5-5. Classes

But before we dig into classes, we should mention **prototypes** in JavaScript. *5-5-1*

Prototype is **an object** with properties and methods applied to it. *5-5-2*

All objects have a prototype linked to it.

Lets create an object **food**: *5-5-3*

```
const fruit = {
    init: (type) => this.type = type,
    eat: () => console.log(`You are eating the ${this.type}`)
};

fruit.init("banana");
fruit.eat(); // You are eating the banana

fruit.init("apple");
fruit.eat(); // You are eating the apple
```

We might want to create several fruits and not overwriting each created fruit with a new one. *5-5-4*

Lets use the `Object.create()`

```
const fruit = {
    init: (type) => this.type = type,
    eat: () => console.log(`You are eating the ${this.type}`)
};

let banana = Object.create(fruit);
banana.init("banana");
banana.eat()

let apple = Object.create(fruit);
apple.init("apple");
apple.eat()
```

When we do this, we *do not copy* the fruit object into a new. *5-5-5*

We create a new empty object that has fruit as its *prototype*.

```
const fruit = {
    init: (type) => this.type = type,
    eat: () => console.log(`You are eating the ${this.type}`)
};

let banana = Object.create(fruit);

fruit.isPrototypeOf(banana); // true
```

Banana and apple is built on its prototype, fruit

The banana and apple objects themselves doesn't have the `.init()` and `.eat()` methods. *5-5-6*

But their prototype does, so they inherit from their prototype.

When we call a method on an object, it first checks if *that* object have the called method. *5-5-7*

If it does not it goes back in the chain and look in its prototype.

Due to this behavior we could overwrite the `eat` method on banana for example *5-5-8*

```
const fruit = {
    init: (type) => this.type = type,
    eat: () => console.log(`You are eating the ${this.type}`)
};

let banana = Object.create(fruit);
banana.init("banana");
banana.eat = () => console.log(`BANANANANANA`)
banana.eat() // BANANANANANA

let apple = Object.create(fruit);
apple.init("apple");
apple.eat() // You are eating the apple
```

When calling `banana.eat()` it first look at object of its own: does the method exits? *5-5-9*

Yes, then run that functions.

In turn of the `apple` object, it will keep walking to its prototype since the method is not found on the object of its own.

The same with the prototypes we have not created by ourselves, `Object` and `Array` for example. *5-5-10*

In the previous example, `Object.prototype` would be the prototype of `fruit`.

```
Object.prototype.isPrototypeOf(fruit); // true
```

If you create an array for example *5-5-11*

```
let arr = [1,2,3,4,5,6,7];
```

You can access various methods on the array

```
arr.length;
arr.slice();
arr.find();
```

These are no methods we have created by ourselves, but comes from `Array.prototype`.

Note that it is very rare to use prototypal inheritance in code though. *5-5-12*

In the above `fruit` example, we could also have created a new fruit with the basic functions by doing `let banana = Object.assign({}, fruit);`

Class *5-5-13*

ES6 introduces JavaScripts classes, which are syntactical sugar over JavaScript's existing prototype-based inheritance that we've recently talked about.

A class is simply a reusable program-code-template for creating objects, whichs can hold properties and implementations of behavior, also known as methods.

We create classes using the `class` keyword. *5-5-14*

```
class Fruit {
    constructor (type) {
        this.type = type;
    }
    eat () {
        console.log(`You are eating ${this.type}`);
    }
}
```

The equivalent to this class with JavaScript in previous version of EcmaScript could look like this: *5-5-15*

```
function Fruit(type) {
    this.type = type;
};
Fruit.prototype.eat = function() {
    console.log(`You are eating the ${this.type}`)
};
```

We see how the ES6 features facilitates us to write simpler and clearer code.

`constructor` *5-5-16*

A class's constructor function is where you'll focus your initialization logic.

```
    constructor (type) {
        this.type = type;
    }
```

When creating a Fruit for example, we always want a type.

The **constructor** acts as the definition of the class, what parameters we defined into the constructor must be passed when creating the object.

You cannot call the constructor function directly *5-5-17*

```
const fruit = Fruit('Banana'); // TypeError
```

We need to instantiate our new object using the `new` keyword.

Instantiation

5-5-18

We instantiate classes using the `new` keyword to create an object with the given structure, properties and methods.

```
let banana = new Fruit('Banana');

console.log(banana); // Fruit {type: 'Banana'}
```

When you call the functions with `new`, four things happen under the hood:

5-5-19

- A new object gets created (`banana`)
- `banana` gets linked to another object (`Fruit`), called its prototype
- The this value is set to refer to `banana`
- It implicitly returns banana

It's between steps three and four that the engine executes your function's specific logic.

Since a class is an object, we can access the properties and mehtods of that object as usual.

5-5-20

```
let banana = new Fruit('Banana');

console.log(banana.type); // Banana
banana.eat(); // You are eating Banana
```

Subclasses

5-5-21

Classes created with `extends` are called subclasses.

Subclasses are classes that it built on top of another class, child of another class.

Lets say we have our fruit example

5-5-22

```
class Fruit {
    constructor (type) {
        this.type = type;
    }
    eat () {
        console.log(`You are eating ${this.type}`);
    }
}
```

We might want to have a more specific fruit category, `SweetFruits` maybe *5-5-23*

```
class SweetFruit extends Fruit {
    constructor (type) {
        super(type);
    }
    cook () {
        console.log(`You are cooking the ${this.type}`);
    }
}

let sweetBanana = new SweetFruit('very Sweet Banana');
sweetBanana.eat();
sweetBanana.cook();
```

The `SweetFruit` object will have the properties and methods both from its parent class as well as properties and methods of its own. *5-5-24*

super(params) *5-5-25*

The super keyword is used to access properties and methods on an instance's parent.

In the case of our example, the super method sends the passed params into the constructor of the parent class.

This makes it possible to set `this.type`.