UNIVERSITÀ DEGLI STUDI DI NAPOLI FEDERICO II

WEB TECHNOLOGIES — LECTURE 06

JAVASCRIPT: PART II

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 learned the key concepts of the JavaScript language:

- **Variables**, **functions**, **scope** (and **hoisting**)
- **Objects**, **constructors**

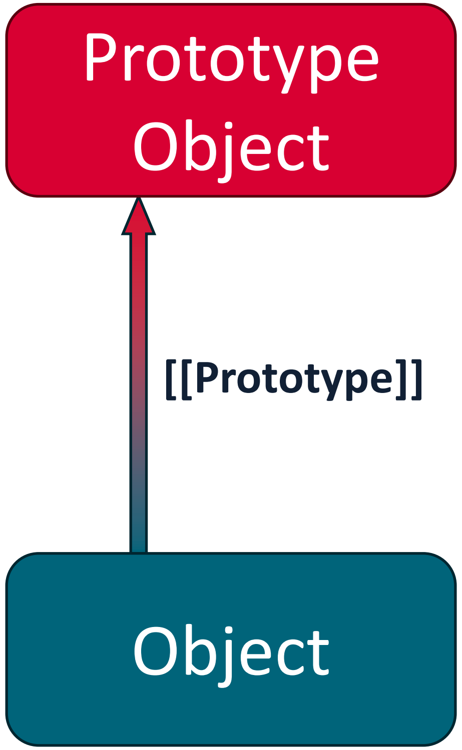
Today we'll continue dwelling into the JavaScript language and learn some other core concepts:

- **Prototypes and Inheritance**
- **Data Structures (Arrays, Maps, Sets)**
- **Classes**
- **Modules**

PROTOTYPES AND INHERITANCE

OBJECTS: PROTOTYPES AND INHERITANCE

- Inheritance is an essential aspect of object-oriented programming
- JavaScript features **Prototypal inheritance**
- Every object has a **special hidden property** called **[[Prototype]]**
- **[[Prototype]]** is either **null**, or references another objects
- When we access a property of an object, and it's missing, JavaScript searches for the property by traversing the prototype chain



OBJECTS: PROTOTYPES

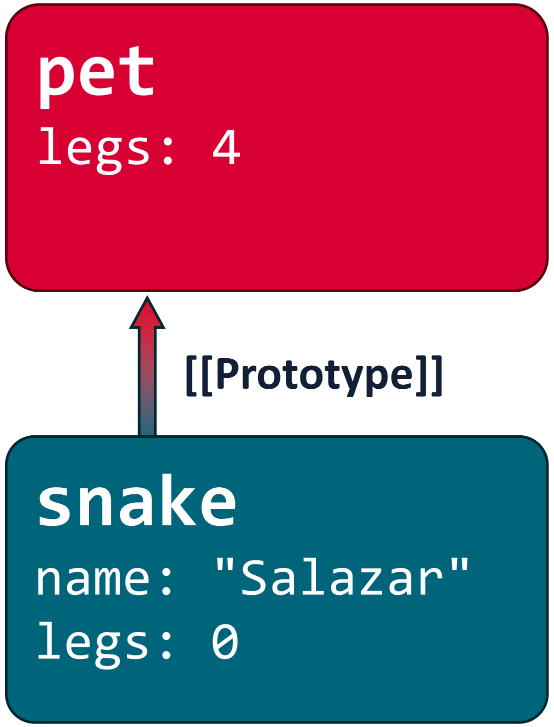
```
let pet = {
  legs: 4, greet(){ console.log("..."); }
}
let cat = {
  __proto__: pet //setting the prototype
}
cat.greet(); // "..."
console.log(cat.legs); //4
console.log(cat.__proto__); //Object { legs: 4, greet: greet() }
```

- **__proto__** is a getter/setter for the actual **[[Prototype]]** property
- In modern JavaScript, it is possible to use **Object.getPrototypeOf()** and **Object.setPrototypeOf()**

OBJECTS: WORKING WITH PROTOTYPES

- Prototypes are **only used when reading properties**
- **Write/delete operations work directly on the object**

```
let pet   = { legs: 4 }

let cat   = { name: "Garfield" }
let snake = { name: "Salazar" }

Object.setPrototypeOf(cat, pet);
Object.setPrototypeOf(snake, pet);

snake.legs = 0
console.log(`Cat legs: ${cat.legs}`);     // Cat legs: 4
console.log(`Snake legs: ${snake.legs}`); // Snake legs: 0
console.log(`Pet legs: ${pet.legs}`);     // Pet legs: 4
```



pet
legs: 4

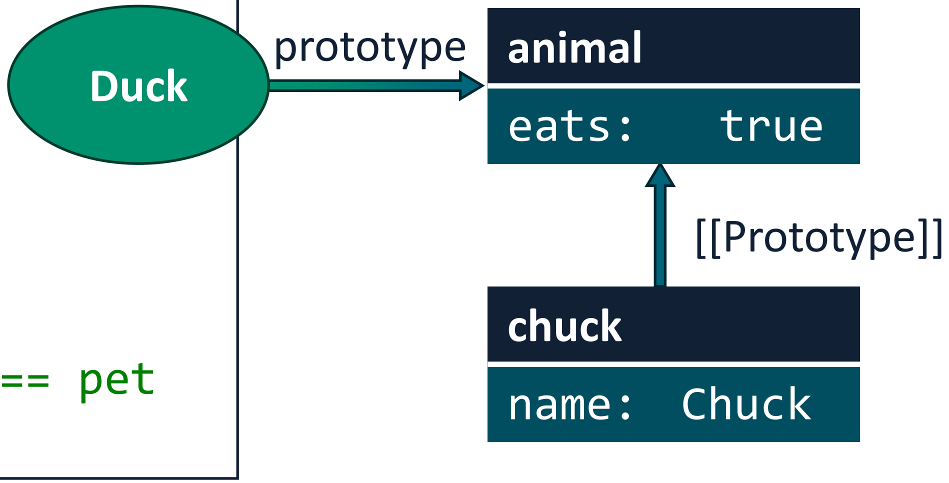
[[Prototype]]

snake
name: "Salazar"
legs: 0

CONSTRUCTORS AND PROTOTYPES

- We can create objects using Constructor functions, like «**`new Pet()`**»
- When **`Pet.prototype`** is an object, the **`new`** operator uses it to set the **`[[Prototype]]`** for the newly created object

```
let pet = {
  eats: true
};
function Duck(name) {
  this.name = name;
}
Duck.prototype = pet;
let chuck = new Duck("Chuck"); //chuck.__proto__ == pet
console.log(chuck.eats); //true
```



CONSTRUCTOR AND PROPERTIES

What happens when the constructor function has no **`prototype`** property?

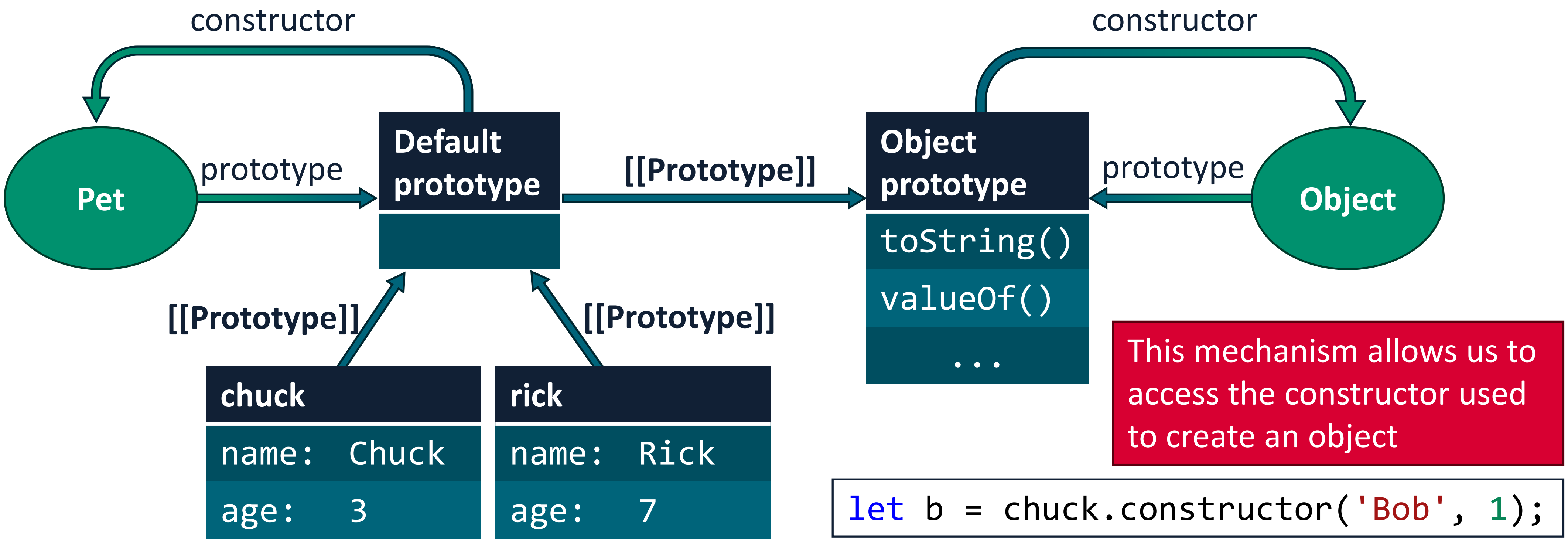
```
function Pet(name, age) {
  this.name = name; this.age = age;
}
let chuck = new Pet('Chuck', 3); let rick  = new Pet('Rick' , 7);
```

- In that case, a **default prototype** is used
- The default prototype is an object with a **`constructor`** property, pointing back to the constructor function

```
Pet.prototype = { constructor: Pet }
```

CONSTRUCTORS AND PROTOTYPES

```
function Pet(name, age) {
  this.name = name; this.age = age;
}
let chuck = new Pet('Chuck', 3); let rick  = new Pet('Rick' , 7);
```



This mechanism allows us to access the constructor used to create an object

```
let b = chuck.constructor('Bob', 1);
```

CONSTRUCTORS AND PROTOTYPES

- Two **alternative** ways to add methods to created objects:

```
function describe(){ console.log(`I'm ${this.name} and my age is ${this.age}`); }

function Pet(name, age) {
  this.name = name; this.age = age;
  this.describe = describe; // (1) adds a describe method to each created object
}

let chuck = new Pet('Chuck', 3); let rick  = new Pet('Rick', 7);

Pet.prototype.describe = describe; // (2) adds a describe method to the prototype
                                   // of each created object

chuck.describe(); rick.describe();
```

DATA STRUCTURES

ARRAYS

- Arrays allow to store **ordered** sequences of values
- Can be declared using the array literal syntax with square brackets **[]** or the **Array** constructor

```
//array literal syntax
let a = ["HTML", "CSS", "JS"];
console.log(a); //Array(3) [ "HTML", "CSS", "JS" ]

//constructor syntax
let b = new Array("HTML","CSS","JS");
console.log(b); //Array(3) [ "HTML", "CSS", "JS" ]
```

ARRAYS: INDEXING

- Arrays are indexed, starting at **0**.
- Specific values can be accessed in read/write mode using square brackets notation
- The **length** property contains the **maximum index, plus one**

```
let a = ["HTML", "CSS", "JS"];
a[2] = "JavaScript";
a[3] = "React";

console.log(a[0]);     // HTML
console.log(a[1]);     // CSS
console.log(a[2]);     // JavaScript
console.log(a[3]);     // React
console.log(a.length); // 4
```

ARRAYS: LENGTH

- You probably noticed we defined **array.length** in a strange way…
- That’s because the length **is not** actually the count of values in the array!

```
let a = [1, 2, 3, 4, 5];

a[999] = 998;
console.log(a)
console.log(a.length); // 1000 (!)
console.log(a[5]);     // undefined
console.log(a[999]);   // 998
console.log(a[2000]);  // undefined
```

```
// an interesting thing about the
// length is that it is writeable

a.length = 3;
console.log(a); // [1,2,3]
a.length = 5;
console.log(a); // [1,2,3,<2 empty>]
a.length = 0; // clears the array
```

ARRAYS: MIXED TYPES

- An array can contain **heterogeneous** data types

```
let a = [
  "JavaScript",
  {name: "John", job: "Dev"},
  12,
  function(name){
    console.log(`Hello ${name}`);
  }
]

console.log(a[1].name);   // John
a[3]("Web Technologies"); // Hello Web Technologies
```

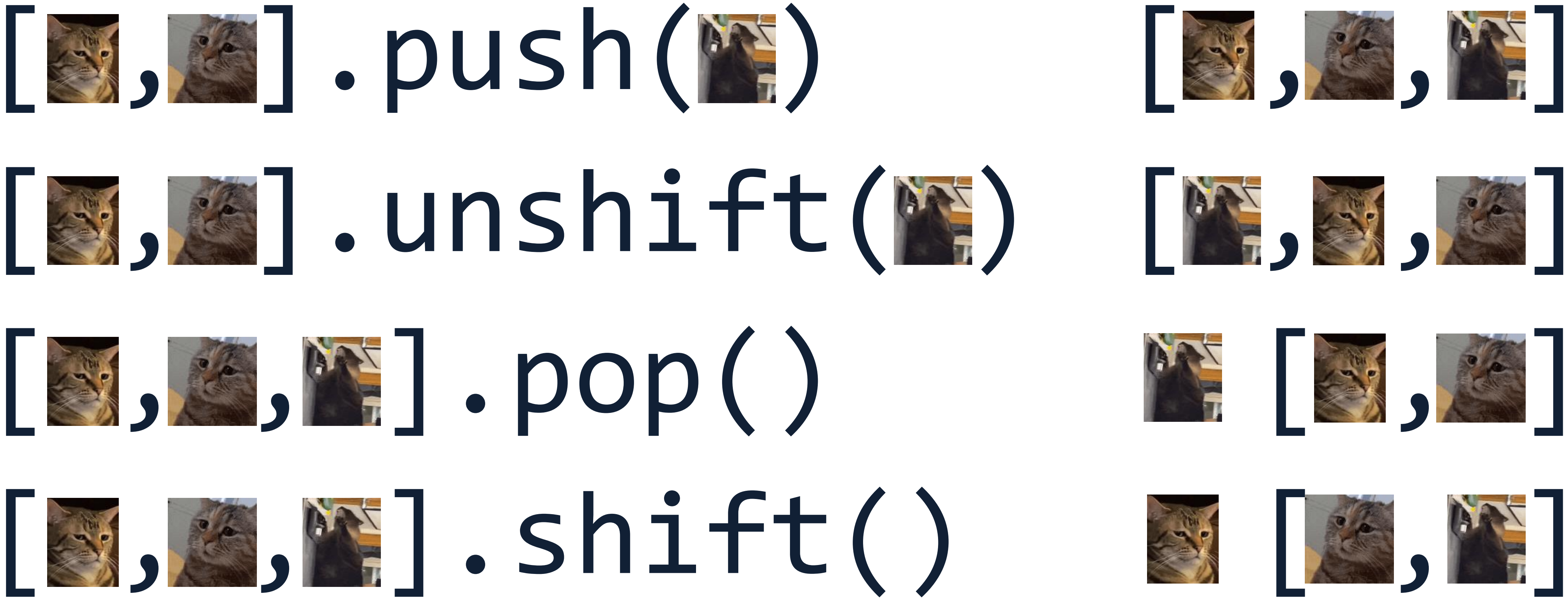
ARRAYS: METHODS

Arrays provide dedicated methods to add/remove items

- **push():** add an item to the end
- **shift():** take an item from the beginning
- **pop():** take an item from the end
- **unshift():** add an item at the beginning

```
let a = [1];       // [1]

a.push(2);         // [1, 2]
a.unshift(0);      // [0, 1, 2]
x = a.pop();       // [0, 1]
y = a.unshift();   // [1]

console.log(x);    // 2
console.log(y);    // 0
```

ARRAY METHODS: VISUALIZED



```
[ , ].push( )         [ , , ]
[ , ].unshift( )      [ , , ]
[ , , ].pop()           [ , ]
[ , , ].shift()         [ , ]
```

For a visualization of all array methods, you can check out: https://js-arrays-visualized.com/

ARRAYS: ITERATION

```
let a = ["a", "b", "c"];
a[4] = "e";

for(let i = 0; i < a.length; i++)
  console.log(a[i]); //a,b,c,undefined,e

for(let item of a)
  console.log(item); //a,b,c,undefined,e

a.forEach( (value, index, array) => {
  console.log(`a[${index}]=${value}`);
});

for(let key in a){
  console.log(a[key]); //a,b,c,e
}
```

- Array can be iterated over using **for** loops over indexes, by using the alternative **for..of** syntax, or the **forEach** method.
- Since Arrays are objects, it is also possible to use **for..in**, but that’s not a good idea
  - It’s 10-100 times **slower**
  - There might be other **enumerable** properties outside the array values…

MULTIDIMENSIONAL ARRAYS

- Array items can also be other arrays
- We can use this to define multidimensional arrays (e.g.: matrices)

```
let matrix = [
  [1,2,3],
  [4,5,6],
  [7,8,9],
]

console.log(matrix[0][0]); //1
console.log(matrix[1][1]); //5
console.log(matrix[2][2]); //9
```

DESTRUCTURING ASSIGNMENTS

A special syntax that allows to **unpack arrays** into variables

```
let [x, y] = ["a","b","c"]; // remaining array elements are discarded
console.log(x); //a
console.log(y); //b

let [a, b, ...rest] = [1,2,3,4,5]; // remaining array elements are stored in rest
console.log(a);    //1
console.log(b);    //2
console.log(rest); //[3, 4, 5]

//similar syntax can also be used in function parameters
function greet(msg, ...names){
  console.log(`${msg} to ${names}`);
}
greet("Hello", "Ann", "Bob", "Carl"); //Hello to Ann,Bob,Carl
```

ITERABLES

- The **for..of** loop can be used with **iterable** objects
- Arrays are iterable objects (so are strings)

```
let string = "Web Technologies!";
for(let char of string){
  console.log(char); //W, e, b, , T, e, ...
}
```

- What if we have a custom object that represents a collection of values and we want to use it in a **for..of** loop construct?

ITERABLES

```
//range represents a set of integers:
//starting at min, and arriving up to max, with steps of the specified size
let range = {
  min: 1,
  max: 10,
  step: 2
}

for(let value of range){ //TypeError: range is not iterable
  console.log(value);
}
```

- To make an object iterable, it is necessary to implement a special **Symbol.iterator** method

ITERABLES: SYMBOL.ITERATOR

- When a **`for..of`** loop starts, it calls the **`Symbol.iterator`** method on the object once (and throws a TypeError if the method does not exist)
- Onward, the **`for..of`** loop only works with the object (Iterator) returned by the `Symbol.iterator` call
- When the **`for..of`** loop needs to access the next value, it calls the **`next()`** method on the Iterator
- The result of the invocation of **`next()`** is an object of the form **`{done: boolean, value: any}`** where **`done=true`** means that the loop is finished, otherwise **value** is the next value.

IMPLEMENTING SYMBOL.ITERATOR

```
let range = { min: 1, max: 10, step: 3 }

range[Symbol.iterator] = function(){
  return {
    current: this.min, max: this.max, step: this.step,
    next(){
      let n = {done: false, value: this.current};
      if(n.value > this.max)
        n.done = true;
      this.current += this.step;
      return n;
    }
  }
}

for(let value of range)
  console.log(value); //1,4,7,10
```

WORKING WITH ITERATORS EXPLICITLY

- It is also possible to work with iterators directly, as shown below
- The below while loop replicates the **for..of** loop of the previous slide

```
let iterator = range[Symbol.iterator]();
while(true){
  let result = iterator.next();
  if(result.done)
    break;
  console.log(result.value);
}
```

MAPS

- Maps are collection of **key-value pairs.**
- So, they’re just like objects? Not really, Maps **allow keys of any type**!
- Methods and properties are:

new Map()	Creates the map
map.set(key, value)	Stores the value by the key
map.get(key)	Returns the value by the key, or undefined if not present
map.has(key)	Returns true if the key exists, false otherwise
map.delete(key)	Removes the key-value pair by the key
map.clear()	Removes everything from the map
map.size	Is the current element count

MAPS VS OBJECTS

```
let map = new Map();
let o = {};

map.set(1, "Num");          map.set("1", "Str");
map.set(true, "Bool");      map.set("true", "Str");

console.log(map); //Map { 1 → "Num", "1" → "Str", true → "Bool", "true" → "Str" }
console.log(map.size); // 4

o[1]      = "Num";  o["1"]    = "String";
o[true]   = "Bool"; o["true"] = "String";
console.log(o); //Object { 1: "String", true: "String" }
```

MAPS: ITERATIONS

```javascript
let map = new Map();

map.set("Hello", "JS"); map.set(1, true); map.set(false, 42);

for(let key of map.keys()){ //map.keys() returns an iterable over keys
  console.log(key); //Hello, 1, false
}
for(let value of map.values()){ //map.values() returns an iterable over values
  console.log(value); //JS, true, 42
}
for(let [key, value] of map.entries()){ //iterable over entries
  console.log(`${key}: ${value}`);
}
for(let [key, value] of map){ //equivalent to the one before
  console.log(`${key}: ${value}`);
}
```

SETS

- Sets are a data structure to store collections of values without repetitions
- Main methods of a set are:

new Set([iterable])	Creates the Set. If an iterable is provided, copies its values
set.add(value)	Stores the value, returns the set itself
set.delete(value)	Deletes value from Set. Returns true if value existed, false otherwise
set.has(value)	Returns true if value exists in the set, false otherwise
set.clear()	Removes everything from the set
set.size	Is the current element count

SETS: EXAMPLES

```
let set = new Set();

let eric = { name: "Eric" };
let stan = { name: "Stan" };
let kyle = { name: "Kyle" };

set.add(eric); set.add(stan);
set.add(kyle); set.add(eric);
set.add(kyle);

// set keeps only unique values
console.log( set.size ); // 3

for (let user of set) {
  console.log(user.name); //Eric, Stan, Kyle
}
```

CLASSES

CLASSES IN JAVASCRIPT

We are familiar with the concept of classes in object-oriented software

- Classes are basically templates for creating objects
- **Constructors** and **new** can help with that in JavaScript
- JavaScript also features a more-advanced **class** construct, which provides some benefits

```
class Pet {
  constructor(name){ this.name = name }
  method1(){/*...*/}
  method2(){/*...*/}
  /* more methods */
}

let pet = new Pet("Chuck");
```

CLASSES IN JAVASCRIPT

```
class Pet {
  constructor(name){ this.name = name }
  method1(){/*...*/}
  method2(){/*...*/}
  /* more methods */
}

let pet = new Pet("Chuck");
```

- **new Pet()** creates a new object and invokes the constructor method with the given arguments, which sets the name property on the object. The new object is then returned.

CLASSES IN JAVASCRIPT

- What exactly is a class? It’s not a new language-level entity
- `typeof(Pet); //function`
- In JavaScript, classes are **special kinds of functions**
- What does the `class Pet {...}` construct really do then?
    1. Creates a constructor function named Pet. The code of the function is taken from the `constructor()` method (or is empty if there is no such method).
    2. Stores other class methods in `Pet.prototype`

CLASSES IN JAVASCRIPT

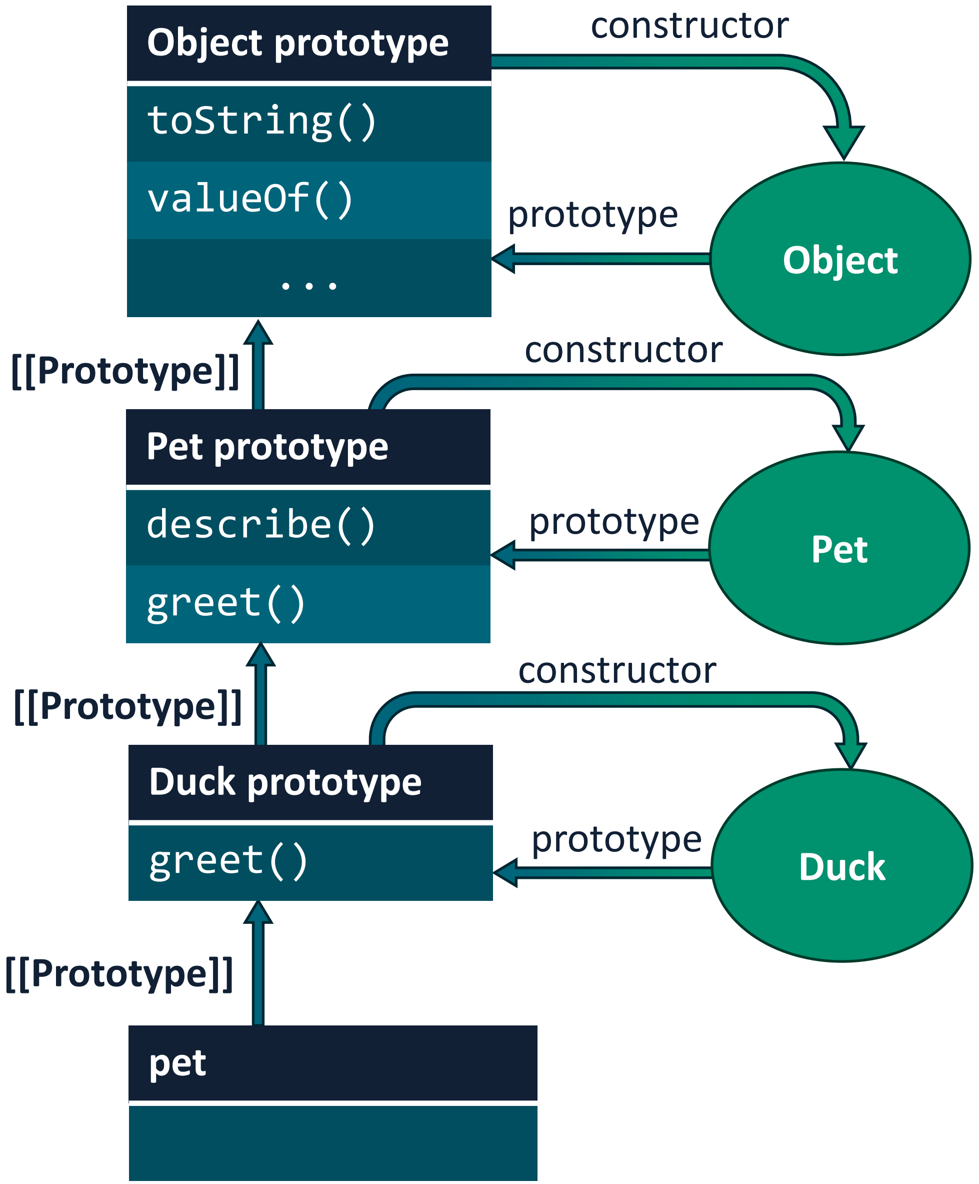
- Similarly to objects, classes support properties and getter/setters

```
class Duck {
  species = "Duck"; //property
  constructor(name){this.name = name}
  greet(){
    console.log("Quack!")
  }
  get fullDescription(){return `${this.name} the ${this.species}`};

  [Symbol.iterator]() {/*...*/}
}

let duck = new Duck("Chuck");
duck.greet(); //Quack!
console.log(duck.fullDescription); //Chuck the Duck
```

CLASS INHERITANCE

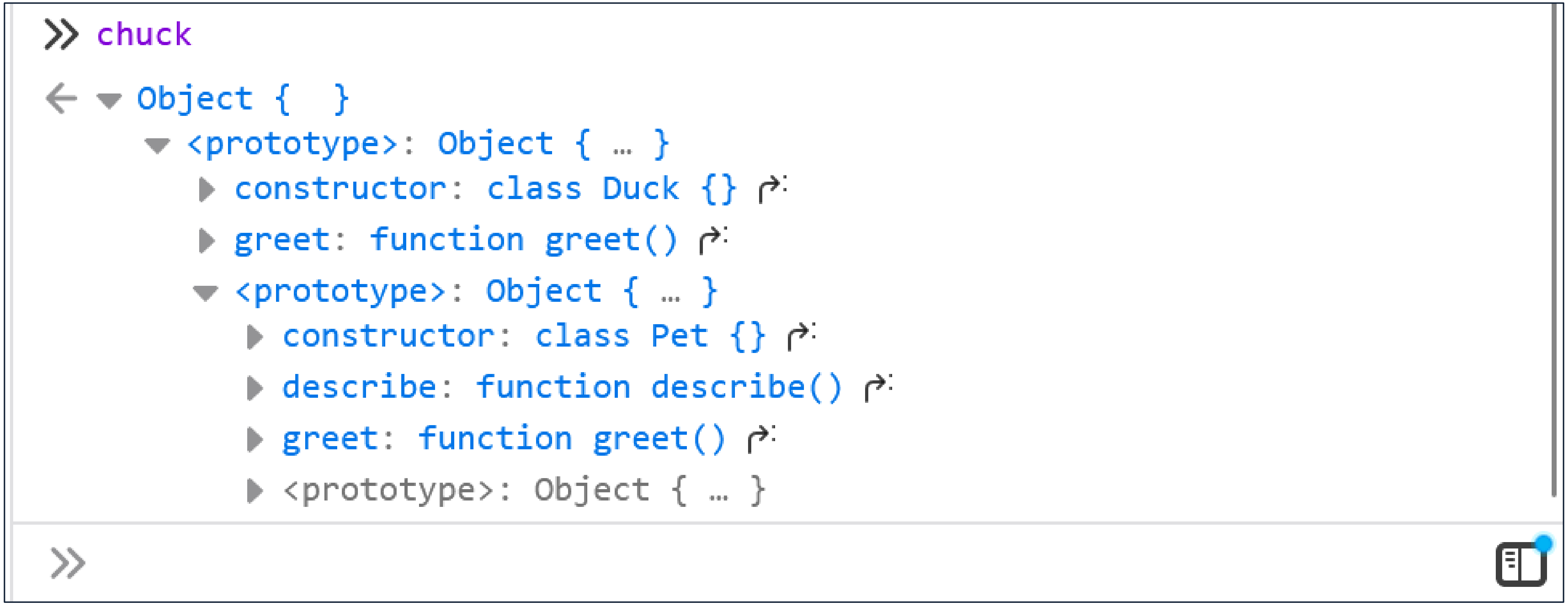
- JavaScript classes can inherit from other classes using the **extends** keyword

```
class Pet {
  describe(){ console.log("I'm a pet"); }
  greet(){ console.log("..."); }
}

class Duck extends Pet {
  greet(){ console.log("Quack!"); }
}

let chuck = new Duck();
chuck.describe(); //I'm a pet
chuck.greet();    //Quack!
```



CLASS INHERITANCE

- Internally, extends is implemented using the **[[Prototype]]** property



```
>> chuck
← ▼ Object {  }
     ▼ <prototype>: Object { … }
       ▶ constructor: class Duck {}
       ▶ greet: function greet()
       ▼ <prototype>: Object { … }
         ▶ constructor: class Pet {}
         ▶ describe: function describe()
         ▶ greet: function greet()
         ▶ <prototype>: Object { … }
>>
```

ERROR HANDLING

ERRORS

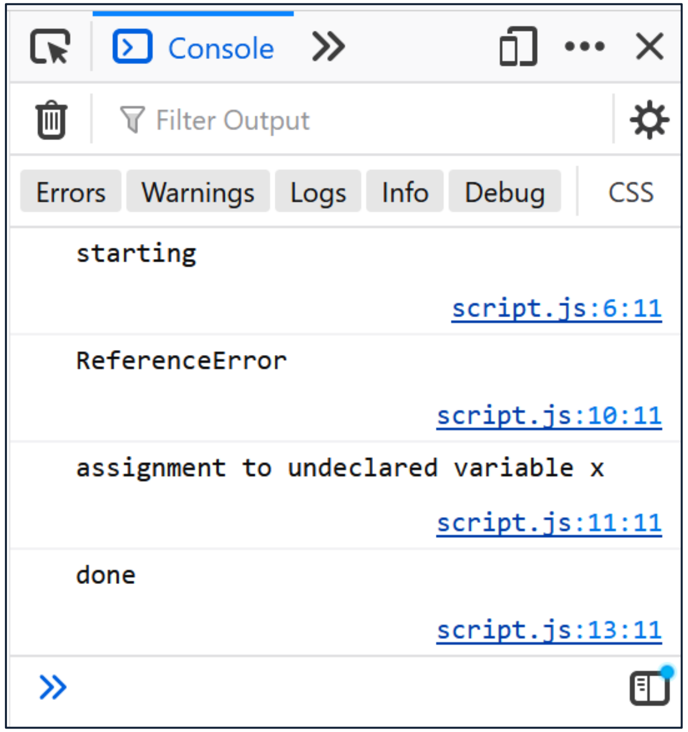
- When running a script, errors can happen
- They can happen because programmers make mistakes, because users provided unexpected inputs, and so on…
- When errors happen, scripts typically stop immediately, printing an error message in the console

```
let myvar = 0;

for(let num of [1,2,3,4,5]){
  mvvar += num; // ReferenceError: mvvar is not defined
}

console.log(myvar);
```

HANDLING ERRORS: TRY/CATCH/FINALLY

- A **try/catch/finally** syntax construct provides ways to handle errors.
- Conceptually, it’s the same construct you know from Java
  - With a few minor differences (we’ll see them!)…

```
try {
  console.log("starting");
  x = 1; //forgot the let keyword
  console.log("assignment done"); // not executed
} catch (error) {
  console.log(error.name);
  console.log(error.message);
} finally {
  console.log("done");
}
```



```
starting
                                   script.js:6:11
ReferenceError
                                   script.js:10:11
assignment to undeclared variable x
                                   script.js:11:11
done
                                   script.js:13:11
```

HANDLING ERRORS: TRY/CATCH/FINALLY

Peculiarities of try/catch constructs in JavaScript:

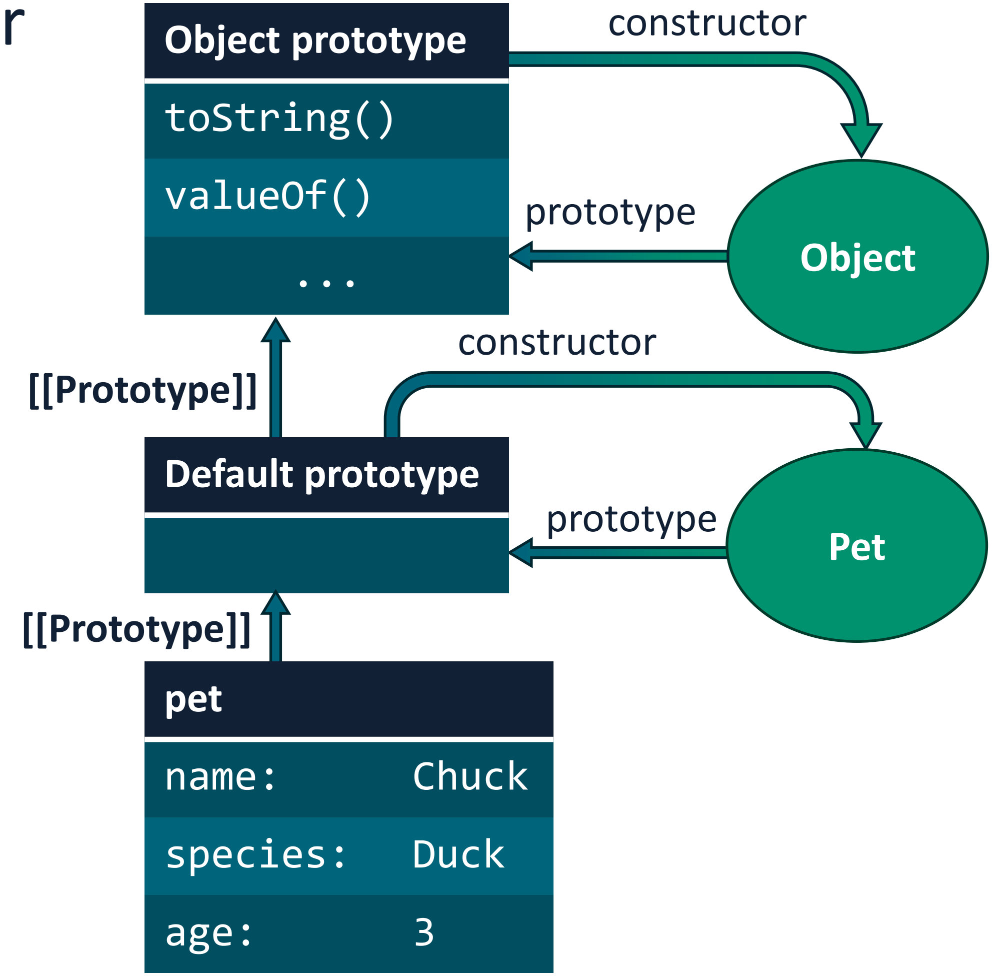
- There must be **at most one** catch block (try/finally is admissible)
- Different types of errors are handled inside the catch block

```
try {
  x = 1; //forgot the let keyword
} catch (error) {
  if(error instanceof ReferenceError){
    console.log("Got a ReferenceError");
  } else {
    console.log(`Got something else: ${error.name}`);
  }
}
```

THE INSTANCEOF OPERATOR

- The `instanceof` operator tests whether the prototype property of a given constructor appears anywhere in the prototype chain of an object

```
function Pet(name, species, age) {
  this.name = name;
  this.species = species;
  this.age = age;
}
const pet = new Pet('Chuck', 'Duck', 3);

console.log(pet instanceof Pet); //true
console.log(pet instanceof Object); //true
```



THROWING ERRORS

- Errors in JavaScript propagate in the same way as in Java
  - upwards until cought or until the top of the call tree is reached (in which case script evaluation is abruptly interrupted and the error is logged to console)
- Errors can be thrown using the `throw` keyword

```
try {
  throw new Error("Oops!");
} catch (err) {
  console.log(err.name);  // Error
  console.log(err.message); // Oops!
}

class MyError extends Error {
  constructor(message = "Whoops") {
    super(message);
    this.name = "MyError";
  }
}

try {
  throw new MyError();
} catch (err) {
  console.log(err.name);  // MyError
  console.log(err.message); // Whoops
}
```

MODULES

MODULARITY

- Modern JavaScript allows language-level **module** definition
- Modules are ways to split complex programs in multiple files (modules), with each module containing classes or functions for a specific purpose
  - Improves **maintainability**, promotes **separation of concerns**
  - Can help disambiguate and prevent naming conflicts (improves **reusability**)

MODULES

- A module is just a JavaScript file
- Modules can load each other and use special directives `export` and `import` to **interchange** functionality

- `export` labels variables and functions that should be accessible **outside** of the current module
- `import` allows the import of specific functionality (e.g.: variables, functions) from other modules (provided that these functionality are exported!)

MODULES: EXAMPLE

- Suppose you want to re-use some code you developed for some other projects: a **pet.js** file and a **greet.js** file

```
//pet.js
function Pet(name, age){
  this.name=name; this.age=age;
}
let msg = "Hello";
function greet(pet){
  console.log(`${msg}, I'm ${pet.name}`);
}
```

```
//greet.js
let msg = "Howdy";

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

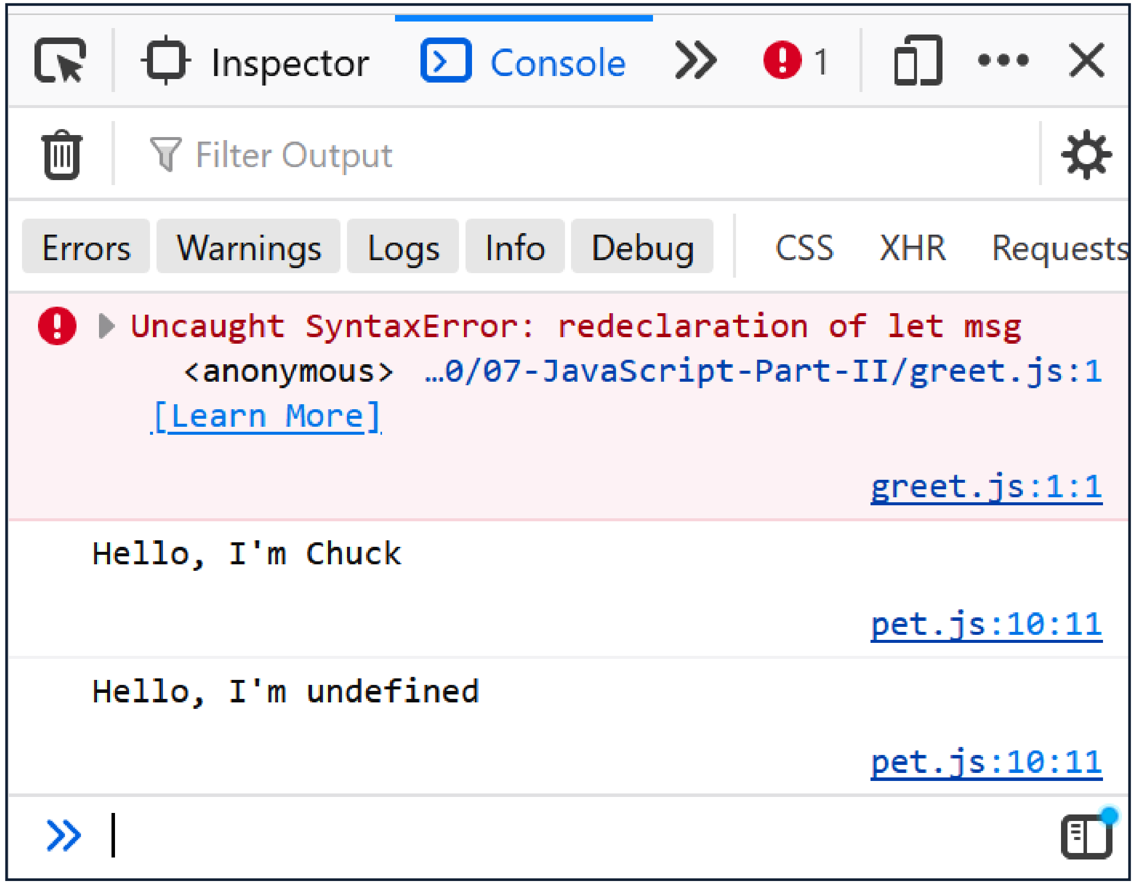
```
//script.js (The new code we need to implement)
let chuck = new Pet("Chuck", 7); // Pet is defined in pet.js
greet(chuck); // Desiderata: Hello, I'm Chuck
greet("Web Technologies"); // Desiderata: Howdy, Web Technologies
```

MODULES: EXAMPLE

- One might try to include all the scripts in the web page

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

```
//script.js (The new code)
let chuck = new Pet("Chuck", 7);
// Pet is defined in pet.js
greet(chuck);
// Desiderata: Hello, I'm Chuck
greet("Web Technologies");
// Desiderata: Howdy, Web Technology
```



```
Uncaught SyntaxError: redeclaration of let msg
    <anonymous>  …0/07-JavaScript-Part-II/greet.js:1
    [Learn More]
                                                  greet.js:1:1
Hello, I'm Chuck
                                                  pet.js:10:11
Hello, I'm undefined
                                                  pet.js:10:11
```

MODULES: EXAMPLE

- To make this work, we would need to change the code we want to re-use, for example using different identifiers for variables and functions
  - That’s **not ideal**, and defeats the main purpose of re-using existing code as is
- Modules come to the rescue!

```
//pet-module.js
export function Pet(name, age){
  this.name=name; this.age=age;
}

let msg = "Hello"; //no need to export

export function greet(pet){
  console.log(`${msg}, I'm ${pet.name}`);
}
```

```
//greet-module.js
let msg = "Howdy"; //no need to export

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

MODULES: EXAMPLE

- In the HTML document, we just need to include the main script, **as a module**:

```
<script type="module" src="./script-module.js"></script>
```

```
//script-module.js, other modules are imported given their URLs
import {Pet, greet as greetPet} from './pet-module.js';
import {greet} from './greet-module.js';

let chuck = new Pet("Chuck", 7);

greetPet(chuck); //Hello, I'm Chuck

greet("Web Technologies"); //Howdy, Web Technologies
```

REFERENCES

- **The Modern JavaScript Tutorial**
  Freely available at https://javascript.info/ or on GitHub
  Part 1: Prototypes and inheritance, Data types (5.4 to 5.7), Classes (9.1 to 9.4, 9.6), Generators and advanced iteration (12.1), Modules (13.1, 13.2)
- **Eloquent JavaScript (3rd edition)**
  By Marijn Haverbeke
  Freely available at https://eloquentjavascript.net/
  Chapters 6, 10
- **JavaScript Reference**
  MDN web docs
  https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference