OOP

Encapsulation – only user functionalities are exposed. All programming logic is hidden inside of objects and their functionalities are available only through methods.

Polymorphism – different objects can share the same methods and override them for customization.

Inheritance – using the same features of an existing object and adding new features.

class – defines the blueprint for an object

prototype-based language – building prototype objects that can be reused or added to

Constructor Functions

A function that creates objects with properties and methods.

Object Constructor	Object literal

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

create an instance – The parenthesis are only necessary when an argument is used:

const redDice = new Dice(4); => Dice with 4 sides

const redDice = new Dice; => Dice with 6 sides

check if object is an instance of a class – Confirm instance evaluation:

redDice instanceof Dice => true;

Built-in Constructor Functions

	literal syntax	constructor Function
Object	const literalObject = {};	constructedObject = new Object();
Array	const literalArray = [1,2,3]; This method is better!	constructedArray = new Array(1,2,3);

Class Declarations

Class Declaration	Object Constructor

class Dice {
constructor(sides=6) { this.sides = sides; }
roll() {
return Math.floor(this.sides * Math.random() + 1) }
} This method is better!
	const Dice = function(sides=6){
	this.sides = sides;
	this.roll = function() {
	return Math.floor(this.sides * Math.random() + 1)} }

Cunstructor Property

All objects have the constructor property. It returns the constructor function that created it.

Object created using class declaration - blueDice.constructor=

[Function: Dice]

Object created using object literal - literalObject.constructor=

[Function: Object]

Use constructor property to create a copy of object – You don't need to reference the constructor function or class declaration.

Static Method

Not available to instances of the class:

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

Prototypal inheritance

Every class has a shared prototype property between instances.A prototype is just an object. If you add new properties or methods after a class was already made, they will be inherited by any instances created after.

Add new properties – by assignment

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

Add new methods – by assignment

```
Turtle.prototype.attack = function(){ return `Feel the power of my ${this.weapon}!`; }
```

How to find prototype of object –

Prototype property – returns an object

Turtle.prototype => Turtle { attack: [Function], weapon: 'Hands' }

Object.getPrototypeOf() method – takes object as parameter and returns object

Object.getPrototypeOf(raph) => Turtle { attack: [Function], weapon: 'Hands' }

__proto__ property – not part of official specs and not recommended; deprecated

raph.__proto__ => Turtle { attack: [Function], weapon: 'Hands' }

How to check if object is prototype of instance – Turtle.prototype.isPrototypeOf(raph) => true

Check if property is considered it's own – If an object has inherited prototype properties they aren't considered to be a class's own property:

raph.hasOwnProperty('name') => true

Prototypes are live – If an instance has already been created and a new property or method is added to a prototype, that instance will automatically inherit them.

Use object instance to overwrite Prototype property – Instance "own" properties take precedence over the same prototype properties.

Private Methods

Use getters and setters to access private variables that start with _ .

```
class Turtle {

constructor(name,color) {

 this.name = name;

let _color = color;

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

this.getColor = () => _color; }

}
```

Object Constructor

All objects inherit from Object() constructor

1. object calls method =>
    2. checks if object has that method =>
        3. checks if object prototype has that method =>
            4. checks if Object() constructor function prototype has method =>
                5. TypeError: method isn't a function

Enumerable Properties

Properties are enumerable, meaning they will show up in a for-in loop.

propertyIsEnumerable() – Every object inherits this method. All properties and methods created through assignment are enumerable:

Turtle.prototype.propertyIsEnumerable('eat') => true

non-enumerals – built-in methods are non-enumeral, user methods are enumeral:

Object.prototype.propertyIsEnumerable('toString') => false

Inheritance

extends – classes inherit other classes in a class declaration:

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

Polymorphism

Different objects with the same methods but customized using override.

toString() – Object.prototype automatically gives all objects this method. Override to display meaningful message:

```
class Turtle {
    toString() { return `A turtle called ${this.name}`;} }
```

Property attributes

property descriptor – an object that stores values of attributes that provide info about properties: value, writable, enumerable, configurable

```
const me = { name: 'DAZ' }
```

When an assignment is made, descriptor attributes will be set to true:

```
{ value: 'DAZ', writable: true, enumerable: true, configurable: true }
```

Get property descriptor – Use Object.getOwnPropertyDescriptor(object, property) method

```
Object.getOwnPropertyDescriptor(me,'name') =>
    { value: 'DAZ', writable: true, enumerable: true, configurable: true }
```

Set property descriptor – Use Object.defineProperty(object, property, new property descriptor) method

```
Object.defineProperty(me, 'eyeColor', { value: 'blue', writable: false, enumerable: true }) =>
```

Creating Object from other Object

Using object literal can act as a prototype for other classes. It is capitalized to act as as a class:

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

creating object using prototype – An instance of Human can be created:

```
const lois = Object.create(Human);
```

Human is prototype of lois object:

```
Human.isPrototypeOf(lois) => true
```

add new properties to instances – using assignment:

```
lois.name = 'Lois Lane' => 'Lois Lane'
lois.job = 'Reporter' => 'Reporter'
```

inheritance – Human acts as superclass and prototype of subclass. Subclassess can add new properties/methods.

```
const Superhuman = Object.create(Human);
Superhuman.change = function() {
    return `${this.realName} goes into a phone box and comes out as ${this.name}!`; };
```

default values – set values in its prototype so above method will always work:

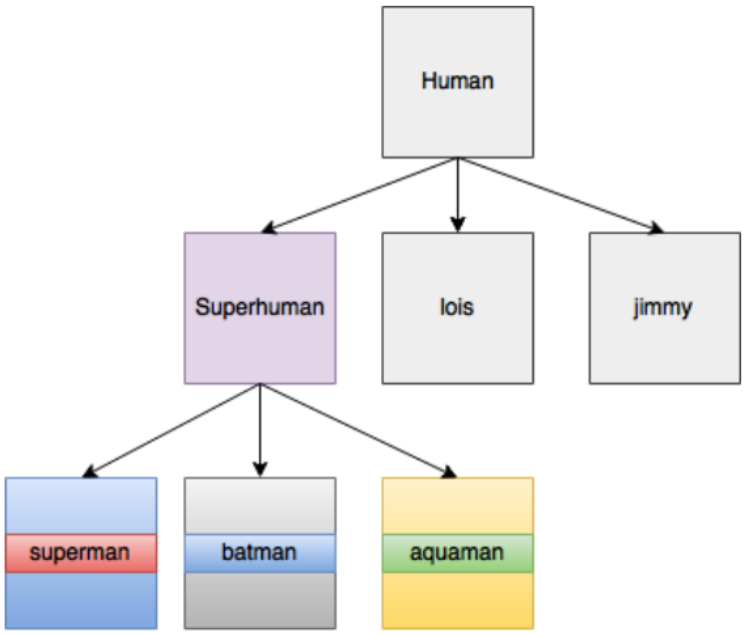
```
Superhuman.name = 'Name Needed';
Superhuman.realName = 'Real Name Needed';
```

2 ways to add custom values:

create superman object – by assignment	**using constructor function** – create an init method that takes values

const superman = Object.create(Superhuman);
superman.name = 'Superman';
superman.realName = 'Clark Kent';
	Superhuman.init = function(name,realName){
	this.name = name;
	this.realName = realName;
	this.init = undefined; // should only be called once so this removes it
	return this;
	
	const batman = Object.create(Superhuman);
	batman.init('Batman','Bruce Wayne');
	OR
	const aquaman = Object.create(Superhuman).init('Aquaman', 'Arthur Curry');
	!!This is the better way!!
	
	batman.change() =>
	'Bruce Wayne goes into a phone box and comes out as Batman!'

Object Prototype Chain -



12-2. The prototype chain

Mixins

How to avoid inheritance -> Mix objects together.

Object assign - Assign object properties to another object to make shallow copy. A shallow copy is only a reference and with every change the original will change:

const a = {};

const b = { name: 'JavaScript' };

Object.assign(a,b);

Deep copy – Prevents a shallow copy:

function mixin(target,...objects) {

```
    for (const object of objects) {
    if(typeof object === 'object') {
        for (const key of Object.keys(object)) {
            if (typeof object[key] === 'object') {
            target[key] = Array.isArray(object[key]) ? [] : {};
            mixin(target[key],object[key]);
            } else {
            Object.assign(target,object);
            }
        }
        }
    }
    return target;
}
```

Useful for adding a large number of properties to an object at once

const wonderWoman = Object.create(Superhuman);	
Mix using Object literal	**Assign one at a time**
mixin(wonderWoman, { name: 'Wonder Woman', realName: 'Diana Prince' }); !!This is the better way!!	wonderWoman.name = 'Wonder Woman'; wonderWoman.realName = 'Diana Prince';

Create a copy Function – make a deep copy of an object:

```
function copy(target) {
const object = Object.create(Object.getPrototypeOf(target));
mixin(object, target);
return object;}

const bizarro = copy(superman);
```

Factory function – a function used to return an object. Uses the copy function created above and allows you to use an object literal as an argument:

```
function createSuperhuman(...mixins) {
    const object = copy(Superhuman);
    return mixin(object,...mixins); }

const hulk = createSuperhuman({name: 'Hulk', realName: 'Bruce Banner'});
```

mixin objects – Add superpower objects to superhero

```
const flight = {
fly() {
 console.log(`Up, up and away! ${this.name} soars through the
↪ air!`);
    return this;
}
}

const superSpeed = {
move() {
 console.log(`${this.name} can move faster than a speeding
↪ bullet!`);
    return this;
}
}

const xRayVision = {
xray() {
    console.log(`${this.name} can see right through you!`);
    return this;
}
}

mixin(superman,flight,superSpeed,xRayVision);
```

Add mixins as argument to factory function – one assignment to create superhero object inheriting Superhuman object, custom name details and relevant powers:

```
const flash = createSuperhuman({ name: 'Flash', realName: 'Barry Allen' }, superSpeed);
```

Chaining Functions

form a sequence of method calls– chain methods together by having them return this, then you can call multiple methods at once:

```
superman.fly().move().xray();
```

Binding this

this – points to the object calling the method. Functions inside functions create problems. The value of *this* loses scope. It actually points to the global object:

```
superman.friends = [batman,wonderWoman,aquaman]

superman.findFriends = function(){

    this.friends.forEach(function(friend) {

        console.log(`${friend.name} is friends with ${this.name}`); }); }
```

this.name is undefined.

Solutions:

set this to that	**bind**
create a reference before the nested function	sets the value of this in a function. bind(this) binds to the object calling the method

superman.findFriends = function(){
const that = this;
this.friends.forEach(function(friend) {
console.log(
`${friend.name} is friends with ${that.name}`); });}
	superman.findFriends = function() {
	this.friends.forEach(function(friend) {
	console.log(
	`${friend.name} is friends with{this.name}`); }.bind(this);)}

for-of loop	**arrow function**
prevents using nested functions, and *this* remains bound	*this* remains bound to object calling method

superman.findFriends = function() {
for(const friend of this.friends) {
console.log(
`${friend.name} is friends with ${this.name}`);}; }
	superman.findFriends = function() {
	this.friends.forEach((friend) => {
	console.log(
	`${friend.name} is friends with ${this.name}`); }); }

Borrowing from objects

By making a reference to a method, an object can borrow a method from another object without inheriting.

```
const fly = superman.fly;

fly.call(batman);
```

Borrowing from arrays

Take the slice method from array	Array literal	Slice with no arguments
const slice = Array.prototype.slice; slice.call(arguments, 1, 3);	[].slice.call(arguments, 1, 3)	const argumentsArray = Array.prototype.slice.call(arguments);

ES6 and onward methods:

Array.from()	Spread operator
const argumentsArray = Array.from(arguments);	const argumentsArray = [...arguments];

Composition VS Inheritance

Inheritence causes bloating. Composition is building small blocks of single tasks/behaviors that help to build complex objects. Classes are monolithic structures layered on top of one another. Make classes with short inheritance chains and small amounts of properties/methods.Only inherit once or you risk problems with attempting to make changes since they affect objects in the whole chain. Borrow a method from a class when there are properties/methods not needed.