UNIVERSITÀ DEGLI STUDI DI NAPOLI FEDERICO II

WEB TECHNOLOGIES — LECTURE 05

JAVASCRIPT: PART I

Luigi Libero Lucio Starace, PhD

luigiliberolucio.starace@unina.it
https://luistar.github.io
https://www.docenti.unina.it/luigiliberolucio.starace

PREVIOUSLY, ON WEB TECHNOLOGIES

So far, we’ve learnt to create modern, beautiful web pages

- With **HTML** we define their structure
- With **CSS** we define their appearence on possibly different media

Still, our web pages are inherently **static**

- There is no way their content can change while we are browsing (unless we edit the html file and re-load the page)

JAVASCRIPT

High-level loosely-typed scripting programming language

- First introduced by the Netscape web browser in 1995
- Designed to be **included in web pages** and **executed inside a web browser**, with the goal of making HTML documents more dynamic
- JavaScript programs (scripts) are interpreted as plain text by the JS Engine

JAVASCRIPT

- Nowadays it’s used **not only in web pages** (we’ll see that!)
  - Backend software, Desktop and Mobile applications, general scripting…
- JavaScript is the most popular programming language
  - Used by 66% of professional developers on StackOverflow in 2023
- JavaScript implements the **ECMAScript** specification

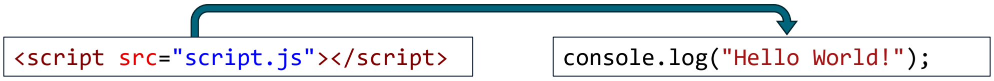
INCLUDING JAVASCRIPT IN WEB PAGES

JavaScript code can be **included** in an HTML documents in two ways:

- **Internal JavaScript**
  - Code is written inside a **<script>** element in the **<head>** or in the **<body>**

```
<script>
  console.log("Hello World!");
</script>
```

- **External JavaScript**
  - Using **<script src="url/of/script.js">** (external file must have «**.js**» extension)

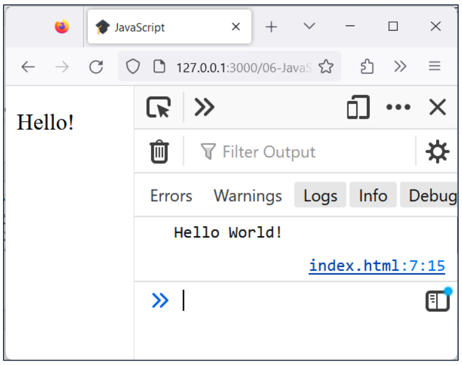


```
<script src="script.js"></script>
```

```
console.log("Hello World!");
```

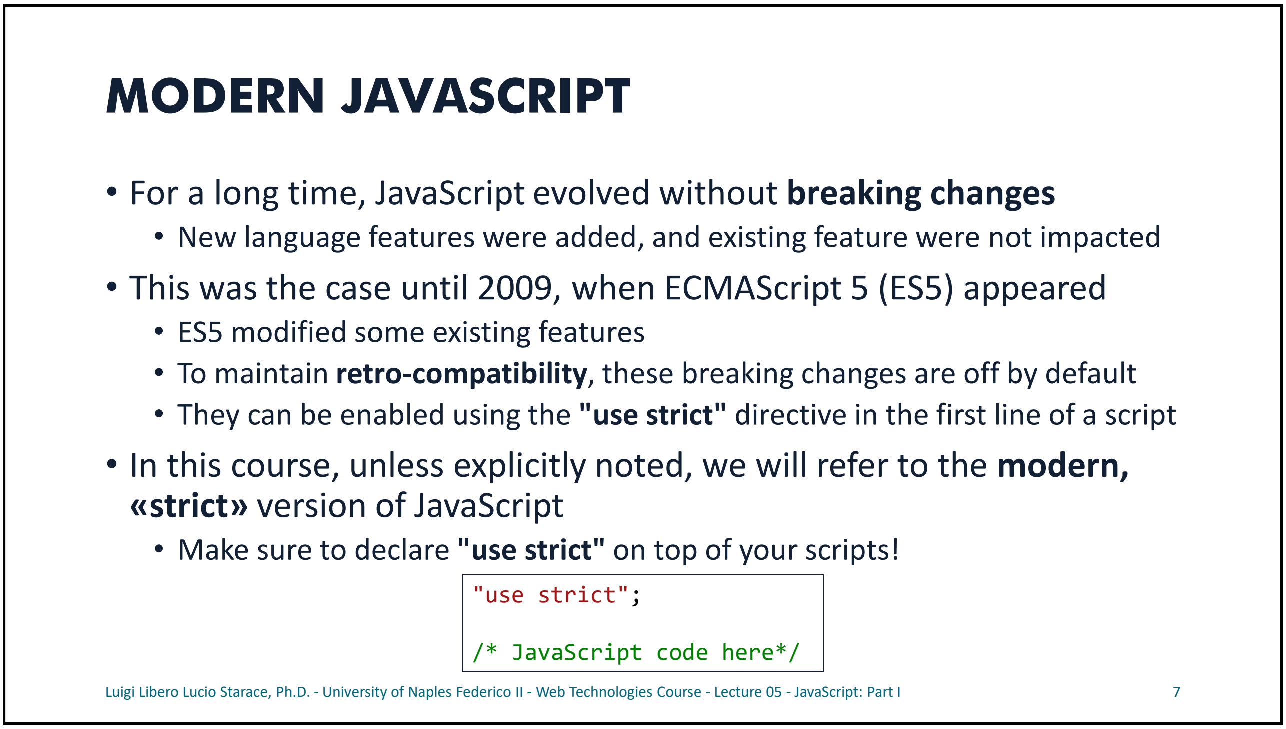
JAVASCRIPT: HELLO WORLD!

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8">
    <title>JavaScript</title>
    <script>
      console.log("Hello World!");
    </script>
  </head>
  <body>
    <h1>Hello!</h1>
  </body>
</html>
```



Hello!

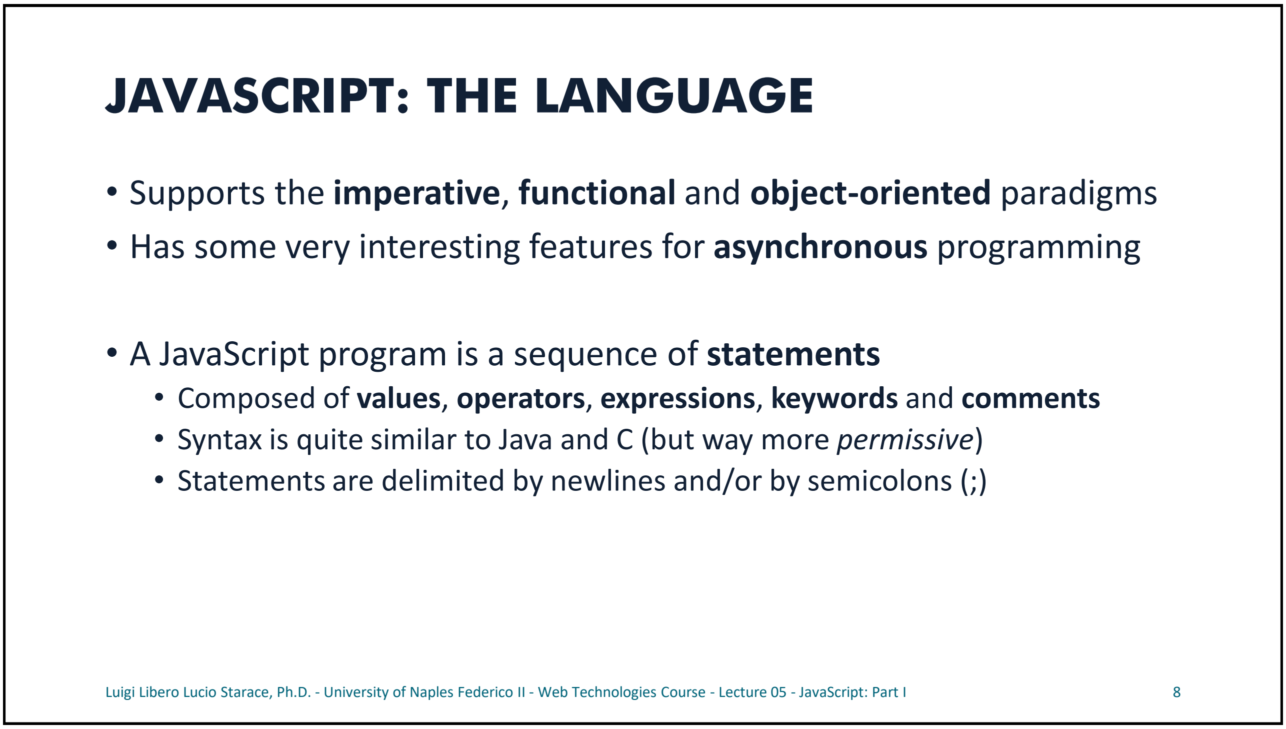
```
Hello World!
index.html:7:15
```



MODERN JAVASCRIPT

- For a long time, JavaScript evolved without **breaking changes**
  - New language features were added, and existing feature were not impacted
- This was the case until 2009, when ECMAScript 5 (ES5) appeared
  - ES5 modified some existing features
  - To maintain **retro-compatibility**, these breaking changes are off by default
  - They can be enabled using the **"use strict"** directive in the first line of a script
- In this course, unless explicitly noted, we will refer to the **modern, «strict»** version of JavaScript
  - Make sure to declare **"use strict"** on top of your scripts!

```
"use strict";

/* JavaScript code here*/
```



JAVASCRIPT: THE LANGUAGE

- Supports the **imperative**, **functional** and **object-oriented** paradigms
- Has some very interesting features for **asynchronous** programming

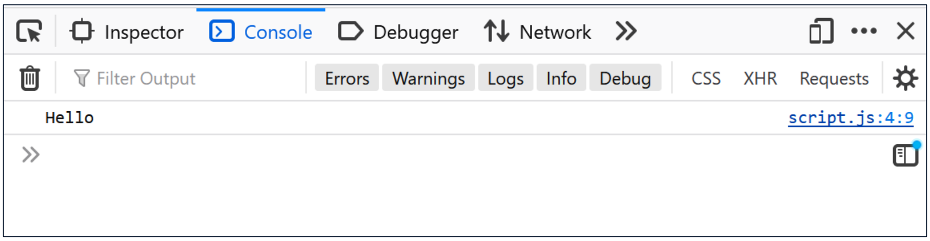
- A JavaScript program is a sequence of **statements**
  - Composed of **values**, **operators**, **expressions**, **keywords** and **comments**
  - Syntax is quite similar to Java and C (but way more *permissive*)
  - Statements are delimited by newlines and/or by semicolons (;)

STATEMENTS

```
//with semicolons
msg = "Hello";
num = 1;
console.log(msg);
```

```
//without  semicolons
msg = "Hello"
num = 1
console.log(msg)
```

```
/* statements on the same line
   (and multi-line comment) */

msg="Hello"; num=1; console.log(msg);
```



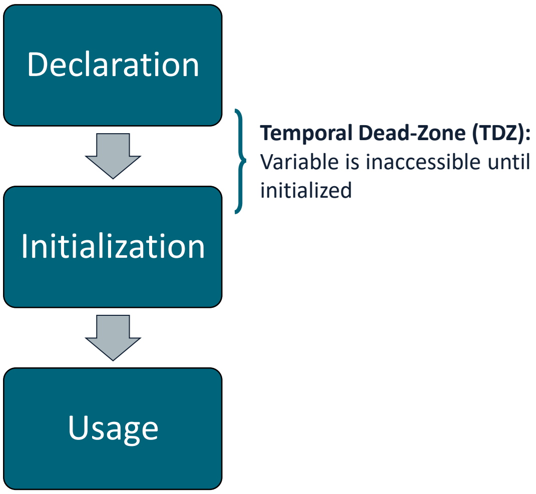
```
Hello                                                script.js:4:9
```

- Delimiting statements with a semicolon is a preferred **good practice**

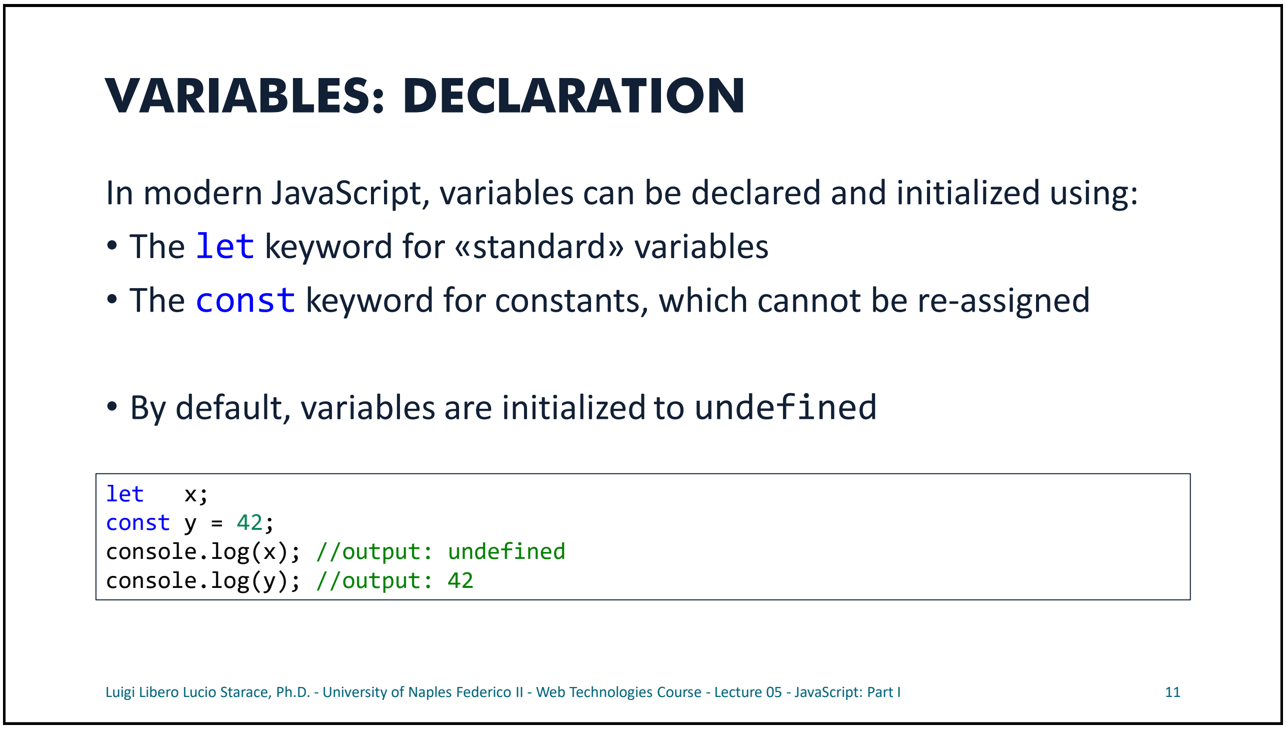
VARIABLES: LIFECYCLE

Declaring a variable in JavaScript consists of three distinct steps:

1. **Declaration:** variable name is bound to the current scope
2. **Initialization**: variable is initialized
3. **Usage**: variable can be referenced



Temporal Dead-Zone (TDZ): Variable is inaccessible until initialized



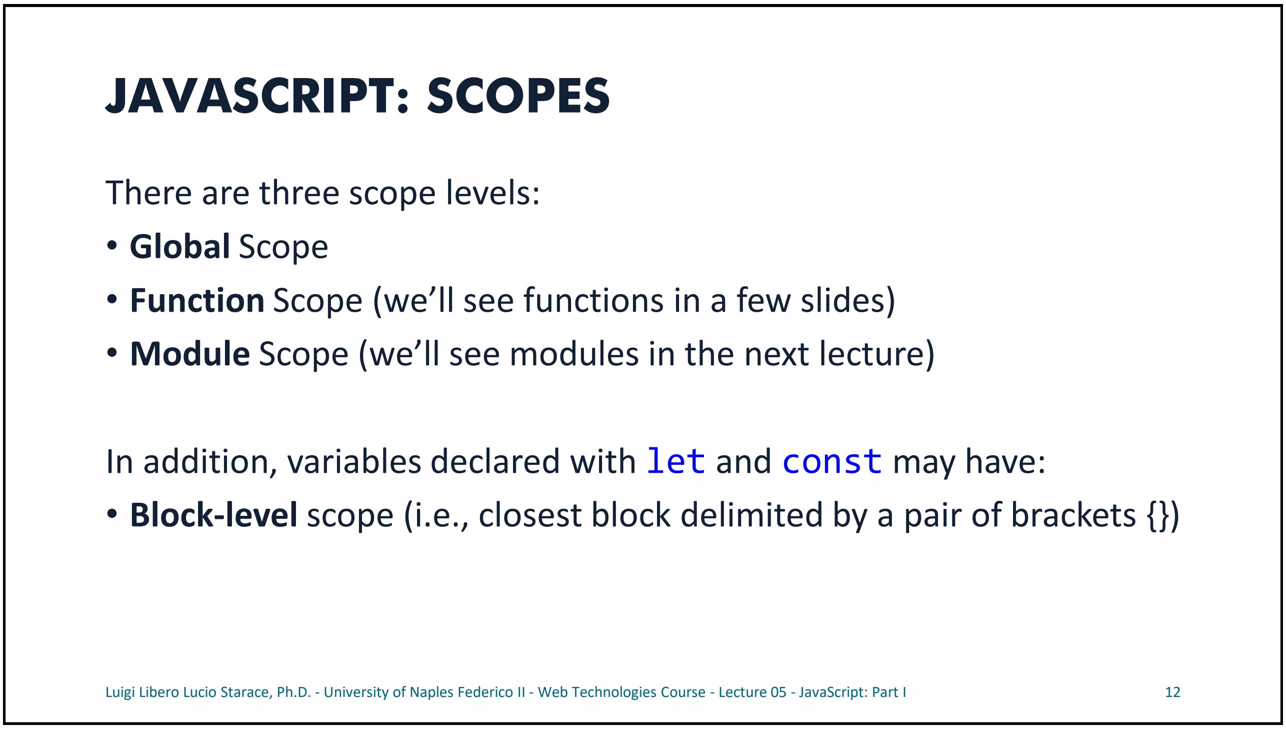
VARIABLES: DECLARATION

In modern JavaScript, variables can be declared and initialized using:

- The `let` keyword for «standard» variables
- The `const` keyword for constants, which cannot be re-assigned

- By default, variables are initialized to `undefined`

```
let   x;
const y = 42;
console.log(x); //output: undefined
console.log(y); //output: 42
```



JAVASCRIPT: SCOPES

There are three scope levels:

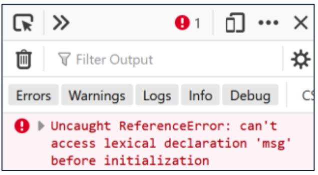
- **Global** Scope
- **Function** Scope (we'll see functions in a few slides)
- **Module** Scope (we'll see modules in the next lecture)

In addition, variables declared with `let` and `const` may have:

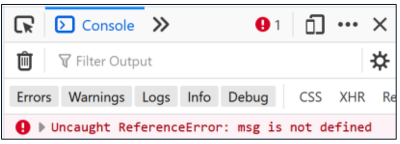
- **Block-level** scope (i.e., closest block delimited by a pair of brackets {})

VARIABLES AND SCOPE: LET KEYWORD

```
let bool; //variable declaration
bool = true;

if(bool){
  //console.log(msg);//msg not accessible here
  let msg = "Hello"; //declaration + assignment
  console.log(msg);  //output: Hello
}

//console.log(msg); //msg not defined here
console.log(bool);  //output: true
```



```
Uncaught ReferenceError: can't
access lexical declaration 'msg'
before initialization
```



```
Uncaught ReferenceError: msg is not defined
```

- The scope of variables declared using **let/const** is the closest block
- Variables are accessible only after the line they are declared in is executed

JAVASCRIPT: HOISTING

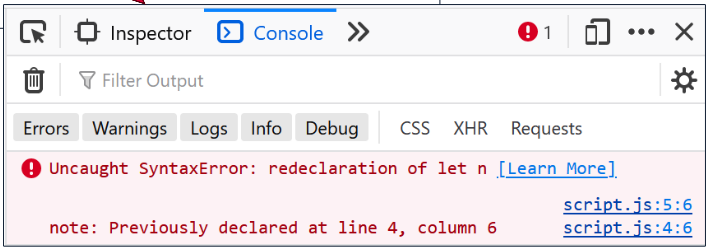
- **Hoisting** is the default behaviour of implicitly moving declarations of variables (and functions) at the beginning of their scope
- When using let and const, only the **declaration** is hoisted to the beginning of the scope, not the **initialization**
- The **initialization** happens on the line that initially contained the variable declaration

```
// x variable not defined
{
// Declaration for x is hoisted here
// TDZ for the x variable
// TDZ for the x variable
console.log(x); //raises error
// TDZ for the x variable
let x = 1; // TDZ ends, x initialized
// x variable initialized to 1
// x variable initialized to 1
}
// x variable not defined
```

```
ReferenceError: can't access
lexical declaration 'x' before
initialization
```

VARIABLES AND SCOPE: LET KEYWORD

```
let n; //variable declaration
n=1;
if(n>0){
  let n=2; //shadows n variable from external scope
  //let n; //error: cannot re-declare in the same block
  console.log(n); //output: 2
}
console.log(n); //output: 1
```

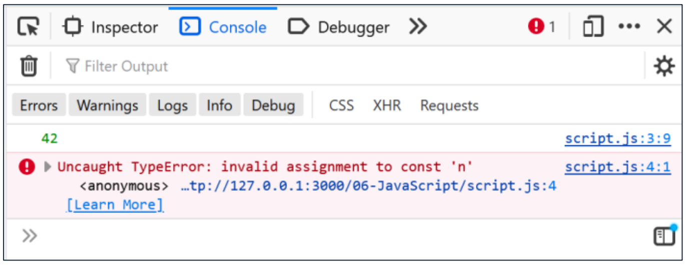


```
Uncaught SyntaxError: redeclaration of let n [Learn More]
                                                        script.js:5:6
note: Previously declared at line 4, column 6           script.js:4:6
```

VARIABLES AND SCOPE: CONST KEYWORD

- `const` can be used to declare block-scoped local constants
  - Variables whose value cannot be re-assigned
- Hoisting behaviour is similar to `let`

```
const n=42;
console.log(n); //output: 42
n=43; //raises a TypeError
```



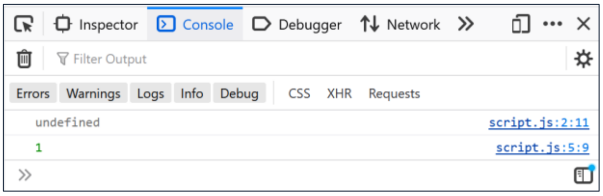
```
42                                                       script.js:3:9
Uncaught TypeError: invalid assignment to const 'n'      script.js:4:1
    <anonymous> …tp://127.0.0.1:3000/06-JavaScript/script.js:4
  [Learn More]
```

VARIABLES: BEFORE ECMASCRIPT 6

Before the introduction of ECMAScript 6 (2015), variables could be declared using the `var` keyword, or even **implicitly**

- Unless you **really** need to support very old browsers, you should not declare variable in these ways
- They have some rather **confusing** and **error-prone** properties

```
if(true){
  console.log(n);
  var n=1;
}
console.log(n);
```



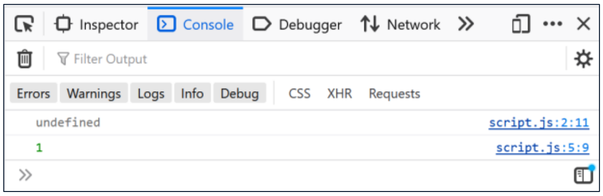
```
undefined                                  script.js:2:11
1                                           script.js:5:9
```

VARIABLES AND SCOPE: VAR HOISTING

- Variables declared using `var` are hoisted to the closest function or global scope (no block-level scope) and are also initialized to undefined!

```
if(true){
  console.log(n); //output: undefined
  var n=1;
}
console.log(n); //output: 1
```

```
var n;
if(true){
  console.log(n); //output: undefined
  n=1;
}
console.log(n); //output: 1
```

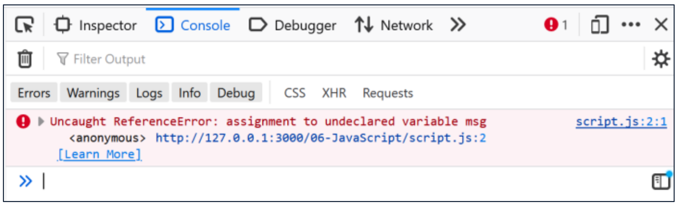


```
undefined                                  script.js:2:11
1                                           script.js:5:9
```

VARIABLES AND SCOPE: IMPLICIT DECL.

- Variables can also be implicitly declared, for example by simply using them in an assignment without any keyword
- Doing so results in the creation of a **global variable**
- **It's a bad practice, very error prone, and you should never do this**
- It is also **forbidden** in the JavaScript **«strict mode»**, and results in a ReferenceError

```
"use strict"

msg = "pls don't do this"
```

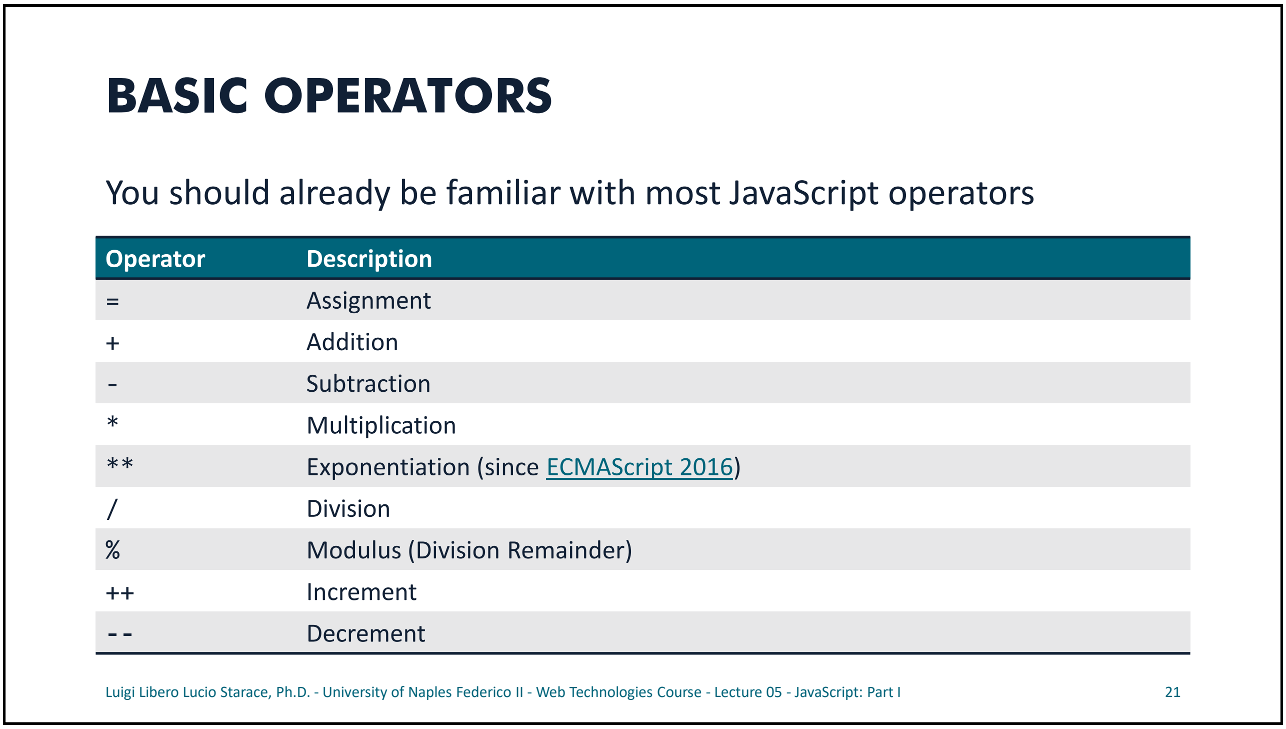


```
Uncaught ReferenceError: assignment to undeclared variable msg        script.js:2:1
    <anonymous> http://127.0.0.1:3000/06-JavaScript/script.js:2
   [Learn More]
```

PRIMITIVE DATA TYPES

```
let x; // variable declared and initialized to undefined
console.log(typeof x); // undefined

x=42;
console.log(typeof x); // number
x=3.14;
console.log(typeof x); // number
x="hello";
console.log(typeof x); // string
x='hello';
console.log(typeof x); // string
x=false;
console.log(typeof x); // boolean
x=10e12;
console.log(typeof x); // number
```

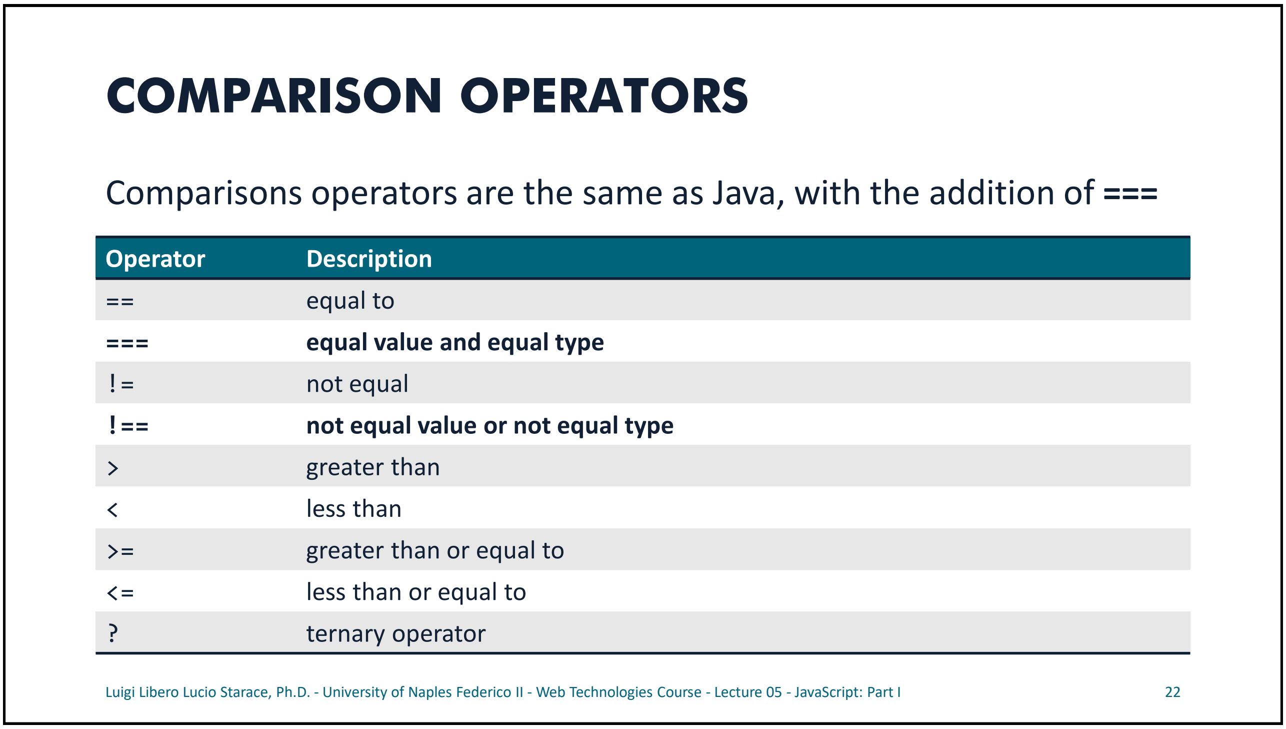
```
x=42/0;                 // Infinity
console.log(typeof x); // number
x=42 * "Hello";         // NaN
console.log(typeof x); // number
```



BASIC OPERATORS

You should already be familiar with most JavaScript operators

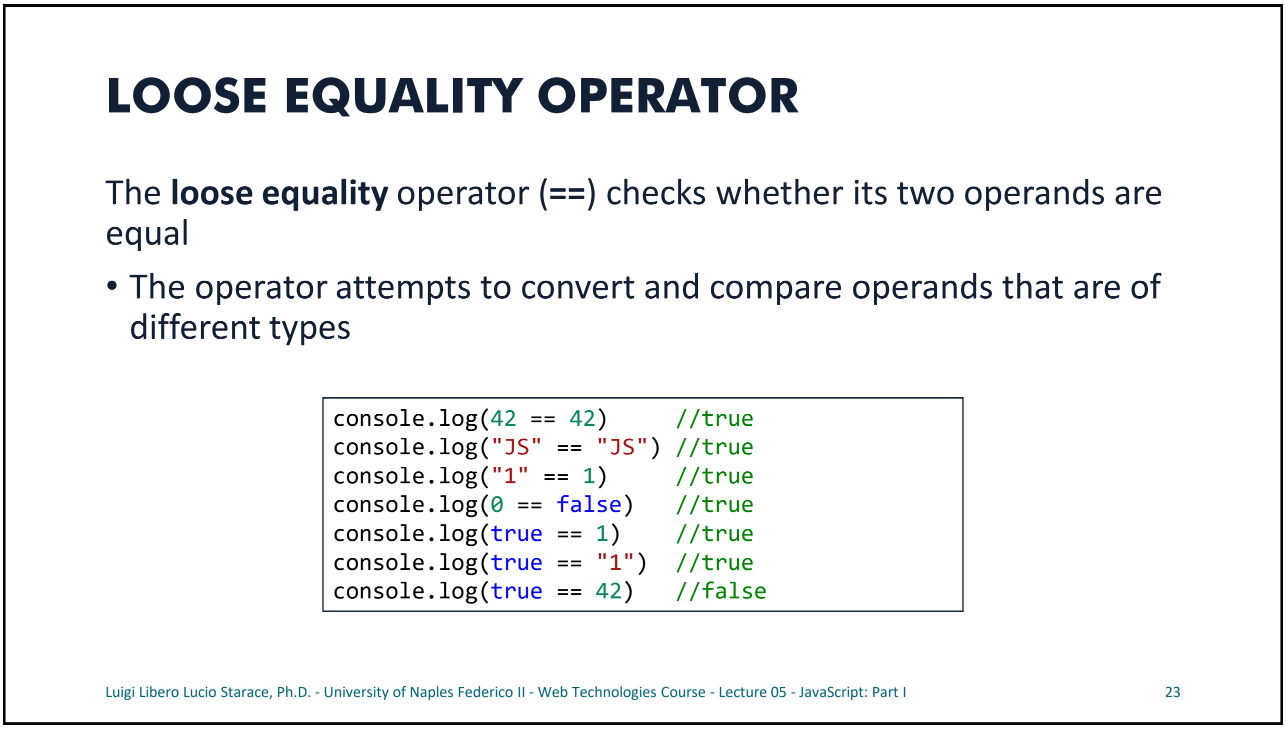
Operator	Description
=	Assignment
+	Addition
-	Subtraction
*	Multiplication
**	Exponentiation (since ECMAScript 2016)
/	Division
%	Modulus (Division Remainder)
++	Increment
--	Decrement



COMPARISON OPERATORS

Comparisons operators are the same as Java, with the addition of ===

Operator	Description
==	equal to
===	**equal value and equal type**
!=	not equal
!==	**not equal value or not equal type**
>	greater than
<	less than
>=	greater than or equal to
<=	less than or equal to
?	ternary operator

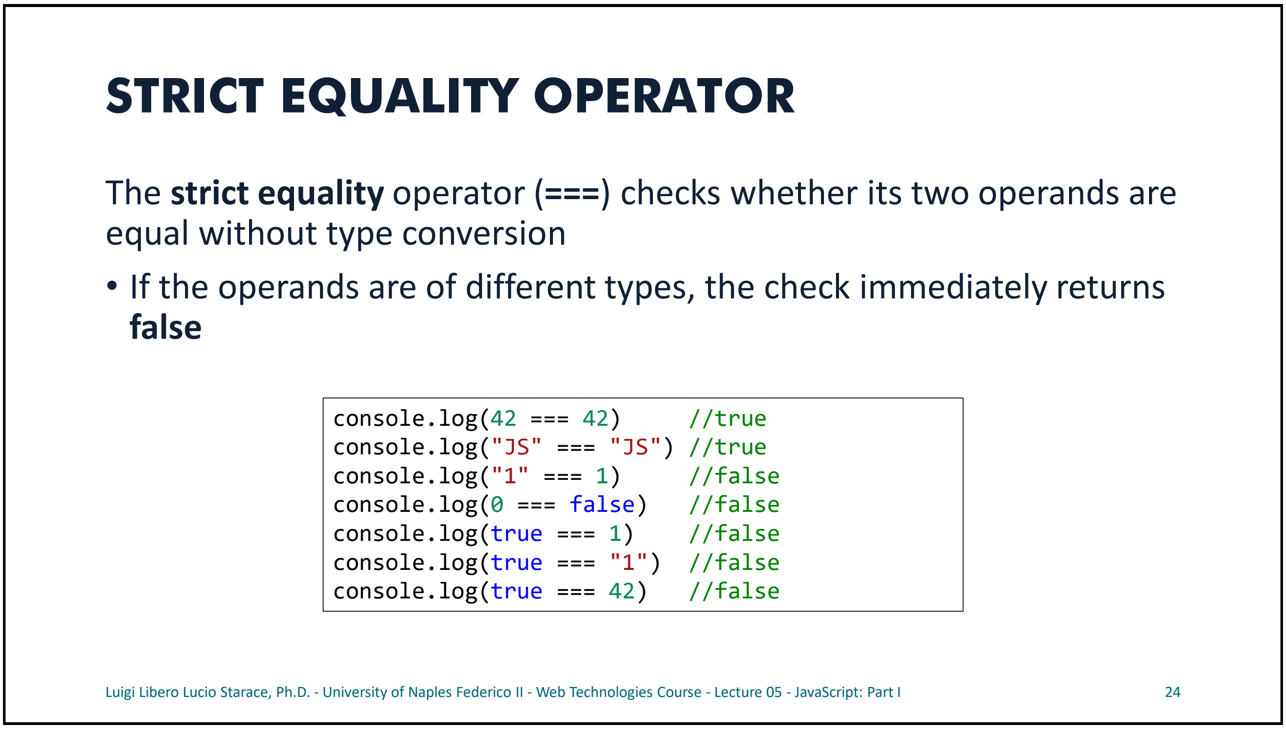


LOOSE EQUALITY OPERATOR

The **loose equality** operator (==) checks whether its two operands are equal

- The operator attempts to convert and compare operands that are of different types

```
console.log(42 == 42)     //true
console.log("JS" == "JS") //true
console.log("1" == 1)     //true
console.log(0 == false)   //true
console.log(true == 1)    //true
console.log(true == "1")  //true
console.log(true == 42)   //false
```

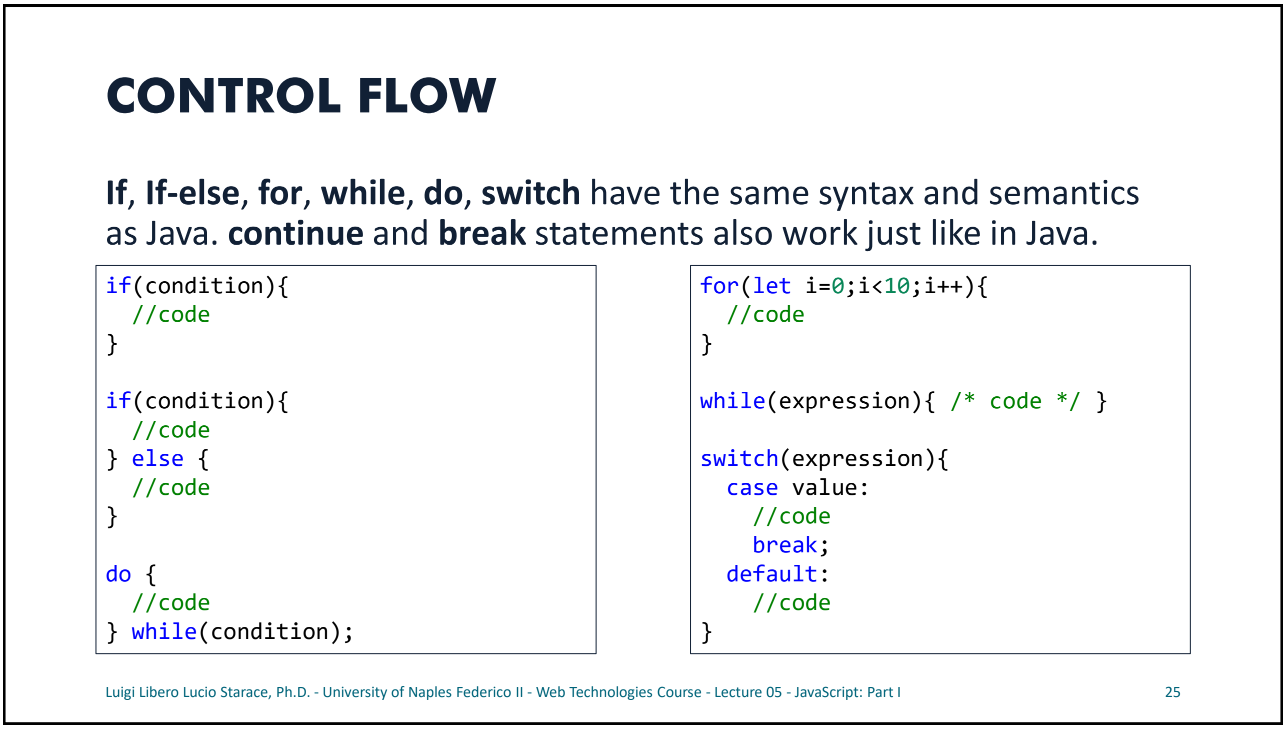


STRICT EQUALITY OPERATOR

The **strict equality** operator (===) checks whether its two operands are equal without type conversion

- If the operands are of different types, the check immediately returns **false**

```
console.log(42 === 42)     //true
console.log("JS" === "JS") //true
console.log("1" === 1)     //false
console.log(0 === false)   //false
console.log(true === 1)    //false
console.log(true === "1")  //false
console.log(true === 42)   //false
```



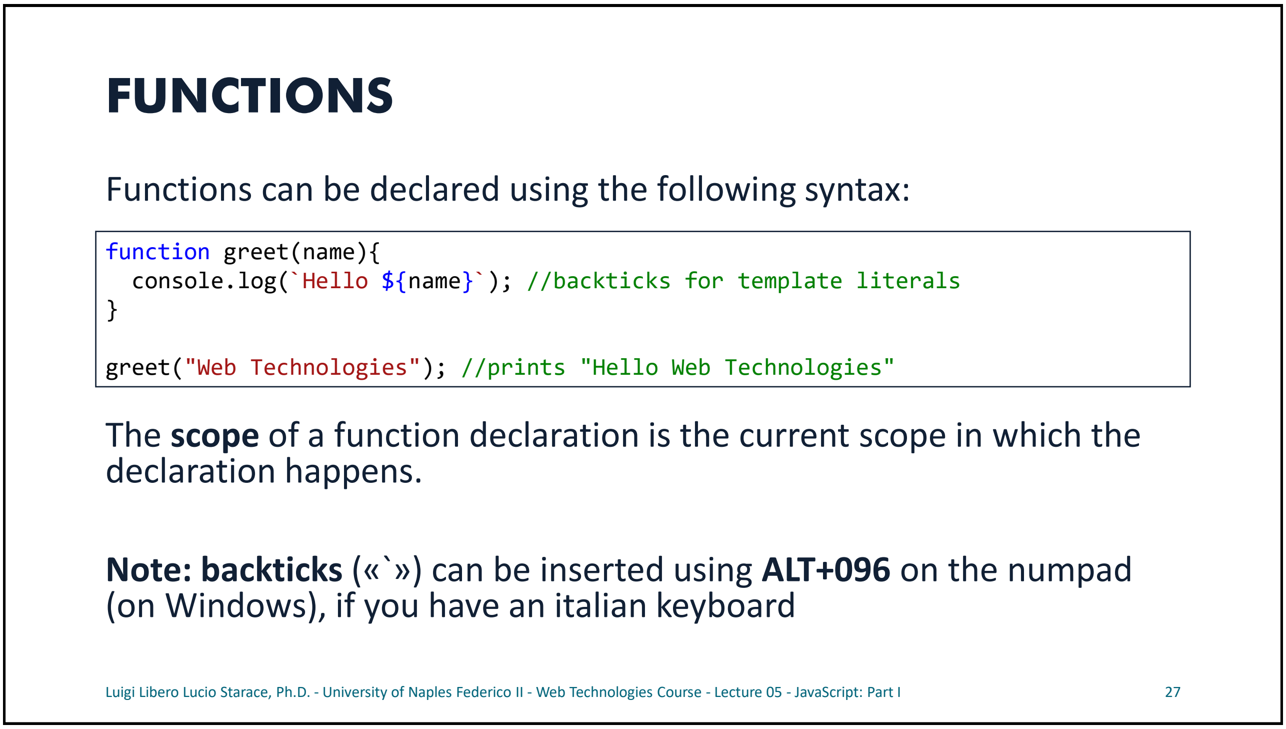
CONTROL FLOW

If, **If-else**, **for**, **while**, **do**, **switch** have the same syntax and semantics as Java. **continue** and **break** statements also work just like in Java.

```
if(condition){
  //code
}

if(condition){
  //code
} else {
  //code
}

do {
  //code
} while(condition);
```

```
for(let i=0;i<10;i++){
  //code
}

while(expression){ /* code */ }

switch(expression){
  case value:
    //code
    break;
  default:
    //code
}
```

FUNCTIONS



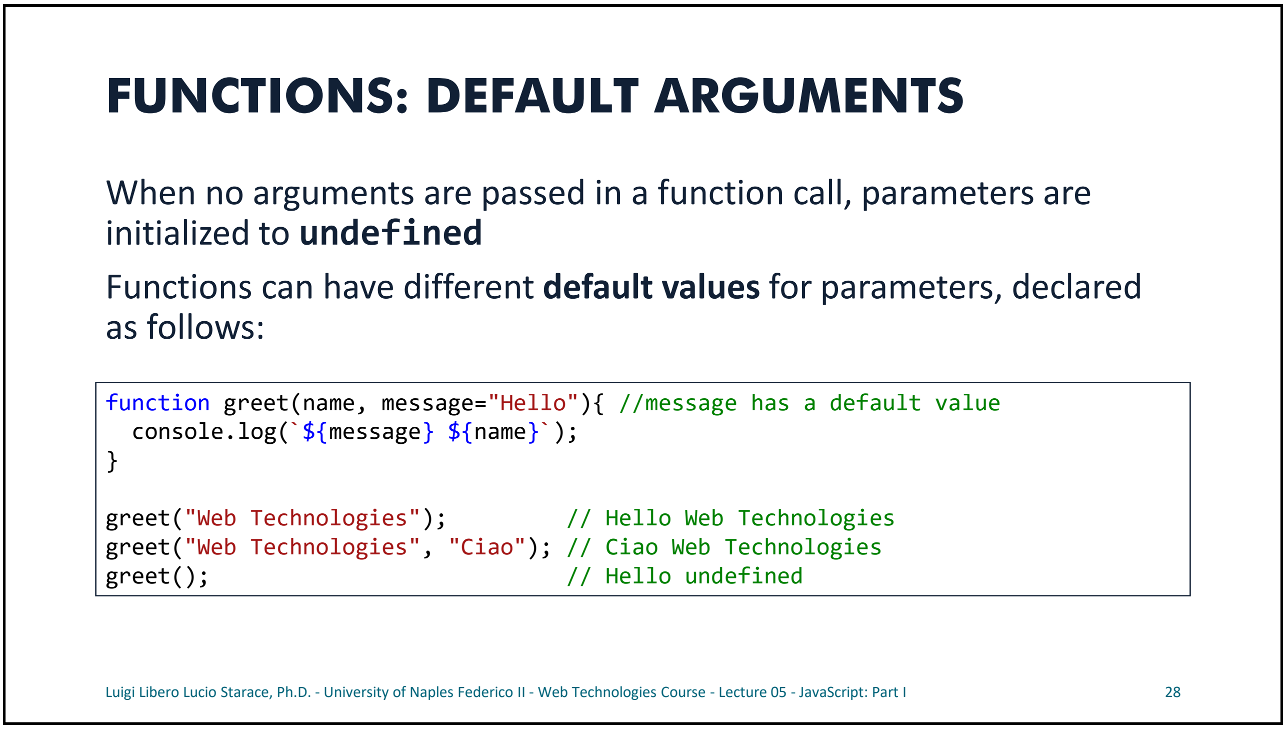
FUNCTIONS

Functions can be declared using the following syntax:

```
function greet(name){
  console.log(`Hello ${name}`); //backticks for template literals
}

greet("Web Technologies"); //prints "Hello Web Technologies"
```

The **scope** of a function declaration is the current scope in which the declaration happens.

Note: backticks («`») can be inserted using **ALT+096** on the numpad (on Windows), if you have an italian keyboard

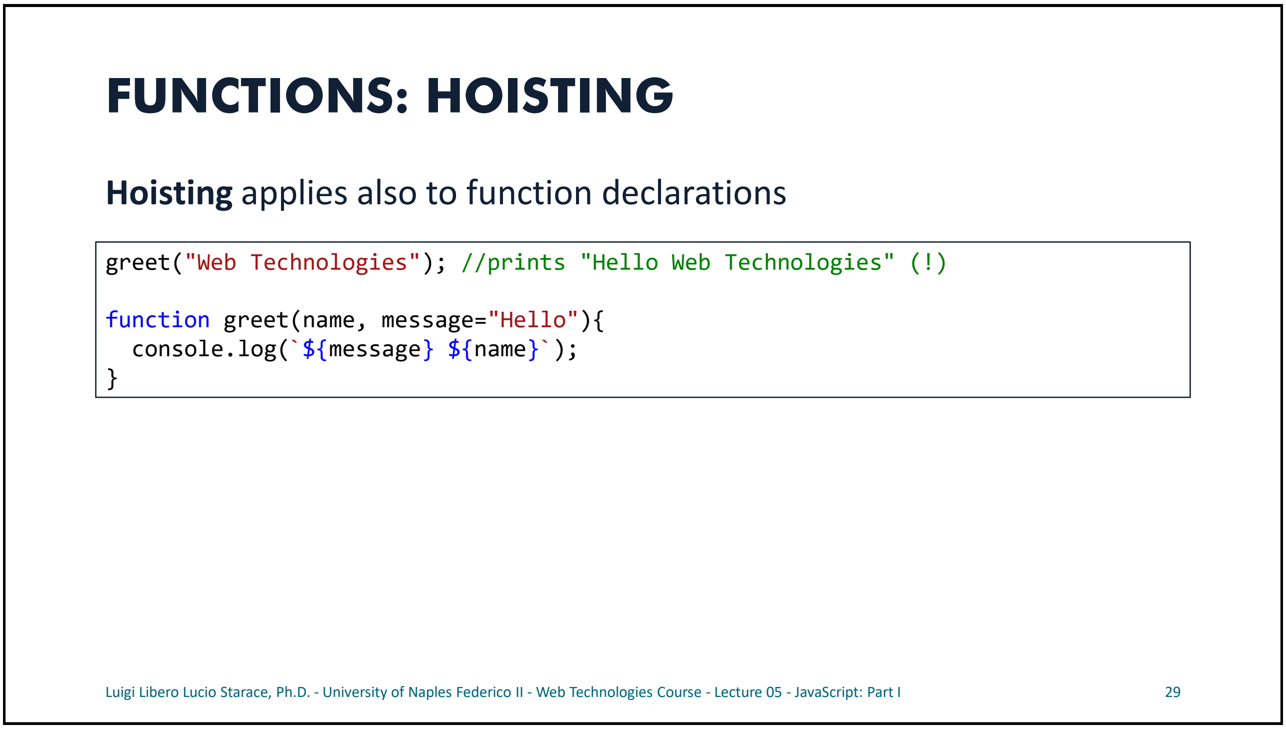


FUNCTIONS: DEFAULT ARGUMENTS

When no arguments are passed in a function call, parameters are initialized to **`undefined`**

Functions can have different **default values** for parameters, declared as follows:

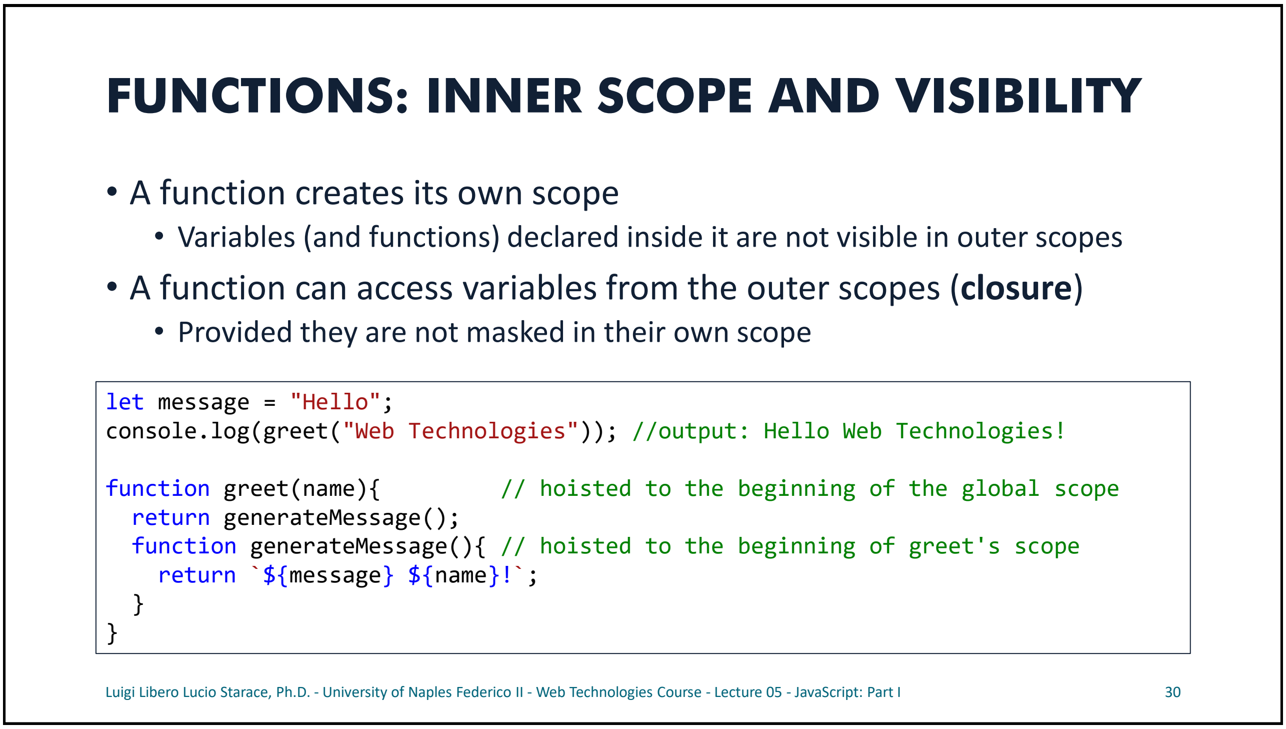
```
function greet(name, message="Hello"){ //message has a default value
  console.log(`${message} ${name}`);
}

greet("Web Technologies");         // Hello Web Technologies
greet("Web Technologies", "Ciao"); // Ciao Web Technologies
greet();                           // Hello undefined
```



FUNCTIONS: HOISTING

Hoisting applies also to function declarations

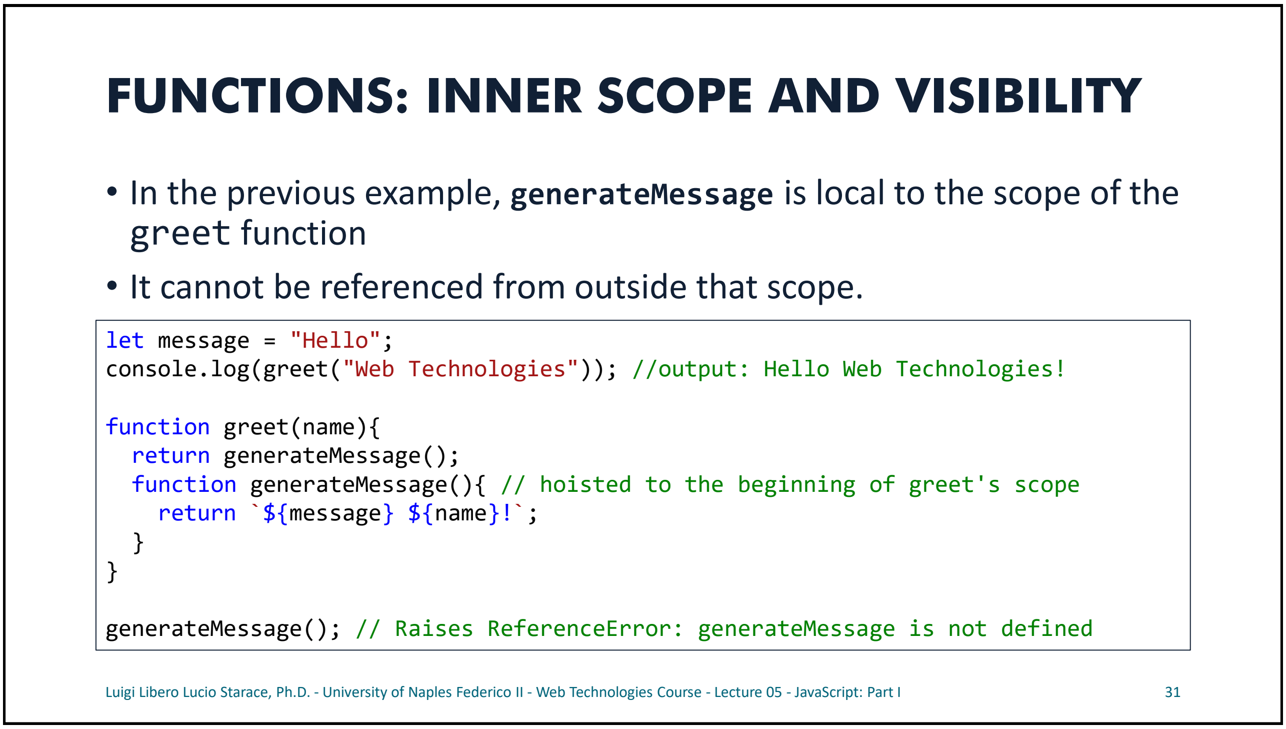
```
greet("Web Technologies"); //prints "Hello Web Technologies" (!)

function greet(name, message="Hello"){
  console.log(`${message} ${name}`);
}
```



FUNCTIONS: INNER SCOPE AND VISIBILITY

- A function creates its own scope
  - Variables (and functions) declared inside it are not visible in outer scopes
- A function can access variables from the outer scopes (**closure**)
  - Provided they are not masked in their own scope

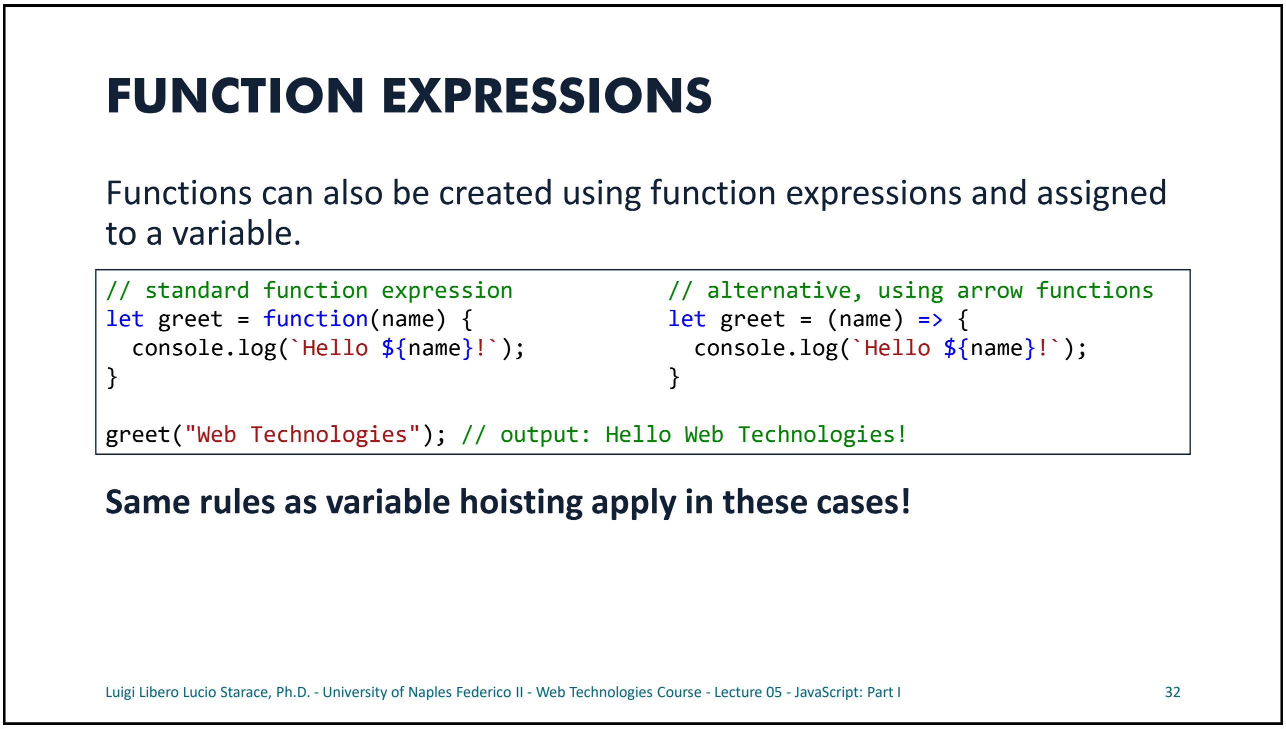
```
let message = "Hello";
console.log(greet("Web Technologies")); //output: Hello Web Technologies!

function greet(name){         // hoisted to the beginning of the global scope
  return generateMessage();
  function generateMessage(){ // hoisted to the beginning of greet's scope
    return `${message} ${name}!`;
  }
}
```



FUNCTIONS: INNER SCOPE AND VISIBILITY

- In the previous example, **`generateMessage`** is local to the scope of the `greet` function
- It cannot be referenced from outside that scope.

```
let message = "Hello";
console.log(greet("Web Technologies")); //output: Hello Web Technologies!

function greet(name){
  return generateMessage();
  function generateMessage(){ // hoisted to the beginning of greet's scope
    return `${message} ${name}!`;
  }
}

generateMessage(); // Raises ReferenceError: generateMessage is not defined
```

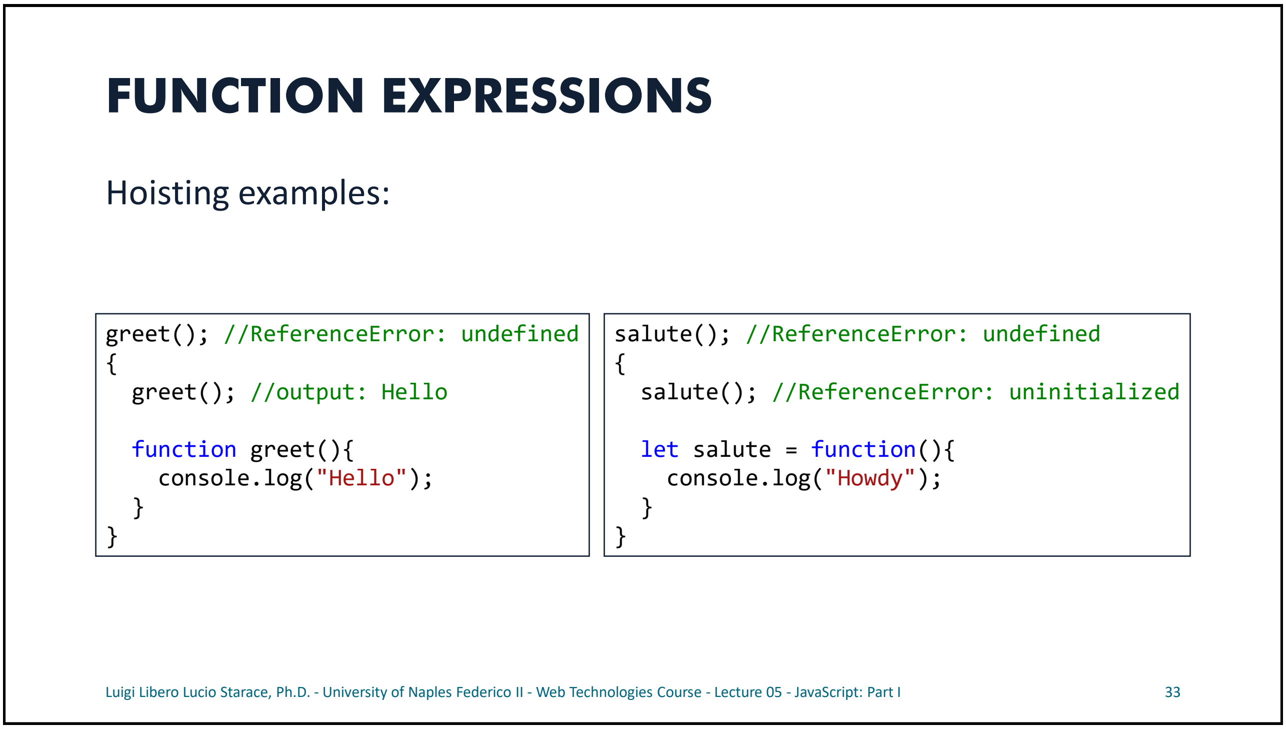


FUNCTION EXPRESSIONS

Functions can also be created using function expressions and assigned to a variable.

```
// standard function expression             // alternative, using arrow functions
let greet = function(name) {                let greet = (name) => {
  console.log(`Hello ${name}!`);              console.log(`Hello ${name}!`);
}                                           }

greet("Web Technologies"); // output: Hello Web Technologies!
```

Same rules as variable hoisting apply in these cases!

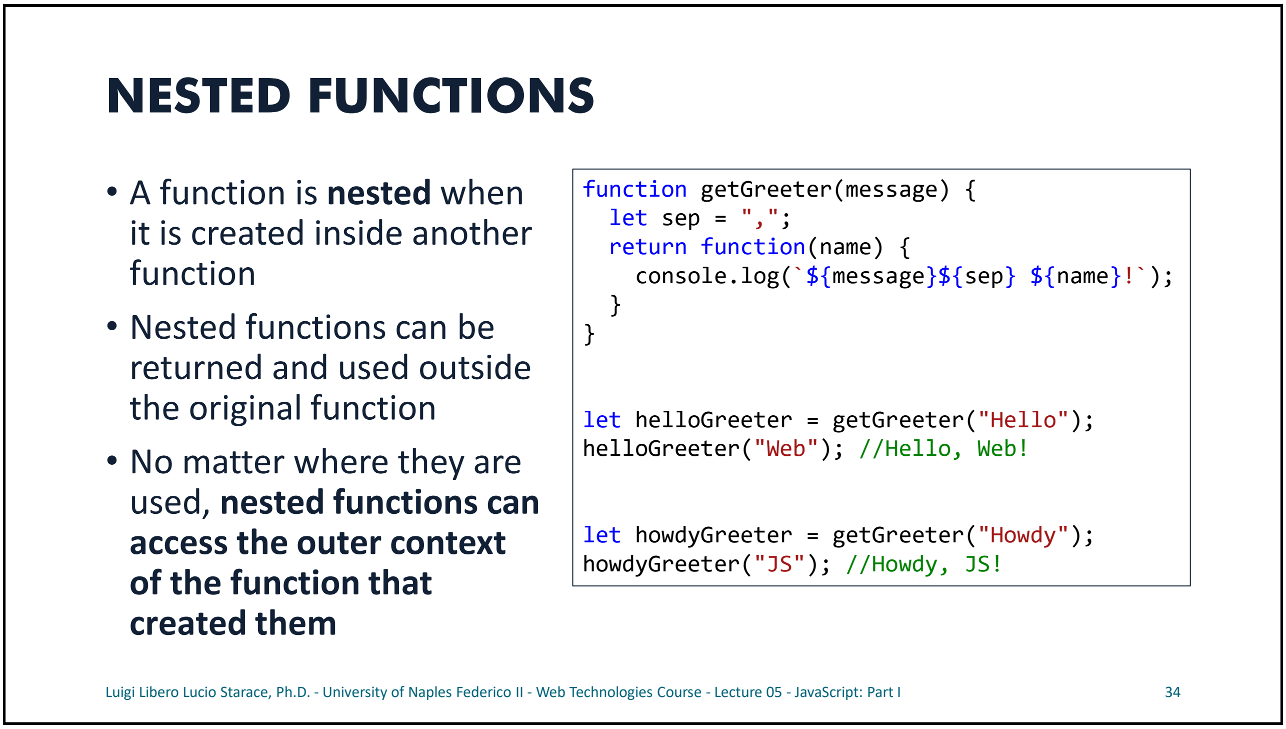


FUNCTION EXPRESSIONS

Hoisting examples:

```
greet(); //ReferenceError: undefined
{
  greet(); //output: Hello

  function greet(){
    console.log("Hello");
  }
}
```

```
salute(); //ReferenceError: undefined
{
  salute(); //ReferenceError: uninitialized

  let salute = function(){
    console.log("Howdy");
  }
}
```

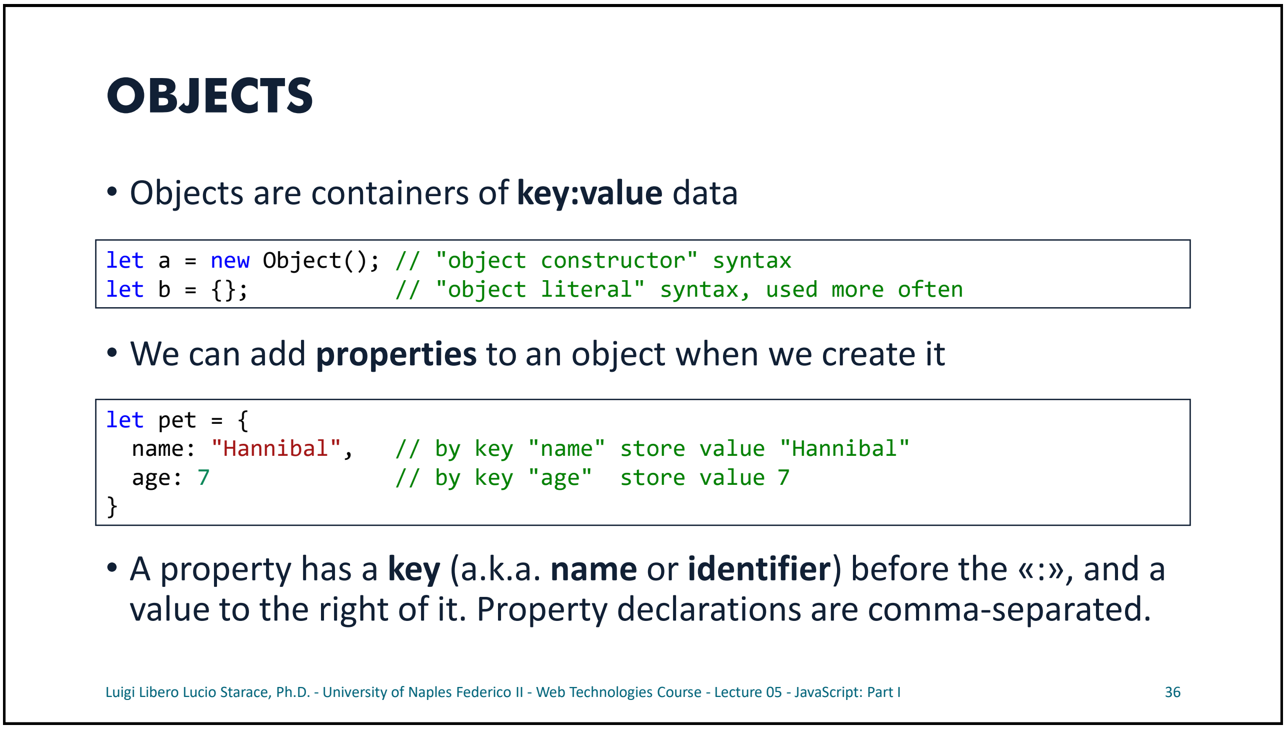


NESTED FUNCTIONS

- A function is **nested** when it is created inside another function
- Nested functions can be returned and used outside the original function
- No matter where they are used, **nested functions can access the outer context of the function that created them**

```
function getGreeter(message) {
  let sep = ",";
  return function(name) {
    console.log(`${message}${sep} ${name}!`);
  }
}


let helloGreeter = getGreeter("Hello");
helloGreeter("Web"); //Hello, Web!


let howdyGreeter = getGreeter("Howdy");
howdyGreeter("JS"); //Howdy, JS!
```

OBJECTS



OBJECTS

- Objects are containers of **key:value** data

```
let a = new Object(); // "object constructor" syntax
let b = {};           // "object literal" syntax, used more often
```

- We can add **properties** to an object when we create it

```
let pet = {
  name: "Hannibal",   // by key "name" store value "Hannibal"
  age: 7              // by key "age"  store value 7
}
```

- A property has a **key** (a.k.a. **name** or **identifier**) before the «:», and a value to the right of it. Property declarations are comma-separated.

OBJECTS: ACCESSING PROPERTIES

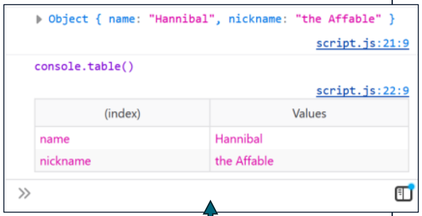
- Properties can be accessed using the **dot** notation

```
let pet = {
  name: "Hannibal",
  age: 7
}

console.log(pet.name);     // output: Hannibal
console.log(pet.age);      // output: 7
console.log(pet.nickname); // output: undefined
```

OBJECTS: ADDING/DELETING PROPERTIES

- Properties can also be added using the dot notation

```
let pet = {
  name: "Hannibal",
  age: 7
}

pet.nickname = "the Affable"; // new property
delete pet.age;               // delete property

console.log(pet.name);        // "Hannibal"
console.log(pet.age);         // undefined
console.log(pet.nickname);    // "the Affable"

console.log(pet);   // print object in console
console.table(pet); // alternative way to print objects in console
```



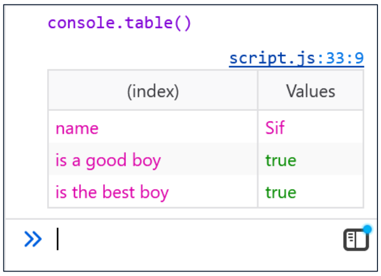
```
▸ Object { name: "Hannibal", nickname: "the Affable" }
                                          script.js:21:9
console.table()
                                          script.js:22:9
```

(index)	Values
name	Hannibal
nickname	the Affable

OBJECTS: SQUARE BRACKETS NOTATION

- Property keys can also contain spaces and be accessed using the square bracket notation:

```
let dog = {
  name: "Sif",
  "is a good boy": true
}

console.log(dog["name"]);
console.log(dog["is a good boy"]);
//expressions can be used as keys
dog["is the "+"best boy"] = true;

console.table(dog);
```



console.table()

script.js:33:9

(index)	Values
name	Sif
is a good boy	true
is the best boy	true

OBJECTS: REFERENCES

Variables assigned to an object store a **reference** to the object, not the object itself

```
let firstPet = {
  name: "Richard",
  species: "Lizard"
}

let secondPet = firstPet;
secondPet.name = "Chuck";
secondPet.species = "Duck";

console.log(firstPet);  // Object { name: "Chuck", species: "Duck" }
console.log(secondPet); // Object { name: "Chuck", species: "Duck" }
console.log(firstPet == secondPet);  // true, same value
console.log(firstPet === secondPet); // true (both are references, same value)
```

OBJECTS: CLONING

- How can we actually create a **clone** of an object?
- We need to iterate over each property, and copy them one by one

```
function clone(object){
  let copy = {}; // new object
  for(let key in object){
    copy[key] = object[key];
  }
  return copy;
}
```

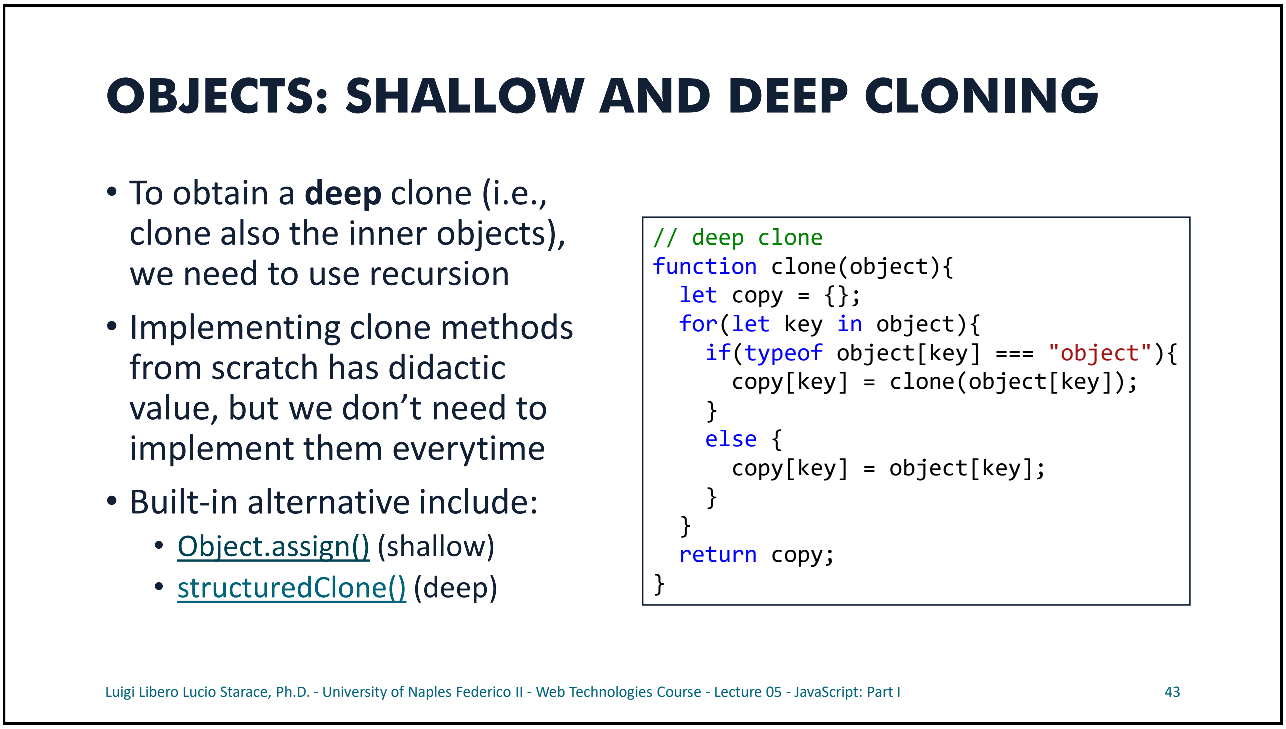
```
let pet = {
  name: "Richard", species: "Lizard"
}

let copy = clone(pet);

copy.name = "Chuck";
console.log(pet.name);  // Richard
console.log(copy.name); // Chuck
```

OBJECTS: SHALLOW AND DEEP CLONING

- The value of an object’s property might be another object
- The clone function we implemented creates a **shallow** clone
  - Inner objects are not cloned themselves, but only references are copied

```
function clone(object){
  let copy = {};
  for(let key in object){
    copy[key] = object[key];
  }
  return copy;
}
```

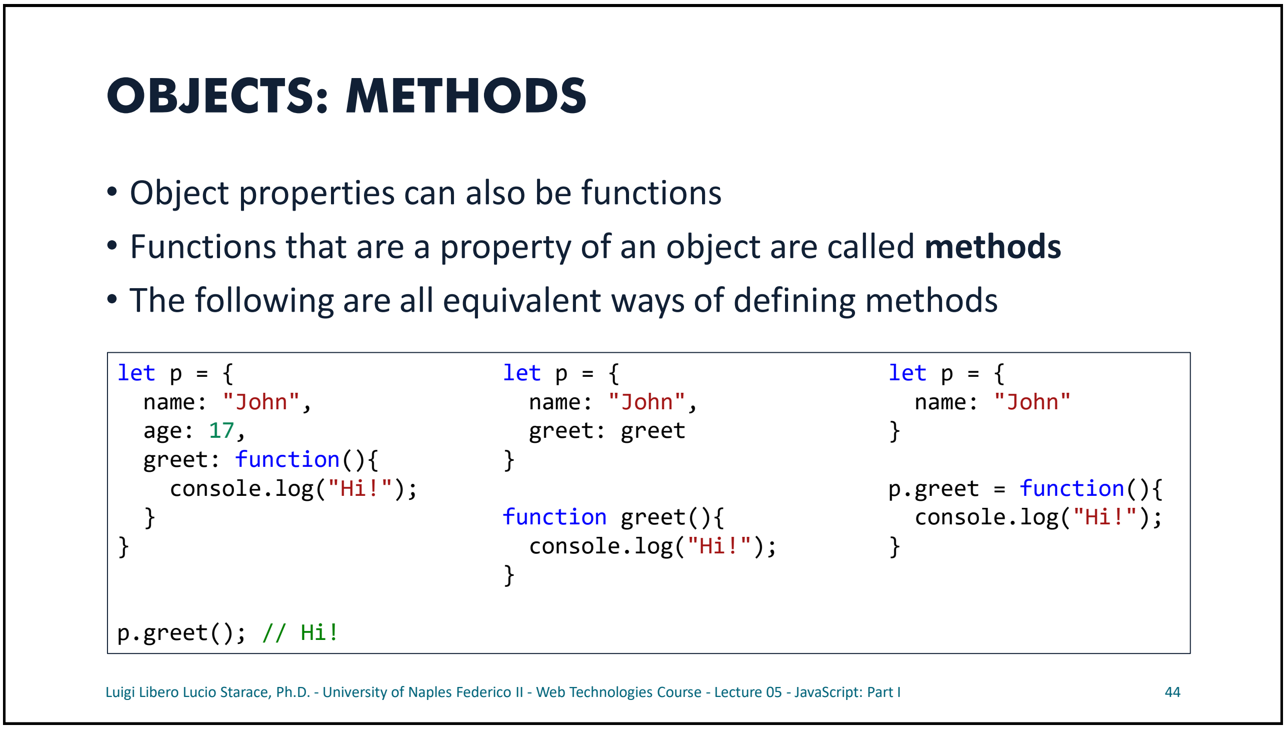
```
let pet = {
  name: "Garfield", species: "Cat",
  owner: {
    name: "John", age: 17
  }
}

let copy = clone(pet);
copy.owner.name = "Liz";
console.log(pet.owner.name);  // Liz
console.log(copy.owner.name); // Liz
```



OBJECTS: SHALLOW AND DEEP CLONING

- To obtain a **deep** clone (i.e., clone also the inner objects), we need to use recursion
- Implementing clone methods from scratch has didactic value, but we don't need to implement them everytime
- Built-in alternative include:
  - Object.assign() (shallow)
  - structuredClone() (deep)

```
// deep clone
function clone(object){
  let copy = {};
  for(let key in object){
    if(typeof object[key] === "object"){
      copy[key] = clone(object[key]);
    }
    else {
      copy[key] = object[key];
    }
  }
  return copy;
}
```



OBJECTS: METHODS

- Object properties can also be functions
- Functions that are a property of an object are called **methods**
- The following are all equivalent ways of defining methods

```
let p = {
  name: "John",
  age: 17,
  greet: function(){
    console.log("Hi!");
  }
}
```

```
let p = {
  name: "John",
  greet: greet
}

function greet(){
  console.log("Hi!");
}
```

```
let p = {
  name: "John"
}

p.greet = function(){
  console.log("Hi!");
}
```

```
p.greet(); // Hi!
```

OBJECTS: METHOD SHORTHAND

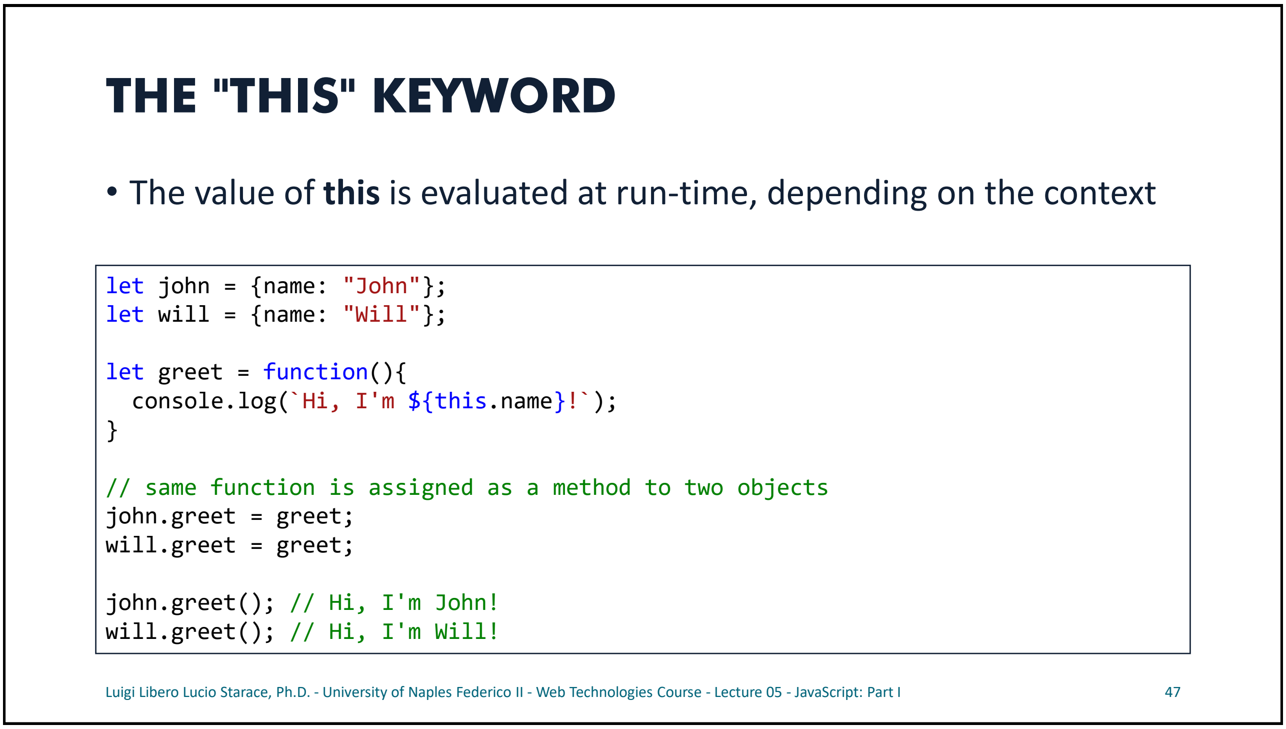
- There also exists a shorter syntax for declaring methods in an object literal

```
// classic version              // shorthand version
let p = {                       let p = {
  name: "John",                   name: "John",
  greet: function(){              greet(){
    console.log("Hi!");             console.log("Hi!");
  }                               }
}                               }


p.sayHi(); // Hi!
```

THE "THIS" KEYWORD

- It’s common for object methods to refer to other properties of the same object
- Methods can access the object containing them via the **this** keyword

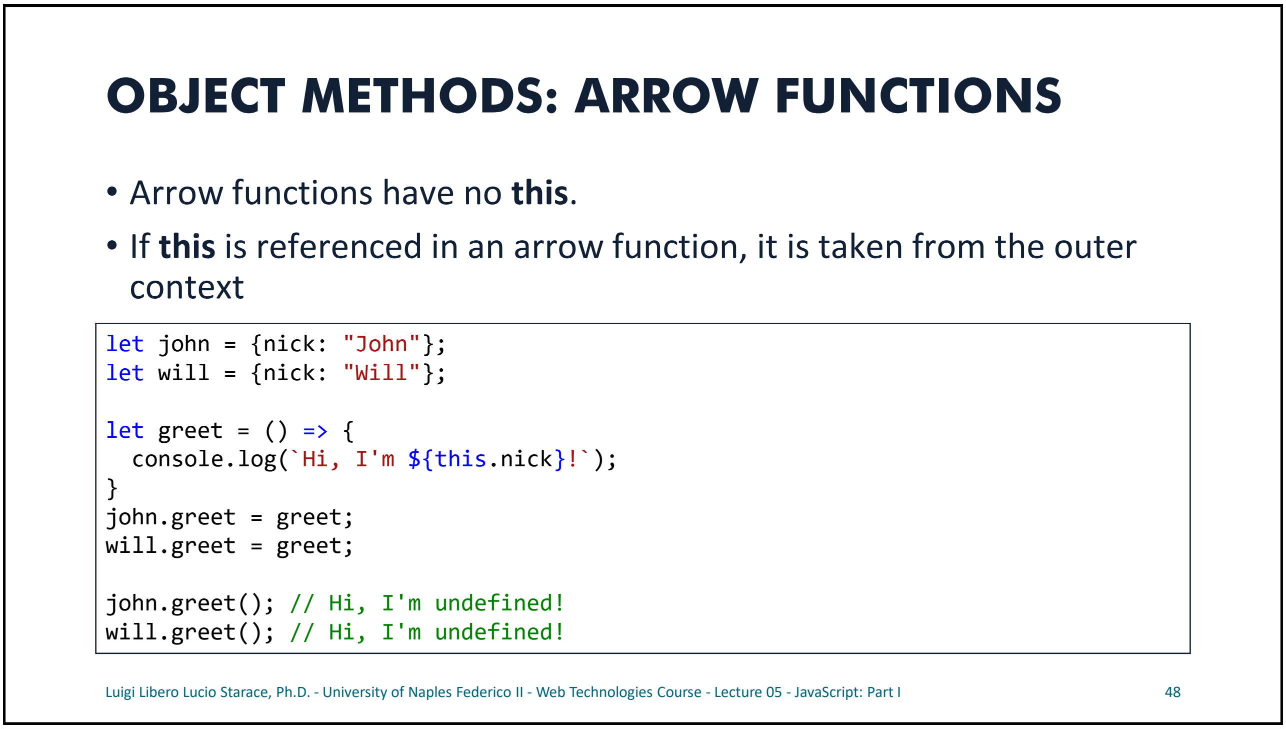
```
let john = {
  name: "John",
  greet(){
    console.log(`Hi, I'm ${this.name}!`);
  }
}

john.greet(); // Hi, I'm John!
```



THE "THIS" KEYWORD

- The value of **this** is evaluated at run-time, depending on the context

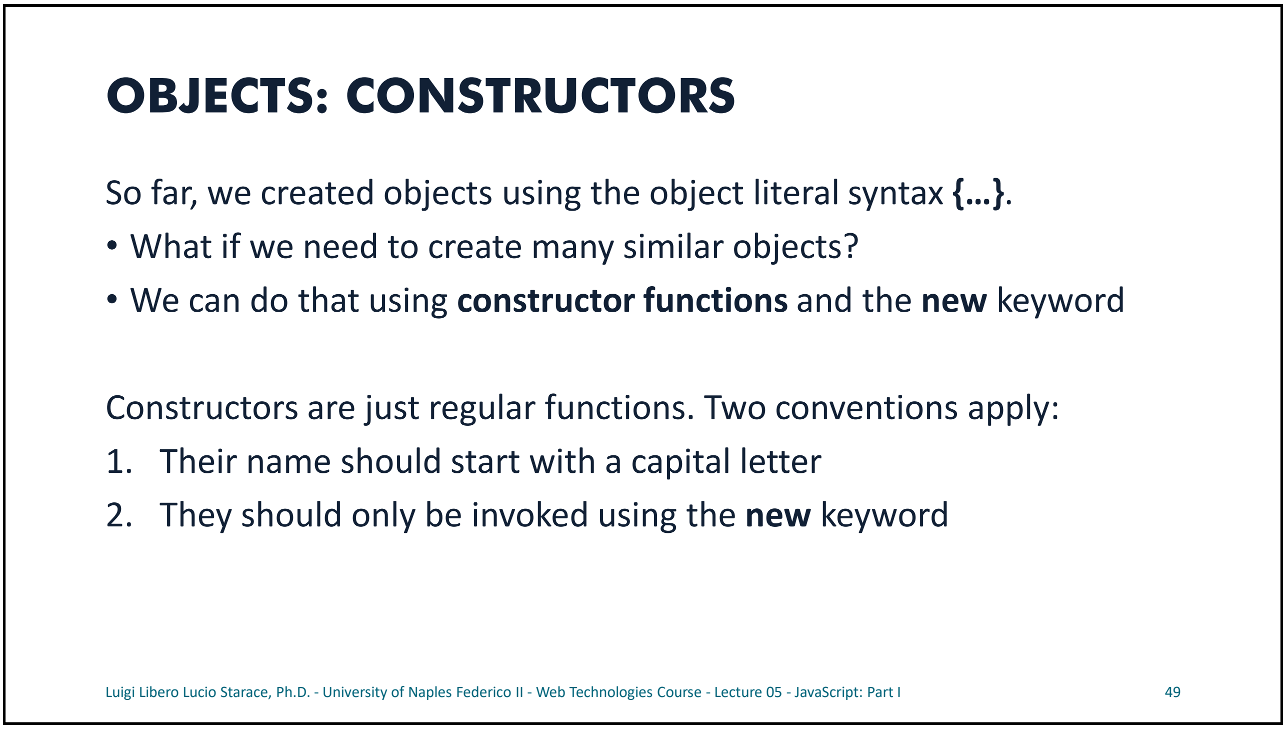
```
let john = {name: "John"};
let will = {name: "Will"};

let greet = function(){
  console.log(`Hi, I'm ${this.name}!`);
}

// same function is assigned as a method to two objects
john.greet = greet;
will.greet = greet;

john.greet(); // Hi, I'm John!
will.greet(); // Hi, I'm Will!
```



OBJECT METHODS: ARROW FUNCTIONS

- Arrow functions have no **this**.
- If **this** is referenced in an arrow function, it is taken from the outer context

```
let john = {nick: "John"};
let will = {nick: "Will"};

let greet = () => {
  console.log(`Hi, I'm ${this.nick}!`);
}
john.greet = greet;
will.greet = greet;

john.greet(); // Hi, I'm undefined!
will.greet(); // Hi, I'm undefined!
```



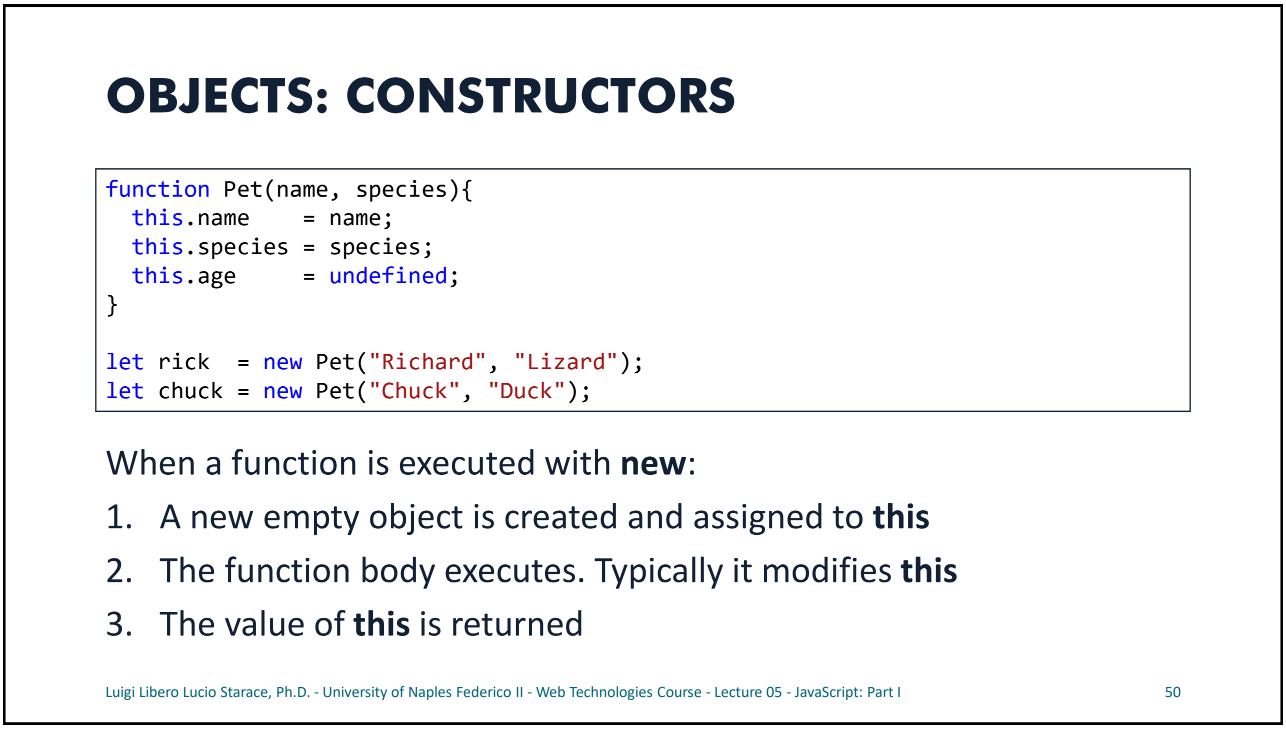
OBJECTS: CONSTRUCTORS

So far, we created objects using the object literal syntax **{…}**.

- What if we need to create many similar objects?
- We can do that using **constructor functions** and the **new** keyword

Constructors are just regular functions. Two conventions apply:

1. Their name should start with a capital letter
2. They should only be invoked using the **new** keyword



OBJECTS: CONSTRUCTORS

```
function Pet(name, species){
  this.name    = name;
  this.species = species;
  this.age     = undefined;
}

let rick  = new Pet("Richard", "Lizard");
let chuck = new Pet("Chuck", "Duck");
```

When a function is executed with **new**:

1. A new empty object is created and assigned to **this**
2. The function body executes. Typically it modifies **this**
3. The value of **this** is returned

OBJECTS: OPTIONAL CHAINING

- If we access an undefined property we get an undefined value

```
let pet = {
  name: "Garfield"
}
console.log(pet.species); // undefined
```

- Sometimes, we might try to access a property of an undefined property, which results in an error

```
console.log(pet.owner.name); //throws ReferenceError
```

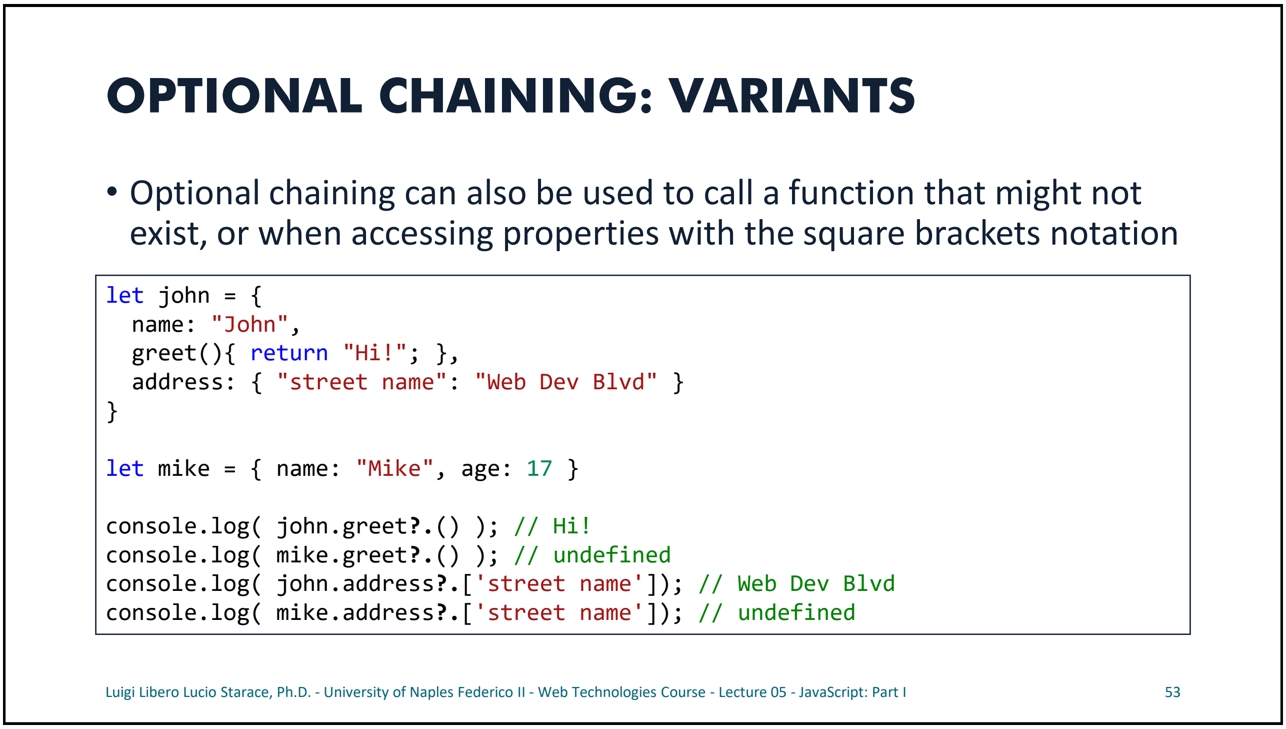
OBJECTS: OPTIONAL CHAINING

- In many practical cases, we might prefer getting an undefined value (e.g.: meaning that there is no known pet owner name) rather than an error.
- We could explicitly check that a property is defined before accessing its inner properties, but that’s quite repetitive and unelegant

```
let ownerName = pet.owner ? pet.owner.name : undefined;
```

- The optional chaining operator «**?.**» is handy in these situations
  - It immediately stops («short-circuits») the evaluation if the left part is undefined, returning **undefined**

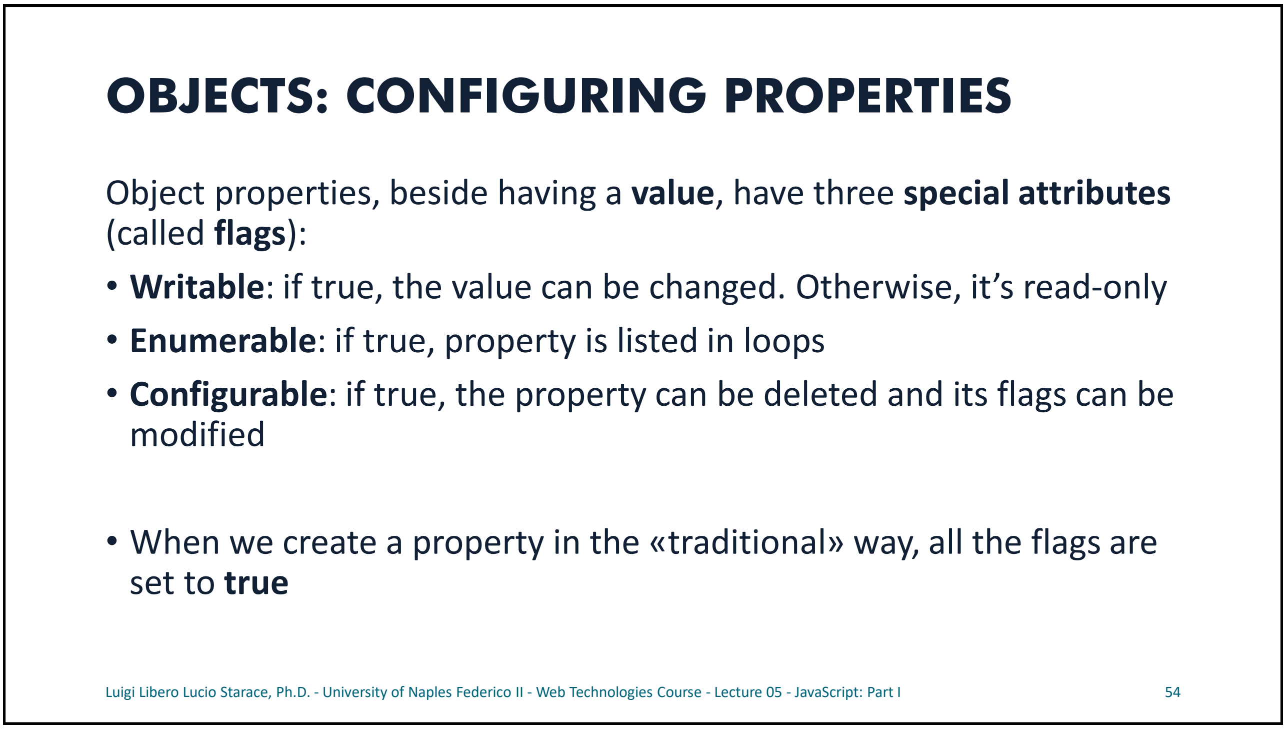
```
let ownerName = pet.owner?.name;
```



OPTIONAL CHAINING: VARIANTS

- Optional chaining can also be used to call a function that might not exist, or when accessing properties with the square brackets notation

```
let john = {
  name: "John",
  greet(){ return "Hi!"; },
  address: { "street name": "Web Dev Blvd" }
}

let mike = { name: "Mike", age: 17 }

console.log( john.greet?.() ); // Hi!
console.log( mike.greet?.() ); // undefined
console.log( john.address?.['street name']); // Web Dev Blvd
console.log( mike.address?.['street name']); // undefined
```

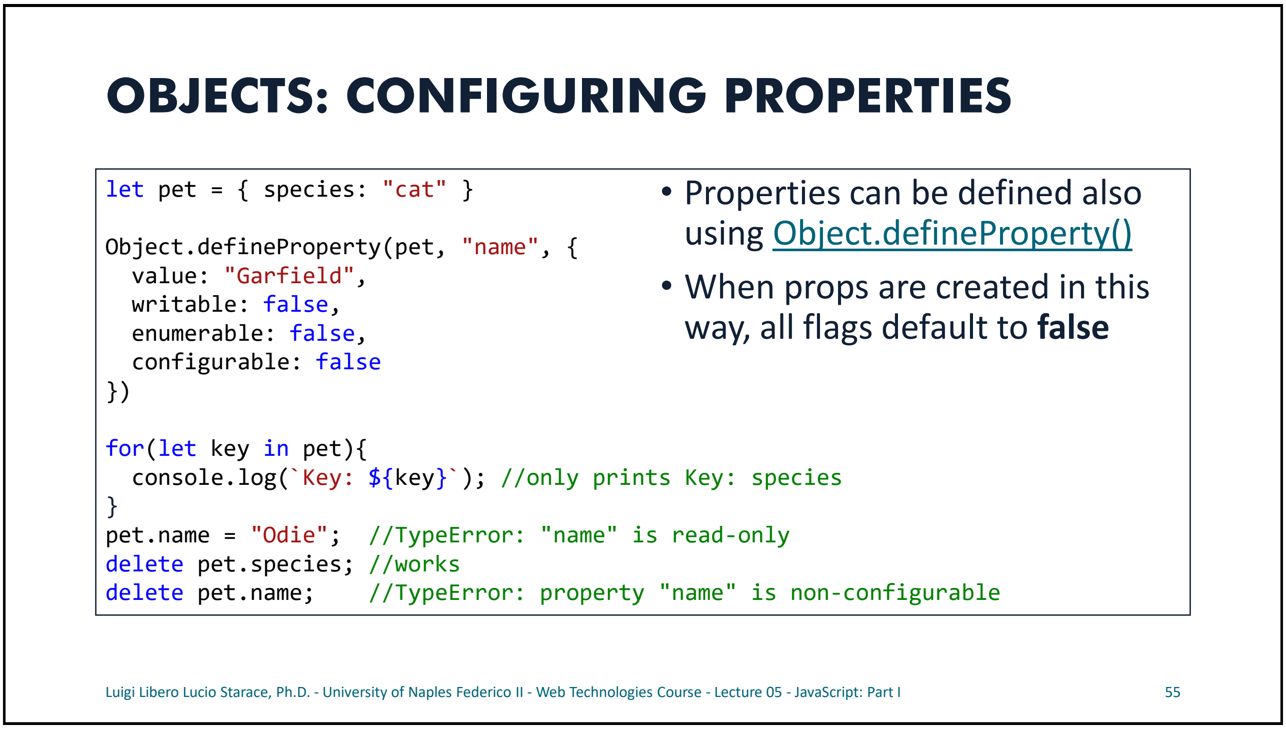


OBJECTS: CONFIGURING PROPERTIES

Object properties, beside having a **value**, have three **special attributes** (called **flags**):

- **Writable**: if true, the value can be changed. Otherwise, it's read-only
- **Enumerable**: if true, property is listed in loops
- **Configurable**: if true, the property can be deleted and its flags can be modified

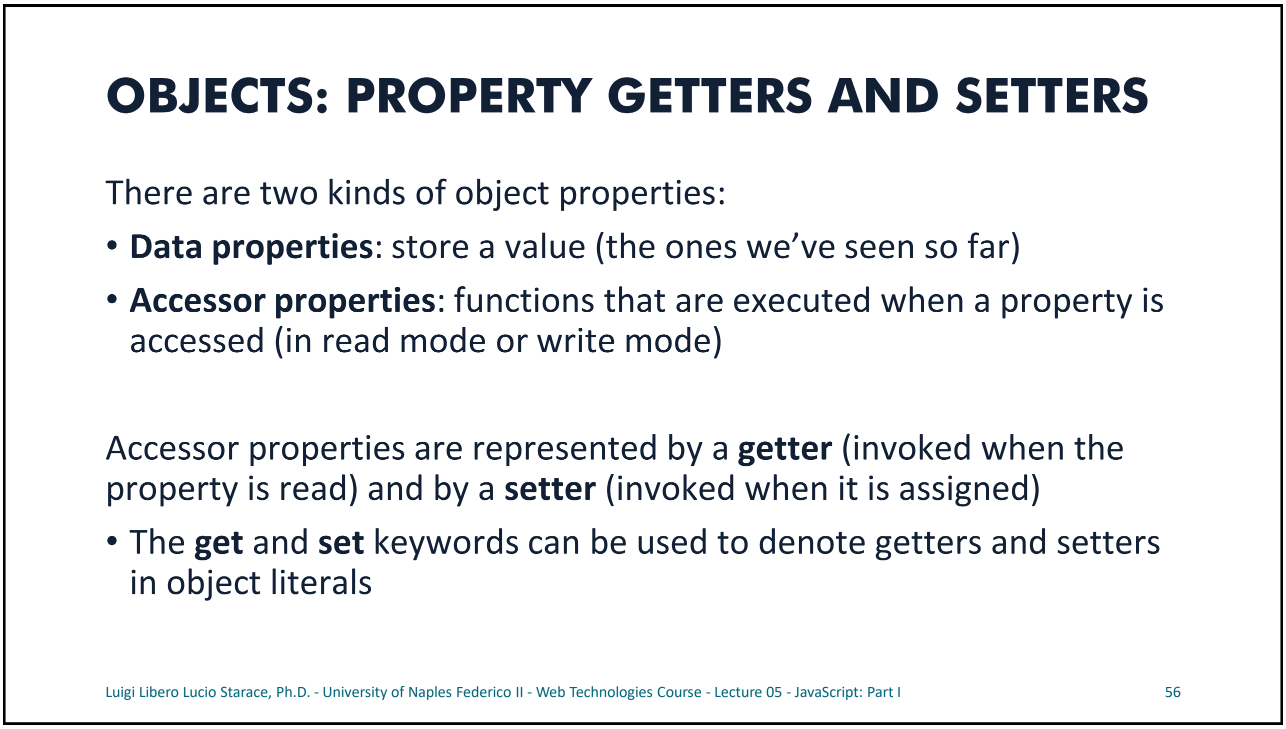
- When we create a property in the «traditional» way, all the flags are set to **true**



OBJECTS: CONFIGURING PROPERTIES

```
let pet = { species: "cat" }

Object.defineProperty(pet, "name", {
  value: "Garfield",
  writable: false,
  enumerable: false,
  configurable: false
})

for(let key in pet){
  console.log(`Key: ${key}`); //only prints Key: species
}
pet.name = "Odie";  //TypeError: "name" is read-only
delete pet.species; //works
delete pet.name;    //TypeError: property "name" is non-configurable
```

- Properties can be defined also using Object.defineProperty()
- When props are created in this way, all flags default to **false**



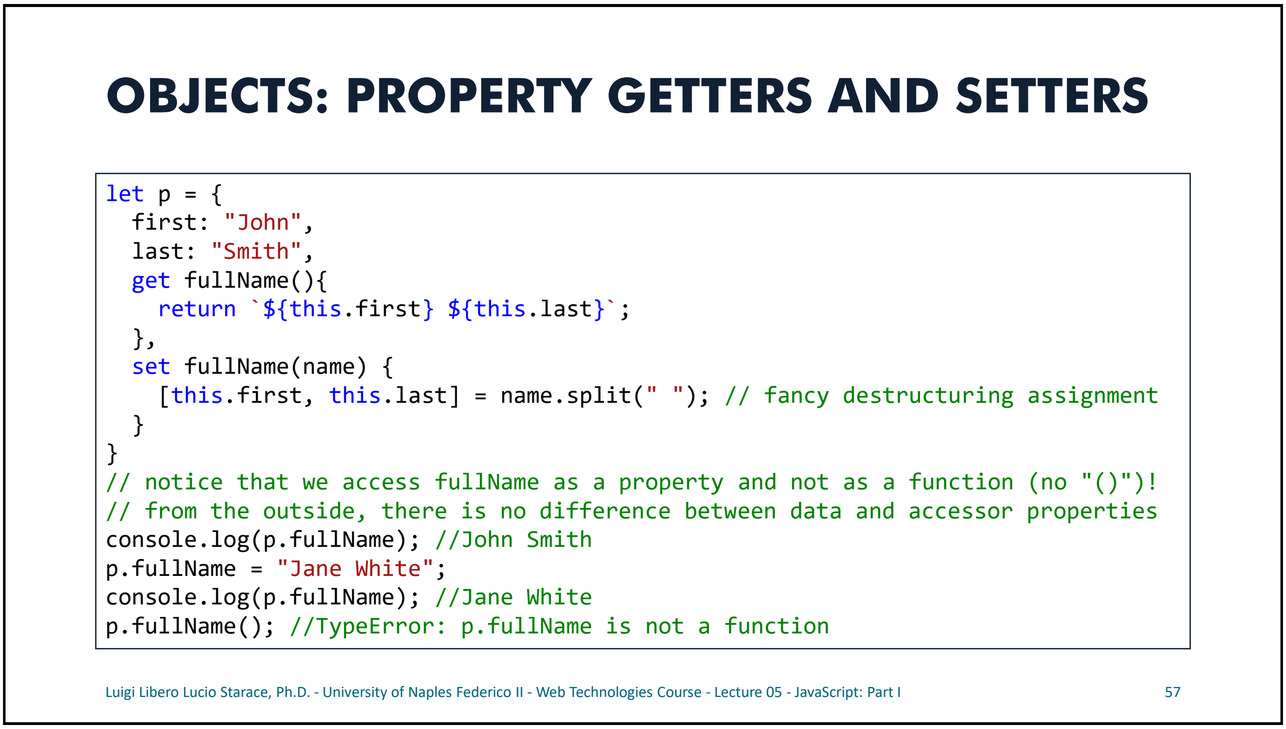
OBJECTS: PROPERTY GETTERS AND SETTERS

There are two kinds of object properties:

- **Data properties**: store a value (the ones we’ve seen so far)
- **Accessor properties**: functions that are executed when a property is accessed (in read mode or write mode)

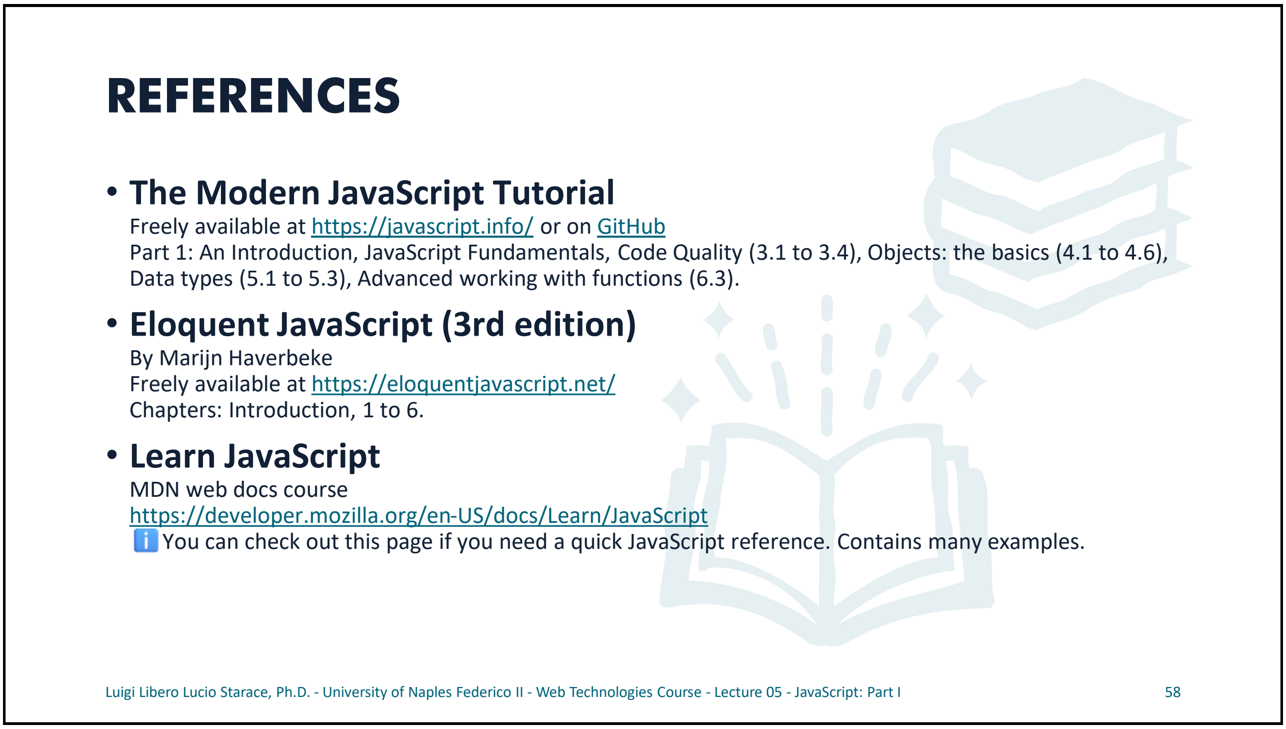
Accessor properties are represented by a **getter** (invoked when the property is read) and by a **setter** (invoked when it is assigned)

- The **get** and **set** keywords can be used to denote getters and setters in object literals



OBJECTS: PROPERTY GETTERS AND SETTERS

```
let p = {
  first: "John",
  last: "Smith",
  get fullName(){
    return `${this.first} ${this.last}`;
  },
  set fullName(name) {
    [this.first, this.last] = name.split(" "); // fancy destructuring assignment
  }
}
// notice that we access fullName as a property and not as a function (no "()")!
// from the outside, there is no difference between data and accessor properties
console.log(p.fullName); //John Smith
p.fullName = "Jane White";
console.log(p.fullName); //Jane White
p.fullName(); //TypeError: p.fullName is not a function
```



REFERENCES

- **The Modern JavaScript Tutorial**
  Freely available at https://javascript.info/ or on GitHub
  Part 1: An Introduction, JavaScript Fundamentals, Code Quality (3.1 to 3.4), Objects: the basics (4.1 to 4.6), Data types (5.1 to 5.3), Advanced working with functions (6.3).
- **Eloquent JavaScript (3rd edition)**
  By Marijn Haverbeke
  Freely available at https://eloquentjavascript.net/
  Chapters: Introduction, 1 to 6.
- **Learn JavaScript**
  MDN web docs course
  https://developer.mozilla.org/en-US/docs/Learn/JavaScript
  ℹ️ You can check out this page if you need a quick JavaScript reference. Contains many examples.