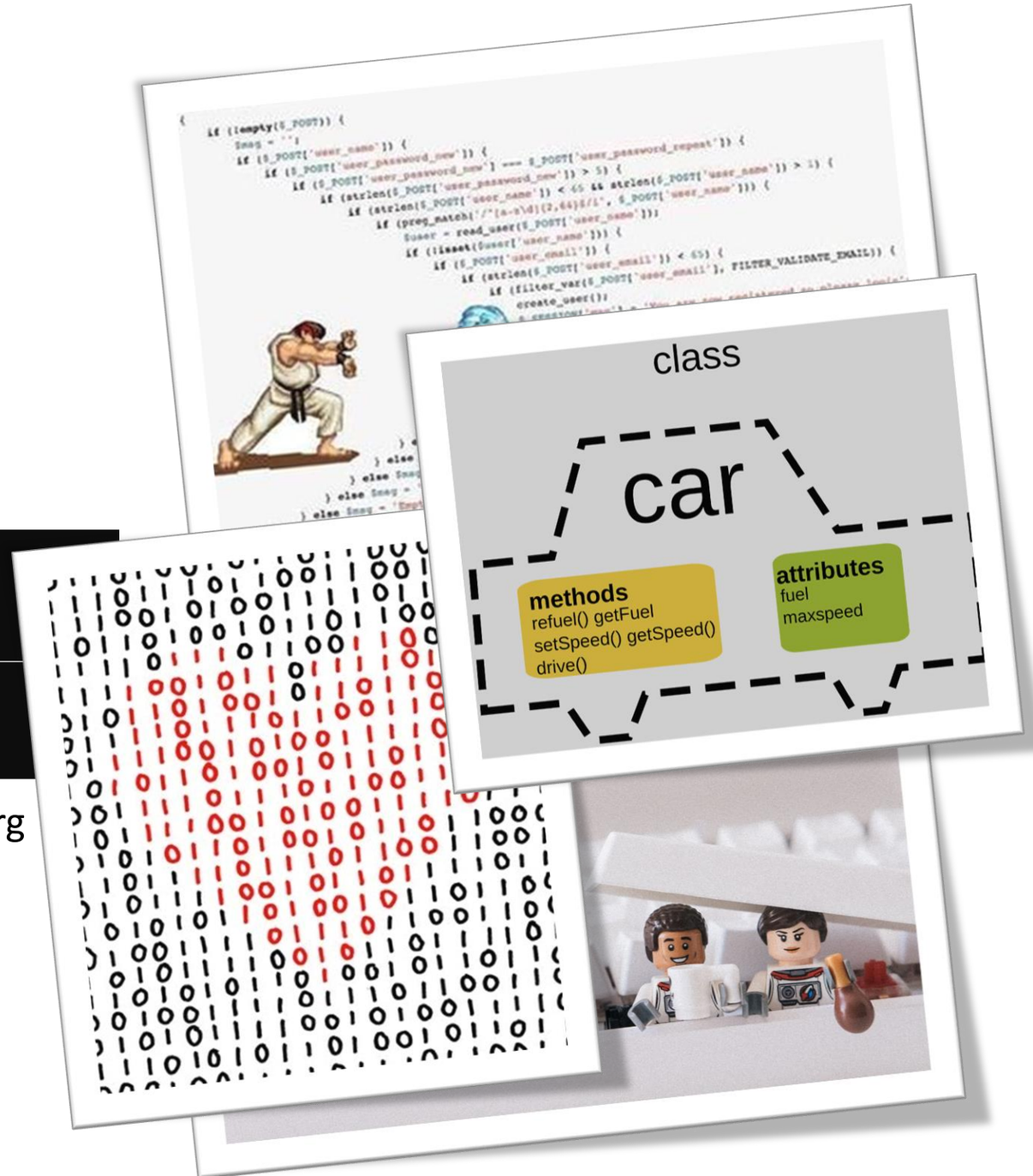
Programming Fundamentals (in JavaScript) 2: OOP

Stefano Balietti

Center for European Social Science Research at Mannheim University (MZES)
Alfred-Weber Institute of Economics at Heidelberg University

@balietti | stefanobalietti.com | @nodegameorg | nodegame.org

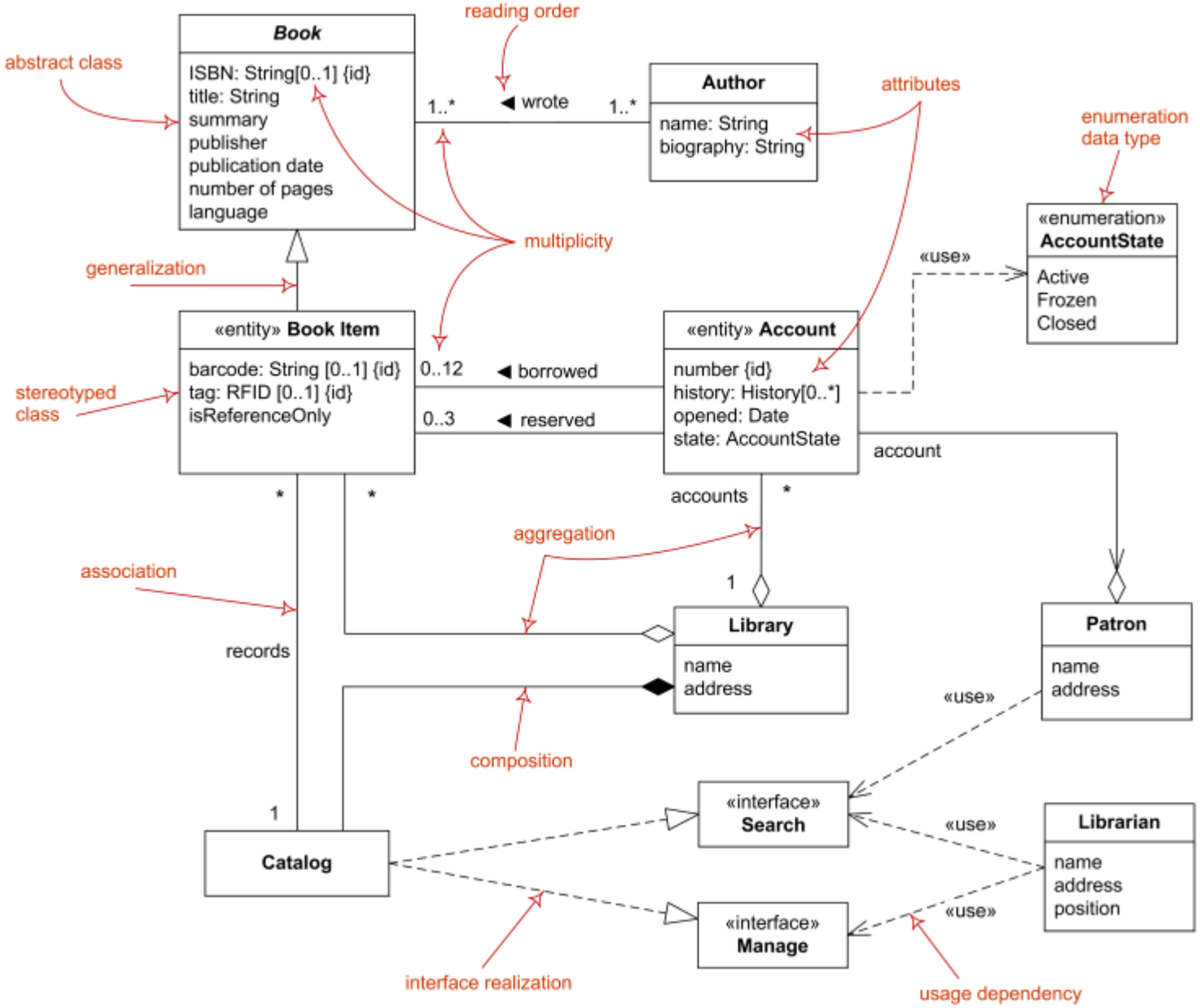


Part 2: Object Oriented Programming (OOP)

Object Oriented Programming (OOP)

- JavaScript is **multi-paradigm**, it has features of the OOP paradigm and of the procedural programming (PP) paradigm
- OOP and PP are two conceptually opposite coding philosophy
- PP revolves *stateless* **procedures** (functions)
- OOP revolves around *stateful* **objects** and **classes**, and on precise relationships between them.

Objects and Classes Diagram



https://www.uml-diagrams.org/class-diagrams-overview.html

Objects and Classes Diagram

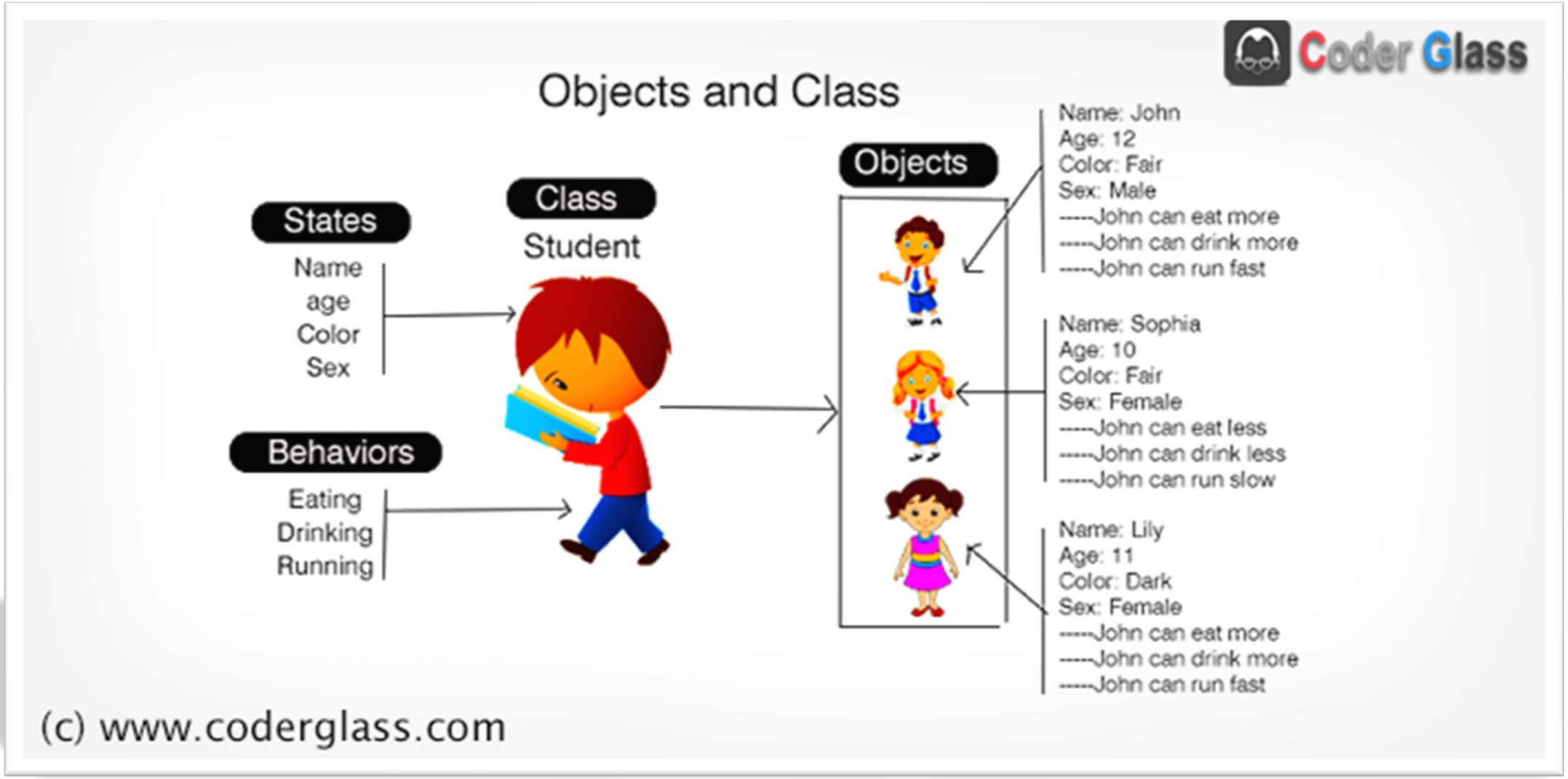
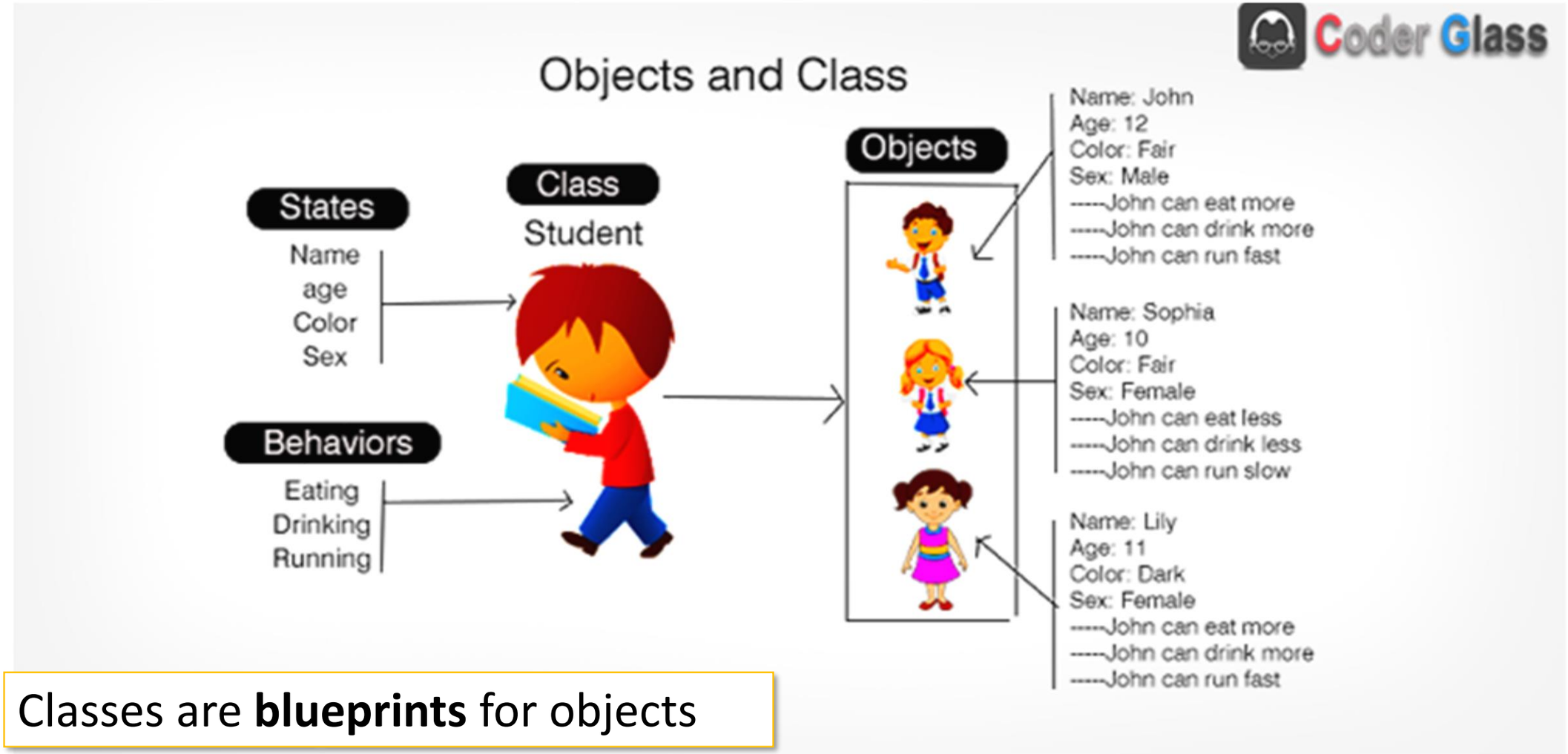


Image source: https://www.coderglass.com/java/java-object-and-class.php

Objects and Classes Diagram

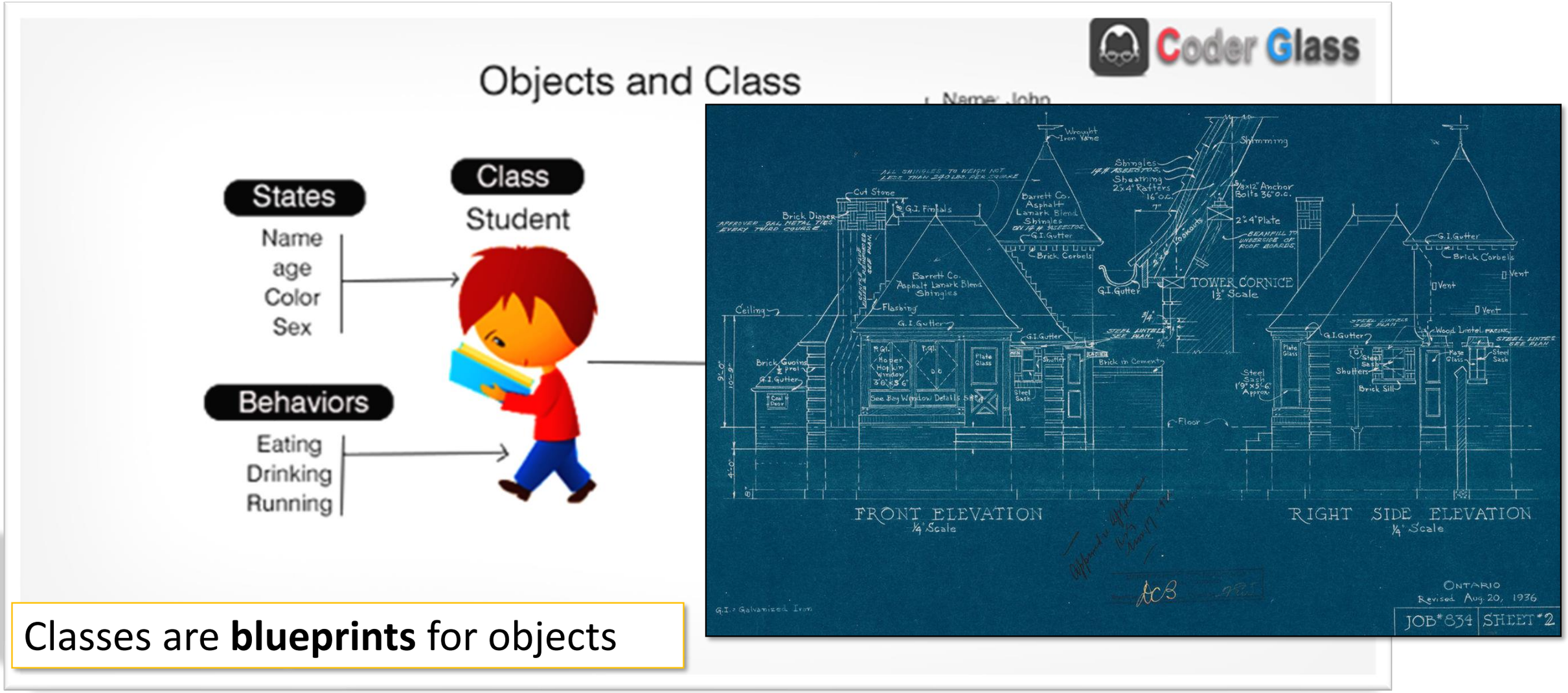


Classes are **blueprints** for objects

Image source: https://www.coderglass.com/java/java-object-and-class.php

https://en.wikipedia.org/w

Objects and Classes Diagram



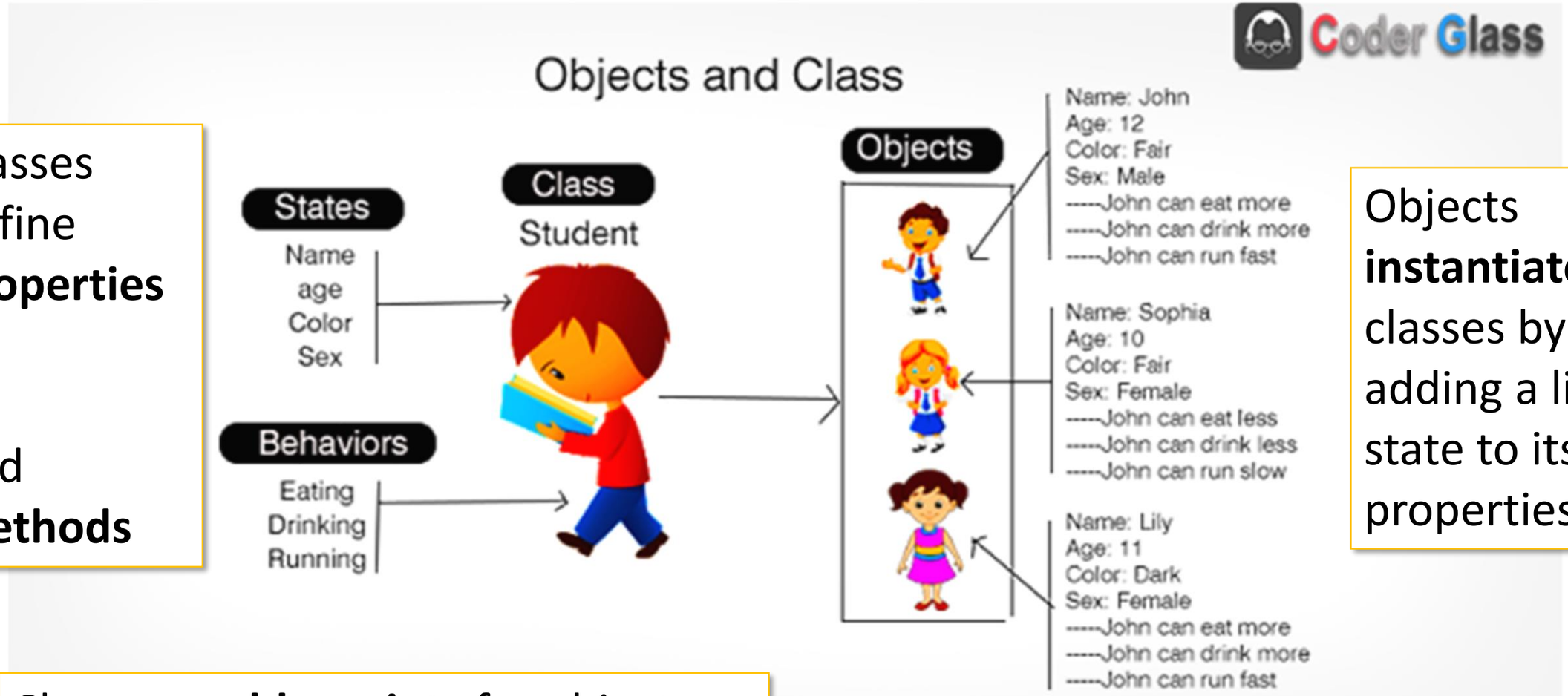
Classes are **blueprints** for objects

Image source: https://www.coderglass.com/java/java-object-and-class.php

Objects and Classes Diagram

Classes define **properties**

and **methods**



Objects **instantiate** classes by adding a live state to its properties

Classes are **blueprints** for objects

Image source: https://www.coderglass.com/java/java-object-and-class.php

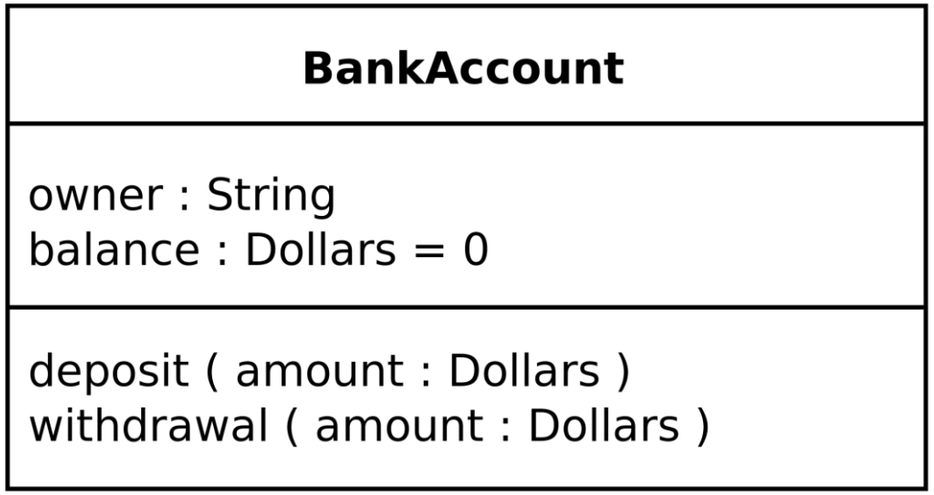
Discuss Exercise

- Let's pick a topic from the list below. Discuss and draw a **UML (Unified Modelling Language) class diagram** for at least one super class and two child classes.

- SHAPES
- ANIMALS
- PROFESSORS
- ACTORS
- SPORTLERS
- CELESTIAL BODIES
- NATIONALITIES

Public Properties

Public Methods



Wikipedia: https://en.wikipedia.org/wiki/Unified_Modeling_Language

JavaScript Classes

```
class Person {

  constructor() {
    this.name = 'Stefano Balietti';
  }

  sayHi() {
    console.log('Hi! I am ' + this.name);
  }
}
```

JavaScript Classes

```
class Person {

  constructor() {
    this.name = 'Stefano Balietti';
  }

  sayHi() {
    console.log('Hi! I am ' + this.name);
  }
}
```

Notice! This is the news ES6 definition of a class.
It is much easier than using ES5 prototypical definition, even if behind the scenes it is exactly the same. *Exercise available!*

JavaScript Classes

```
class Person {
  constructor() {
    this.name = 'Stefano Balietti';
  }
  sayHi() {
    console.log('Hi! I am ' + this.name);
  }
}

// Create an object using the new operator
let stefano = new Person();
```

JavaScript Classes

```
class Person {
  constructor() {
    this.name = 'Stefano Balietti';
  }
  sayHi() {
    console.log('Hi!
  }
}

// Create an object using the new operator
let stefano = new Person();
```

The new operator invokes the **constructor** method of the class. The constructor is a special method which is executed only once, upon creation.

JavaScript Classes

```
class Person {
  constructor() {
    this.name = 'Stefano Balietti';
  }
  sayHi() {
    console.log('Hi!
  }
}

// Create an object
let stefano = new Pe
```

The new operator invokes the **constructor** method of the class. The constructor is a special method which is executed only once, upon creation.

In this case, it is adding the property 'name' with the value 'Stefano Balietti'.

The Constructor

```
constructor() {
   this.name = 'Stefano Balietti';
}
```

The constructor is a compact way of creating new objects. What it does is the following:

The Constructor

```
constructor() {
   this.name = 'Stefano Balietti';
}
```

The constructor is a compact way of creating new objects. What it does is the following:

```
constructor() {
  let person = {};
  person.name = 'Stefano Balietti';
  return person;
}
```

The Constructor

```
constructor() {
   this.name = 'Stefano Balietti';
}
```

The constructor is a compact way of creating new objects. What it does is the following:

```
constructor() {
  let this = {};
  this.name = 'Stefano Balietti';
  return this;
}
```

The Instantiated Object

```
// Create an object using the new operator
let stefano = new Person();
console.log(stefano)
```

```
{
    name: 'Stefano Balietti
}
```

In the technical language the variable stefano is the live "instance" of the class Person.

Couldn't we directly create the object? What is the advantage of using a constructor function?

The Instantiated Object

```
// Create an object using the new operator
let stefano = new Person();
console.log(stefano)
```

```
{
    name: 'Stefano Balietti
}
```

In the technical language the variable stefano is the live "instance" of the class Person.

Couldn't we directly create the object? What is the advantage of using a constructor function?

1. For complex object is faster because the blueprint is already loaded in memory
2. It allows for complex objects!

The Instantiated Object

```
// Create an object using the new operator
let stefano = new Person();
console.log(stefano)
```

```
{
    name: 'Stefano Balietti
}
```

In the technical language the variable stefano is the live "instance" of the class Person.

Couldn't we directly create the object? What is the advantage of using a constructor function?

1. For complex object is faster because the blueprint is already loaded in memory
2. It allows for complex objects! `stefano.sayHi();//I am Stefano Balietti`

A More Complex Person

```
class Person {
  constructor(name, year) {
    this.name = name;
    this.year = year;
  }
  sayHi(to) {
      return 'Hello ' + to + '. I am ' + this.name;
      ', and I was born in ' + this.year;
  }
}
```

Here the constructor is accepting input parameters to customize the instance.

A More Complex Person

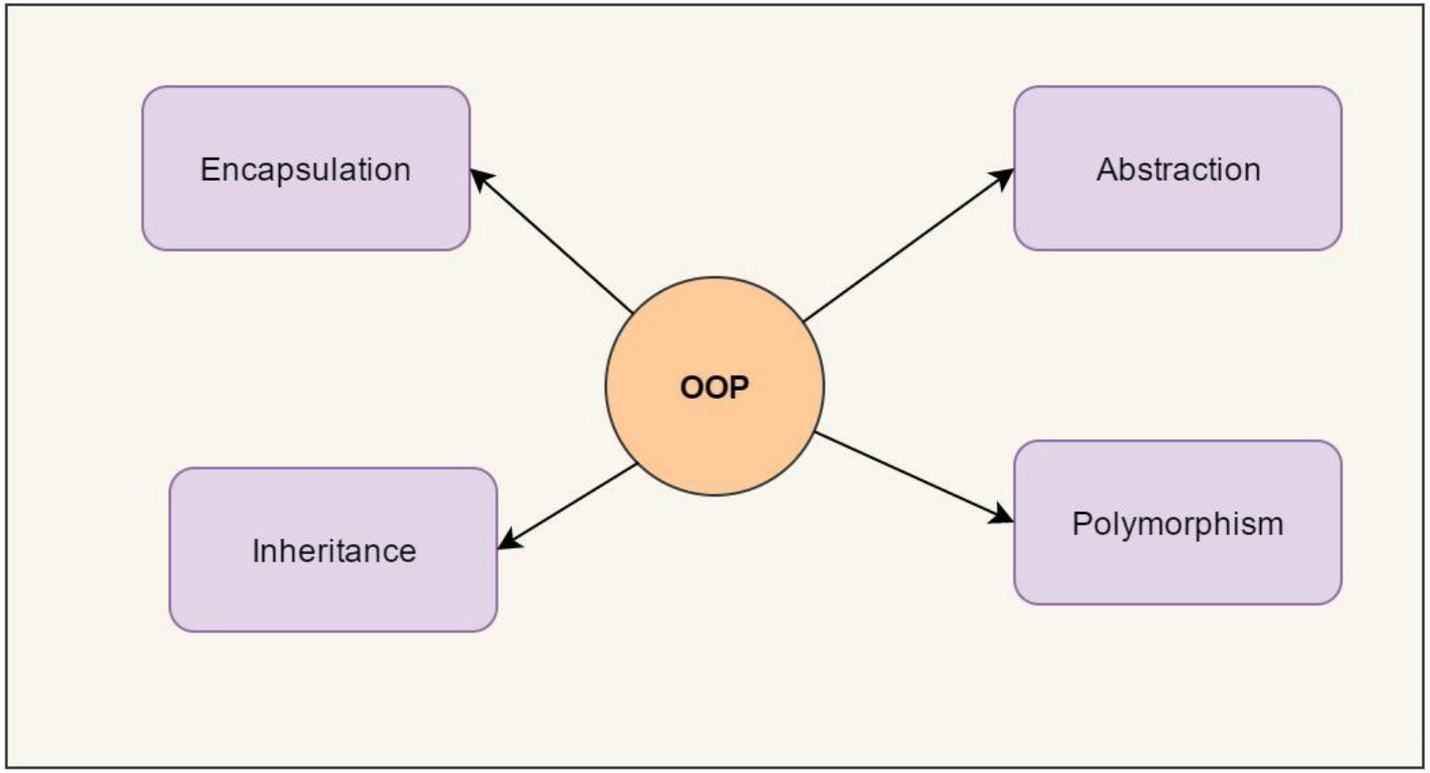
```
class Person {
  constructor(name, year) {
    this.name = name;
    this.year = year;
  }
  sayHi(to) {
      return 'Hello ' + to + '. I am ' + this.name;
      ', and I was born in ' + this.year;
  }
}
let brendan = new Person('Brendan', 1961);
brendan.sayHi('Stefano');
// 'Hello Stefano. I am Brendan and I was born in 1961'
```

Here the constructor is accepting input parameters to customize the instance.

Exercises

Part_2_OOP/classes.js

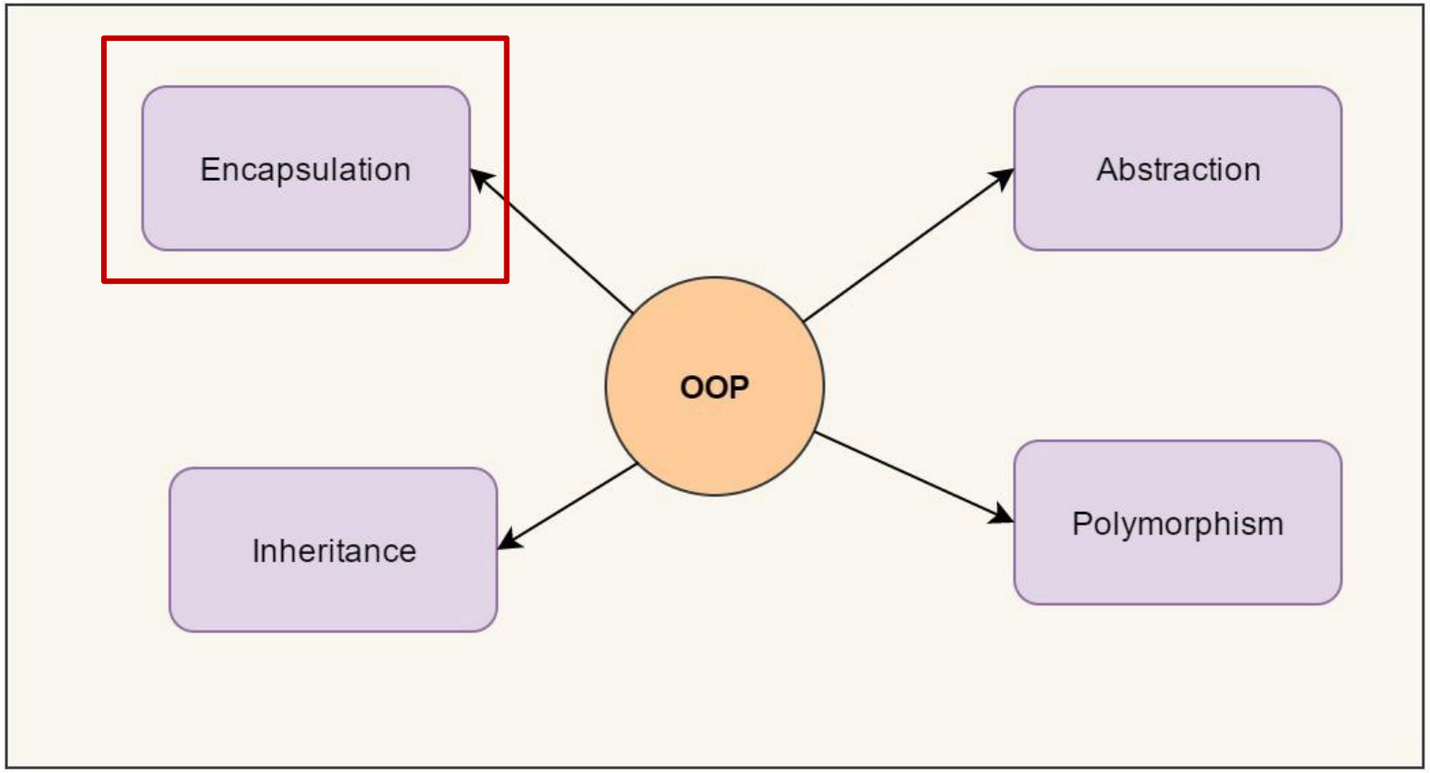
4 Pillars of OOP



Four Pillars of Object Oriented Programming

Picture source: https://medium.com/@cancerian0684/what-are-four-basic-principles-of-object-oriented-programming-645af8b43727

4 Pillars of OOP



Four Pillars of Object Oriented Programming

Picture source: https://medium.com/@cancerian0684/what-are-four-basic-principles-of-object-oriented-programming-645af8b43727

Encapsulation

- Encapsulation means that you can hide some of the methods and properties of a class declaring them as **private**, so they are *not* accessible outside of the class
- This prevents erroneous or malicious manipulation of the object by other entities
- It also reduces the complexity of the API for other external developers

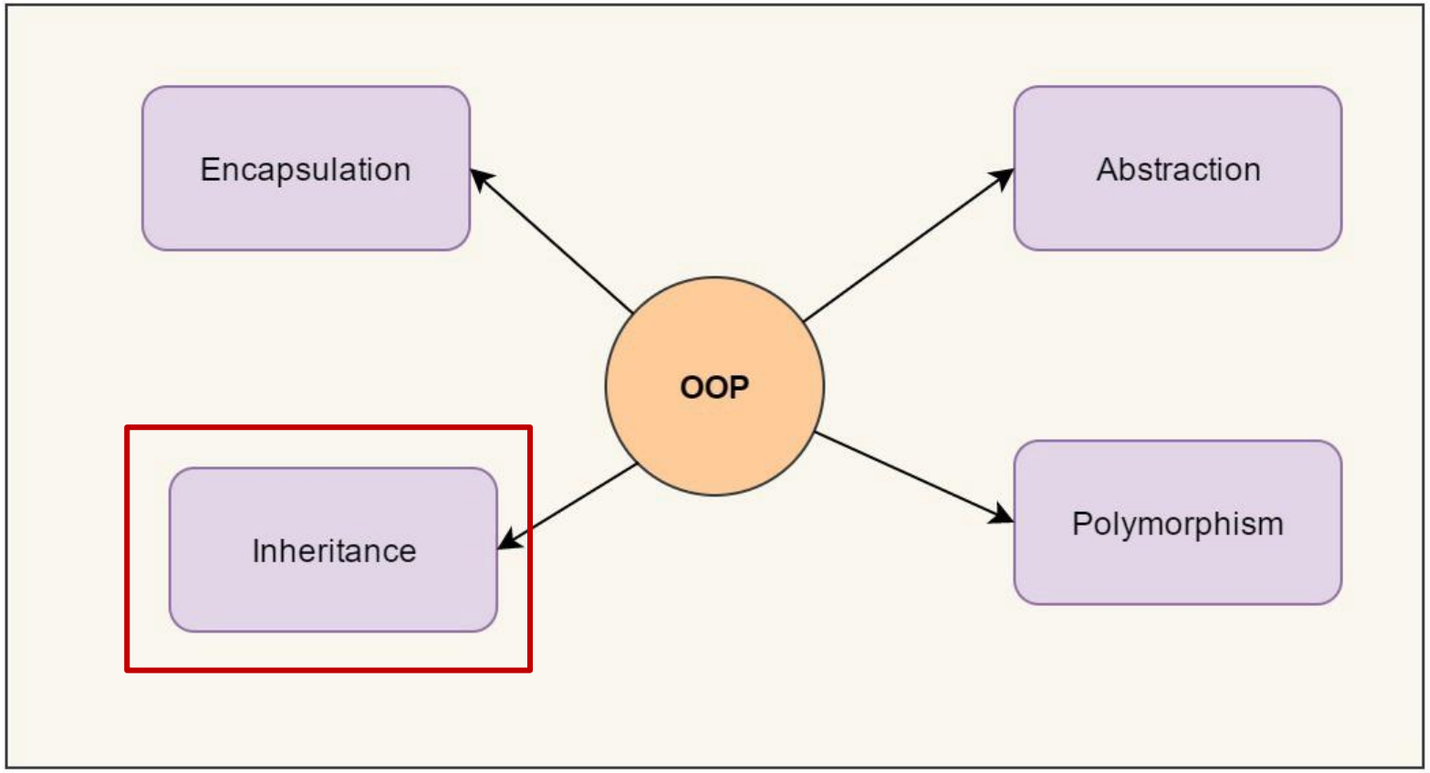
Encapsulation

- Encapsulation means that you can hide some of the methods and properties of a class declaring them as **private**, so they are *not* accessible outside of the class.
- This prevents erroneous or malicious manipulation of the object by other entities
- It also reduces the complexity of the API for other external developers

- *JavaScript does not natively support encapsulation*
- You can do it with **closures**, but it is complex topic, so we don't apply it here

- Here some references for the curious ones:
- https://medium.com/@luke_smaki/javascript-es6-classes-8a34b0a6720a
- https://www.intertech.com/Blog/encapsulation-in-javascript/

4 Pillars of OOP



Four Pillars of Object Oriented Programming

Picture source: https://medium.com/@cancerian0684/what-are-four-basic-principles-of-object-oriented-programming-645af8b43727

Inheritance

- Inheritance means that classes can share portion of codes with each other, by defining directional relationships of dependence, such as Parent/Child
- *JavaScript has native support for this feature*

OOP Pillar 1: Inheritance

```
class Liar extends Person {

    // We are going to add code here.

}
```

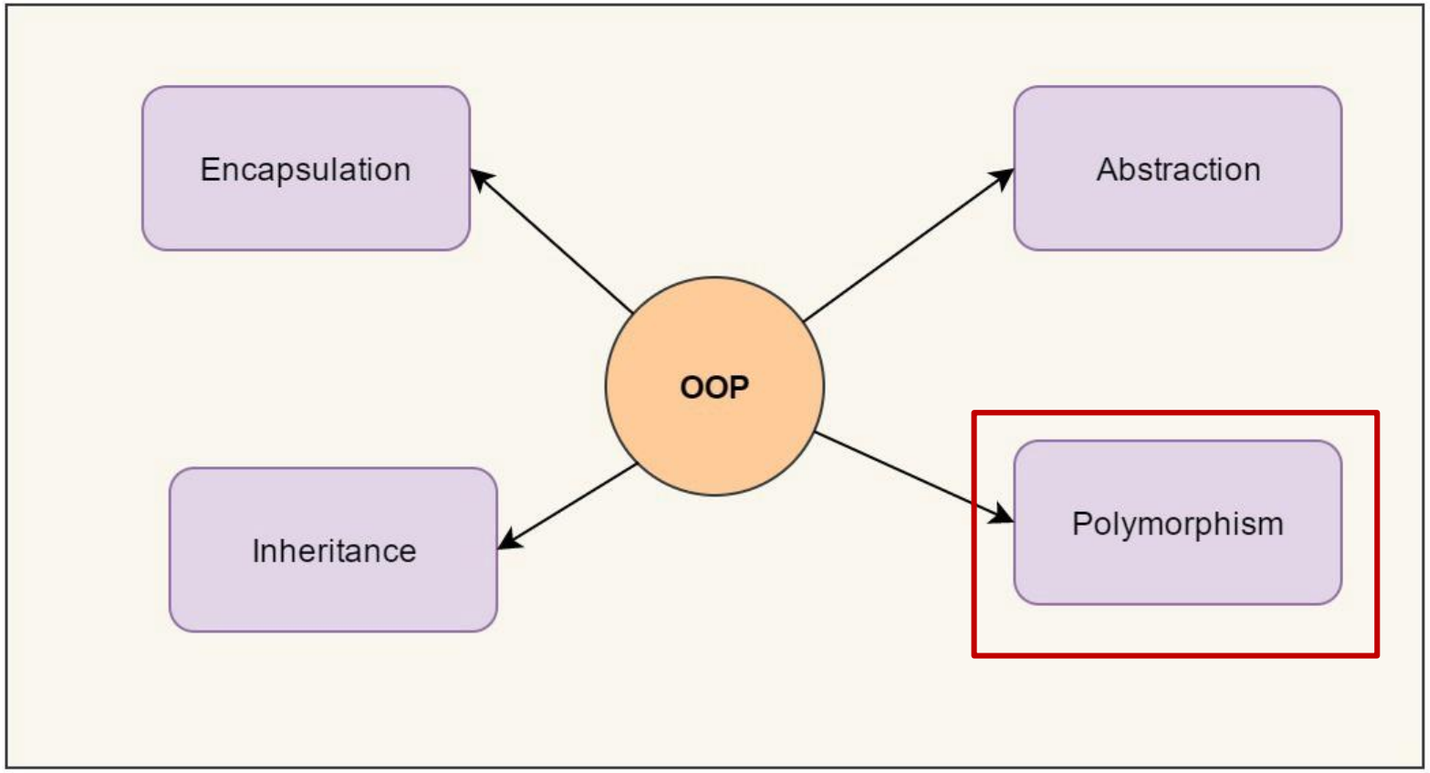
OOP Pillar 1: Inheritance

```
class Liar extends Person {

    // We are going to add code here.

}
```

Here we extend the previously defined Person class.

It means that the Liar class will have all the methods (including the constructor) and properties of the parent class.

4 Pillars of OOP



Four Pillars of Object Oriented Programming

Picture source: https://medium.com/@cancerian0684/what-are-four-basic-principles-of-object-oriented-programming-645af8b43727

Polymorphism

- Inheritance means that classes can share portion of codes with each other, by defining directional relationships of dependence, such as Parent/Child
- *JavaScript has native support for this feature*

- You can't really separate polymorphism from inheritance
- It means one get take many forms
- More specifically, the same method can morph into another one

OOP Pillar 2: Polymorphism

```
class Liar extends Person {

  sayHi(to) {
      return 'Hello ' + to + '. I am ' + this.name +
      ', and I was born in ' + (this.year + 15);
  }
}
```

OOP Pillar 2: Polymorphism

```
class Liar extends Person {

  sayHi(to) {
      return 'Hello ' + to + '. I am ' + this.name +
      ', and I was born in ' + (this.year + 15);
  }
}
```

Here we replace ("**override**") the body of the sayHi method with another one.

OOP Pillar 2: Polymorphism

```
class Liar extends Person {

  sayHi(to) {
      return 'Hello ' + to + '. I am ' + this.name +
      ', and I was born in ' + (this.year + 15);
  }
}
```

Here we replace ("**override**") the body of the sayHi method with another one.

This person is faking to be 15 younger than he or she is.

OOP Pillar 2: Polymorphism

```
class Liar extends Person {

  sayHi(to) {
      return 'Hello ' + to + '. I am ' + this.name +
      ', and I was born in ' + (this.year + 15);
  }
}
```

Here we replace ("**override**") the body of the sayHi method with another one.

This person is faking to be 15 younger than he or she is.

Can we control the degree of lying?

OOP Pillar 2: Polymorphism

```
class Liar extends Person {

  sayHi(to, degree) {
      return 'Hello ' + to + '. I am ' + this.name +
      ', and I was born in ' + (this.year + degree);
  }


}
```

15 can become a parameter

OOP Pillar 2: Polymorphism

```
class Liar extends Person {

  sayHi(to, degree) {
      return 'Hello ' + to + '. I am ' + this.name +
      ', and I was born in ' + (this.year + degree);
  }
```

Note for the Nerds! This type of polyphormism is called **"overloading":** the same method is accepting different combination of input parameters.

```
}
```

OOP Pillar 2: Polymorphism

```
class Liar extends Person {

  sayHi(to, degree) {
      return 'Hello ' + to + '. I am ' + this.name +
      ', and I was born in ' + (this.year + degree);
  }

}
```

Note for the Nerds! This type of polymorphism is called **"overloading":** the same method is accepting different combination of input parameters. However, *JavaScript does not support overloading* and the method is technically **overridden**, so that only one method sayHi exists in the end. Other programming languages will generate two methods, distinguishing them by their input parameters.

OOP Pillar 2: Polymorphism

```
class Liar extends Person {

  sayHi(to, degree) {
      return 'Hello ' + to + '. I am ' + this.name +
      ', and I was born in ' + (this.year + degree);
  }
}
```

However, it is kind of weird that who is invoking the sayHi method gets to decide the degree of lying. It should rather be a fixed property of the person.
What is another approach?

More Polymorphism

```
class Liar extends Person {
  constructor(name, year, degree) {
    this.name = name;
    this.year = year;
    this.degree = degree;
  }
  sayHi(to) {
      return 'Hello ' + to + '. I am ' + this.name +
      ', and I was born in ' + (this.year + this.degree);
  }
}
```

Here we create a new constructor with three input parameters

More Polymorphism

```
class Liar extends Person {
  constructor(name, year, degree) {
    this.name = name;
    this.year = year;
    this.degree = degree;
  }
  sayHi(to) {
      return 'Hello ' + to + '. I am ' + this.name +
      ', and I was born in ' + (this.year + this.degree);
  }
}
let liar = new Liar('Rosie Ruiz', 1953, 5);
liar.sayHi('Stefano'); // Hello Stefano. I am Rosie Ruiz and I was born in 1953
```

Here we create a new constructor with three input parameters

More Polymorphism

```
class Liar extends Person {
  constructo
    this.name
    this.year
    this.degr
  }
  sayHi(to)
      return                    this.name +
      ', and                    + this.degree);
  }
}
let liar = new Liar('Rosie Ruiz', 1953, 5);
liar.sayHi('Stefano'); // Hello Stefano. I am Rosie Ruiz and I was born in 1953
```

constructor with three

More Polymorphism

```
class Liar extends Person {
  constructor(name, year, degree) {
    this.name = name;
    this.year = year;
    this.degree = degree;
  }
  sayHi(to) {
      return 'Hello ' + to + '. I am ' + this.name +
      ', and I was born in ' + (this.year + this.degree);
  }
}
let liar = new Liar('Rosie Ruiz', 1953, 5);
liar.sayHi('Stefano'); // Hello Stefano. I am Rosie Ruiz and I was born in 1953
```

Here we create a new constructor with three input parameters

Can we do better?

More Polymorphism

```
class Liar extends Person {
  constructor(name, year, degree) {
    super(name, year);
    this.degree = degree;
  }
  sayHi(to) {
      return 'Hello ' + to + '. I am ' + this.name +
      ', and I was born in ' + (this.year + this.degree);
  }
}
```

super means the super class, that is, the *parent* class. Here we are invoking its constructor.

More Polymorphism

```
class Liar extends Person {
  constructor(name, year, degree) {
    super(name, year);
    this.degree = degree;
  }
  sayHi(to) {
      return 'Hello ' + to + '. I am
      ', and I was born in ' + (this
  }
}
```

super means the super class, that is, the *parent* class. Here we are invoking its constructor.

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

More Polymorphism

```
class Liar extends Person {
  constructor(name, year, degree) {
    super(name, year);
    this.degree = degree;
  }
  sayHi(to) {
      return 'Hello ' + to + '. I am
      ', and I was born in ' + (this
  }
}
```

super means the super class, that is, the *parent* class. Here we are invoking its constructor.

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

It's just two lines saved, what is the big advantage here?

More Polymorphism

```
class Liar extends Person {
  constructor(name, year, degree) {
    super(name, year);
    this.degree = degree;
  }
  sayHi(to) {
      return 'Hello ' + to + '. I am
      ', and I was born in ' + (this
  }
}
```

super means the super class, that is, the *parent* class. Here we are invoking its constructor.

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

It's just two lines saved, what is the big advantage here?

We <u>avoid code duplication</u>, this makes maintaining the code much easier.
Some constructors can set up many variables at the same time, even methods.

Exercises

Part_2_OOP/encapsulation.js
Part_2_OOP/inheritance_and_poly.js

More Polymorphism and Inheritance

```
class ConfusedLiar extends Liar {
  sayHi(to) {
      if (Math.random() > 0.5) return 'Who am I?';
      else return super.sayHi(to);
  }
}
```

More Polymorphism and Inheritance

```
class ConfusedLiar extends Liar {
  sayHi(to) {
      if (Math.random() > 0.5) return 'Who am I?';
      else return super.sayHi(to);
  }
}
```

We can extend extending classes.

More Polymorphism and Inheritance

```
class ConfusedLiar extends Liar {
  sayHi(to) {
      if (Math.random() > 0.5) return 'Who am I?';
      else return super.sayHi(to);
  }
}
```

We can use *super* to access any method of the parent class.
Here, the confused liar with probably 0.5 will not remember who he or she is (or is it just faking?), otherwise he or she will lie as before.

More Polymorphism and Inheritance

```
class ConfusedLiar extends Liar {
  sayHi(to) {
      if (Math.random() > 0.5) return 'Who am I?';
      else return super.sayHi(to);
  }
}
```

We can use *super* to access any method of the parent class.
Here, the confused liar with probably 0.5 will not remember who he or she is (or is it just faking?), otherwise he or she will lie as before.

We just 6 lines of code, we created a relatively complex personality thanks to inheritance and polymorphism: *a confused liar*! Isn't that amazing?

More Polymorphism and Inheritance

```
class ConfusedLiar extends Liar {
  sayHi(to) {
      if (Math.random() > 0.5) return 'Who am I?';
      else return super.sayHi(to);
  }
}
```

We can use *super* to access any method of the parent class.
Here, the confused liar with probably 0.5 will not remember who he or she is (or is it just faking?), otherwise he or she will lie as before.

We just 6 lines of code, we created a relatively complex personality thanks to inheritance and polymorphism: *a confused liar*! Isn't that amazing?

How can we do better?

More Polymorphism and Inheritance

```
class ConfusedLiar extends Liar {
  sayHi(to) {
      if (Math.random() > 0.5) return 'Who am I?';
      else return super.sayHi(to);
  }
}
```

The else word is *not* needed here.

More Polymorphism and Inheritance

```
class ConfusedLiar extends Liar {
  sayHi(to) {
      if (Math.random() > 0.5) return 'Who am I?';
      return super.sayHi(to);
  }
}
```

Two return statements are not needed either.

More Polymorphism and Inheritance

```
class ConfusedLiar extends Liar {
  sayHi(to) {
      return Math.random() > 0.5 ? 'Who am I?' : super.sayHi(to);
  }
}
```

With the ternary operator we saved one extra line without losing readability.
5 lines! Amazing!

Advanced Topic: Context

- The value of this is called **context**

```
sayHi(to) {
      return 'Hello ' + to + '. I am ' + this.name +
      ', and I was born in ' + (this.year + this.degree);
  }
```

Advanced Topic: Context

- The value of this is called **context**

```
sayHi(to) {
      return 'Hello ' + to + '. I am ' + this.name +
      ', and I was born in ' + (this.year + this.degree);
}
```

- In JavaScript, surprisingly, it is not fixed, but it changes dynamically depending on where the function is executed

Advanced Topic: Context

- The setTimeout function lets you execute some code after a given amount of time (here 2 seconds).

```
setTimeout(function() {
  // Code to be added

}, 2000);
```

Advanced Topic: Context

- If you use the setTimeout function inside our sayHi method the result might be disappointing.

```
setTimeout(function() {
   // Code to be added

}, 2000);
```

- The context, i.e., the value of this, inside the setTimeout function is the setTimeout function itself.
- This is generally terribly confusing to JS beginners

Advanced Topic: Context

```
sayHi(to) {
    setTimeout(function() {
      return 'Hello ' + to + '. I am ' + this.name +
      ', and I was born in ' + (this.year + this.degree);
    }, 2000);
}

sayHi('Stefano');
// Hello Stefano. I am undefined, and I was born in undefined.
```

Advanced Topic: Context

- You can circumvent this problem, by storing the value of this inside another variable.
- For historical reason, it is customary to call this variable *that*

```
sayHi(to) {
    let that = this;
    setTimeout(function() {
      return 'Hello ' + to + '. I am ' + that.name +
      ', and I was born in ' + (that.year + that.degree);
    }, 2000);
}
```

Advanced Topic: Context

- You can circumvent this problem, by storing the value of this inside another variable.
- For historical reason, it is customary to call this variable *that*

```
sayHi(to) {
    let that = this;
    setTimeout(function() {
      return 'Hello ' + to + '. I am ' + that.name +
      ', and I was born in ' + (that.year + that.degree);
    }, 2000);
}
```

- Alternatively, you can use an *arrow function* as a parameter of the setTimeout function

Advanced Topic: Arrow Functions

- Introduced in ES6
- They look weird
- They can shorten function definitions

```
// Standard way.
function() {
   return 'I am a normal function';
}

// Arrow functions.
() => {
   return 'I am an arrow function';
}
```

It isn't much shorter though…

Advanced Topic: Arrow Functions

- Introduced in ES6
- They look weird
- They can shorten function definitions

```
// Standard way.
function() {
   return 'I am a normal function';
}

// Arrow functions.
() => {
   return 'I am an arrow function';
}
```

It isn't much shorter though…There are conditions in which parentheses can be omitted.

Exercises

Part_2_OOP/4_this.js

Objected Oriented Cooperation Tournament

Part_2_OOP/5_final_exercise.js

But first the theory!