Institute of Data

Software Engineering

Module 3

Javascript Language

Agenda: Module 3

- Javascript Fundamentals
- Intermediate Javascript
- Advanced Javascript

Javascript Fundamentals

- **Definition** of Javascript
- Javascript **Engine**
- Code Structure
- Variables
- Data Type
- Type conversion
- Comparison
- Functions
- Objects

Definition of Javascript

- JavaScript was initially called 'LiveScript', which means 'make web pages alive'
- JavaScript is fully independent language with its own specification called ECMAScript. Eg 'ES5(2009),ES6(2015),ES7, etc'
- Comparing 'Javascript' and 'Java' is like comparing 'cart' and 'carpet'

Why Javascript?

Reasons

- Full integration with HTML/CSS.
- Simple things are done simply.
- Support by all major browsers and enabled by default.

JavaScript is the only browser technology that combines these three things.

Javascript Engine

Javascript can execute in any environment(browser, server etc), where there is Javascript engine.
The browser has an embedded engine called 'JavaScript virtual machine' with different 'code names', eg

- V8 - in chrome and opera.
- SpiderMoney - in FireFox
- JavasScriptCore, Nitro, SquirrelFish - in Safari

The basic task for engine is to convert ('compiles') the Javascript to machine executable binaries.

Code structure

- JavaScript command and constructs are written in statements, which are separated by semicolon.

```
alert('hello'); alert('world');
```

- Semicolon can be omitted when a line break exist.

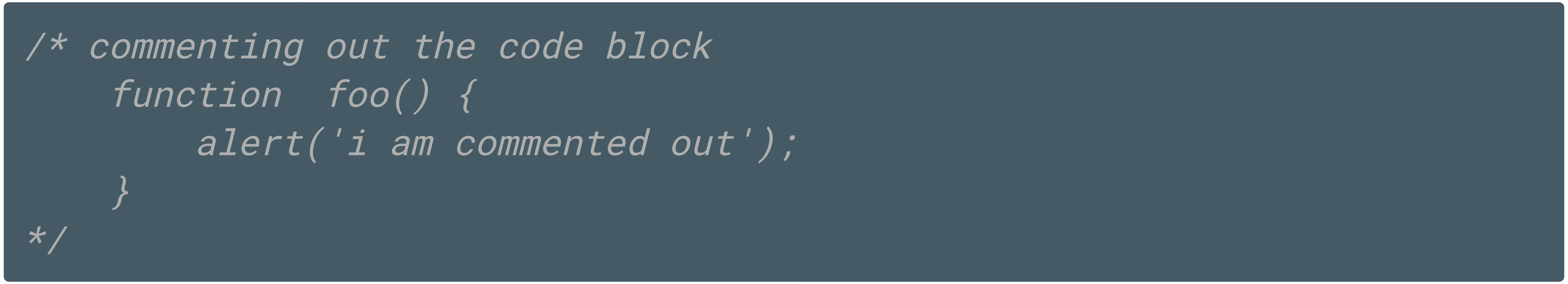
```
alert('hello')
alert('world')
```

- Line comment

```
alert('world'); // This comment follows the statement
```

Code structure

- Block comment.



```
/* commenting out the code block
    function  foo() {
        alert('i am commented out');
    }
*/
```

Variables

A variable is a 'named storage' for data, `const, var, let` are used to declare a variable.

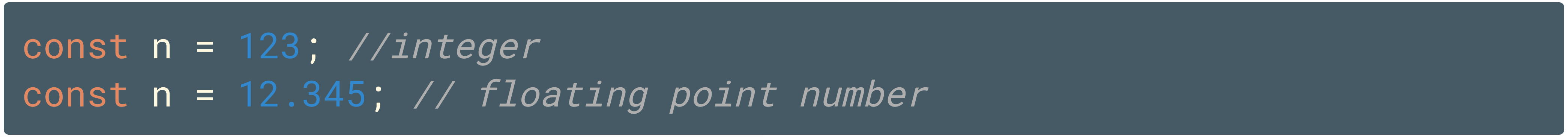
Variable naming

- The name contains only **letters, digits, or symbols $ and _**
- The first character can not be a digit.
- The reserved names can not be used, eg **let, class, return, function**.
- The name is **case-sensitive**.
- **Non-latin** letter are allowed, but not recommended.

Data Type

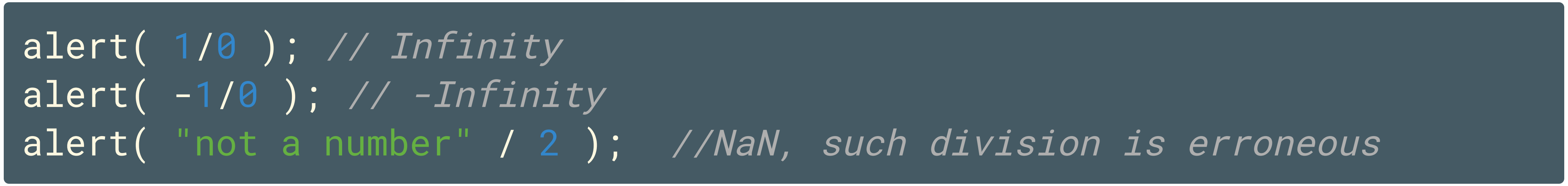
Number

- The number type represents both integer and floating point numbers.



```
const n = 123; //integer
const n = 12.345; // floating point number
```

- Special numeric values:**Infinity, -Infinity, NaN**



```
alert( 1/0 ); // Infinity
alert( -1/0 ); // -Infinity
alert( "not a number" / 2 );  //NaN, such division is erroneous
```

Data Type

BigInt

In JavaScript, the `number` type cannot represent integer values larger than (2**53-1) (that's 9007199254740991), or less than -(2**53-1) for negatives. It's a technical limitation caused by their internal representation.

BigInt value is created by appending n to the end of an integer.

```
const bigInt = 123456789012345678912345n;
```

Data Type

String

A string in JavaScript must be surrounded by quotes. There are 3 types of quotes.

- Double quotes.

```
const str = "hello";
```

- Single quotes.

```
const str = 'hello';
```

- Backticks.

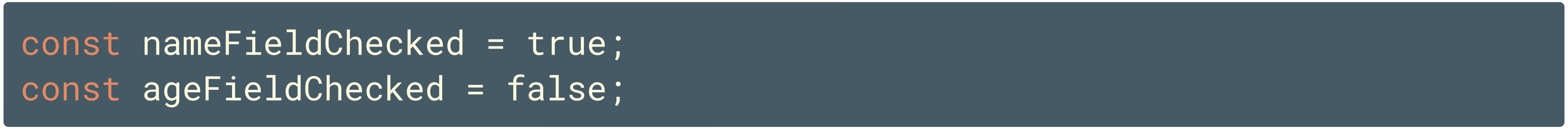
```
const phrase = `can embed another ${str}`;
```

Data Type

Boolean

The boolean type has only two values: true and false.

This type is commonly used to store yes/no values: true means "yes, correct", and false means "no, incorrect".



```
const nameFieldChecked = true;
const ageFieldChecked = false;
```

Data Type

Null

In JavaScript, null is not a **reference to a non-existing object** or a **null pointer** like in some other languages.

It's just a special value which represents **nothing**, **empty** or **value unknown**.

```
const age = null;
```

Data Type

Undefined

The special value undefined also stands apart. It makes a type of its own, just like null.

The meaning of undefined is **value is not assigned**.

If a variable is **declared**, but **not assigned**, then its value is **undefined**.

```
const age;
alert( age ); // shows 'undefined'
```

Data Type

Object

The object type is special. All other types are called "primitive” because their values can contain only a single thing (be it a string or a number or whatever). In contrast, objects are used to store collections of data and more complex entities.

Symbol

The symbol type is used to create unique identifiers for objects.

Data Type

The typeof operator

The **typeof** operator returns the type of the argument. It's useful when we want to process values of different types differently or just want to do a quick check.

It supports two forms of syntax:

- As an operator: `typeof x`.
- As a function: `typeof(x)`.

Data Type

Type check Examples:

```
typeof undefined // "undefined"
typeof 0 // "number"
typeof 10n // "bigint"
typeof true // "boolean"
typeof "foo" // "string"
typeof Symbol("id") // "symbol"
typeof Math // "object"
typeof null // "object"
typeof alert // "function"
```

Note: The result of `typeof null` is `"object"`. That's an officially recognized error in typeof behavior.

Type conversion

String conversion

String conversion happens when we need the string form of a value.

- Explicit conversion by using constructor function.

```
String( false ) //"false"
```

- Implicit conversion

```
"1" + 2 + 2  //"122"
2 + 2 + "1" //"41"
```

Type conversion

Numeric conversion

Numeric conversion happens in mathematical functions and expressions automatically.

- Explicit conversion by using constructor function.

```
Number( "   4    ")  //4
Number( null ) //0
Number( undefined ) // NaN
Number( false ) //0
Number( true ) //1
Number( "" ) // 0
Number( "hello" ) //NaN
```

Type conversion

Numeric conversion

- Implicit conversion by using division `/`, subtraction `-`, unary plus `+`.

```
"6" / "2" //3
"6" - 2 //4
+"" //0
```

Type conversion

Boolean conversion

It happens in logical operations.

- Explicit conversion by using the constructor function.

```
Boolean( "" ) //false
Boolean( 0 ) //false
Boolean( null ) // false
Boolean( undefined ) // false
Boolean( NaN ) // false
Boolean( "hello" ) //true
Boolean( 1 ) //true
```

Type conversion

Boolean conversion

- Implicit conversion by using **if statement**, **ternary operator**, **double NOT !!**.

```
if ( truthy value ) {
    do something;
}

( truthy value )? do one thing : do another thing;

!!null //false
```

Comparison

String Comparison

To see whether a string is greater than another, JavaScript uses the so-called **dictionary** or **lexicographical** order. In other words, strings are compared letter-by-letter.

The algorithm to compare two strings is simple:

1. Compare the first character of both strings.
2. If the first character from the first string is greater (or less) than the other string's, then the first string is greater (or less) than the second. We're done.
3. Otherwise, if both strings' first characters are the same, compare the second characters the same way.
4. Repeat until the end of either string.
5. If both strings end at the same length, then they are equal. Otherwise, the longer string is greater.

Comparision

Comparison of different types

When comparing values of different types, JavaScript converts the values to numbers.

```
"2" > 1 //true
"02" == 2 //true
true == 1 //true
false == 0 //false
null == undefined //true
```

Functions

Definition

Functions are the main **building blocks** of the program. They allow the code to be called many times without repetition.

Function declaration

The function keyword goes first, then goes the name of the function, then a list of **parameters** between the parentheses (**comma-separated**) and finally the code of the function, also named **the function body**, between **curly braces**.

Functions

Exit function call

- Implicit exit by missing `return` keyword

```
function returnUndefined() {
    /* empty */
}
```

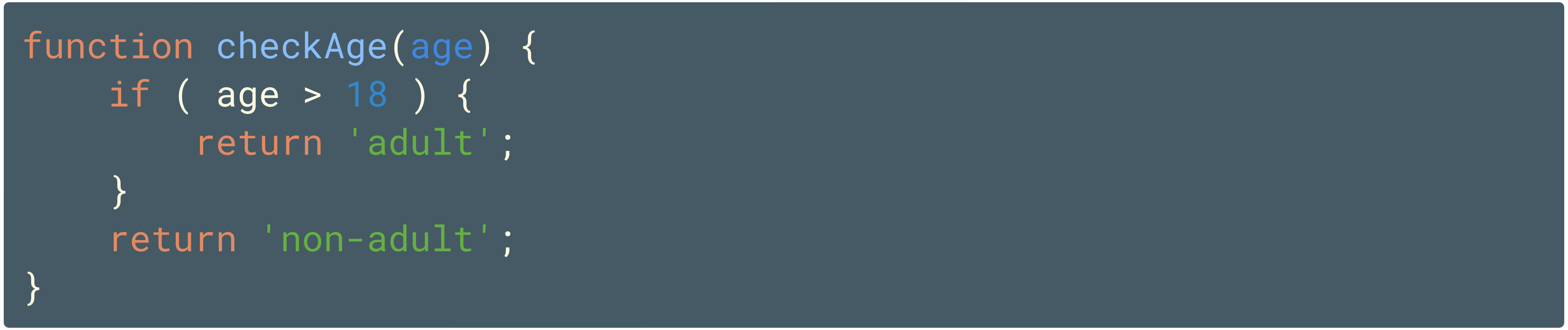
- Explicit exit by returning 'nothing'.

```
function returnUndefined() {
    return; // same as `return undefined`
}
```

Functions

Exit function call

- Explicit exit by returning a value.



```
function checkAge(age) {
    if ( age > 18 ) {
        return 'adult';
    }
    return 'non-adult';
}
```

Functions

Function expression

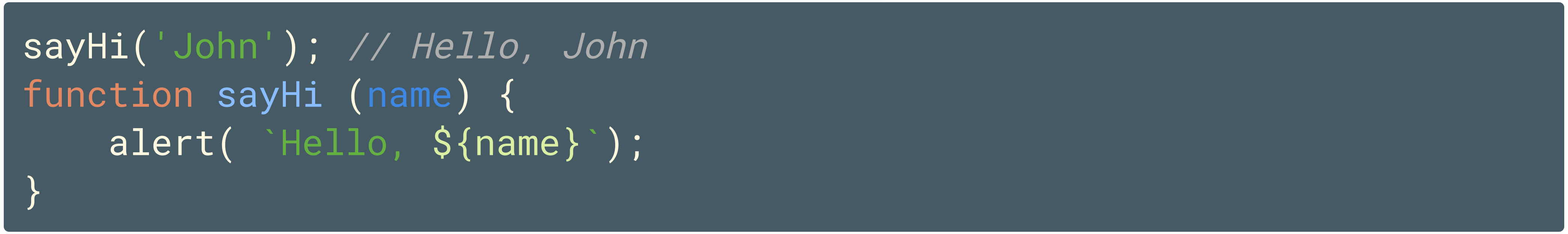
The function is created and assigned to the variable explicitly, like any other value. No matter how the function is defined, it's just a value stored in the variable.

```
const sayHi = function () {
    alert('Hello');
}
```

Functions

Function expression vs Function declaration

- Function Declaration can be **hoisted**, can be called earlier than it is defined.

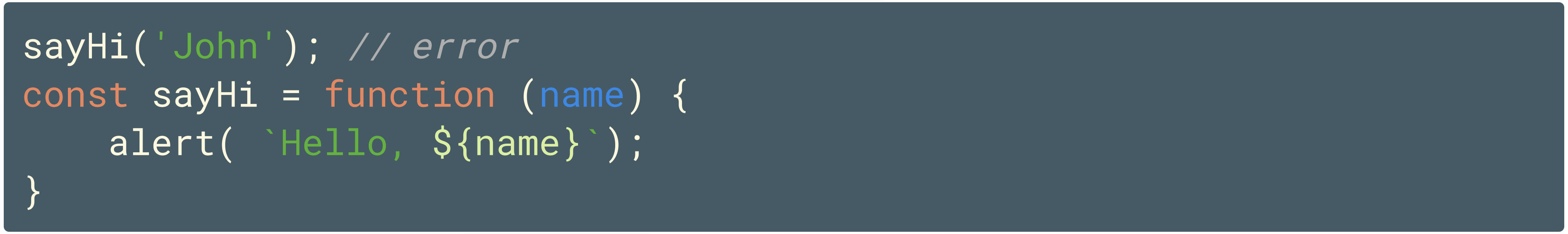


```
sayHi('John'); // Hello, John
function sayHi (name) {
    alert( `Hello, ${name}`);
}
```

Functions

Function expression vs Function declaration

- Function Expression **cannot be hoisted**, it is created when the execution reaches it and is usable only from that moment.



```
sayHi('John'); // error
const sayHi = function (name) {
    alert( `Hello, ${name}`);
}
```

Arrow Functions

Definition

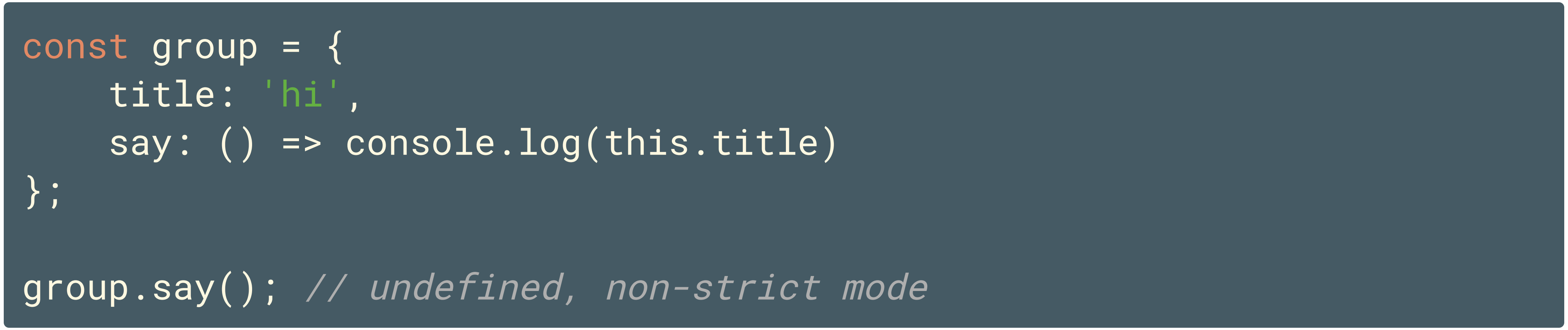
It is a **shorter version** of function expression. This creates a function func that accepts arguments **arg1..argN**, then evaluates the expression on the right side with their use and returns its result.

```
const func = (arg1, arg2, ..., argN) => expression

cont sum = (a, b) => a + b;
/* This arrow function is a shorter form of:
const sum = function(a, b) {
  return a + b;
};
*/
```

Arrow Functions

Features

- Arrow function has no lexical `this`, If this is accessed, it is taken from the outside.



```
const group = {
    title: 'hi',
    say: () => console.log(this.title)
};

group.say(); // undefined, non-strict mode
```

Arrow Functions

Features

- Arrow function has no `arguments` variable.

```
const getArg = () => console.log(arguments);
getArg(1); //ReferenceError:; arguments is not defined.
```

- Arrow function can not be called with `new`, which means it can't be used as constructor.

```
const Func = () => console.log('new Fun');
const fun = new Func(); //TypeError, Func is not a constructor
```

Objects

Definition

Object is an **optional list of properties**, which are used to store **keyed collections** of various data and more complex entities.

A property is a **key: value** pair, where key is a string (also called a **property name**), and **value** can be **anything**.

Creation

An object can be created with figure brackets {…} with an optional list of properties.

Objects

Empty Object

An empty object ('empty cabinet') can be created using one of two syntaxes:

- Object constructor

```
const user = new Object(); // "object constructor" syntax
```

- Object literal

```
const user = {};  // "object literal" syntax
```

Objects

Object with properties

We can immediately put some properties into `{...}` as **key: value** pairs. A property has a key (also known as **name** or **identifier**) before the colon `:` and a value to the right of it.

```
const user = { // an object
    name: 'joe', // by key 'name' store value 'joe'
    age : 20, //by key 'age' store value 20
    'has a dog': false //multi word prop name 'has a dog' store value false
};
```

Objects

Object with operations

We can **get, set, or delete** the value of property key.

```
user.name; //get a prop value of 'name'

user.name = 'Ben'; //set a prop value of 'name'
user['has a dog']; //get a prop value of multi word 'has a dog'

delete use.name; // delete a prop with key of 'name'
delete ['has a dog']; //delete a prop with key of 'has a dog'
```

Objects

Property name limitation

- `__proto__` prop name can not be assigned with a primitive value, except `null`.

```
user.__proto__ = 'Jason'; //no effects
user.__proto__ = null; //has effects
```

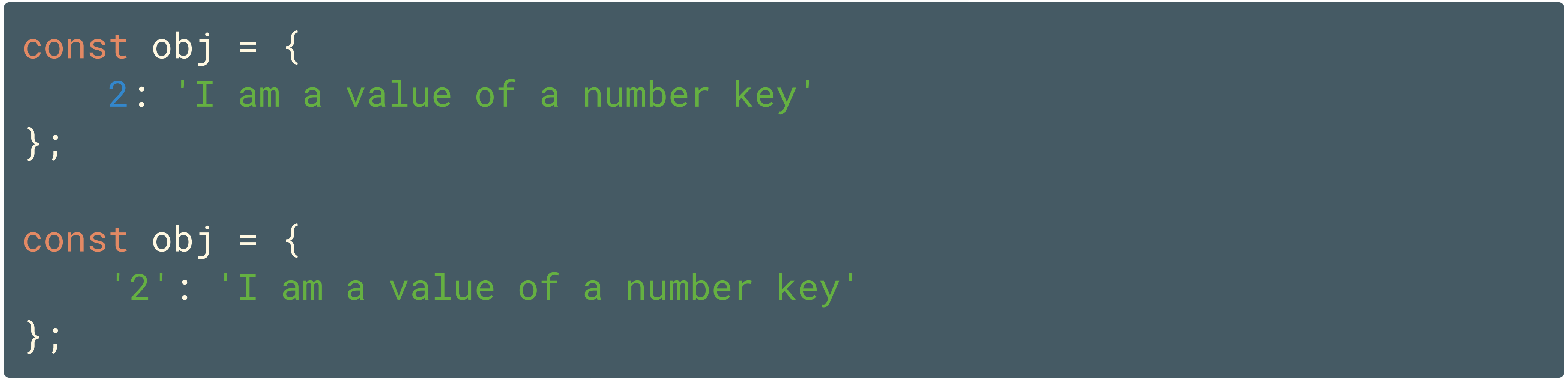
- Anything can be a property name, and be **stringified**.

```
const funcKey = () => console.log('func as key');
const user = {
    [funcKey]: 'I am a value of a function'
};
```

Objects

Property name limitation

- **number or number with quotes** as a property name.



```
const obj = {
    2: 'I am a value of a number key'
};

const obj = {
    '2': 'I am a value of a number key'
};
```

Objects

Property existence test

It is possible to access any property of an object, there will be no error if the property doesn't exist!

Reading a non-existing property just returns `undefined`. So we can easily test whether the property exists.

```
const user = {};

alert( user.noSuchProperty === undefined ); // true means "no such property"
```

Objects

in operator

- in checks if key in an object.

```
const user = { name: "John", age: 30 };
'age' in user ; // true, user.age exists
'blabla' in user ; // false, user.blabla doesn't exist
```

- in also checks if the **inherited** key in an object.

```
const user = { name: 'john' };
const anotherUser = Object.create(user);
anotherUser.age = 15;
'name' in anotherUser; //true,  anotherUser.name exists
```

Objects

Object Iteration

To walk over all keys (**including the inherited keys**) of an object, there exists a special form of the loop: `for..in`.

```
const user = { name: 'john' };
const anotherUser = Object.create(user);

anotherUser.age = 15;

for ( const key in anotherUser ) {
    console.log( key ); // age, name
}
```

Objects

References

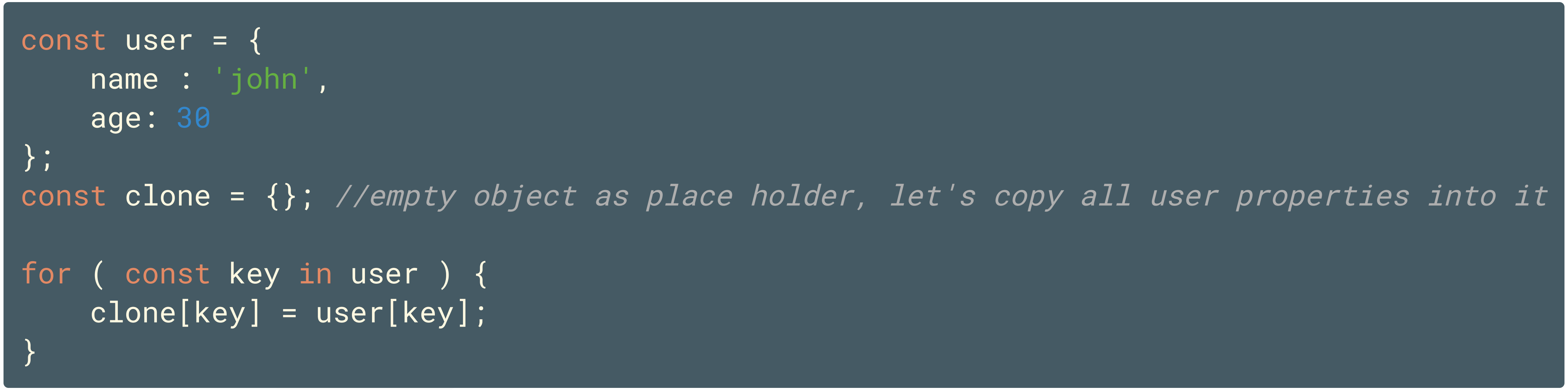
The objects are stored and copied **by reference**, whereas primitive values: **strings, numbers, booleans**, etc, are always copied **as a whole value**. A variable assigned to an object stores not the object itself, but its **address in memory**, in other words ***a reference*** to it.

```
const user = { name : 'john' };
const admin = user;

admin.name =  'Jesse'; // changed by the 'admin' reference

console.log(user.name); // 'Jesse', changes are seen from the 'user' reference
```

Objects

Shallow Copy

create a new object and replicate the structure of the existing one by iterating over its properties and copying them on the **primitive level**.



```
const user = {
    name : 'john',
    age: 30
};
const clone = {}; //empty object as place holder, let's copy all user properties into it

for ( const key in user ) {
    clone[key] = user[key];
}
```

Objects

Shallow Copy

- 'shallow copy' with `Object spread`.

```
const clone = { ...user };
```

- 'shallow copy' with `Object.assign`.

```
const clone = Object.assign( {}, user );
```

Objects

Deep clone

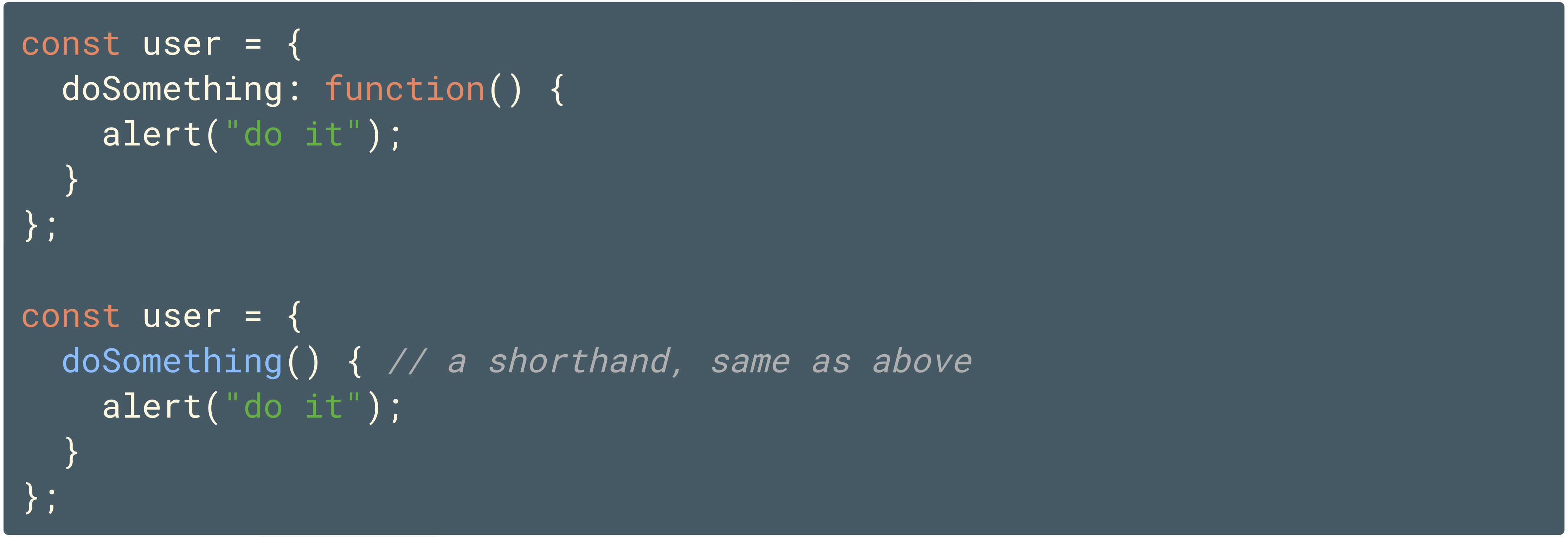
if the **value of property name** in an object is **not a primitive**, we also want it to be **copied over, not referred** to ('deep clone'). we can resort to 3rd party tool. eg https://lodash.com/docs#cloneDeep

```
const box = {
  weight: '20kg',
  dimensions: {
    height: '0cm',
    width: '10cm',
  }
};
const anotherClonedBox = _.cloneDeep(box);
box.dimensions === anotherClonedBox.dimensions; // false, means it is not a reference
```

Objects

Methods

A function that is a property of an object is called its **method**.

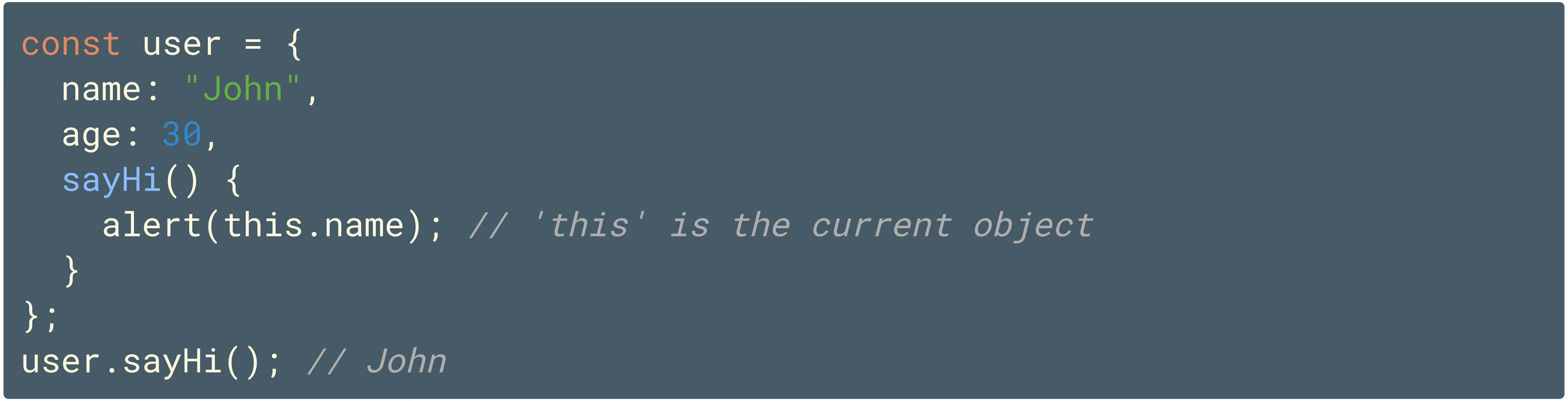


```
const user = {
  doSomething: function() {
    alert("do it");
  }
};

const user = {
  doSomething() { // a shorthand, same as above
    alert("do it");
  }
};
```

Objects

Other Iteration methods

It is common that an object method needs to access the information stored in the object to do its job. To access the object, a method can use the this keyword. The value of `this` is the object **before dot**, the one used to call the method.



```
const user = {
  name: "John",
  age: 30,
  sayHi() {
    alert(this.name); // 'this' is the current object
  }
};
user.sayHi(); // John
```

Objects

this is unbound

In JavaScript, keyword this behaves unlike most other programming languages. It can be used in any function, even if it is not a method of an object.

```
function sayHi() {
  alert( this.name ); // 'this' is not bound, but no syntax error
}
```

The value of this is evaluated during the run-time, depending on the context.

Objects

Constructor function

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.

That can be done using constructor functions and the `new` operator.

- constructor functions are named with capital letter first.
- constructor functions should be executed only with "new" operator.

```
const user = new User(); // User is a constructor function.
```

Objects

Object generations with `new`

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

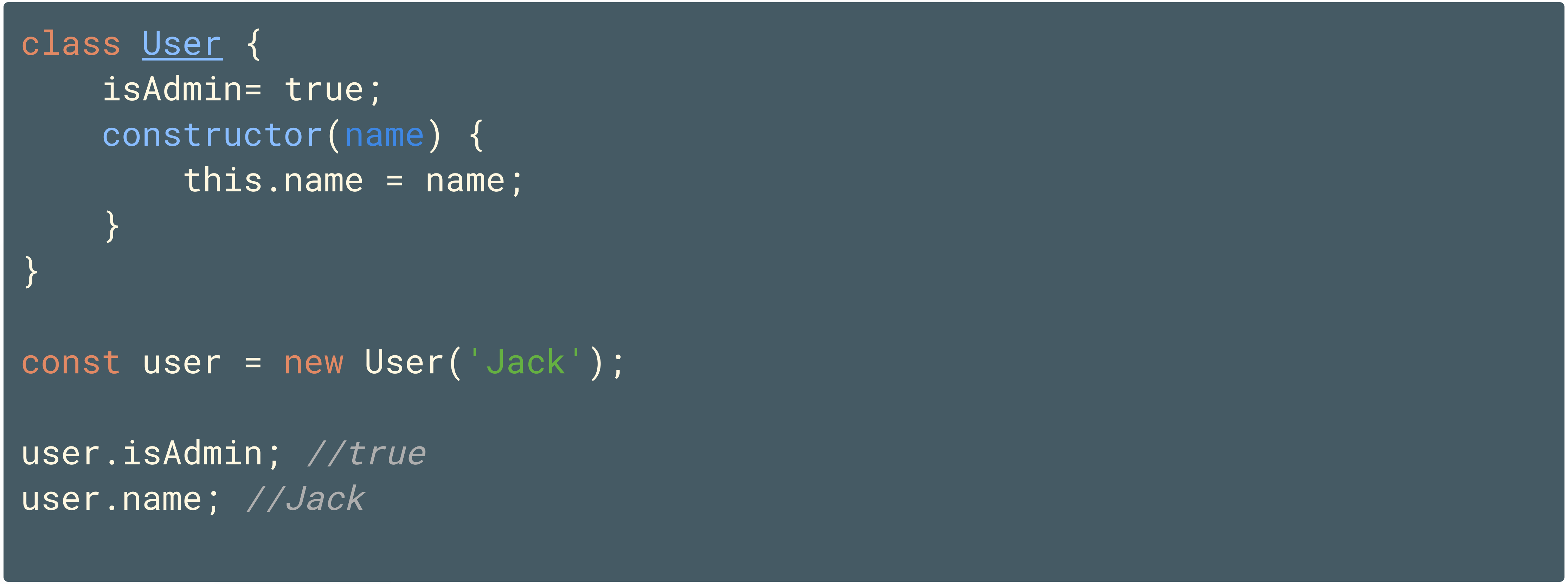
1. A new empty object is created and assigned to `this`.
2. The function body executes. Usually it modifies `this`, adds new properties to it.
3. The value of `this` is returned.

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

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

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

Objects

Object generation in ES6

In ES6, the new syntax keyword `class` is introduced.



```
class User {
    isAdmin= true;
    constructor(name) {
        this.name = name;
    }
}

const user = new User('Jack');

user.isAdmin; //true
user.name; //Jack
```

Questions?