Javascript Repetition Teil 2:

object, and access the property / method on it, without affecting the original.

```
const foo = "bar";
foo.length; // 3
foo === "bar"; // true
```

In the above example, to access the property `length`, JavaScript autoboxed `foo` into a wrapper object, access the wrapper object's `length` property, and discards it afterwards. This is done without affecting `foo` (`foo` is still a primitive string).

This also explains why JavaScript doesn't complain when you try to assign a property to a primitive type, because the assignment is done on that temporary wrapper object, not the primitive type itself.

```
const foo = 42;
foo.bar = "baz"; // Assignment done on temporary wrapper object
foo.bar; // undefined
```

It will complain if you try this with a primitive type which does not have a wrapper object, such as `undefined` or `null`.

```
const foo = null;
foo.bar = "baz"; // Uncaught TypeError: Cannot set property 'bar' of null
```

- **General Behavior of an Object**
  - Objects are similar to dictionaries
  - Every reference type inherits from Object
  - Copy by reference
  - Call by reference
  - Compared by reference

  - *Auto-Unboxing by calling .valueOf()*

Strict Mode

- **Indicates that the code should be executed in "strict mode"**
  - It's a literal expression, ignored by earlier versions of JavaScript
  - Declared at the beginning of a JavaScript file, or a JavaScript function
- ***Strict Mode* converts mistakes into errors**
  - The following condition will throw an error:
    - Assigning a
      - non-writable property
      - a getter-only property
      - a non-existing property
      - a non-existing variable
      - a non-existing object
  - Prohibits keywords (e.g. with())
  - **this can be undefined (or null), if function isn't called in an objects context**
- **EcmaScript 6 (2015) Classes/Methods/Modules are executed in strict mode**

```
function helloWorld(a){
    console.log(a || "No Data");
}

function helloWorld2(){
    console.log(arguments[0]);
}

var sayHello = function(fnOutput)
{
    fnOutput("Hallo")
};

sayHello(helloWorld);
sayHello(helloWorld2);

Hallo
Hallo
```

- **Object is instantiated by using new keyword**
- **BUT a JavaScript class is also a *function***
  - Can also be called regularly without new
  - Context doesn't change; global context is injected

```
function House (color) {                    // class definition, constructor function
  this.facadeColor = color;
  this.paint = function(newColor) {
    this.facadeColor = newColor;
  };
}
let whiteHouse = House("white");            // used without new operator
```

facadeColor property and paint() method are written into the **global context**!

- **A method is called by declaring the object as context**
- **BUT a JavaScript method is also a *function***
  - Can also be called regularly without the context
  - Context doesn't change; global context is injected

```
function House (color) {                    // class definition, constructor function
  this.facadeColor = color;
  this.paint = function(newColor) {
    this.facadeColor = newColor;
  };
}
let whiteHouse = new House("white");
let paintWhiteHouse = whiteHouse.paint;     // copy pointer of function paint
paintWhiteHouse();                          // call function without object (without context)
```

facadeColor property is written into the **global context**!

Context Code Example in JavaScript ES6

```
class House {                                 // class definition
    constructor(color) {                      // constructor definition
        this.facadeColor = color;             // property definition
    }
    paint (newColor) {                        // method definition
        this.facadeColor = newColor;          // do more paint stuff here, colorize windows, etc…
    };
}

let whiteHouse = new House("white");          // whiteHouse represents an instance (House object)
whiteHouse.paint("beige");
```

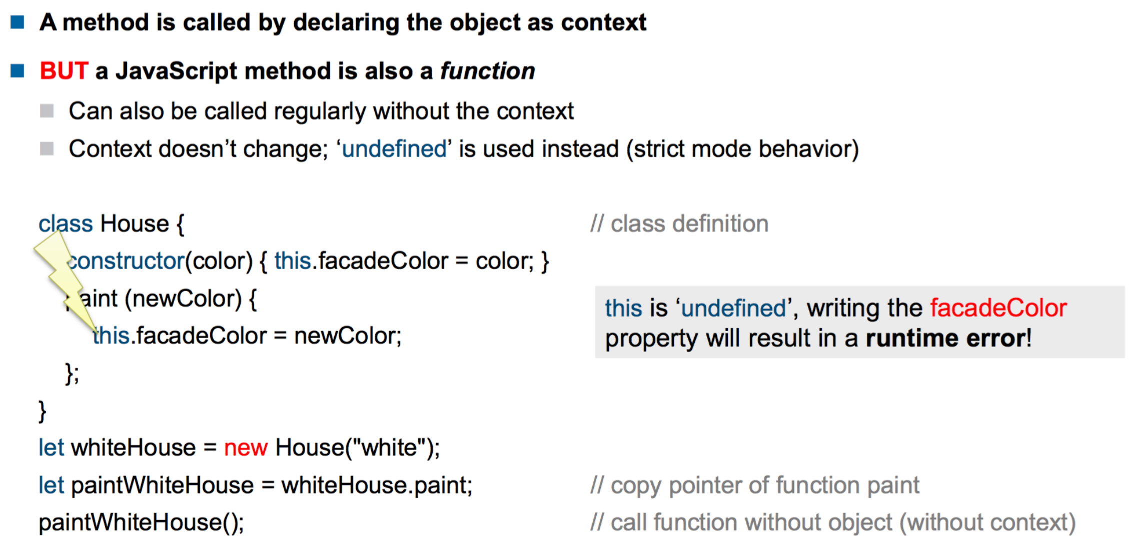
Context Code Example in JavaScript ES6

```
class House {                                 // class definition
    constructor(color) {                      // constructor definition
        this.facadeColor = color;             // property definition
    }
    paint (newColor) {                        // method definition
        this.facadeColor = newColor;          // do more paint stuff here, colorize windows, etc…
    };
}

let whiteHouse = new House("white");          // whiteHouse represents an instance (House object)
whiteHouse.paint("beige");
```

“Abnormal” Context behavior ES6 I

- **Object is instantiated by using new keyword**
- **BUT a JavaScript class is also a *function***
  - typeof operator returns “function”
- **class constructors cannot be invoked without 'new’**
  - Results in a runtime error
  - More deterministic than ES5 approach

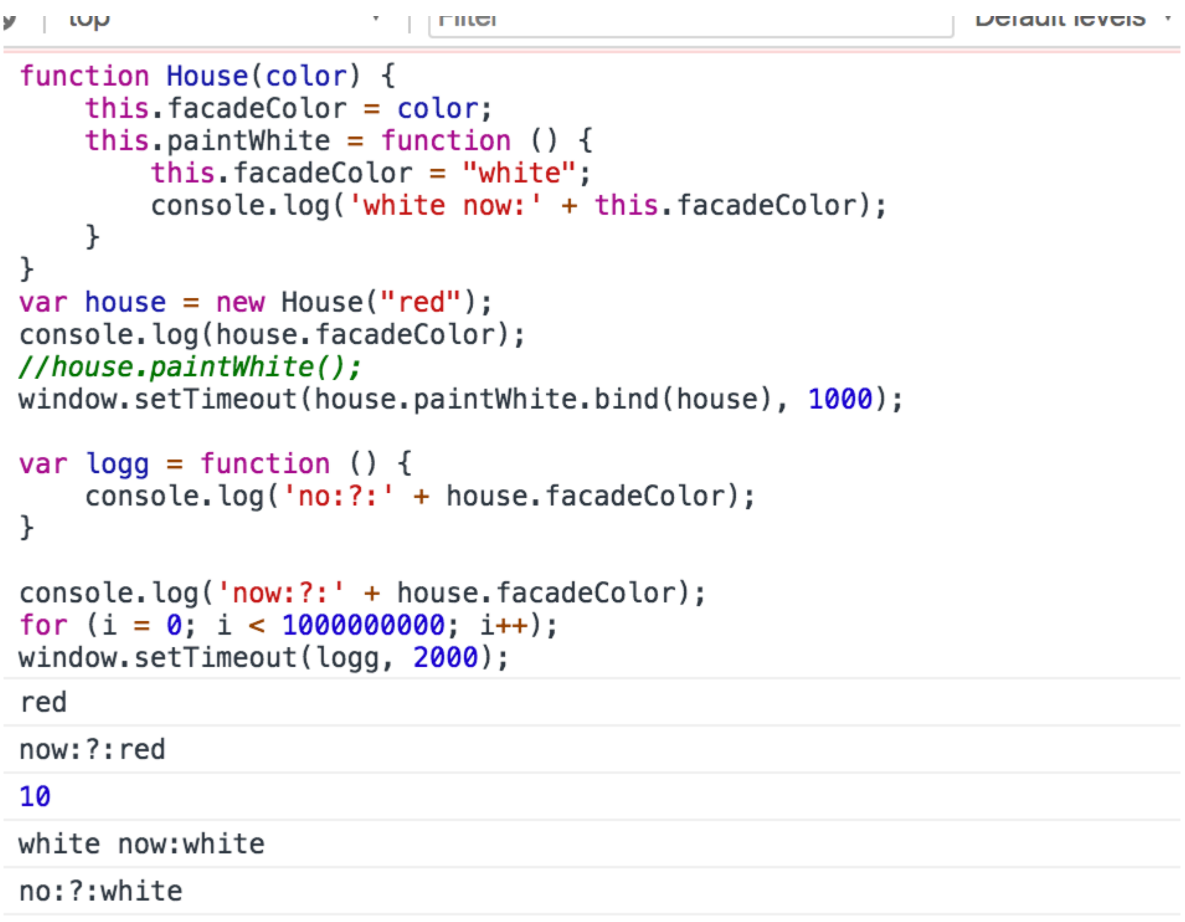


- **A method is called by declaring the object as context**
- **BUT a JavaScript method is also a *function***
  - Can also be called regularly without the context
  - Context doesn't change; 'undefined' is used instead (strict mode behavior)

```
class House {                                            // class definition
   constructor(color) { this.facadeColor = color; }
   paint (newColor) {
      this.facadeColor = newColor;
   };
}
let whiteHouse = new House("white");
let paintWhiteHouse = whiteHouse.paint;                  // copy pointer of function paint
paintWhiteHouse();                                       // call function without object (without context)
```

this is 'undefined', writing the facadeColor property will result in a **runtime error**!

Using Bind:

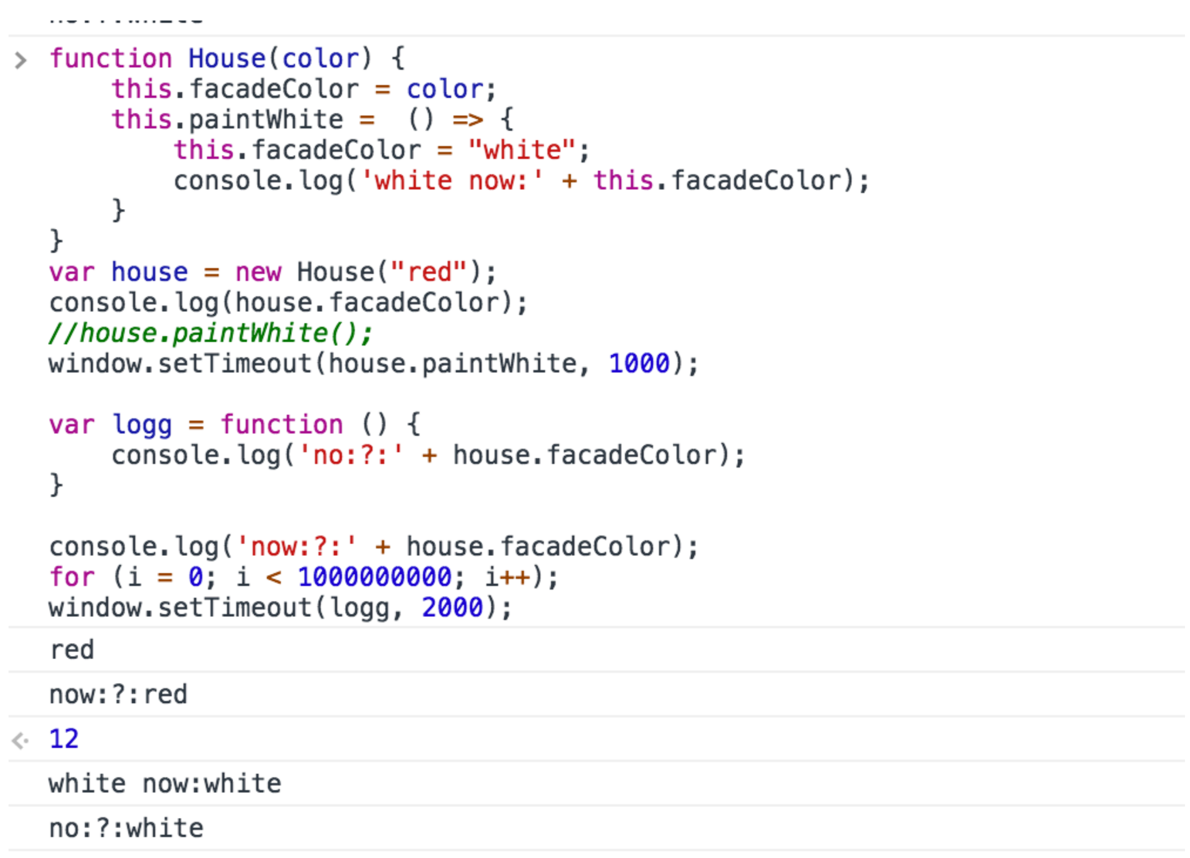


```
function House(color) {
    this.facadeColor = color;
    this.paintWhite = function () {
        this.facadeColor = "white";
        console.log('white now:' + this.facadeColor);
    }
}
var house = new House("red");
console.log(house.facadeColor);
//house.paintWhite();
window.setTimeout(house.paintWhite.bind(house), 1000);

var logg = function () {
    console.log('no:?:' + house.facadeColor);
}

console.log('now:?:' + house.facadeColor);
for (i = 0; i < 1000000000; i++);
window.setTimeout(logg, 2000);
red
now:?:red
10
white now:white
no:?:white
```

Arrow:



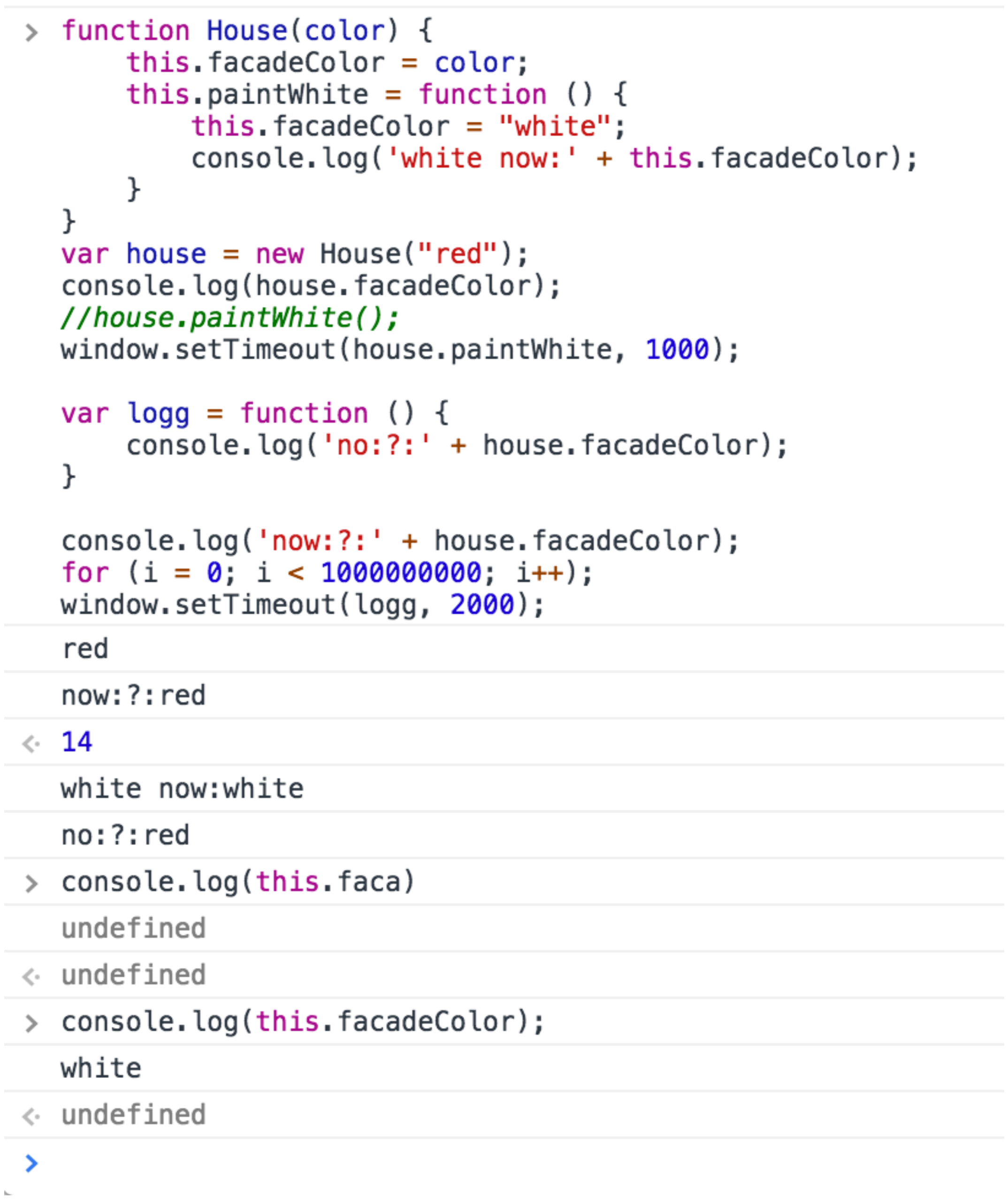
```
function House(color) {
    this.facadeColor = color;
    this.paintWhite =  () => {
        this.facadeColor = "white";
        console.log('white now:' + this.facadeColor);
    }
}
var house = new House("red");
console.log(house.facadeColor);
//house.paintWhite();
window.setTimeout(house.paintWhite, 1000);

var logg = function () {
    console.log('no:?:' + house.facadeColor);
}

console.log('now:?:' + house.facadeColor);
for (i = 0; i < 1000000000; i++);
window.setTimeout(logg, 2000);
```

```
red
now:?:red
12
white now:white
no:?:white
```

Weder Arrow noch bind:



```
> function House(color) {
      this.facadeColor = color;
      this.paintWhite = function () {
          this.facadeColor = "white";
          console.log('white now:' + this.facadeColor);
      }
  }
  var house = new House("red");
  console.log(house.facadeColor);
  //house.paintWhite();
  window.setTimeout(house.paintWhite, 1000);

  var logg = function () {
      console.log('no:?:' + house.facadeColor);
  }

  console.log('now:?:' + house.facadeColor);
  for (i = 0; i < 1000000000; i++);
  window.setTimeout(logg, 2000);
  red
  now:?:red
<· 14
  white now:white
  no:?:red
> console.log(this.faca)
  undefined
<· undefined
> console.log(this.facadeColor);
  white
<· undefined
>
```

window wird versaut (this = window!);

As a recommendation…

…use *Closures (or Lambdas) with scoped variables* or *bind()* if you have

But there are several side effects when applying Closures

- **Access to modified Closure**
- **Breaks some native language features**