Lesson - 27

Constructor, "new" operator

SkillUp by Dobrea Vladislav

Lesson Plan

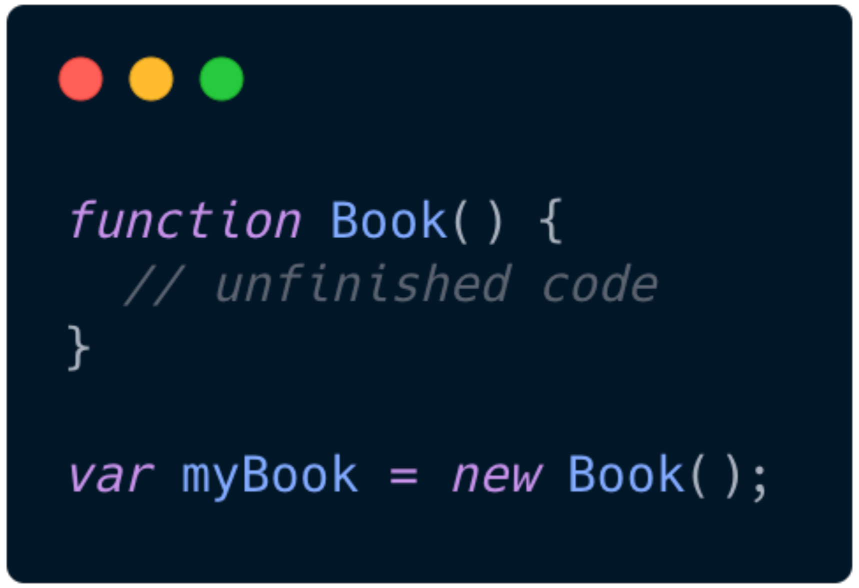
- HW Review
- What is a constructor ?
- Existing constructors in JavaScript

Constructors

The regular `{...}` syntax allows to create one object. But often we need to create many similar objects, like multiple users or menu items and so on.

Constructors are like regular functions, but we use them with the `new` keyword. There are two types of constructors: built-in constructors such as `Array` and `Object`, which are available automatically in the execution environment at runtime; and custom constructors, which define properties and methods for your own type of object.

A **constructor** is useful when you want to create multiple similar objects with the same properties and methods. It’s a convention to capitalize the name of constructors to distinguish them from regular functions.



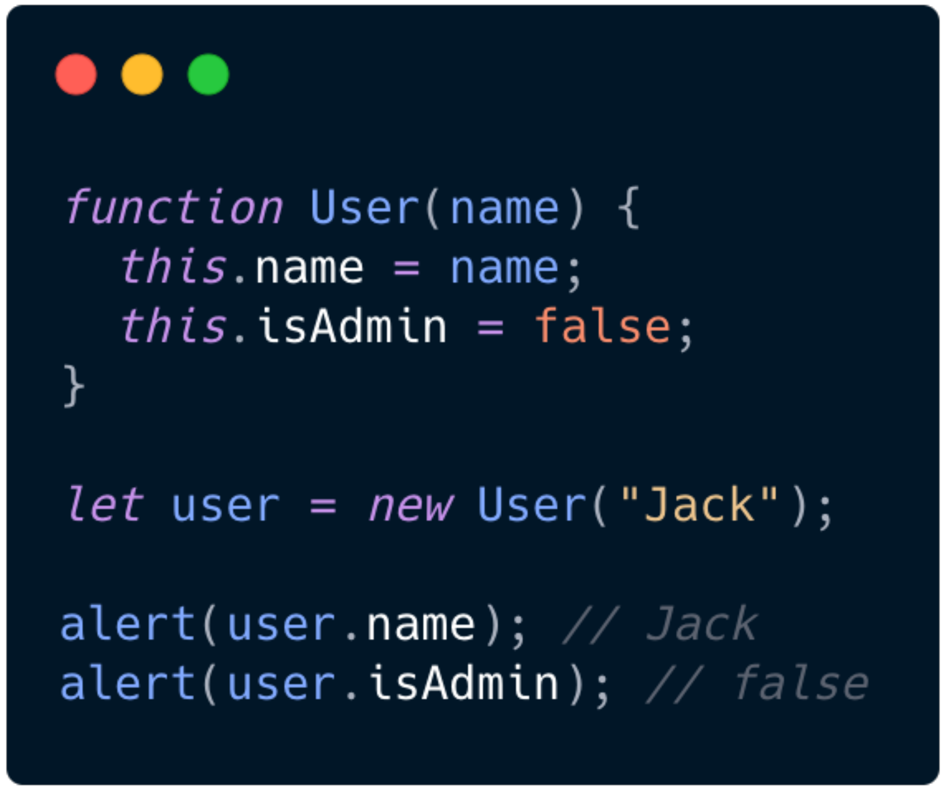
```
function Book() {
  // unfinished code
}

var myBook = new Book();
```

Constructor function

Constructor functions technically are regular functions. There are two conventions though:

1. They are named with capital letter first.
2. They should be executed only with `"new"` operator.

For instance:

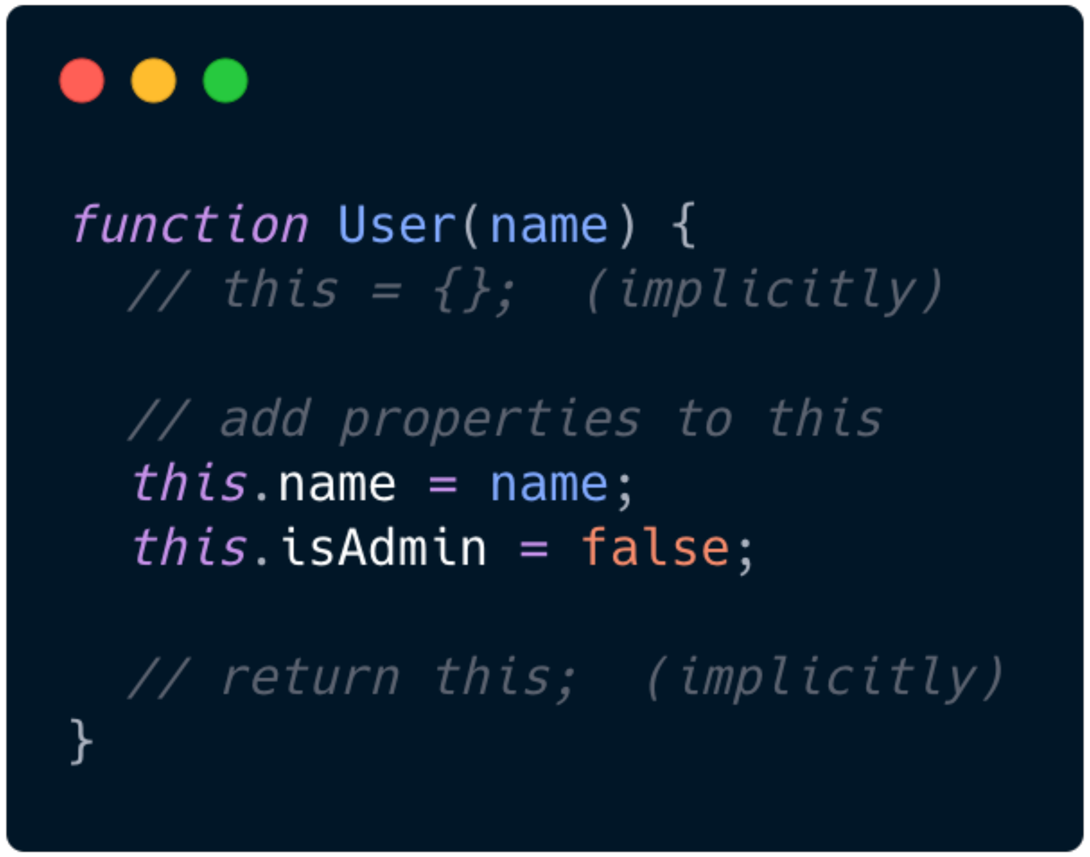


```javascript
function User(name) {
  this.name = name;
  this.isAdmin = false;
}

let user = new User("Jack");

alert(user.name); // Jack
alert(user.isAdmin); // false
```

Constructor function

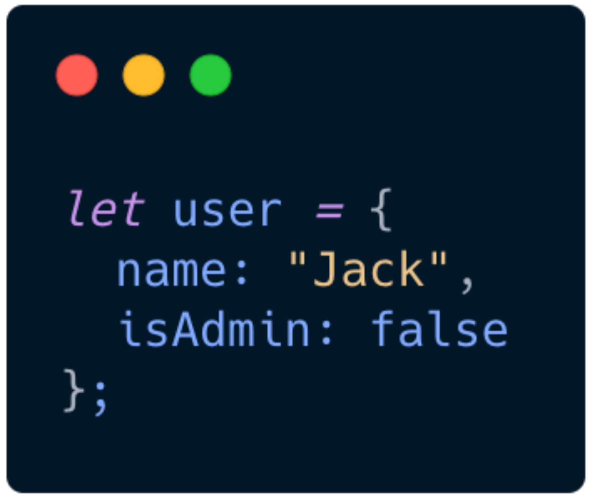
When a function is executed with `new`, it does the following steps:

1. A new empty object is created and assigned to `this`.
2. The function body executes. Usually it modifies `this`, adds new properties to it.
3. The value of `this` is returned.



```
function User(name) {
  // this = {};  (implicitly)

  // add properties to this
  this.name = name;
  this.isAdmin = false;

  // return this;  (implicitly)
}
```

So `let user = new User("Jack")` gives the same result as:



```
let user = {
  name: "Jack",
  isAdmin: false
};
```

Constructor

Now if we want to create other users, we can call `new User("Ann")`, `new User("Alice")` and so on. Much shorter than using literals every time, and also easy to read.

That’s the main purpose of constructors – to implement reusable object creation code.

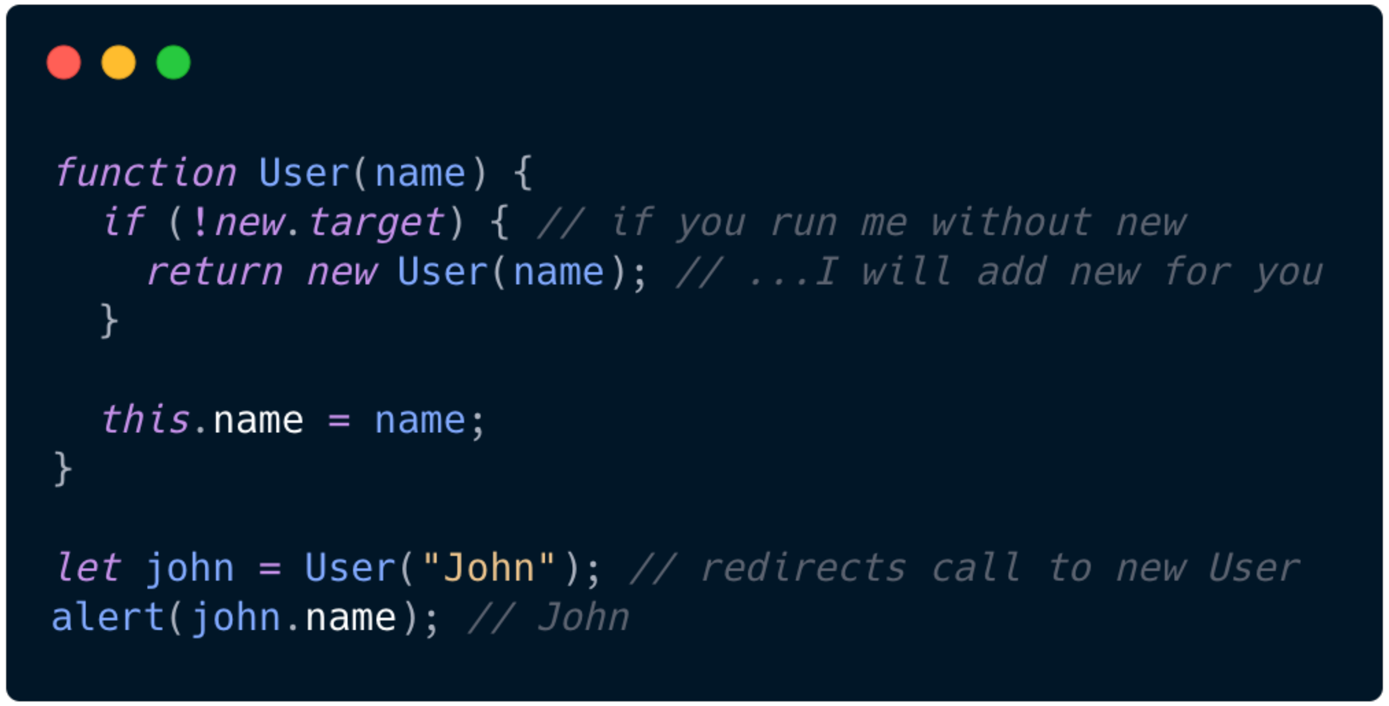
Let’s note once again – technically, any function can be used as a constructor. That is: any function can be run with `new`, and it will execute the algorithm above. The “capital letter first” is a common agreement, to make it clear that a function is to be run with `new`.

Constructor mode test: new.target

Inside a function, we can check whether it was called with `new` or without it, using a special `new.target` property.

That can be used inside the function to know whether it was called with `new`, “in constructor mode”, or without it, “in regular mode”.

We can also make both `new` and regular calls to do the same, like this:



```
function User(name) {
  if (!new.target) { // if you run me without new
    return new User(name); // ...I will add new for you
  }

  this.name = name;
}

let john = User("John"); // redirects call to new User
alert(john.name); // John
```

This approach is sometimes used in libraries to make the syntax more flexible. So that people may call the function with or without `new`, and it still works.

Probably not a good thing to use everywhere though, because omitting `new` makes it a bit less obvious what’s going on. With `new` we all know that the new object is being created.

Return from constructors

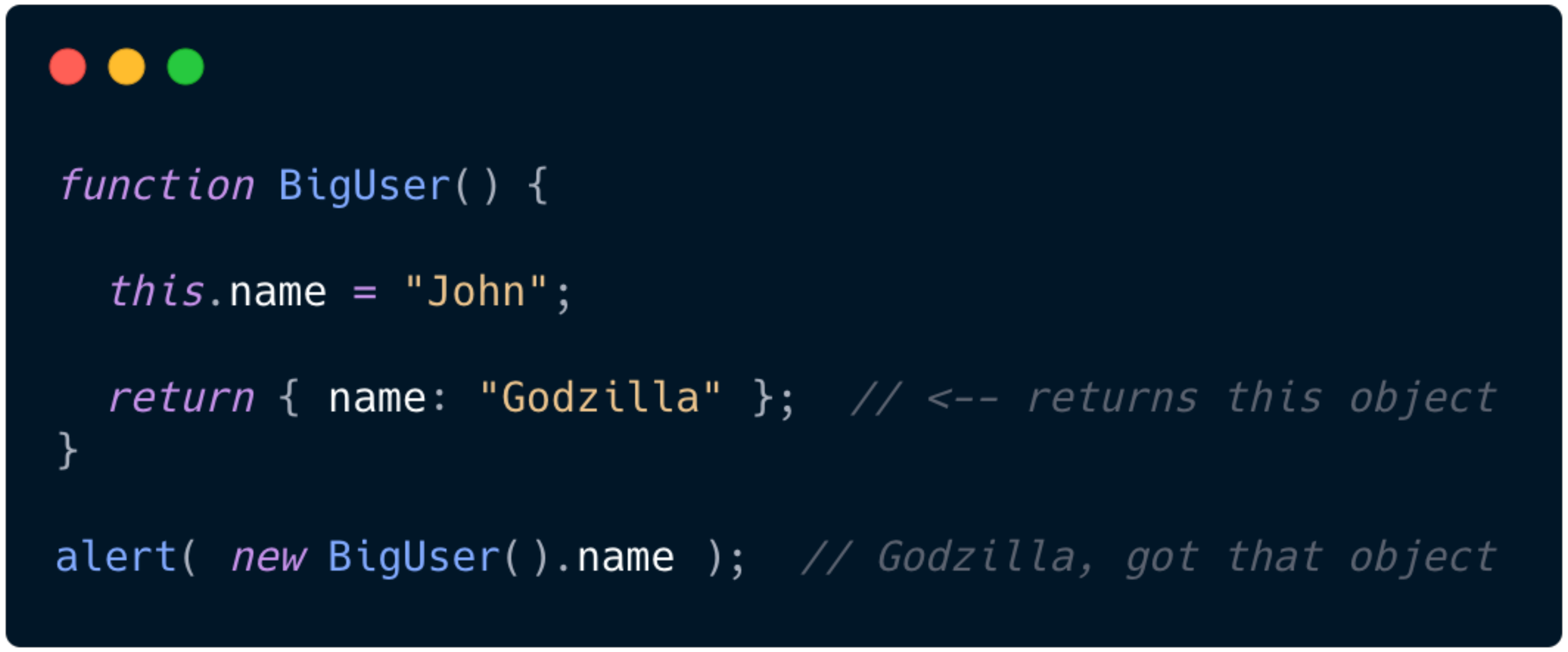
Usually, constructors do not have a return statement. Their task is to write all necessary stuff into this, and it automatically becomes the result.

But if there is a return statement, then the rule is simple:

- If return is called with an object, then the object is returned instead of this.
- If return is called with a primitive, it's ignored.

In other words, return with an object returns that object, in all other cases this is returned.

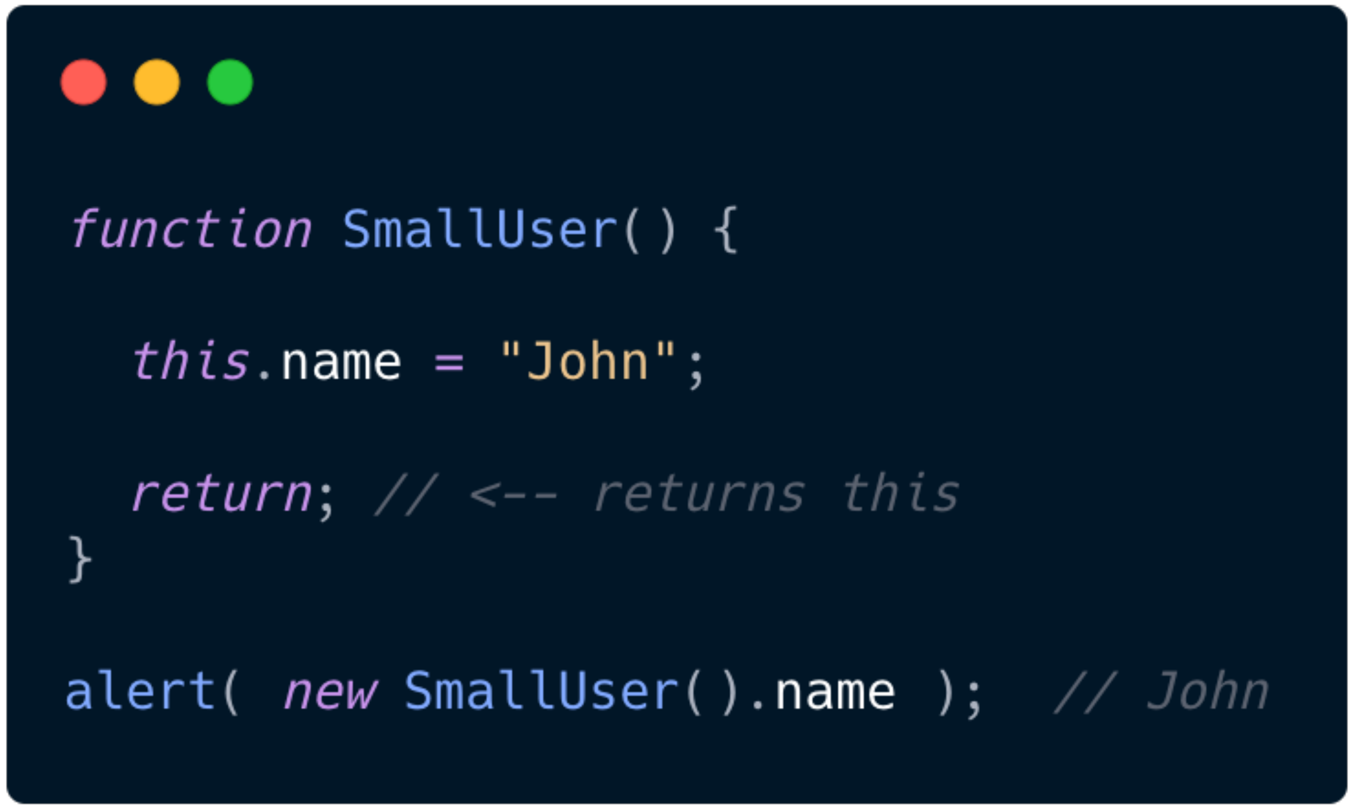
For instance, here return overrides this by returning an object:



```
function BigUser() {

  this.name = "John";

  return { name: "Godzilla" };  // <-- returns this object
}

alert( new BigUser().name );  // Godzilla, got that object
```

Return from constructors

And here’s an example with an empty return (or we could place a primitive after it, doesn’t matter):



```
function SmallUser() {

  this.name = "John";

  return; // <-- returns this
}

alert( new SmallUser().name );  // John
```

Usually constructors don’t have a return statement. Here we mention the special behavior with returning objects mainly for the sake of completeness.

Omitting the parentheses

By the way, we can omit parentheses after `new`, if it has no arguments:

```
let user = new User; // <-- no parentheses
// same as
let user = new User();
```

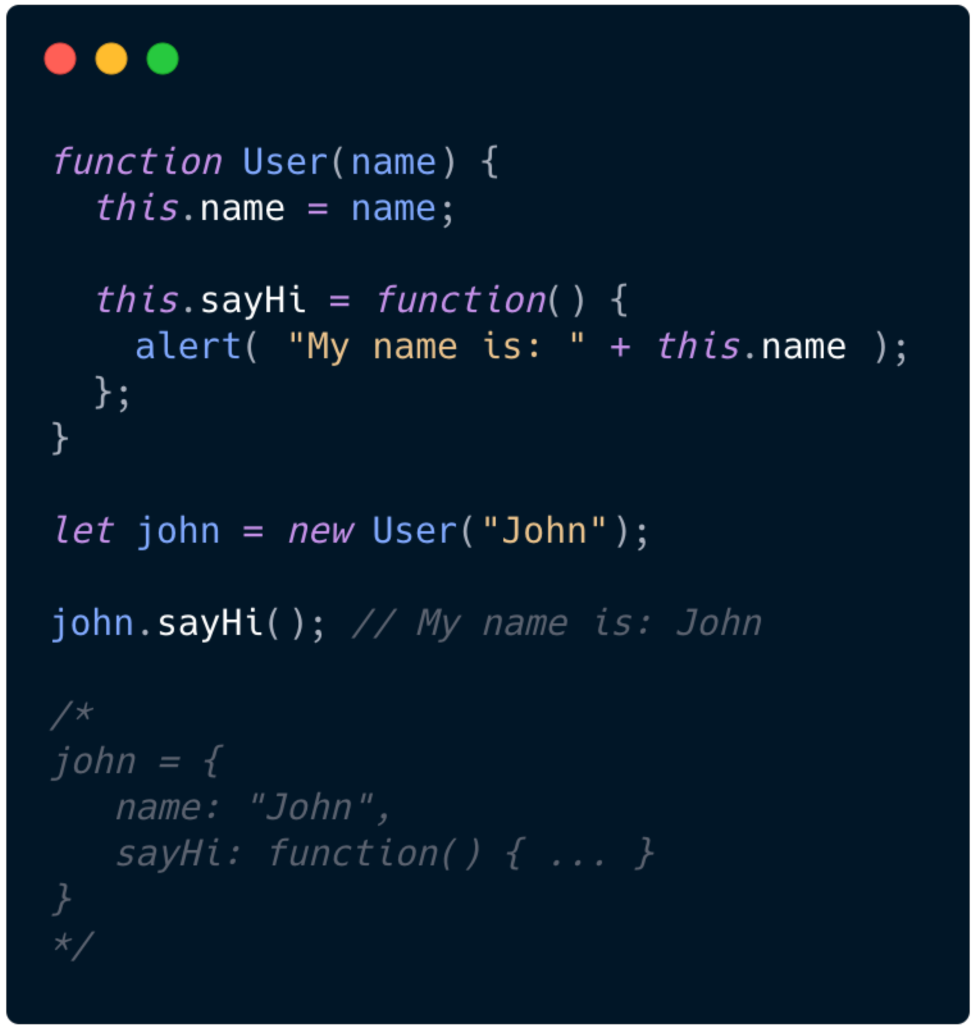
Omitting parentheses here is not considered a “good style”, but the syntax is permitted by specification.

Methods in constructor

Using constructor functions to create objects gives a great deal of flexibility. The constructor function may have parameters that define how to construct the object, and what to put in it.

Of course, we can add to this not only properties, but methods as well.

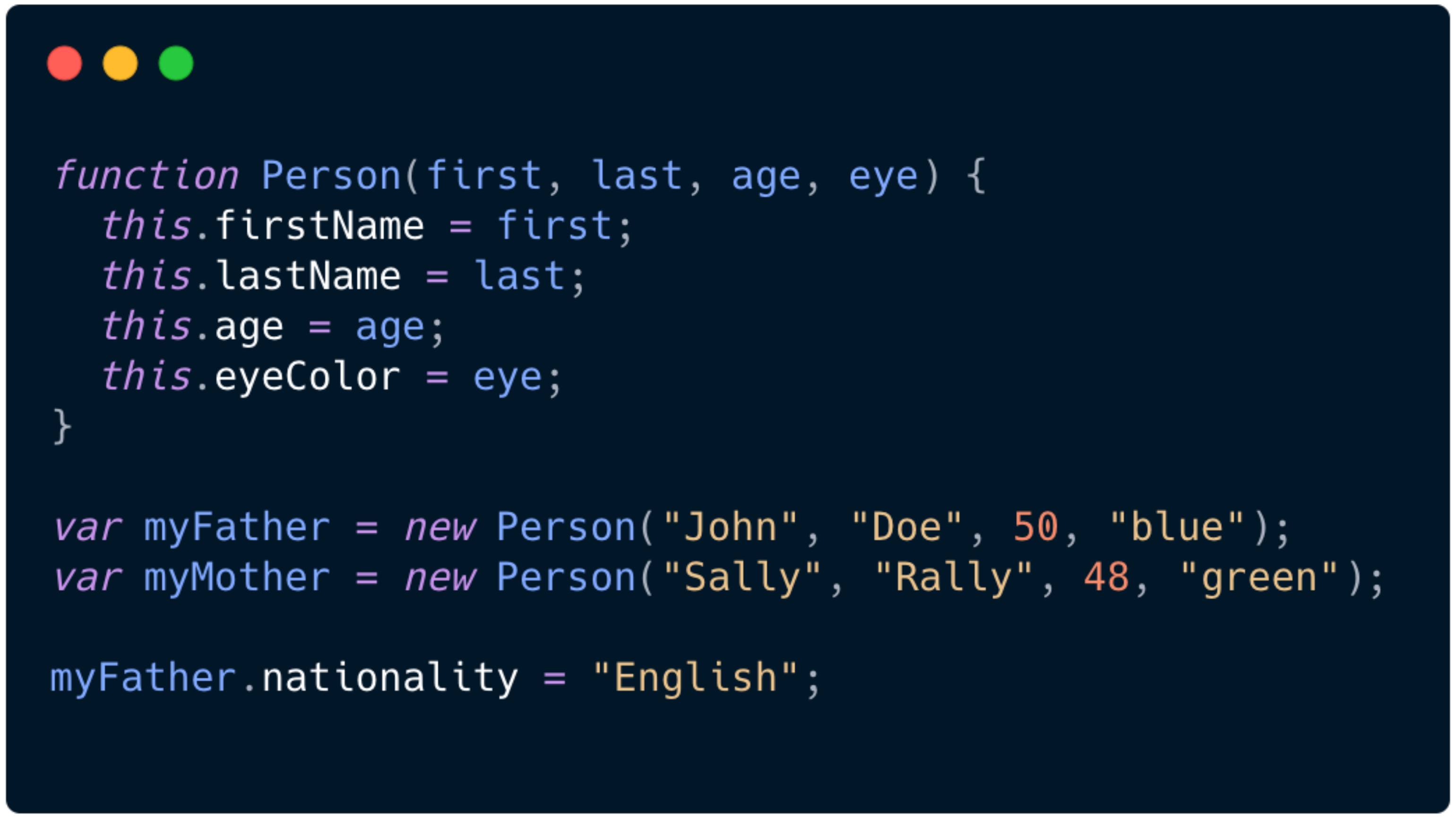
For instance, new User(name) below creates an object with the given name and the method sayHi:



```
function User(name) {
  this.name = name;

  this.sayHi = function() {
    alert( "My name is: " + this.name );
  };
}

let john = new User("John");

john.sayHi(); // My name is: John

/*
john = {
   name: "John",
   sayHi: function() { ... }
}
*/
```

Adding a property to Object

Adding a new property to an existing object is easy:

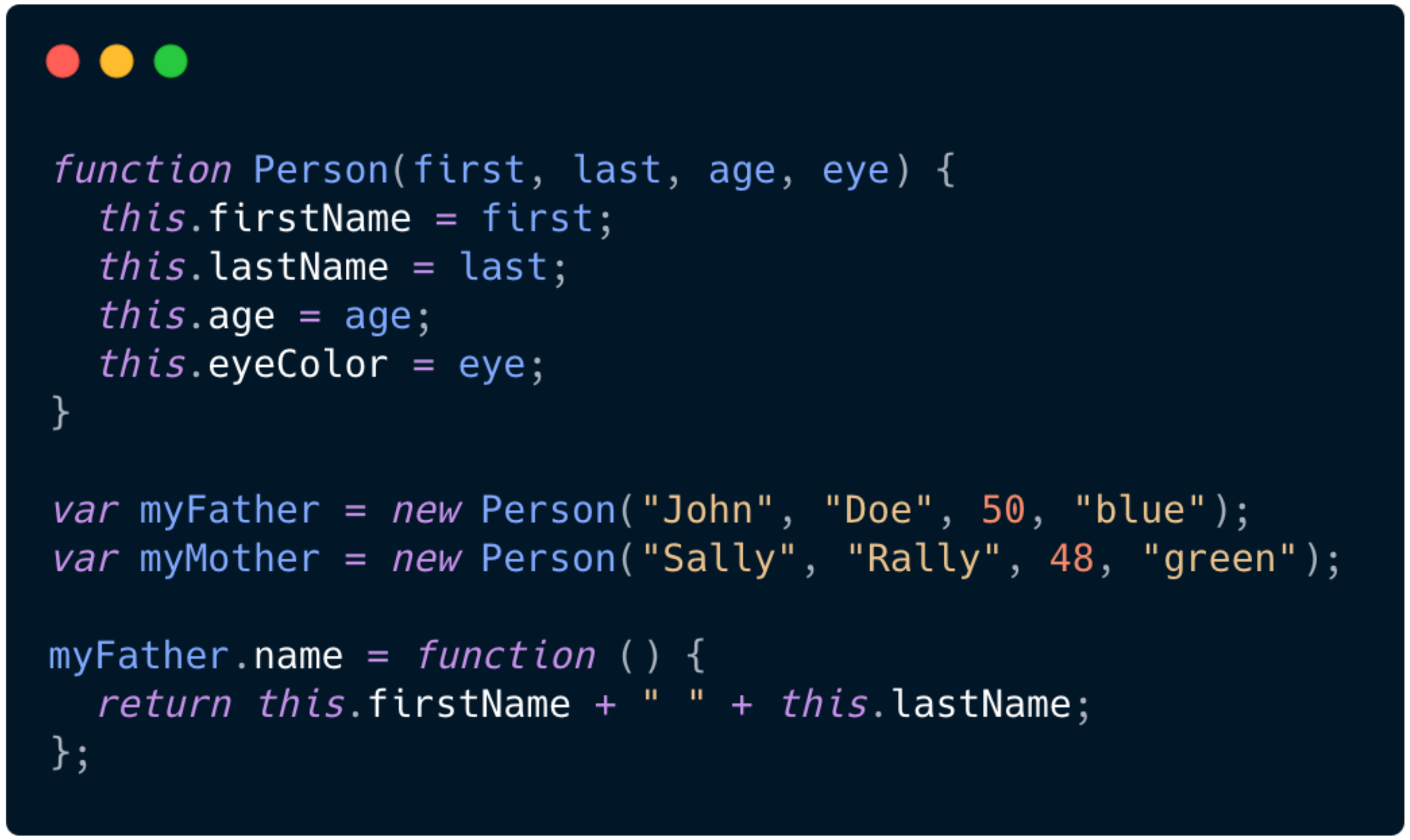


```
function Person(first, last, age, eye) {
  this.firstName = first;
  this.lastName = last;
  this.age = age;
  this.eyeColor = eye;
}

var myFather = new Person("John", "Doe", 50, "blue");
var myMother = new Person("Sally", "Rally", 48, "green");

myFather.nationality = "English";
```

NOTE: The property will be added to myFather. Not to myMother. (Not to any other person objects).

Adding a function to Object

Adding a new function to an existing object is easy:

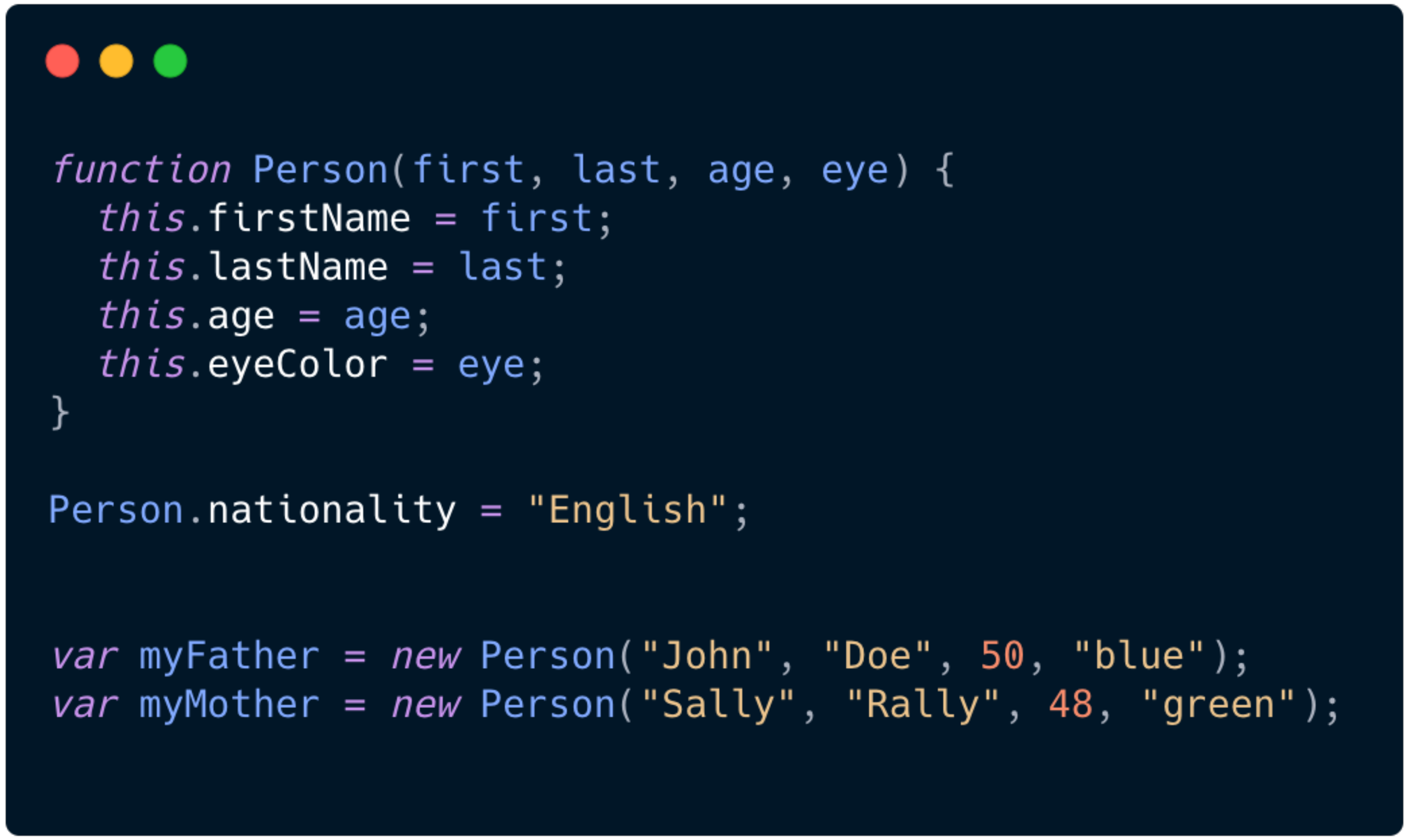


```javascript
function Person(first, last, age, eye) {
  this.firstName = first;
  this.lastName = last;
  this.age = age;
  this.eyeColor = eye;
}

var myFather = new Person("John", "Doe", 50, "blue");
var myMother = new Person("Sally", "Rally", 48, "green");

myFather.name = function () {
  return this.firstName + " " + this.lastName;
};
```

NOTE: The method will be added to myFather. Not to myMother. (Not to any other person objects).

Adding a property to constructor

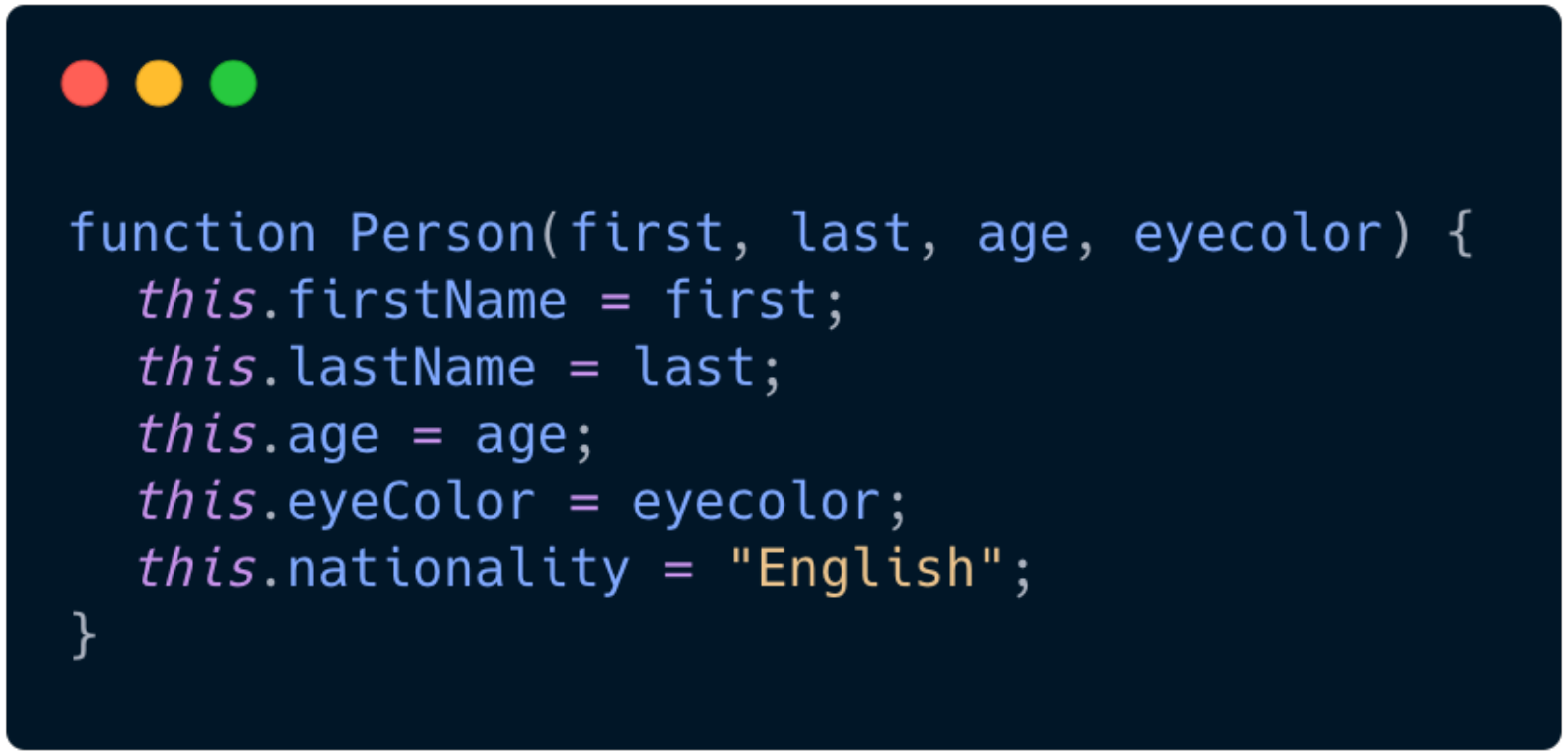
You cannot add a new property to an object constructor the same way you add a new property to an existing object:



```javascript
function Person(first, last, age, eye) {
  this.firstName = first;
  this.lastName = last;
  this.age = age;
  this.eyeColor = eye;
}

Person.nationality = "English";


var myFather = new Person("John", "Doe", 50, "blue");
var myMother = new Person("Sally", "Rally", 48, "green");
```

Adding a property to constructor

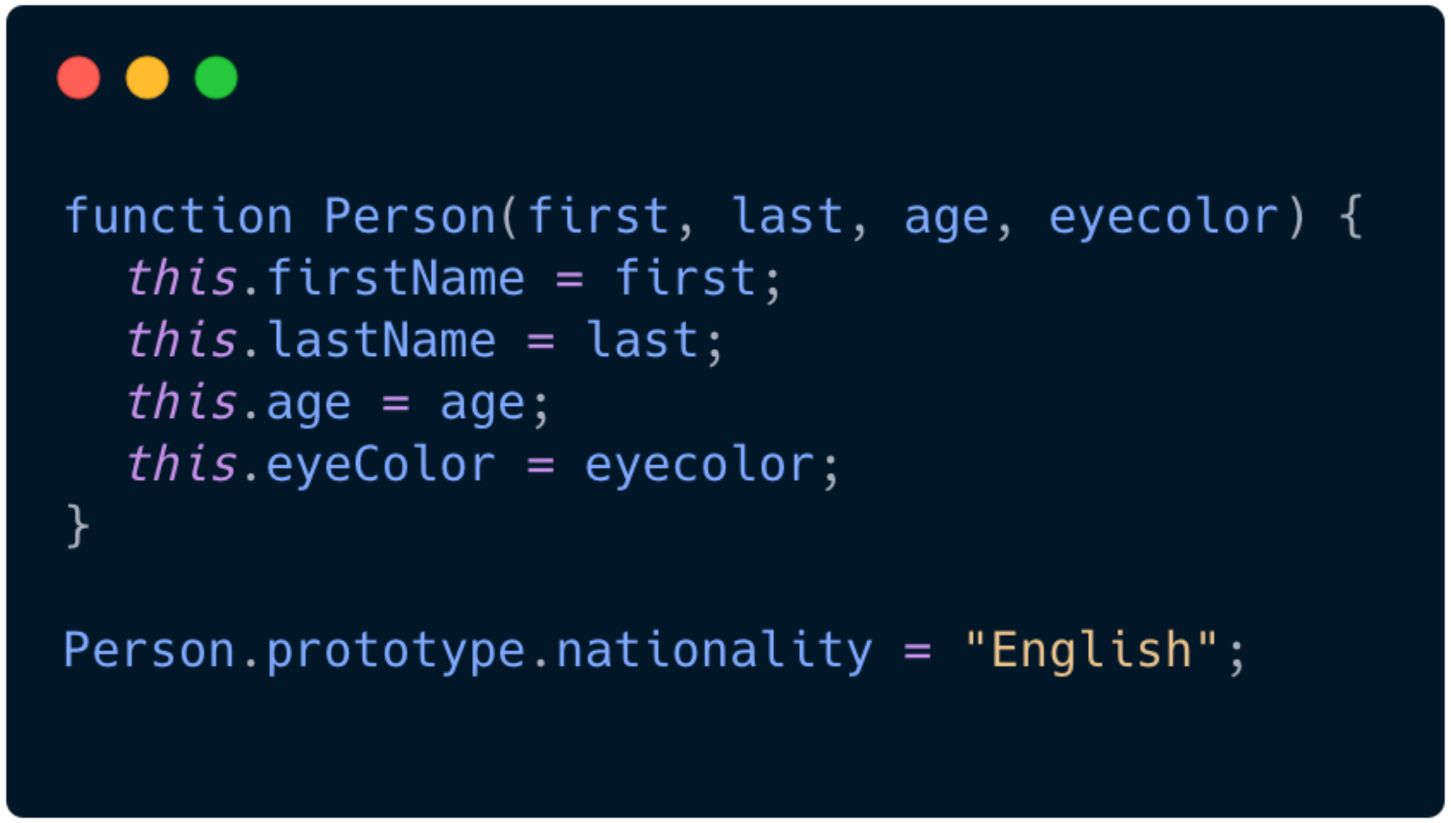
To add a new property to a constructor, you must add it to the constructor function:



```javascript
function Person(first, last, age, eyecolor) {
  this.firstName = first;
  this.lastName = last;
  this.age = age;
  this.eyeColor = eyecolor;
  this.nationality = "English";
}
```

Adding a property to constructor

To add a new property to object prototype, you can add it object special property, ‘prototype’ :

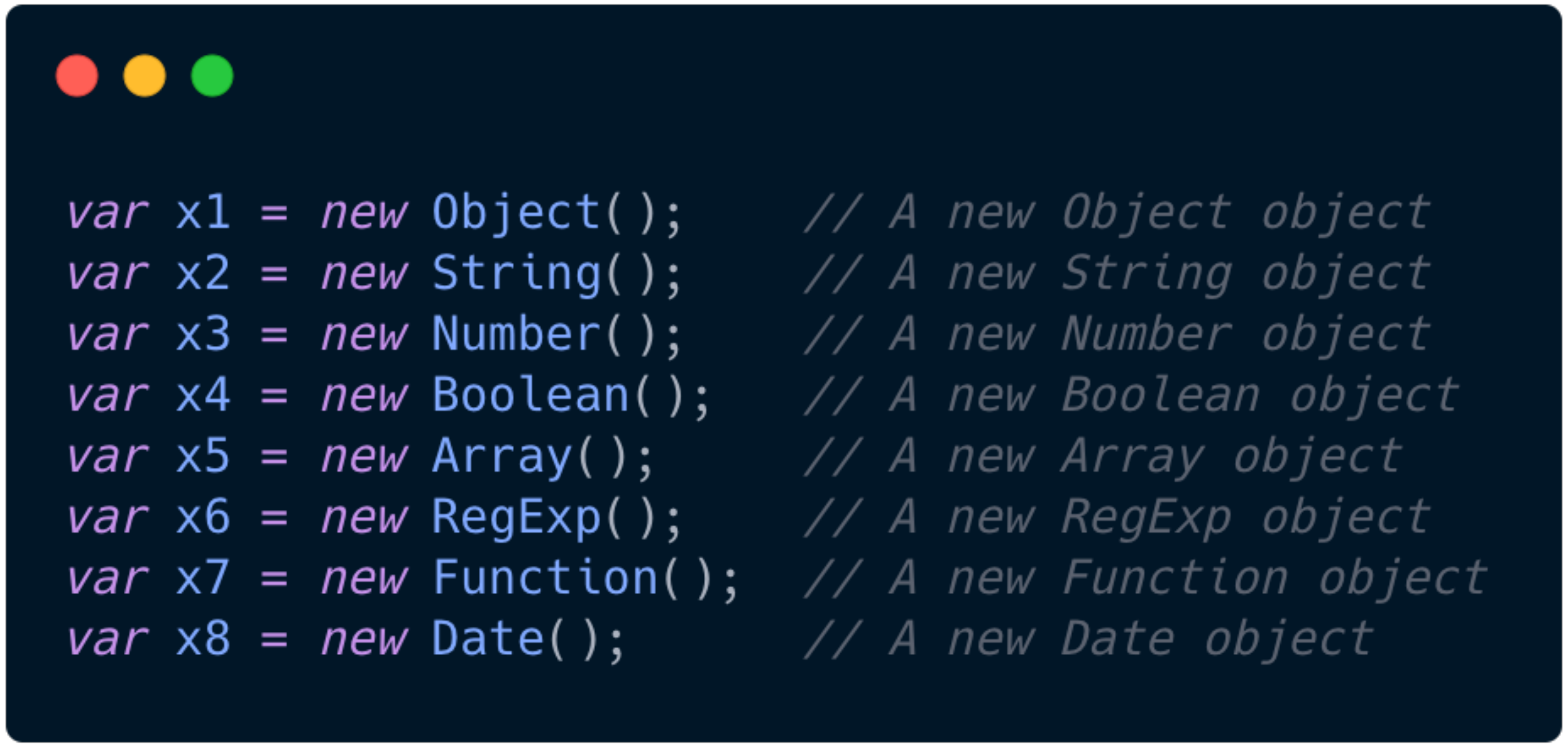


```
function Person(first, last, age, eyecolor) {
  this.firstName = first;
  this.lastName = last;
  this.age = age;
  this.eyeColor = eyecolor;
}

Person.prototype.nationality = "English";
```

NOTE: Only modify your **own** prototypes. Never modify the prototypes of standard JavaScript objects.

Built-in constructors

The JavaScript language has nine built-in constructors: Object(), Array(), String(), Number(), Boolean(), Date(), Function(), Error() and RegExp(). When creating values, we are free to use either object literals or constructors. However, object literals are not only easier to read but also faster to run, because they can be optimize at parse time. Thus, for simple objects it's best to stick with literals:



```
var x1 = new Object();     // A new Object object
var x2 = new String();     // A new String object
var x3 = new Number();     // A new Number object
var x4 = new Boolean();    // A new Boolean object
var x5 = new Array();      // A new Array object
var x6 = new RegExp();     // A new RegExp object
var x7 = new Function();   // A new Function object
var x8 = new Date();       // A new Date object
```

The Math() object is not in the list. Math is a global object. The new keyword cannot be used on Math.

Built-in constructors

As you can see above, JavaScript has object versions of the primitive data types String, Number, and Boolean. But there is no reason to create complex objects. Primitive values are much faster.

ALSO:

Use object literals {} instead of new Object().

Use string literals "" instead of new String().

Use number literals 12345 instead of new Number().

Use boolean literals true / false instead of new Boolean().

Use array literals [] instead of new Array().

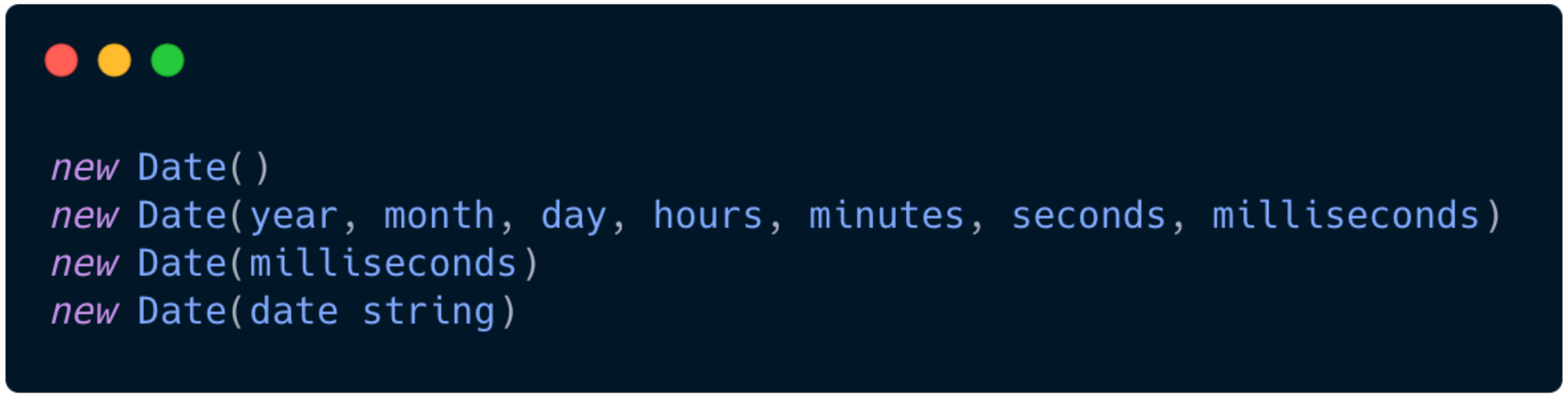
Use pattern literals /()/ instead of new RegExp().

Use function expressions () {} instead of new Function().

new Date()

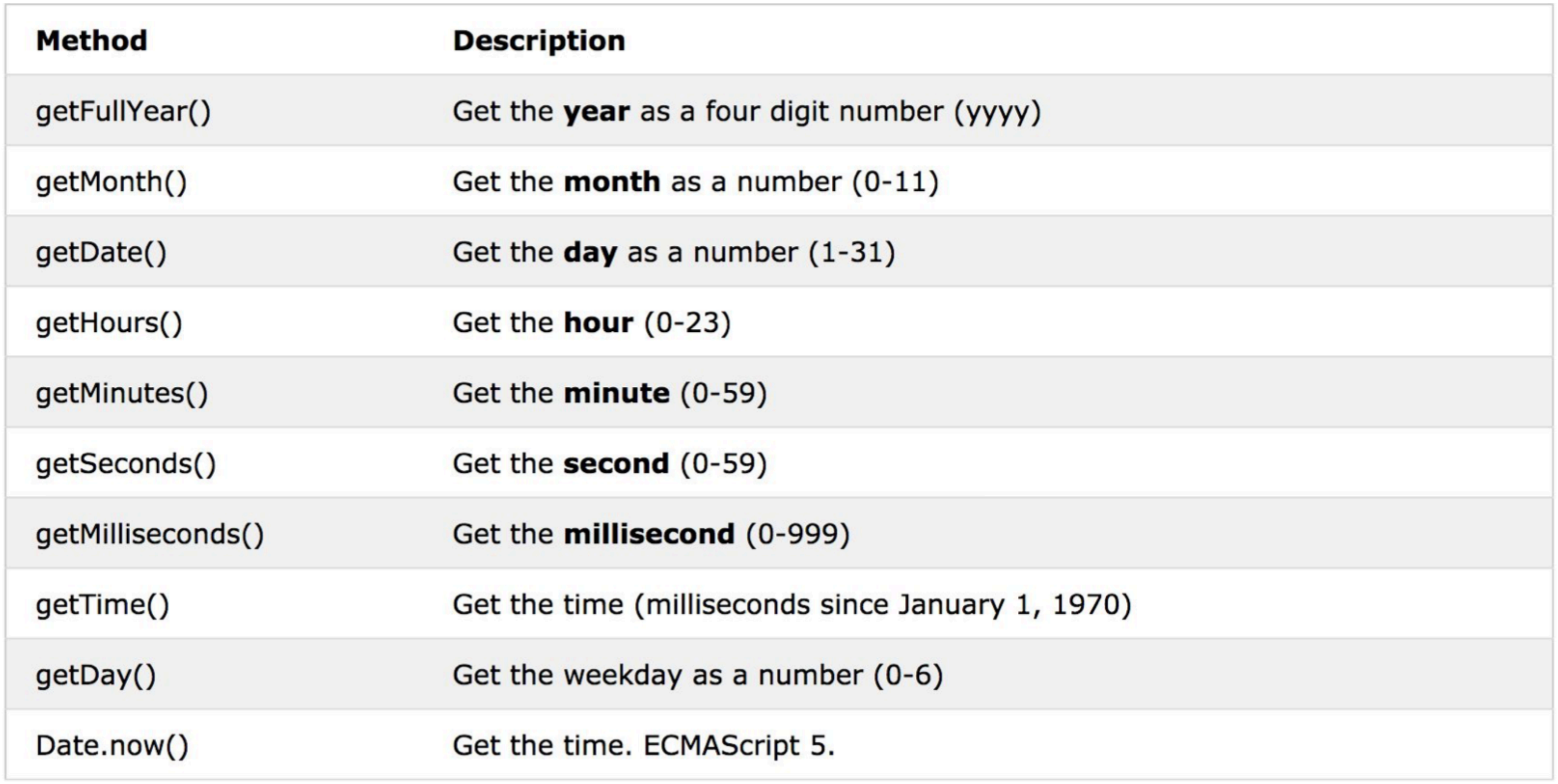
Date objects are created with the new Date() constructor.

There are **4 ways** to create a new date object:



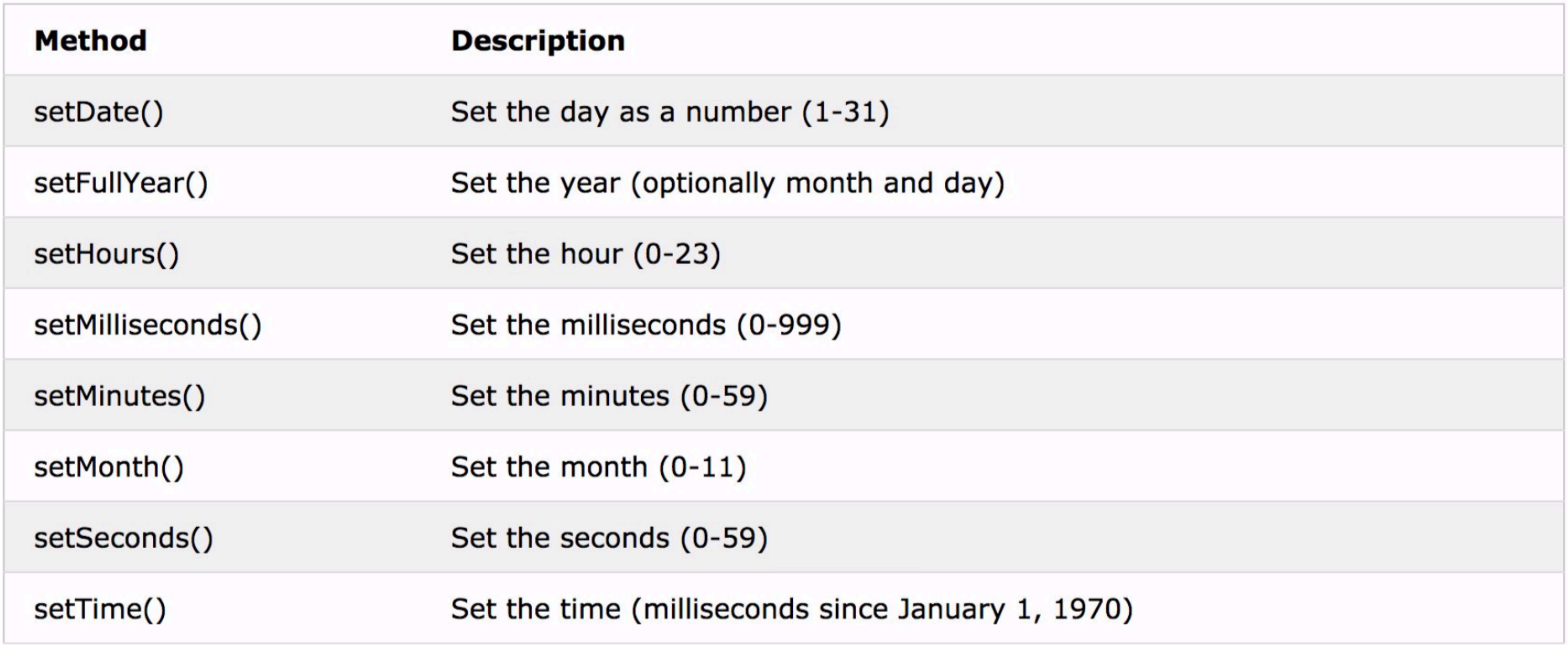
```
new Date()
new Date(year, month, day, hours, minutes, seconds, milliseconds)
new Date(milliseconds)
new Date(date string)
```

Getting date



Method	Description
getFullYear()	Get the **year** as a four digit number (yyyy)
getMonth()	Get the **month** as a number (0-11)
getDate()	Get the **day** as a number (1-31)
getHours()	Get the **hour** (0-23)
getMinutes()	Get the **minute** (0-59)
getSeconds()	Get the **second** (0-59)
getMilliseconds()	Get the **millisecond** (0-999)
getTime()	Get the time (milliseconds since January 1, 1970)
getDay()	Get the weekday as a number (0-6)
Date.now()	Get the time. ECMAScript 5.

Setting date



Method	Description
setDate()	Set the day as a number (1-31)
setFullYear()	Set the year (optionally month and day)
setHours()	Set the hour (0-23)
setMilliseconds()	Set the milliseconds (0-999)
setMinutes()	Set the minutes (0-59)
setMonth()	Set the month (0-11)
setSeconds()	Set the seconds (0-59)
setTime()	Set the time (milliseconds since January 1, 1970)

Home work

Improve APOD (A picture of the day) project:

- Using constructor function, create(initialize) an Apod object, that will create the container of the app (check classwork)
- Create a function **addPictureOfDay()**, that will add new picture of the day to page.
- **addPictureOfDay()** should be called in constructor itself, because on the first load(page open), we need to display today's picture of the day.
- Add a button in html “Load new APOD”, that will have as onclick event **addPictureOfDay(),** by clicking it, it should load new apod(using fetch, as we did in last project)
- In order to get a new picture of the day, you need to work with new Date() object. Check classwork, for hint.
- You can design the app as you wish.

Learning Resources

1. **Constructors:**
    1. **https://javascript.info/constructor-new**
    2. **https://www.w3schools.com/js/js_object_constructors.asp**
    3. **https://css-tricks.com/understanding-javascript-constructors/**
2. **Date Object: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Date**