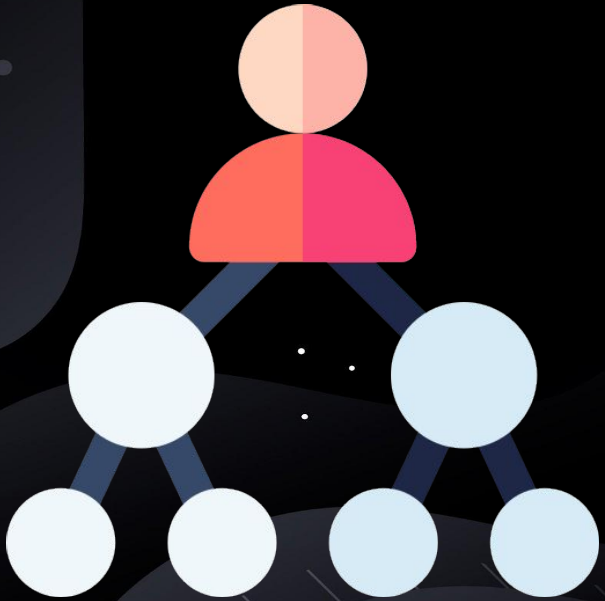
JavaScript

STUDIA PODYPLOMOWE
POLITECHNIKA BIAŁOSTOCKA

#2

Prototypal Inheritance

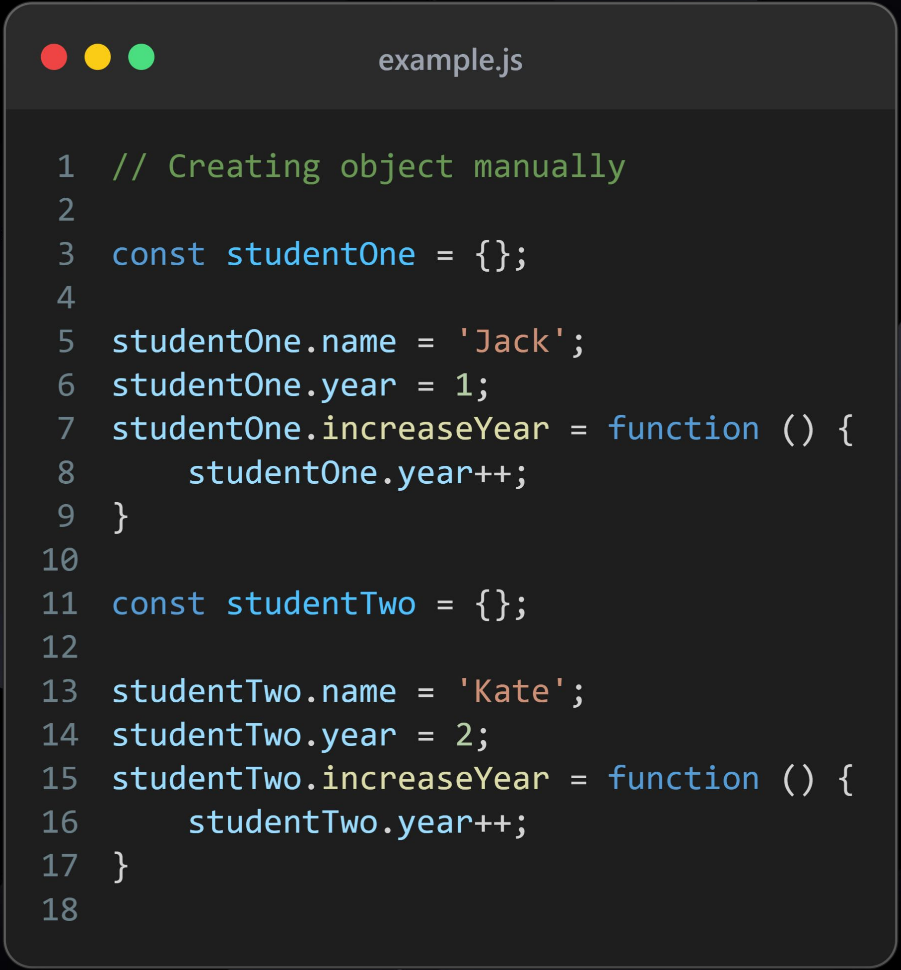


Creating a student

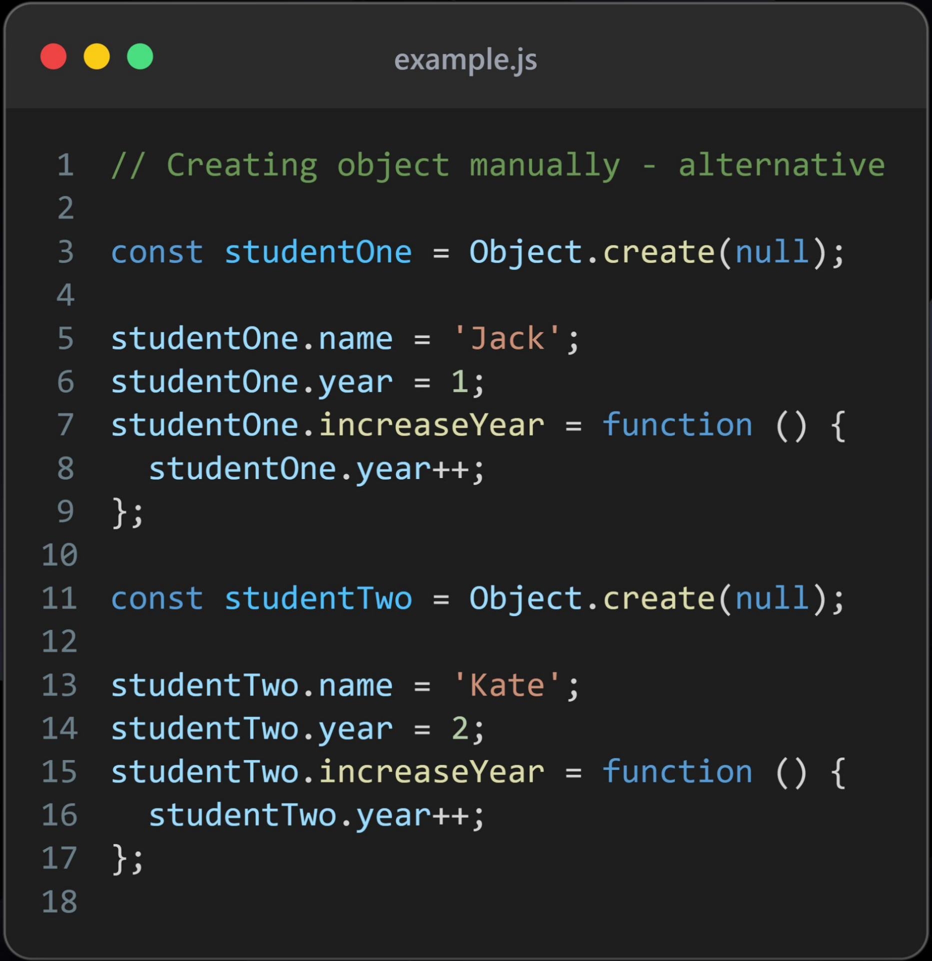
- We want to have a student
- The student has a name and a year
- We need to store the student’s name and year
- We want to add functionality - increase student’s year

Object Oriented Paradigm

- Combine data and functionality into one logical piece - Object
- Store data as object fields
- Store functionality as a method on the object



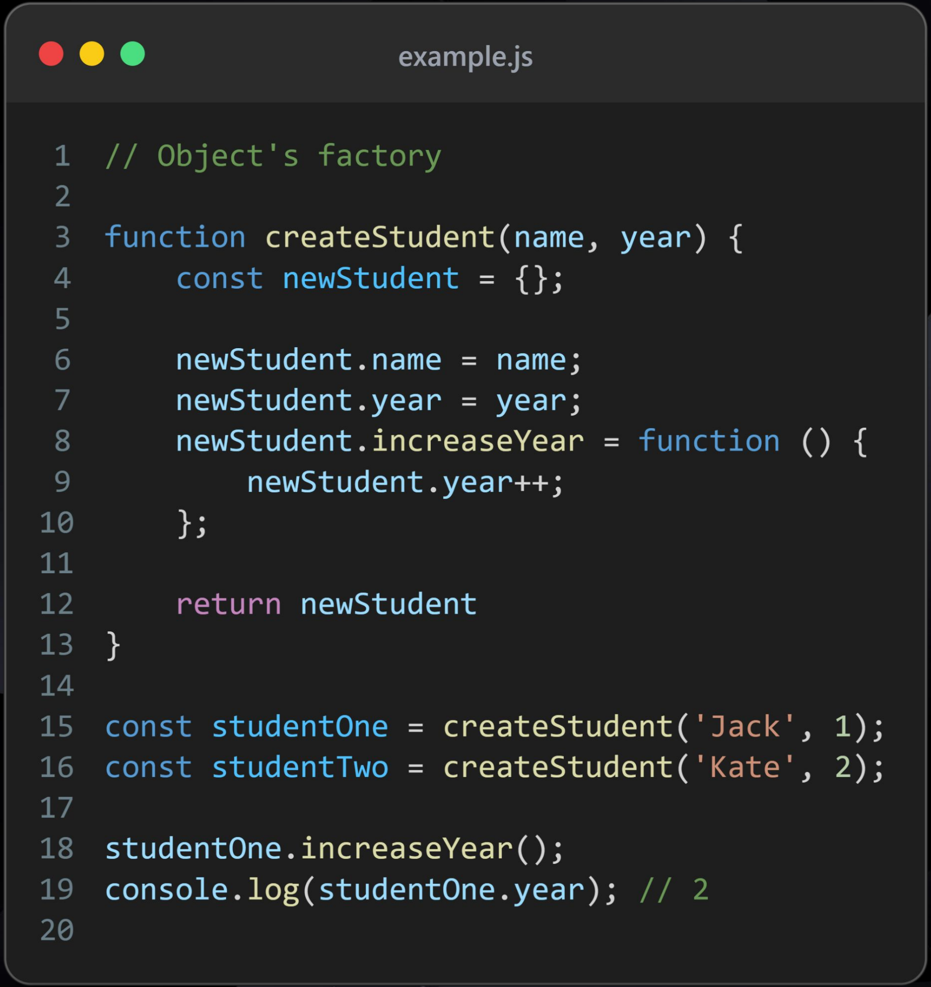
```javascript
// Creating object manually

const studentOne = {};

studentOne.name = 'Jack';
studentOne.year = 1;
studentOne.increaseYear = function () {
    studentOne.year++;
}

const studentTwo = {};

studentTwo.name = 'Kate';
studentTwo.year = 2;
studentTwo.increaseYear = function () {
    studentTwo.year++;
}

```



```javascript
// Creating object manually - alternative

const studentOne = Object.create(null);

studentOne.name = 'Jack';
studentOne.year = 1;
studentOne.increaseYear = function () {
  studentOne.year++;
};

const studentTwo = Object.create(null);

studentTwo.name = 'Kate';
studentTwo.year = 2;
studentTwo.increaseYear = function () {
  studentTwo.year++;
};

```

Improving the solution

- The current approach is manual
- We need to automate it more!



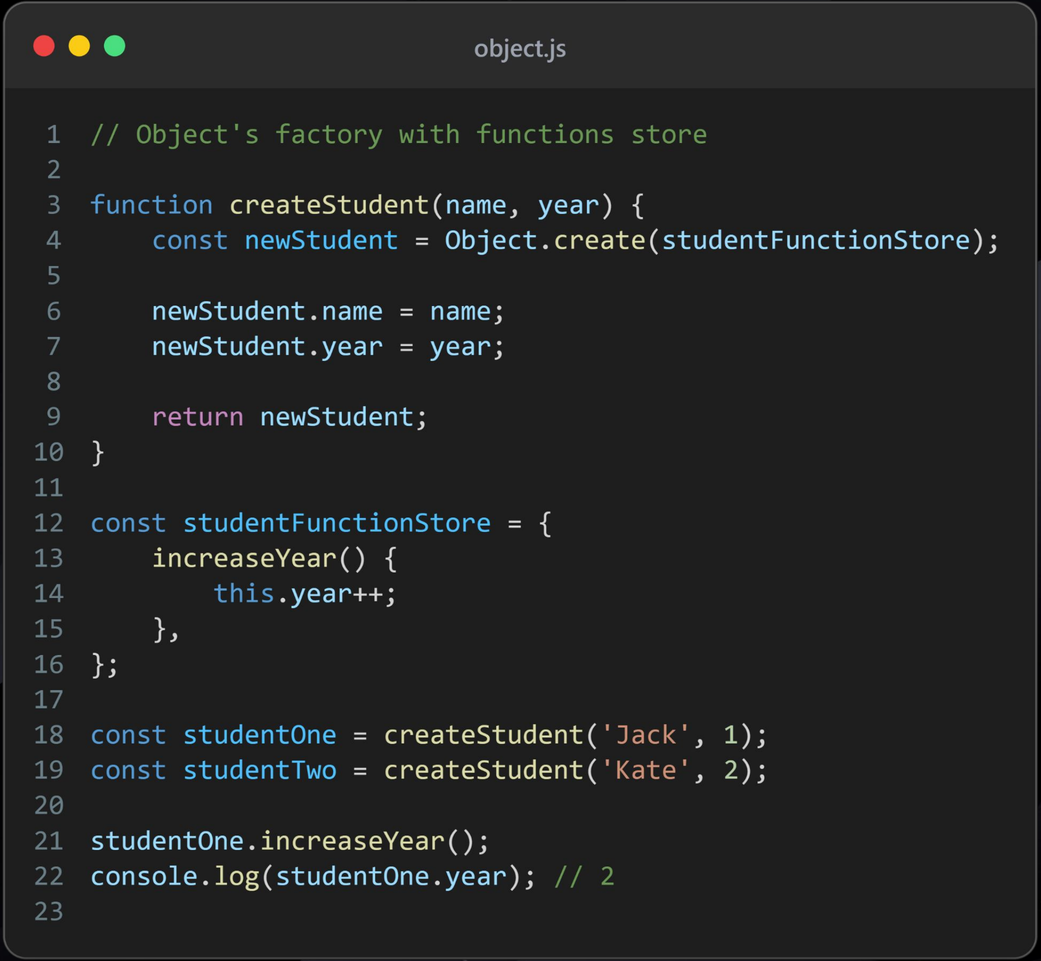
```javascript
// Object's factory

function createStudent(name, year) {
    const newStudent = {};

    newStudent.name = name;
    newStudent.year = year;
    newStudent.increaseYear = function () {
        newStudent.year++;
    };

    return newStudent
}

const studentOne = createStudent('Jack', 1);
const studentTwo = createStudent('Kate', 2);

studentOne.increaseYear();
console.log(studentOne.year); // 2

```

Problem

- We are creating multiple copies of identical functions in memory
- What if student had 50 methods?

Solution

- Create a store that keeps all the functions
- Make a connection that will allow each student to access functions



```javascript
// Object's factory with functions store

function createStudent(name, year) {
    const newStudent = Object.create(studentFunctionStore);

    newStudent.name = name;
    newStudent.year = year;

    return newStudent;
}

const studentFunctionStore = {
    increaseYear() {
        this.year++;
    },
};

const studentOne = createStudent('Jack', 1);
const studentTwo = createStudent('Kate', 2);

studentOne.increaseYear();
console.log(studentOne.year); // 2

```

__proto__

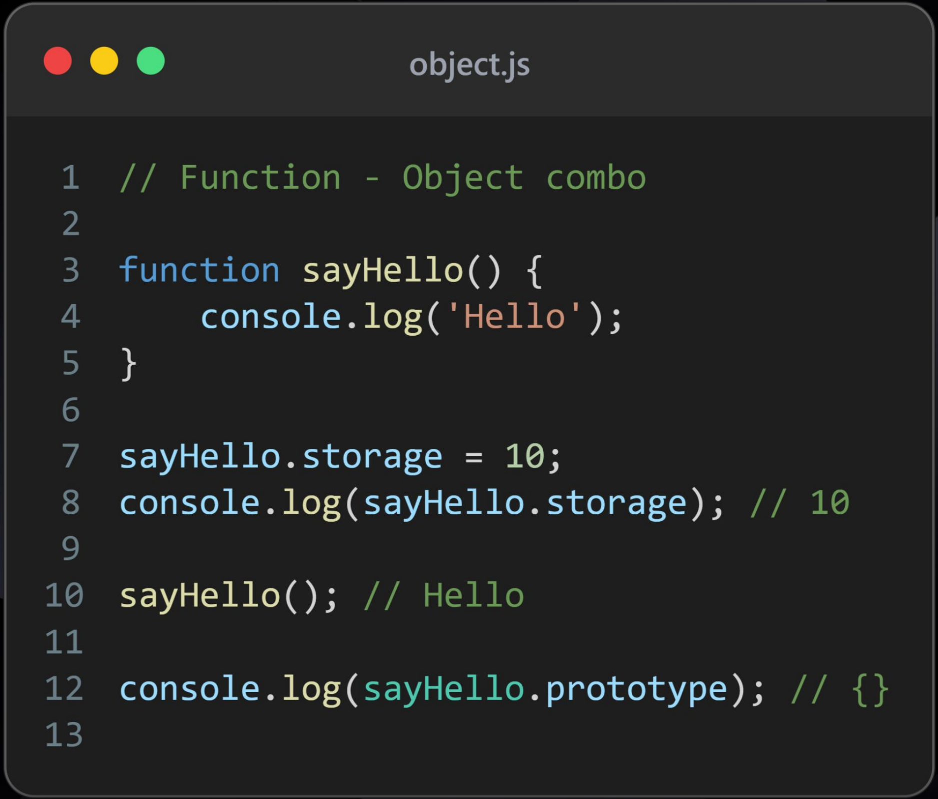
- With Object.create(**functionStore**) we can create a new empty **object**
- BUT this **object** has a hidden property - __proto__
- This property links to the **object** we pass as an argument
- This allows us to call methods and access values that are stored in the passed **object** as if they were on our **object**

This is it!

- That’s the whole secret behind prototypal inheritance
- Objects linked to another Objects
- From now on it’s just expanding this concept, creating chains of prototypal links = prototypal inheritance

Functions in JS

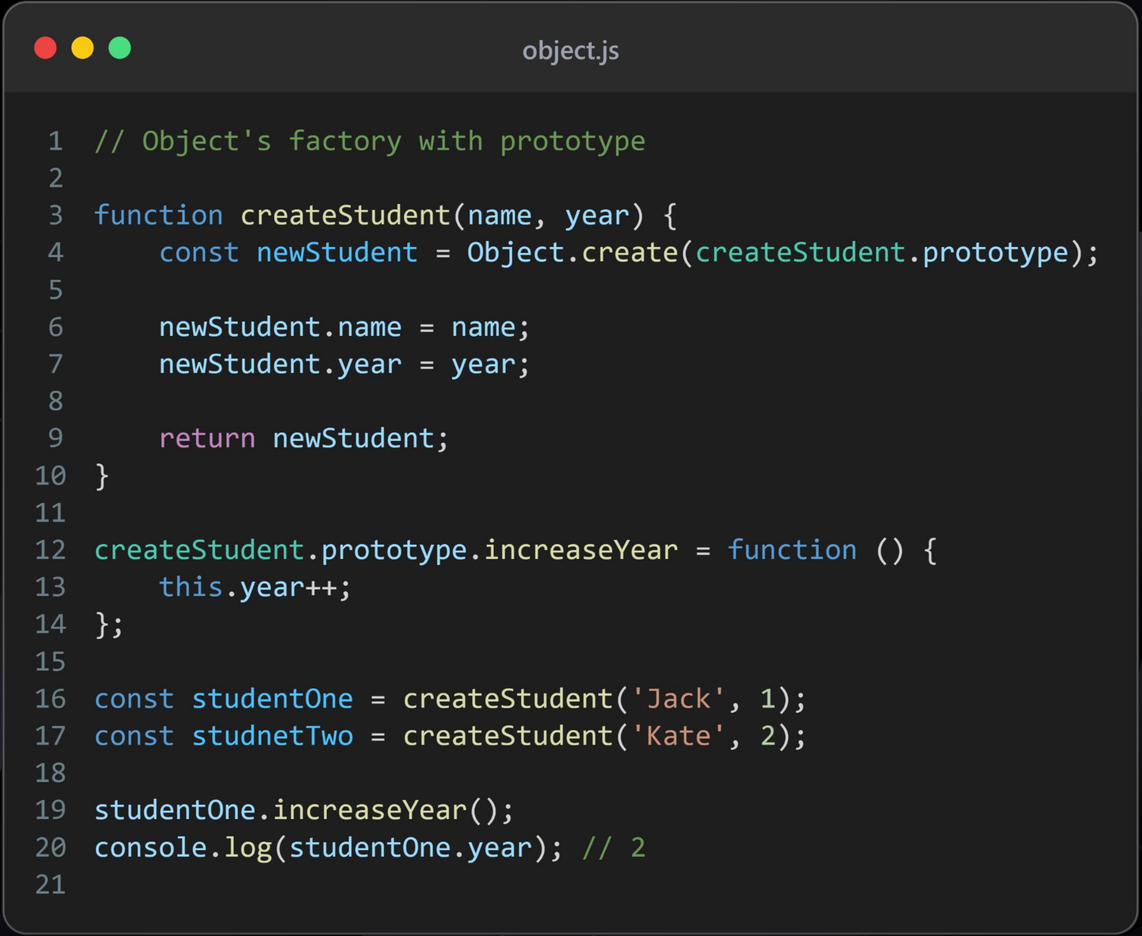
- Function in JS is a special type of Object
- Function - Object combo
- As with all objects - they can store values under keys as fields
- What if we keep our studentFunctionsStore in the createStudent function - object combo?



```js
// Function - Object combo

function sayHello() {
    console.log('Hello');
}

sayHello.storage = 10;
console.log(sayHello.storage); // 10

sayHello(); // Hello

console.log(sayHello.prototype); // {}

```

Prototype

- JS creates such a store for all functions!
- It’s under .prototype key
- functionName.prototype is an object that can store methods used by all objects created with Object.create(functionName) or new functionName()



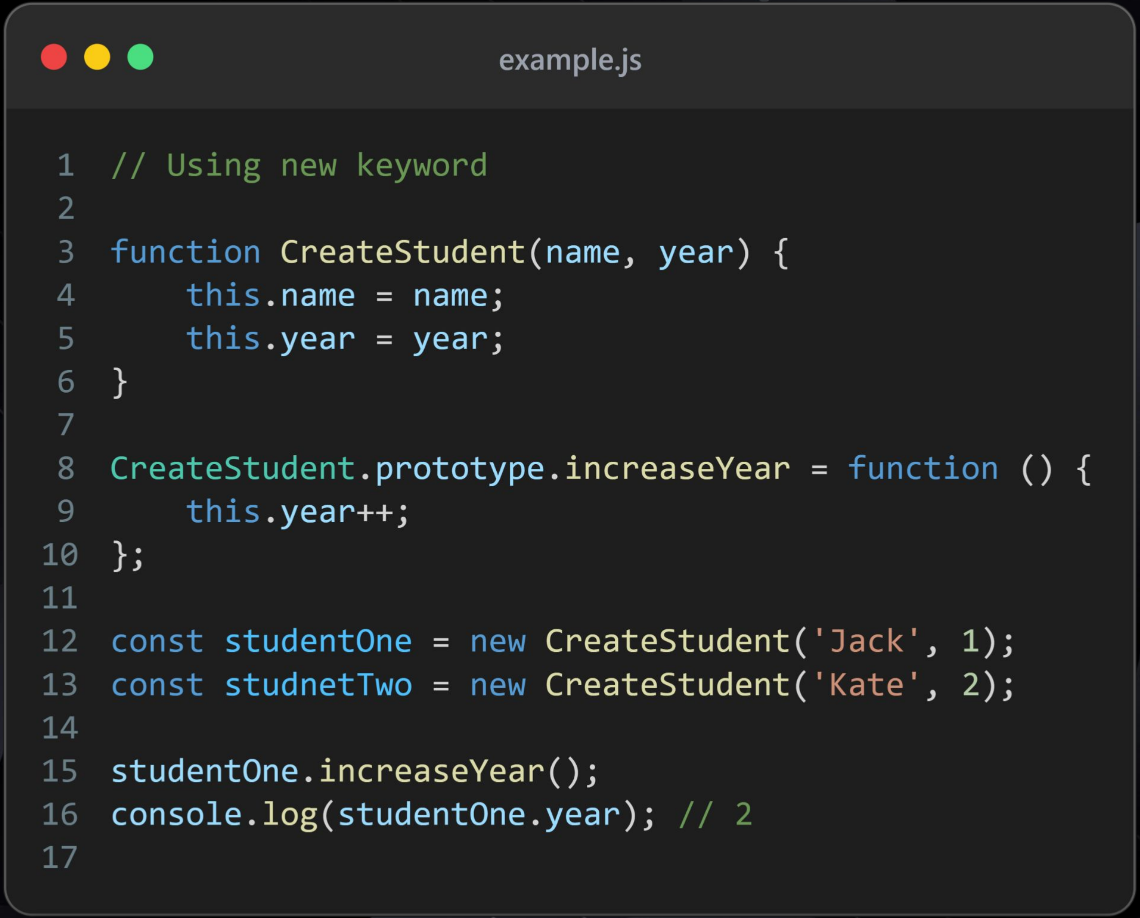
```javascript
// Object's factory with prototype

function createStudent(name, year) {
    const newStudent = Object.create(createStudent.prototype);

    newStudent.name = name;
    newStudent.year = year;

    return newStudent;
}

createStudent.prototype.increaseYear = function () {
    this.year++;
};

const studentOne = createStudent('Jack', 1);
const studnetTwo = createStudent('Kate', 2);

studentOne.increaseYear();
console.log(studentOne.year); // 2

```

First Sugar Coat

- JS creators thought this is too much
- They added new keyword to make things simpler
- But it hides the actual inner workings
- It’s fine to use it, but understand what is going on

New Recap

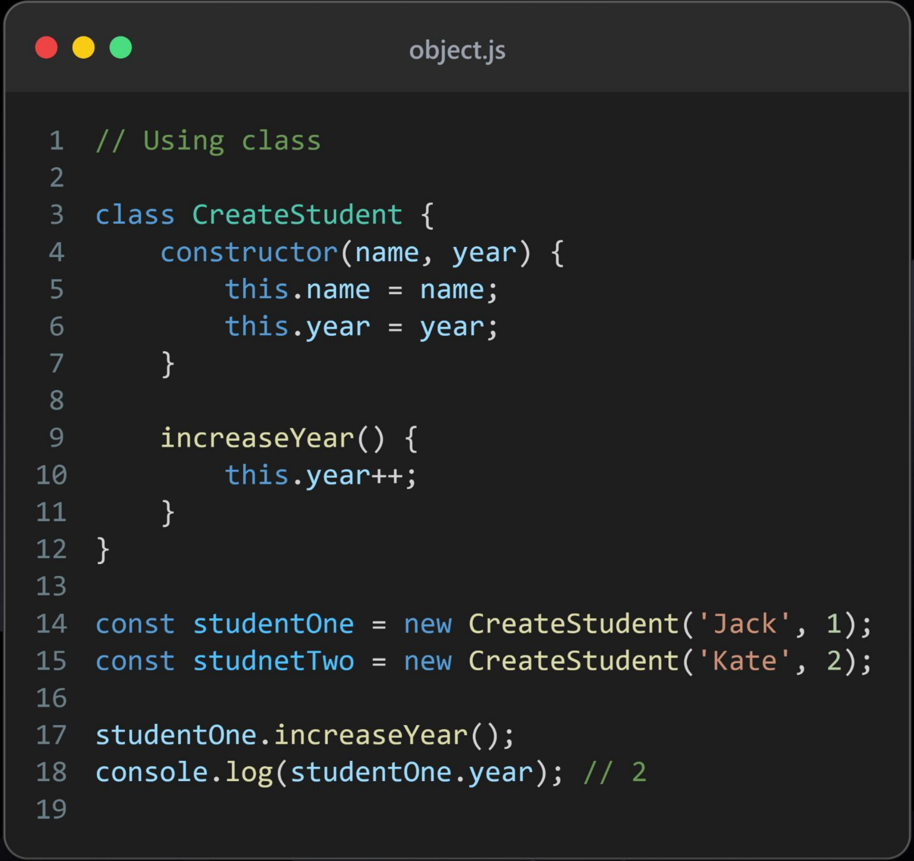
- When used with a function:
  - creates an empty object in the function’s context
  - makes *this* point to that new object
  - sets object.__proto__ = function.prototype
  - returns the object (without *return* keyword!)



```javascript
// Using new keyword

function CreateStudent(name, year) {
    this.name = name;
    this.year = year;
}

CreateStudent.prototype.increaseYear = function () {
    this.year++;
};

const studentOne = new CreateStudent('Jack', 1);
const studnetTwo = new CreateStudent('Kate', 2);

studentOne.increaseYear();
console.log(studentOne.year); // 2
```

This was the way

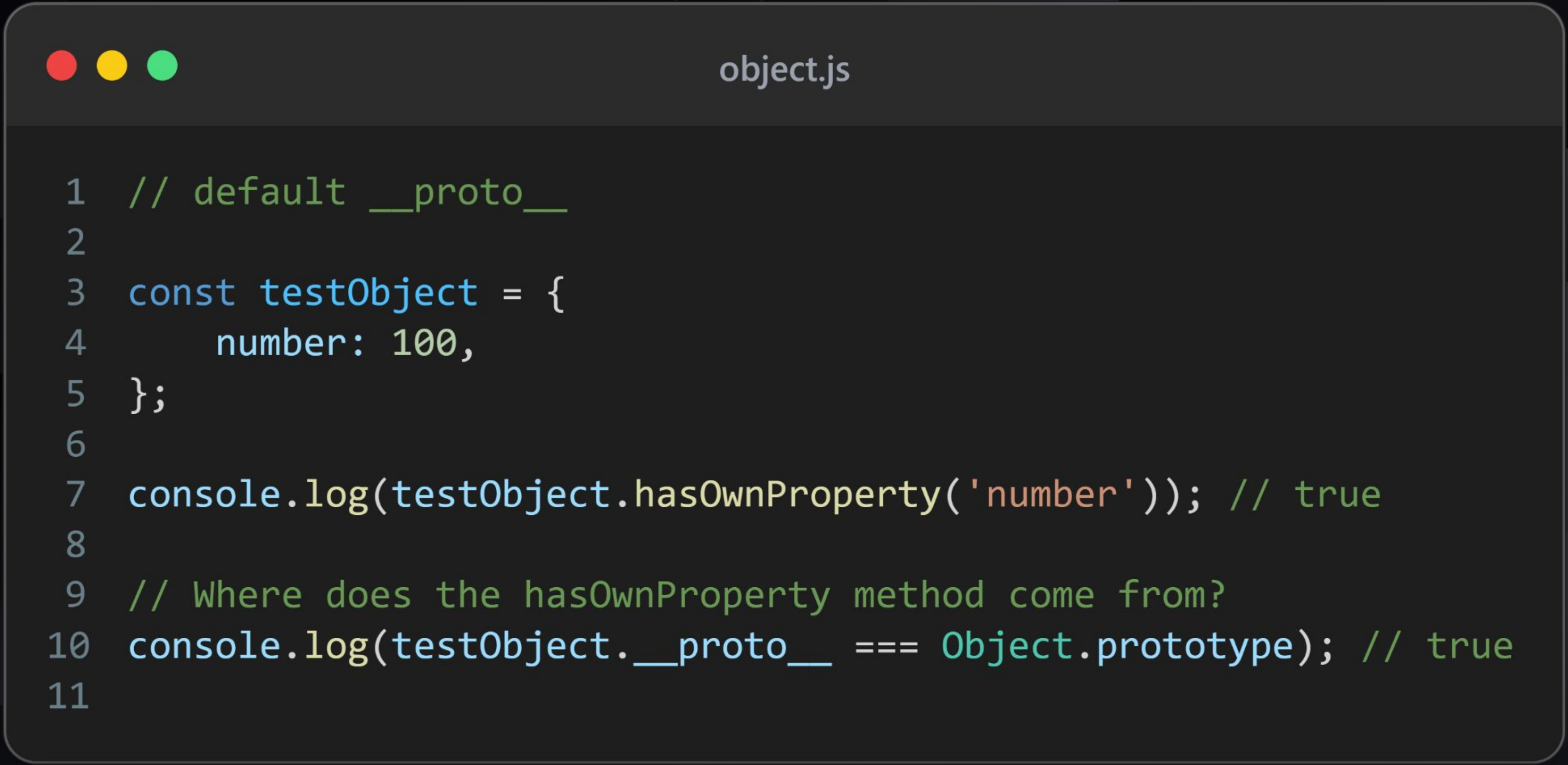
- Before ES6 (2015) it was the default way of class implementation
- Since ES6 we got classes in JS



```javascript
// Using class

class CreateStudent {
    constructor(name, year) {
        this.name = name;
        this.year = year;
    }

    increaseYear() {
        this.year++;
    }
}

const studentOne = new CreateStudent('Jack', 1);
const studnetTwo = new CreateStudent('Kate', 2);

studentOne.increaseYear();
console.log(studentOne.year); // 2
```

Let’s leave classes

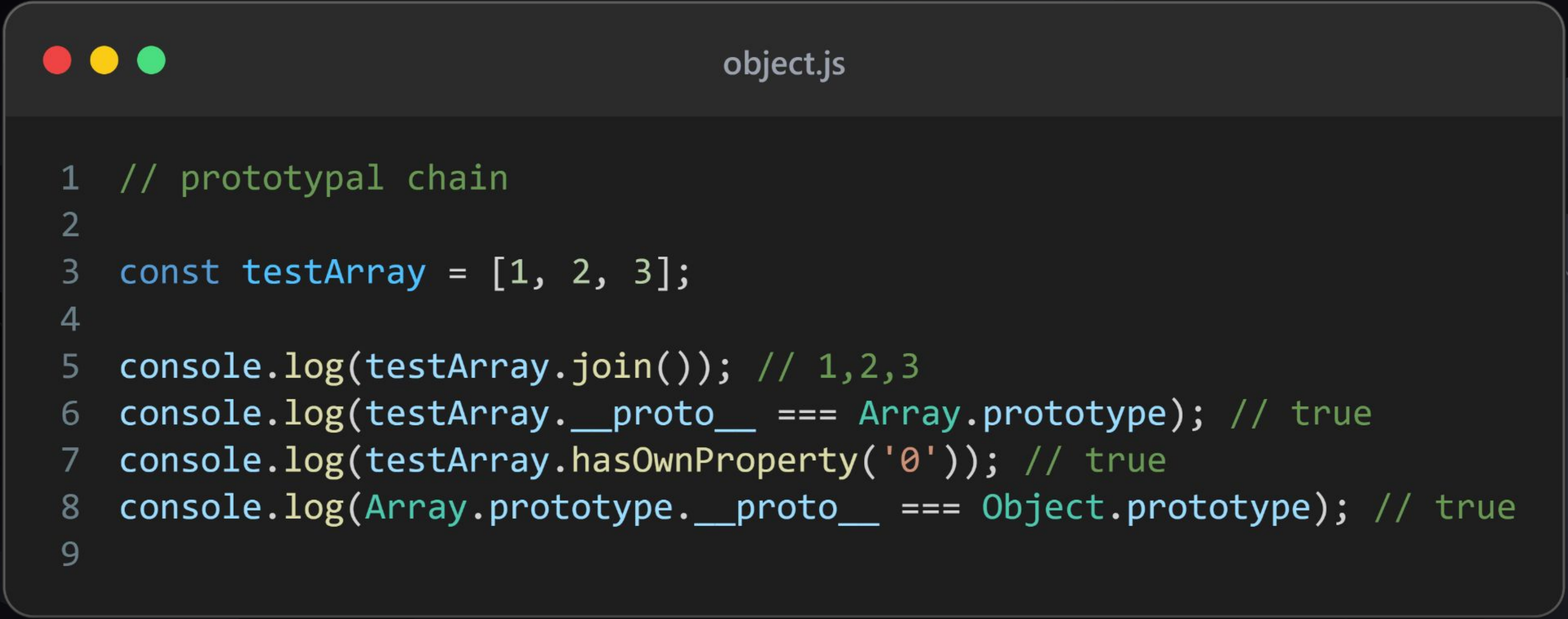
for now…



```javascript
// default __proto__

const testObject = {
    number: 100,
};

console.log(testObject.hasOwnProperty('number')); // true

// Where does the hasOwnProperty method come from?
console.log(testObject.__proto__ === Object.prototype); // true
```

Default __proto__

- Every object in JS has a default __proto__ property
- If we don’t set it - it links to Object.prototype



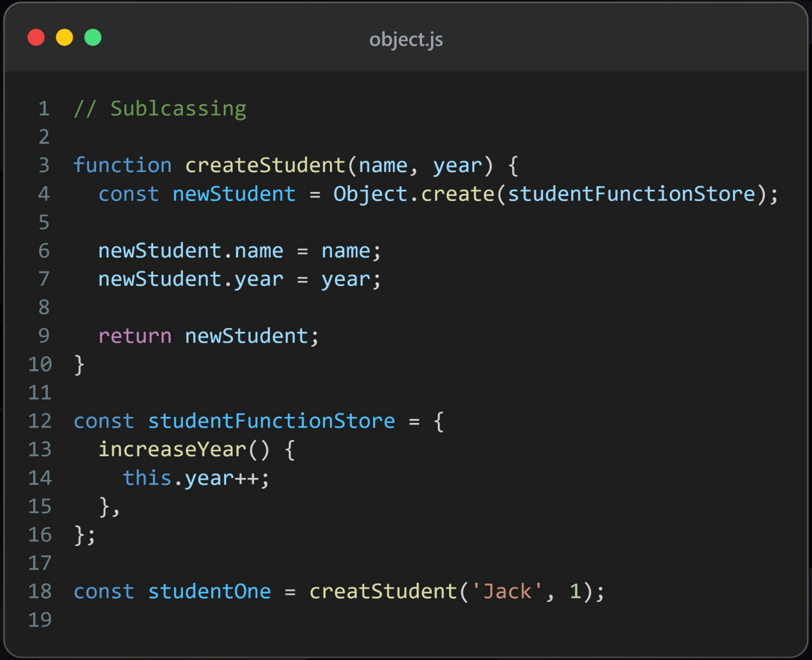
```javascript
// prototypal chain

const testArray = [1, 2, 3];

console.log(testArray.join()); // 1,2,3
console.log(testArray.__proto__ === Array.prototype); // true
console.log(testArray.hasOwnProperty('0')); // true
console.log(Array.prototype.__proto__ === Object.prototype); // true

```

Prototypal chain

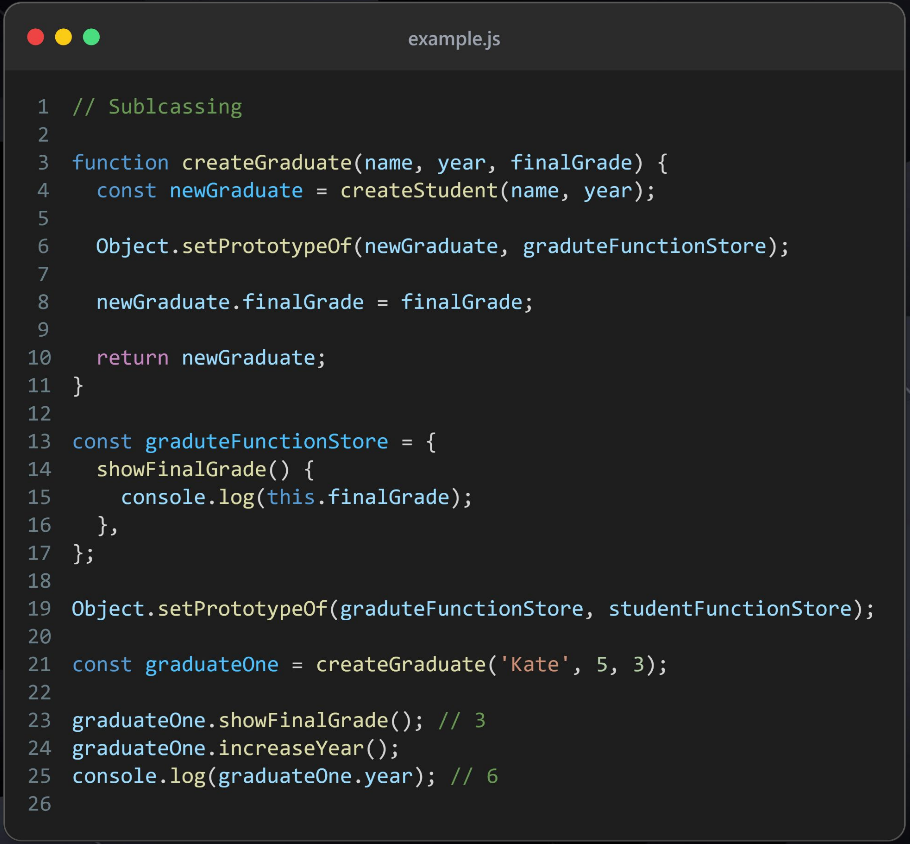
- Prototypes are also objects - so they have __proto__
- Those can link to other prototypes
- So we can have prototypes chains

Subclassing

- We can manually create such chains
- This allows us to natively implement inheritance in JS
- prototypal inheritance



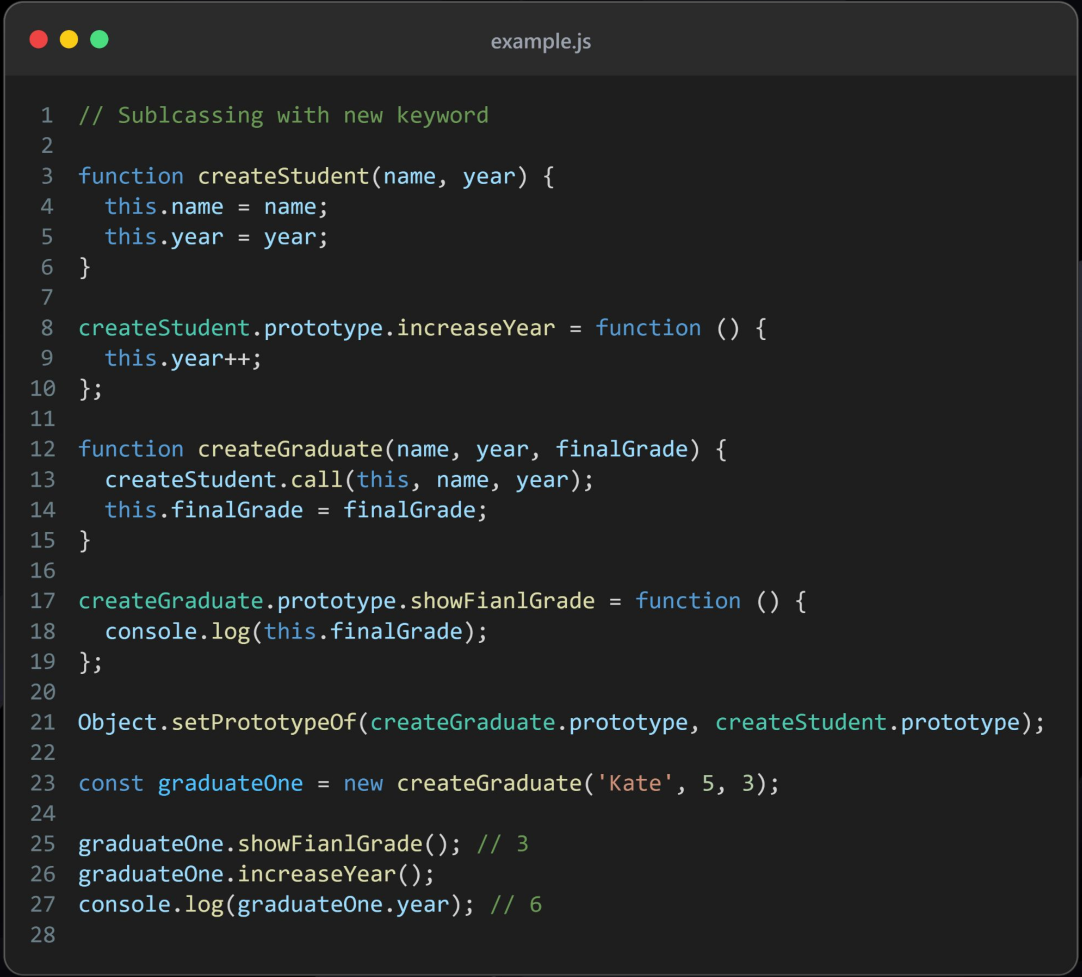
```javascript
// Sublcassing

function createStudent(name, year) {
  const newStudent = Object.create(studentFunctionStore);

  newStudent.name = name;
  newStudent.year = year;

  return newStudent;
}

const studentFunctionStore = {
  increaseYear() {
    this.year++;
  },
};

const studentOne = creatStudent('Jack', 1);

```



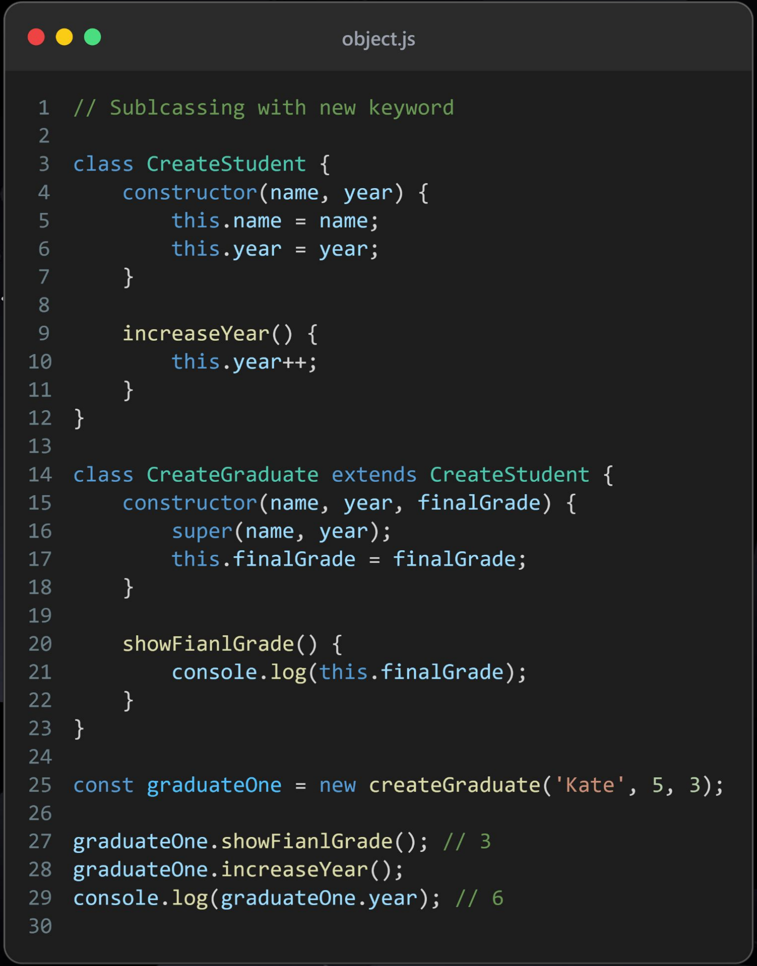
```javascript
// Sublcassing

function createGraduate(name, year, finalGrade) {
  const newGraduate = createStudent(name, year);

  Object.setPrototypeOf(newGraduate, graduteFunctionStore);

  newGraduate.finalGrade = finalGrade;

  return newGraduate;
}

const graduteFunctionStore = {
  showFinalGrade() {
    console.log(this.finalGrade);
  },
};

Object.setPrototypeOf(graduteFunctionStore, studentFunctionStore);

const graduateOne = createGraduate('Kate', 5, 3);

graduateOne.showFinalGrade(); // 3
graduateOne.increaseYear();
console.log(graduateOne.year); // 6

```

Subclassing with new

- We can add new to our solution
- A bit less code, and a bit more weirdness with *this* keyword



```javascript
// Sublcassing with new keyword

function createStudent(name, year) {
  this.name = name;
  this.year = year;
}

createStudent.prototype.increaseYear = function () {
  this.year++;
};

function createGraduate(name, year, finalGrade) {
  createStudent.call(this, name, year);
  this.finalGrade = finalGrade;
}

createGraduate.prototype.showFianlGrade = function () {
  console.log(this.finalGrade);
};

Object.setPrototypeOf(createGraduate.prototype, createStudent.prototype);

const graduateOne = new createGraduate('Kate', 5, 3);

graduateOne.showFianlGrade(); // 3
graduateOne.increaseYear();
console.log(graduateOne.year); // 6

```



```javascript
// Sublcassing with new keyword

class CreateStudent {
    constructor(name, year) {
        this.name = name;
        this.year = year;
    }

    increaseYear() {
        this.year++;
    }
}

class CreateGraduate extends CreateStudent {
    constructor(name, year, finalGrade) {
        super(name, year);
        this.finalGrade = finalGrade;
    }

    showFianlGrade() {
        console.log(this.finalGrade);
    }
}

const graduateOne = new createGraduate('Kate', 5, 3);

graduateOne.showFianlGrade(); // 3
graduateOne.increaseYear();
console.log(graduateOne.year); // 6

```

Back to classes

- This is the most elegant
- But mostly obscures what happens under the hood
- It’s crucial to understand what really happens
- It’s not true class inheritance is like in other programming languages, more like an “emulation” with prototypes

THE
END