**Object oriented programming in JavaScript.**

There are three main concepts in OOP: encapsulation, polymorphism, and inheritance.

**Encapsulation**

Concept: Inner workings are kept hidden inside object and only essential functionalities are exposed to end user such as ‘on’ button. Methods available to implement functionality without outside world needing to know how it's done.

**polymorphism**

Concept: the same process can be used for different objects. And share the same method for also have the ability to override shared methods with a more specific implementation

**inheritance**

concept: taking the features of one object then adding some new features. Improved functionality adding new properties and methods.

**Classes**

the class is used to define a blueprint for an object. Objects are then created as an instance of that class and in here all the properties and methods of the class. An Example: a juicer class would represent the design of a juicer and each juicer that's made on the production line would be instances of that class. JavaScript uses prototype based language in the background even though it uses classes starting with ES6.

**constructor functions**

We can use object literal notation to create new objects:

```
const dice = {
  sides: 6,
  roll() {
    return Math.floor(this.sides * Math.random() + 1)
  }
}
```

An alternative way to create objects is to use a **Constructor function**.

Here is a function that defines the properties and methods of an object.

```
const Dice = function(sides=6){
  this.sides = sides;
  this.roll = function() {
    return Math.floor(this.sides * Math.random() + 1)
  }
}
```

The keyword “this” is used to represent the object that will be returned by the constructor function.

The “sides” property to the argument is 6 if no argument is provided. And also adds a method called roll ().

* **now** we can create an instance of the dice constructor function using the “new” operator.

```
const redDice = new Dice();
<< Dice { sides: 6, roll: [Function] }
```

Example have two instances of a class:

```
const redDice = new Dice;
const whiteDice = new Dice(4);
```

whiteDice has an argument of 4 for the sides. redDice has the default of six sides. to check the sides, you can use

```
redDice.sides
//6
```

```
whiteDice.sides
//4
```

**ES6 class declarations**

New class declaration syntax:

```
class Dice {
   constructor(sides=6) {
      this.sides = sides;
   }

   roll() {
      return Math.floor(this.sides * Math.random() + 1)
   }
}
```

To create an instance of the “Dice” class the “new” operator is used again:

```
const blueDice = new Dice(20);
<< Dice { sides: 20 }
```

* We can create a copy of an object without referencing the actual constructor function or class declaration directly. Example: we can make a copy of the redDice Object:

```
const greenDice = new redDice.constructor(10);

greenDice instanceOf Dice
<< true
```

**static methods**

The “static closed quote keyword can be used in class declarations to create a static method. These can be called class methods in other programming languages. A static method is called at a class rather than by instances of the class. Example- The Dice class could have a method:

```
class Dice {
   constructor(sides=6) {
       this.sides = sides;
   }

   roll() {
       return Math.floor(this.sides * Math.random() + 1)
   }

   static description() {
       return 'A way of choosing random numbers'
   }
}
```

Call this static method with dot notation:

```
Dice.description()
```

*Static Methods not available to instances of the class. redDice and whiteDice cannot call static description() method.

**Prototypal Inheritance**

Every class has a prototype property that is shared by every instance of the class. Any properties or methods of a class's prototype can be accessed by every instantiated object of that class.

```
class Turtle {
    constructor(name) {
        this.name = name;
        this.weapon = 'hands';
    }
    sayHi() {
        return `Hi dude, my name is ${this.name}`;
    }
    attack(){
        return `Feel the power of my ${this.weapon}!`;
    }
}
```

This can then be used to create a new turtle instance:

Copy

```
const leo = new Turtle('Leonardo');
<< Turtle { name: 'Leonardo' }
```

The variableleopoints to an instance of theTurtleclass. It has anameproperty and asayHi()method that references thenameproperty:

Copy

```
leo.name;
<< 'Leonardo'

leo.sayHi();
<< 'Hi dude, my name is Leonardo'
```

**the prototype property**

Use this property to augment the class with extra methods and properties after it's been created. All classes in constructor functions have a prototype property that returns an object:

```
Turtle.prototype;<< Turtle {}
```

All instances of the “turtle” class sure all properties and methods of its prototype. This means that they can call any methods of the prototype and access any of its properties. Since the prototype is an object, we can add new properties by assignment:

```
Turtle.prototype.weapon = 'Hands';<< 'Hands'
```

**finding out the prototype**

there are a few ways to find the prototype of an object. We can use the objects prototype property:

```
raph.constructor.prototype;
```

You can also use:

```
Object.getPrototypeOf(raph);

// both return Turtle { attack: [Function}, weapon: ‘Hands’ }
```

**the prototype is live**

the prototype object is live, so if a new property or method is added to the prototype, any instances of its class will inherit the new properties and methods automatically, even if that instance has already been created.

**over writing prototype properties**

an object instance can overwrite any properties or methods inherited from its prototype by simply assigning a new value to them.

```
leo.weapon = ‘Katana Blades’;

raph.weapon =  ‘Sai’;
```

**what should a prototype be used for?**

the prototype can be used to add any new properties in methods after the class has been declared. It should be used to define any properties that will be the same for every instance of the class. For example, although turtles like pizza. Pizza could be set in the prototype. Weapons would not be a good idea. Each turtle has a different weapon.

**public and private methods**

the default of an object's methods are public in JavaScript. That means they can be queried directly and changed by assignment. Properties and methods can be changed after it's been created. To avoid confusion and disaster from changing of methods, we can use private methods. In the turtle class, you can modify it to include a private color property.

```
class Turtle {

  constructor(name,color) {

    this.name = name;

    let _color = color;

    this.setColor = color => { return _color = color; }
```

```
        this.getColor = () => _color;
    }
}

raph = new Turtle('Raphael','Red');
<< Turtle { name: 'Raphael', setColor: [Function], getColor: [Function] }

raph.getColor();
<< 'Red'

raph.setColor(4);
<< 4
```

**Private methods and properties can only be used inside their scope.

**inheritance**

Properties and methods can be inherited from the prototype, but the prototype is another object, so it has its own prototype, which causes a chain of inheritance.

**the object constructor**

Enumerable properties

You can have enumerable or non enumerable properties. If they are not enumerable, this means they will not show up when a for- in loop is used to loop through an object's properties and methods.

All properties in methods that are created by assignment are innumerable. Objects that inherit a tostring() method from Object.prototype are not enumerable, so it won't show up in any objects.

**inheritance using extends**

A class can inherit from another class using the extends keyword in a class declaration.

```
class Turtle {
  constructor(name) {
    this.name = name;
    }
  sayHi() {
    return `Hi dude, my name is ${this.name}`;
  }

  swim() {
    return `${this.name} paddles in the water`;
  }
}
class NinjaTurtle extends Turtle {
  constructor(name) {
    super(name);
    this.weapon = 'hands';
  }
  attack() { return `Feel the power of my ${this.weapon}!` }
}
```

When a class extends a class it is a subclass or child class. The keyword super refers to the parent class an can be used to access any properties and call any methods of the parent class.

**polymorphism**

Different objects can have the same method but implemented in different ways. Objects are able to override a method with a more specific implementation.

**Getters and Setters**

An object an odd property descriptor can have I get and set method instead of a value attribute objects have one or the other they can have both useful if a property relies on the value of another property

For example, if we addageandretirementAgeproperties to themeobject, we can then create ayearsToRetirementproperty that depends on these properties:

```
me.age = 21;

me.retirementAge = 65;


Object.defineProperty(me, 'yearsToRetirement',{

   get() {

       if(this.age > this.retirementAge) { return 0; }

       else { return this.retirementAge - this.age; }

   },

   set(value) {

       this.age = this.retirementAge - value;

       return value;

   }

});
```

The getter bases theyearsToRetirementproperty on theageandretirementAgeproperties, so returns the relevant value when queried:

Copy

```
me.yearsToRetirement

<< 44
```

The next example shows how we can create aDiceclass that uses agetfunction that will return a description of the number of sides, rather than just the actual number, and asetfunction that prohibits a non-positive number of sides to be set:

Copy

```
class Dice {

constructor(sides=6){
    Object.defineProperty(this, 'sides', {
        get() {
        return `This dice has ${sides} sides`;
        },
        set(value) {
        if(value > 0) {
            sides = value;
            return sides;
        } else {
            throw new Error('The number of sides must be positive');
        }
        }
    });

    this.roll = function() {
        return Math.floor(sides * Math.random() + 1)
    }
    }
}
```

The number of sides can now be assigned in the usual way, but it will act a little differently:

```
Copy
const yellowDice = new Dice;

yellowDice.sides
<< "This dice has 6 sides"

yellowDice.sides = 10;
<< 10
```

```
yellowDice.sides
<< "This dice has 10 sides"
```

```
yellowDice.sides = 0;
<< Error: "The number of sides must be positive"
```

**creating objects from other objects**

we can avoid using classes altogether and create new objects based on another object to act as a blueprint or prototype instead.

The Object() Constructor function has a method called create that can be used to create a new object that is a copy of the object that divided as an argument. The object that is provided for the argument is the prototype for the new object.

```
const Human = {
arms: 2,
legs: 2,
walk() { console.log('Walking'); }
}
const lois = Object.create(Human);
```

You can add extra properties to each instance using assignment.

```
lois.name = 'Lois Lane';
<< 'Lois Lane'
```

```
lois.job = 'Reporter';
<< 'Reporter'
```

**object based inheritance**

an object in can also act like a superclass. It can become the prototype of another object. It will have all the properties and methods the object has but with extra methods. New objects can be created an initialized

```
Superhuman.init = function(name,realName){

    this.name = name;

    this.realName = realName;

    this.init = undefined; // this line removes the init function, so it can only be called once

    return this;

}
```

Now a new object can easily be created and initialized:

Copy

```
const batman = Object.create(Superhuman);

batman.init('Batman','Bruce Wayne');


batman.change();

<< 'Bruce Wayne goes into a phone box and comes out as Batman!'


const aquaman = Object.create(Superhuman).init('Aquaman', 'Arthur Curry');


aquaman.change();

<< 'Arthur Curry goes into a phone box and comes out as Aquaman!'
```

**Mixins**

a mixing is a way of adding properties and methods of some objects to another object without using inheritance. Use Object.assign() method.

You can also use mixin() function to assign all properties of an object to another object as a deep copy instead of a shallow copy. You can use mixins to add properties.

```
const wonderWoman = Object.create(Superhuman);
```

Instead of assigning each property, one at a time:

Copy

```
wonderWoman.name = 'Wonder Woman';
<< 'Wonder Woman'

wonderWoman.realName = 'Diana Prince';
<< 'Diana Prince'
```

We can just mix in an object literal and add both properties at once:

Copy

```
mixin(wonderWoman,{ name: 'Wonder Woman', realName: 'Diana Prince' });

wonderWoman.change()
<< 'Diana Prince goes into a phone box and comes out as Wonder Woman'
```

**modular JavaScript**

a module is a self contained piece of code that provides functions and methods that can be used in other files and by other modules. Helps with code maintainability. The code in a module should have a single purpose an group functions with distinct functionality Modular code helps to make things interchangeable so you can swap one module for another without adding other parts of it All code in modules is in strict mode so don't need to use strict no way to app opt out of this. Module has own global scope variables created in a module can only be accessed within that module. The value of this level of a module is undefined rather than the global object. You can't use HTML I'll comment modules use keyboard to specify any values or functions that are to be made available from the module. Not everything in a module needs to be used.

Default exports

These refer to a single variable, function or class in a module that can be breakout happy explained. Don't use more than one default export it will cause a syntax error.

Aliases

you can use an alias to import a function

```
import sq from './square.js';
```

the function can then be called using sq() instead of square()

## Node.js Modules

these are pre-made modules that can be used in JavaScript