously made
- Make a *sayIntro* **method** that logs out your name and favorite food: "Hi, I'm [] and my favorite food is []"
- Make a *setEyeColor* **method** that takes a color as a parameter:
    - If the color is the same as the current eye color, log out that you failed to set it
    - If the color is different from the current eye color, set it

```
let person = {
 name : "Justin",
 eyeColor : "Hazel",
 "favorite-food" : "pizza",
 address: {
 streetNumber : "133",
 streetName : "West 21st",
 city : "New York",
 zip : "10011"
},
sayIntro: function() {
        ...
```

Note, the next slide is the second half of the code, I just can't fit it within the PDF continuously.

Note, the previous slide is the first half of the code, I just can't fit it within the PDF continuously.

```
...
sayIntro: function() {
 console.log(`Hi, I'm ${this.name}`);
 console.log(`My favorite food is ${this['favorite-food']}.`);
},
setEyeColor: function(color) {
 if(color === this.eyeColor) {
        console.log("You failed to set eyeColor");
 } else {
        this.eyeColor === color;
        console.log(`You set the eye color to ${color}.`);
 }
}
}
```

Test the combined code of the last two slides in repl.it.

```
person.sayIntro();
let eyeColor = "blue";
person.setEyeColor(eyeColor);
```

CONSTRUCTORS

We use a **constructor function** to create **objects** that all have the same structure, i.e. the same property names.

We call a **constructor** like a regular **JavaScript function** and use the **new** keyword.

More on this later, I assume we might be covering too much ground right now.

Empty **object** using a **constructor**.

Let's say we had an empty **construcor function**:

```
function Person() {
// by convention the first letter
// of a constructor is uppercase
}
```

We use the **new** keyword when calling the **constructor function**:

```
let clark = new Person();
let bruce = new Person();
clark; => {}
bruce; => {}
```

Both *clark* and *bruce* are **empty objects** because the **constructor** was empty.

Creating **arrays** and **objects** with **constructors**, maybe less mysterious:

```
let emptyObject = new Object();
emptyObject; // => {}
let emptyArray = new Array();
emptyArray; // => []
```

So far, we've set **properties** by hand every time we've created an **object**.

Constructors help us create a blueprint for our data. Every object created with a constructor has the same property names and methods.

We can use parameters in our constructor function to set the values of the properties.

```
function Superhero(firstName, superHeroName) {
        this.name = firstName;
        this.heroName = superHeroName;
}

let superman = new Superhero('Clark', 'Superman');
let batman = new Superhero('Bruce', 'Batman');
console.log(superman.name); // => 'Clark'
console.log(batman.heroName); // => 'Batman'
```

What does a **constructor function** do?

1. Creates a **new object**
2. Makes the **this** variable point to the new **object** while the **constructor** is executing
3. Automatically return **new object**

Some common methods for a constructor:

```
function Superhero(firstName, superHeroName) {
        this.name = firstName;
        this.heroName = superHeroName;
}

Superhero.prototype.revealIdentity = function() {
        console.log(this.name + ' is ' + this.heroName);
}

let superman = new Superhero('Clark', 'Superman');
let batman = new Superhero('Bruce', 'Batman');
superman.revealIdentity(); // => 'Clark is Superman'
batman.revealIdentity(); // => 'Bruce is Batman'
```

We used the *Superhero* **constructor** function as before.

We added a **method** to *Superhero*'s **prototype**, causing all **objects** created with the *Superhero* **constructor** to have access to that **method**.

When the *revealIdentity* **method** is called, the ***this*** variable in the method refers to whatever **object** it's called on.

This is why in the past we'd see things like **<>Array.prototype.forEach()** on MDN.

Anticipating questions: the prototype object from the previous code example (which you may have seen before) is an **object** from which other **objects** inherit **properties**.

Constructors

Prototypes

Practice with **constructors**

Work in repl.it. Using a **constructor function**, allow for the creation of a bear object that has the following properties and methods:

- *name*: property, string
- *species*: property, string
- *foodsEaten*: property, array
- *eatSomething*: method, takes a thing to eat (as a string) and adds it to *foodsEaten*
- *introduce*: method, log out its name, species, and the last item it ate

Create three bears with **arguments** passed in to your **constructor function**. You'll need to use the **prototype** syntax to give each bear access to *eatSomething* and *introduce*.

```
function Bear(name, species) {
  this.name = name;
  this.species = species;
  foodsEaten = [];
}

Bear.prototype.eatSomething = function(foodToEat){
  foodsEaten.push(foodToEat);
  // console.log(foodsEaten);
};
```

```
Bear.prototype.introduce = function(){
  // console.log(foodsEaten);
  // console.log(foodsEaten.length-1);
  console.log(`Hi, my name is ${this.name} and I'm a ${this.sp
  console.log(`The last thing I ate was ${foodsEaten[foodsEate
};

let grizzley = new Bear("Smokey", "Grizzley");
grizzley.eatSomething("fish");
grizzley.introduce();
```

In class we'll use these concepts to make some sort of generative artwork.