JavaScript

STUDIA PODYPLOMOWE
POLITECHNIKA BIAŁOSTOCKA


#5

Previously on JS

- Asynchronous JS
- Browser/Node features
- Promises
- Async/await
- Try/catch

OBJECT ORIENTED JAVASCRIPT

OOP

- State + behaviour pair
- State - field
- Behaviour - method
- Structures complex code

Object

CHARACTER

name	“Bob”
level	1
attack()	

CHARACTER

name "Bob"

level 1

attack()

CHARACTER

name "Will"

level 1

attack()

CHARACTER

name "Jake"

level 1

attack()

CHARACTER

name "Suzie"

level 1

attack()

CHARACTER

name "John"

level 1

attack()

Class

- A recipe / instruction how to create an object
- Allows creating objects with same state properties and behaviour

CLASS.

CHARACTER

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

CLASS

CHARACTER

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

CHARACTER

name “John”

level 1

attack()

CHARACTER

name “Bob”

level 1

attack()

CHARACTER

name “Suzie”

level 1

attack()

Inheritance

- Classes can extend other classes, they inherit their fields and methods
- They usually add something new to existing class

CLASS

CHARACTER

CLASS ARCHER

EXTENDS

CHARACTER

CLASS ARCHER
EXTENDS CHARACTER

```
constructor(name, level, dexterity) {
  super(name, level);
  this.dexterity = dexterity;
}
dexterity
poisonShot()
```

Encapsulation

- Restriction of access to internals of object
- We don’t want other parts of software to accidentally change something inside our objects

CLASS CHARACTER

```
constructor(name, level, hp) {
  this.name = name;
  this.level = level;
  this.hp = hp;
}
public name
public level
public hp
public attack() {
  this.calculateDamage()
}
private calculateDamage()
```

Interface

- Instruction how to interact with part of software - module / server / object
- No interface in JS yet!

CLASS CHARACTER

name

level

attack()

this.js

```
1
2  // This
3
4  const myCharacter = {
5    name: "Bob",
6    level: 1,
7    attack() {
8      const damage = this.level * 2;
9      console.log(`${this.name} deals ${damage} points of damage`)
10   }
11 }
12
13 myCharacter.attack() // "Bob deals 2 points of damage"
14
```

this

- A ‘variable’ that is created in every context (except arrow functions)
- It points to ‘context’ or ‘scope within context’
- The thing it points to depends only on how the context was created.

this.js

```
1
2  // This
3
4  const myCharacter1 = {
5    name: "Bob",
6    level: 1,
7    attack() {
8      const damage = this.level * 2;
9      console.log(`${this.name} deals ${damage} points of damage`)
10   }
11 }
12
13 myCharacter1.attack() // "Bob deals 2 points of damage"
14
15 const myCharacter2 = {
16   name: "John",
17   level: 1,
18   attack() {
19     const damage = this.level * 2;
20     console.log(`${this.name} deals ${damage} points of damage`)
21   }
22 }
23
24 myCharacter2.attack() // "John deals 2 points of damage"
25
```

this.js

```
1
2  // This
3
4  const myCharacter3 = {
5    name: "Suzie",
6    level: 1,
7    attack() {
8      const damage = this.level * 2;
9      console.log(`${this.name} deals ${damage} points of damage`)
10   }
11 }
12
13 myCharacter3.attack() // "Suzie deals 2 points of damage"
14
15 const myCharacter4 = {
16   name: "Jake",
17   level: 1,
18   attack() {
19     const damage = this.level * 2;
20     console.log(`${this.name} deals ${damage} points of damage`)
21   }
22 }
23
24 myCharacter4.attack() // "Jake deals 2 points of damage"
25
```

object-factory.js

```
1
2  // Object factory
3
4  function getCharacter(name, level) {
5    return {
6      name,
7      level,
8      attack() {
9        console.log(`${this.name} deals ${this.level * 2} points of damage`);
10     }
11   }
12 }
13
14 const myCharacter = getCharacter("Johnny", 1);
15 myCharacter.attack(); // "Johnny deals 2 points of damage"
16
```

object-factory.js

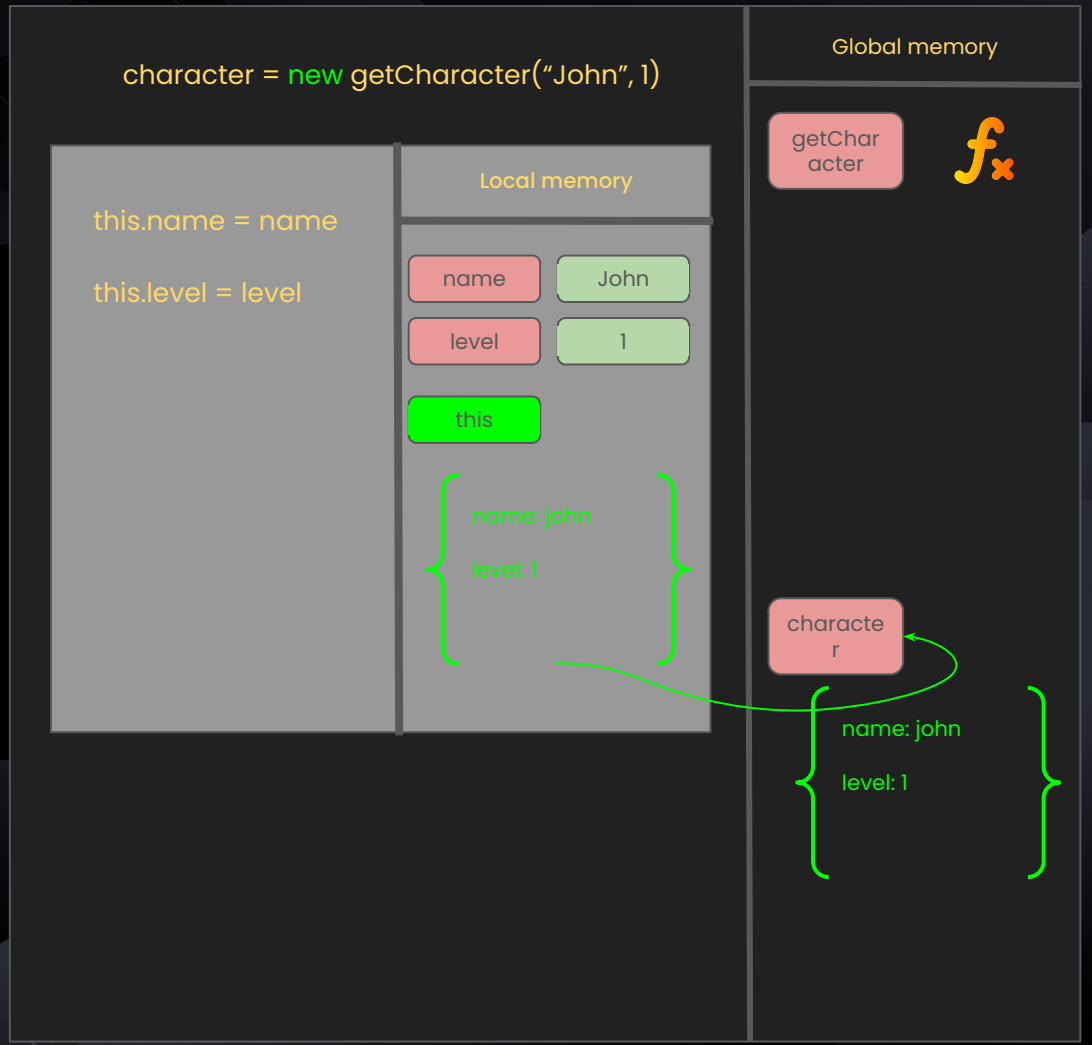
```
1
2  // Object factory
3
4  function getCharacter(name, level) {
5    this.name = name;
6    this.level = level;
7  }
8
9  const character = new getCharacter("John", 1);
10 console.log(character); // { name: "John", level: 1 }
11
```

new

1. Creates new object
2. Points this at the new object
3. Connects the object with the prototype where methods are stored
4. Returns the object

object-factory.js

```
1
2  // Object factory
3
4  function getCharacter(name, level) {
5    this.name = name;
6    this.level = level;
7  }
8
9  const character = new getCharacter("John", 1);
10 console.log(character); // { name: "John", level: 1 }
11
```

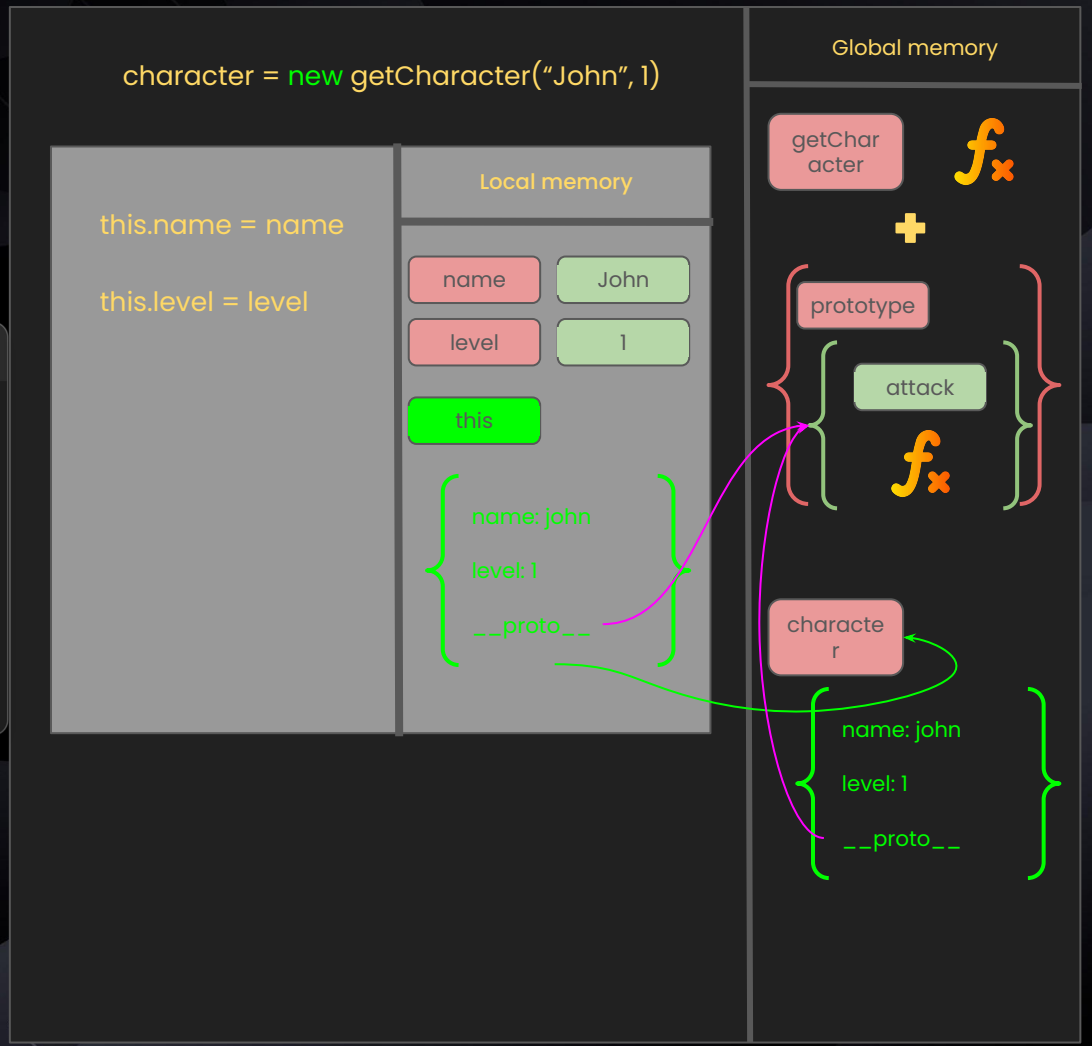


object-factory.js

```
1
2  // Object factory
3
4  function getCharacter(name, level) {
5    this.name = name;
6    this.level = level;
7  }
8
9  getCharacter.prototype.attack = function () { console.log('Attack!') }
10
11 const character = new getCharacter("John", 1);
12 character.attack(); // "Attack!"
13
```

object-factory.js

```
1
2  // Object factory
3
4  function getCharacter(name, level) {
5    this.name = name;
6    this.level = level;
7  }
8
9  getCharacter.prototype.attack = function () { console.log('Attack!') }
10
11 const character = new getCharacter("John", 1);
12 character.attack(); // "Attack!"
13
```



classes

class.js

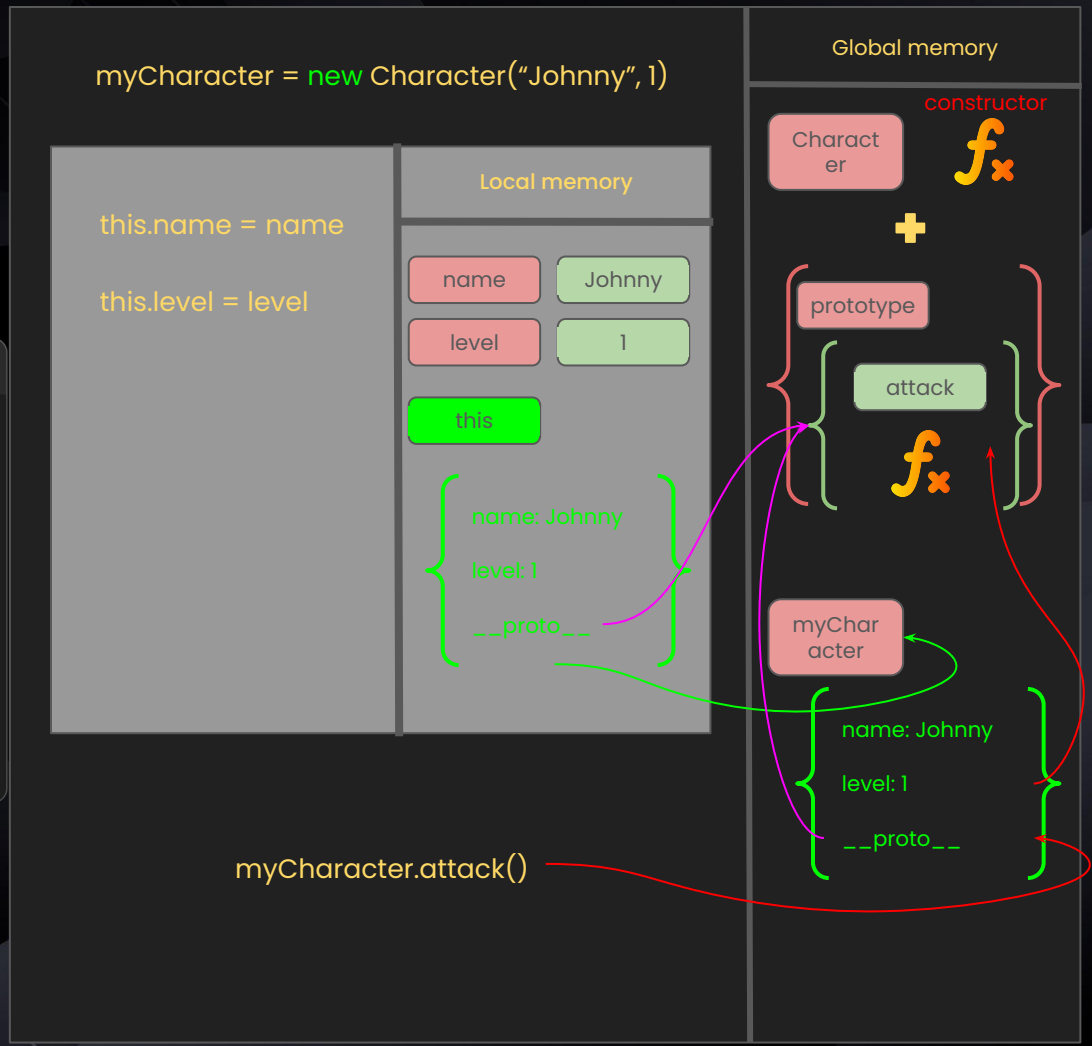
```
1
2  // Classes
3
4  class Character {
5    constructor(name, level) {
6      this.name = name;
7      this.level = level;
8    }
9    attack() {
10     console.log(`${this.name} deals ${this.level * 2} points of damage`);
11   }
12 }
13
14 const myCharacter = new Character("Johnny", 1);
15 myCharacter.attack(); // "Johnny deals 2 points of damage"
16
```

```
function getCharacter(name, level) {
  this.name = name;
  this.level = level;
}
```

```
getCharacter.prototype.attack = function () { console.log('Attack!') }
```

class.js

```
1
2  // Classes
3
4  class Character {
5    constructor(name, level) {
6      this.name = name;
7      this.level = level;
8    }
9    attack() {
10     console.log(`${this.name} deals ${this.level * 2} points of damage`);
11   }
12 }
13
14 const myCharacter = new Character("Johnny", 1);
15 myCharacter.attack(); // "Johnny deals 2 points of damage"
16
```



class

- function + object combo
- The ‘function’ part is the constructor
- The ‘object’ part stores all methods under the name ‘prototype’

inheritance.js

```
1
2  // Inheritance
3
4  class Archer extends Character {
5    constructor(name, level, dexterity) {
6      super(name, level);
7      this.dexterity = dexterity;
8    }
9
10   poisonShot() {
11     const dmgModifier = this.level * 2;
12     console.log(`You have been poisoned. You will lose ${dmgModifier} HP each turn!`);
13   }
14 }
15
16 const myArcher = new Archer("Bob", 10, 50);
17 myArcher.attack(); // "Bob deals 20 points of damage"
18 myArcher.poisonShot(); // "You have been poisoned. You will lose 20 HP each turn!"
19
```

class.js

```
// Classes

class Character {
  constructor(name, level) {
    this.name = name;
    this.level = level;
  }
  attack() {
    console.log(`${this.name} deals ${this.level * 2} points of damage`);
  }
}

const myCharacter = new Character("Johnny", 1);
myCharacter.attack(); // "Johnny deals 2 points of damage"

```

inheritance.js

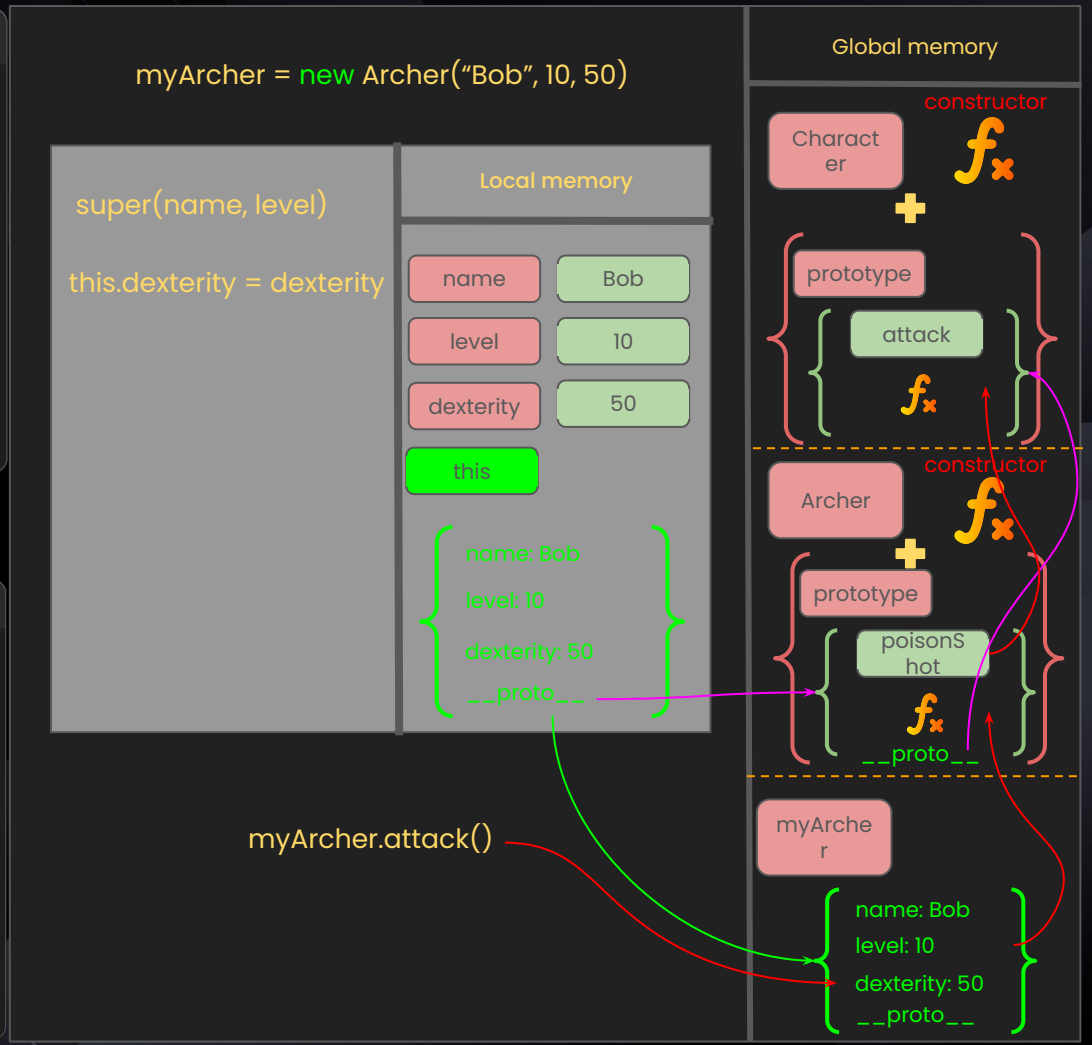
```
// Inheritance

class Archer extends Character {
  constructor(name, level, dexterity) {
    super(name, level);
    this.dexterity = dexterity;
  }

  poisonShot() {
    const dmgModifier = this.level * 2;
    console.log(`You have been poisoned. You will lose ${dmgModifier} HP each turn!`);
  }
}

const myArcher = new Archer("Bob", 10, 50);
myArcher.attack(); // "Bob deals 20 points of damage"
myArcher.poisonShot(); // "You have been poisoned. You will lose 20 HP each turn!"

```



super

- When extending other class you need to use as a function in constructor to be able to use this
- A way to reference parent class

Instanceof

- Instanceof allows us to check if object is of certain class
- Takes inheritance into account
- Returns boolean

instanceof.js

```
1
2  // Instanceof
3
4  class Car {
5    constructor() {
6      this.wheels = 4;
7    }
8  }
9
10 class Bike { }
11
12 const skoda = new Car();
13
14 console.log(skoda instanceof Car); // true
15 console.log(skoda instanceof Bike); // false
16
```

instanceof.js

```
1
2  // Instanceof
3
4  class Car {
5    constructor() {
6      this.wheels = 4;
7    }
8  }
9
10 class ElectricCar extends Car {
11   constructor() {
12     super();
13
14     this.battery = 10000
15   }
16 }
17
18 const tesla = new ElectricCar();
19
20 console.log(tesla instanceof Car); // true
21
```

Public instance field

- Usually omitted but can be included for more self documenting code


public-field.js
// Public field
class Car {
wheels;
constructor() {
this.wheels = 4;
}
checkStatus() {
console.log(`Got ${this.wheels} wheels`);
}
}
const myCar = new Car();
myCar.checkStatus(); // "Got 4 wheels"

public-field.js

```
1
2  // Public field
3
4  class Car {
5    wheels = 4;
6
7    constructor(horsePower) {
8      this.horsePower = horsePower;
9    }
10 }
11
12 const myCar = new Car(140);
13 const mySecondCar = new Car(200);
14
15 console.log(myCar); // { wheels: 4, horsePower: 140 }
16 console.log(mySecondCar); // { wheels: 4, horsePower: 200 }
17
```

Static fields and methods

- Fields and methods that belong to class - not the object created with the class
- Usually used to add utility methods to class itself

static-field.js

```
1
2  // Static fields and methods
3
4  class Car {
5    static sound = "Beep Beep!"
6
7    static makeSound() {
8      console.log(this.sound);
9    }
10
11   makeNonStaticSound() {
12     console.log(this.sound);
13   }
14 }
15
16 console.log(Car.sound); // "Beep Beep!"
17 Car.makeSound(); // "Beep Beep!"
18
19 const myCar = new Car();
20 myCar.makeNonStaticSound(); // undefined
21
```

Private fields and methods

- Quite new addition to JS
- Allows to natively encapsulate data inside class/object
- Add # before the name of field

private-field.js

```
1
2  // Private fields
3
4  class Car {
5    #speed;
6
7    constructor(horsepower) {
8      this.horsepower = horsepower;
9      this.#speed = 10 * horsepower;
10   }
11
12   getSpeed() {
13     return this.#speed;
14   }
15 }
16
17 const myCar = new Car(10);
18 console.log(myCar.getSpeed()); // 100
19 console.log(myCar.#speed);
20 // SyntaxError: Private field '#speed' must be declared in an enclosing class
21
```

getters and setters

- Methods that are used like normal properties
- Allow to add extra logic while setting values or add layer of encapsulation

getter-setter.js

```
1
2  // getters and setters
3
4  class Character {
5    constructor(strength) {
6      this.strength = strength;
7    }
8
9    get damage() {
10     return this.strength * 5;
11   }
12
13   set damage(damage) {
14     this.strength = Math.floor(damage / 5);
15   }
16 }
17
18 const myCharacter = new Character(10);
19 console.log(myCharacter.damage); // 50
20
21 myCharacter.damage = 10;
22 console.log(myCharacter.damage); // 10
23
24
```


THE
END