Introduction to JavaScript and TypeScript

Tom Södahl Bladsjö

tom.sodahl.bladsjo@svenska.gu.se

January 22, 2026

Why JS?

Why JavaScript?

JavaScript

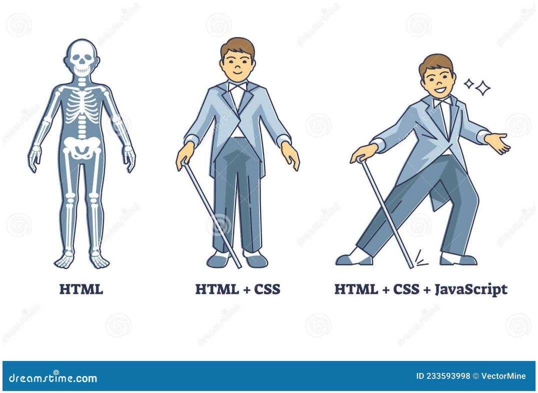
- is used for web development
- runs in browsers
- makes websites interactive

Why JavaScript in *this course*?

- It lets us build programs that can be run directly in the browser

$\rightarrow$ people can use our dialogue systems online via a web page (they don't have to download and run the code themselves)

JS and HTML



Basically, JavaScript can be embedded in HTML to add interactivity to webpages

JS and HTML

We can insert JS code into our HTML:

To write code directly into the document:

```
<script>
  console.log("hello world!");
</script>
```

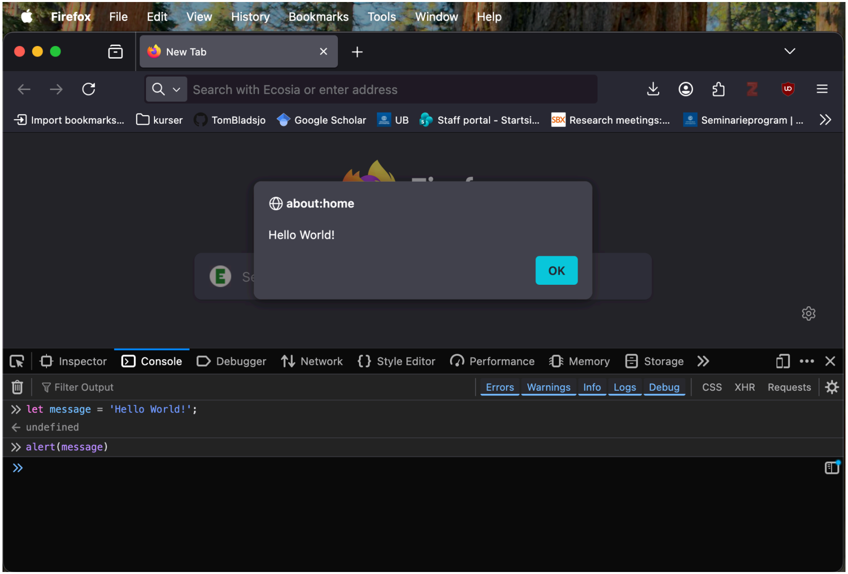
To link an external script file:

```
<script src="path/to/my/script.js"></script>
```

We can also interact with the HTML from our JavaScript code:

```
const element = document.getElementById("intro");
element.innerHTML = "hello world!";
```

What does it mean that JavaScript runs in the browser?



Hello World!

```
>> let message = 'Hello World!';
<- undefined
>> alert(message)
>>
```

Try it yourself!

Open a browser window.

If you're on a Mac:

- Try pressing ⌘ + ⌥ + I, or ⌘ + ⌥ + J

If you're on a Windows:

- Try ctrl + shift + I, or ctrl + shift + J

If you don't see the console:

- Look for something like **Tools** ⟩ **Browser Tools** ⟩ **Web Developer Tools** in the browser menu

JS vs Python - general

JS is similar to Python in many ways:

- both are object oriented programming languages
- they have similar underlying structures – variables, functions, loops, if-statements etc

But:

- they look different: some things that are implicit/inferred in Python are explicit in JS
- some things are actually different (we will get to that)

Basic structure

In Python, statements and blocks of code are separated by whitespace (linebreaks and indentation)

Python

```python
if condition:
    function1()
function2()
```

Basic structure

In JS, statements are followed by semicolon ; and blocks are declared with curly brackets { }

JavaScript

```javascript
if (condition) {
  function1();
}
function2();

// this also works:
if (condition) {function1();} function2();
```

Comments

- // for single line comments
- /* and */ for multi-line comments

Example

```
// this is a single line comment

/* this is
a multi-line
comment
*/
```

Types

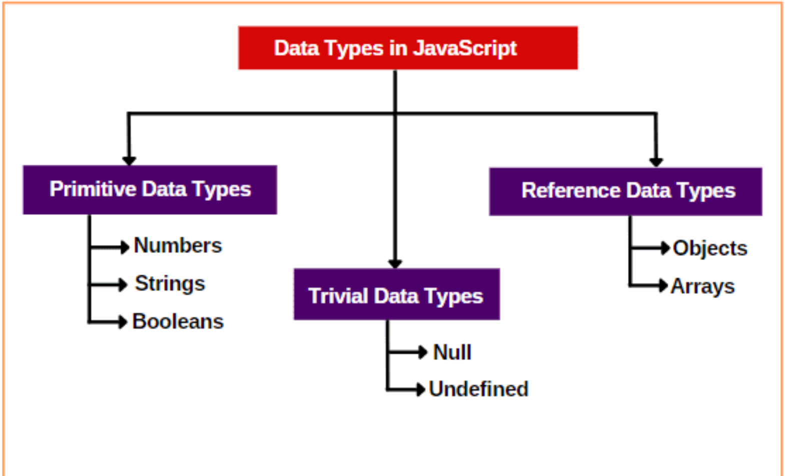


Fig: Types of Data types in JavaScript

Variable declaration

let

```
let x = 1
console.log(x)
// console.log works sort of like print in Python
x = 2
console.log(x) // x is now 2
let x = 3 // SyntaxError: redeclaration of let x
```

Variable declaration

const

```
// constants can only be assigned once:
const name = 'Tom'
name = 'Tim'
// TypeError: invalid assignment to const 'name'
```

Use constants for things that will not be reassigned during the running of the script

Variable declaration

There's also `var`, which

- is rarely used anymore
- is similar to `let`, but
- has some unexpected behaviors

Bottom line:

You can use `var` if you want, but you will be fine just using `let` and `const`

Strings

`""` and `''` work just like in Pyhton:

```
"this is a string"
'this is also a string'
"it's a string"
'a "string" is a string'
```

` `` ` works like f-strings in Python:

```
let user = 'Jane'

console.log(`hello ${user}`) // "hello Jane"
```

Conditionals

- A condition is something that evaluates to `true` or `false`
- Based on the result of the evaluation, execute different blocks of code

if statements

```
let fruit = 'apple';

if (fruit == 'apple') {
console.log("it's an apple!");
} else {
console.log("it's not an apple");
}
```

Conditionals

JavaScript does not have `elif` like Python.
Instead, we have to use `else` and `if`:

else if

```
if (fruit == 'apple') {
console.log("it's an apple!");
} else if (fruit == 'pear') {
console.log("it's a pear");
} else if (fruit == 'banana') {
console.log("it's a banana!");
} else {
console.log("I don't know this fruit");
}
```

Loops

Pyhton

```python
for i in range(10):
    print(i)
```

JavaScript

```javascript
// syntax: (start clause; stop clause; step clause)
for (let i = 0; i < 10; i += 1) {
    console.log(i);
}

// you can also try this:
for (let i = 100; i >= 0; i -= 5) {
    console.log(i);
}
```

Looping over a list

Example

```
// you can iterate over arrays using (let ... of [array]):
let l = ['a', 'b', 'a', 'b', 'c']
for (let item of l) {
    console.log(item);
}
// but watch out!
for (let item in l) {
    console.log(item);
}  // output: 0 1 2 3 4

//  using "in" like in Python gives you indices, not the
    items themselves. to get the items themselves using "
    in":
for (let i in l) {
    console.log(l[i]);
}  // output: a b a b c
```

Indefinite loops

while loop

```
// you can also do while loops like in python:
let n = 1
while (n <= 5) {
    console.log(n);
    n ++
}
```

Functions

Example

```javascript
// a simple function:
function greet() {
    alert('Hello world!');
}
// to call the function:
greet();


// just like in Python, functions can have optional arguments with default values:
function add(a, b=2) {
  return a + b;
}
```

Functions

There is also a format for writing functions with arrows:

Example

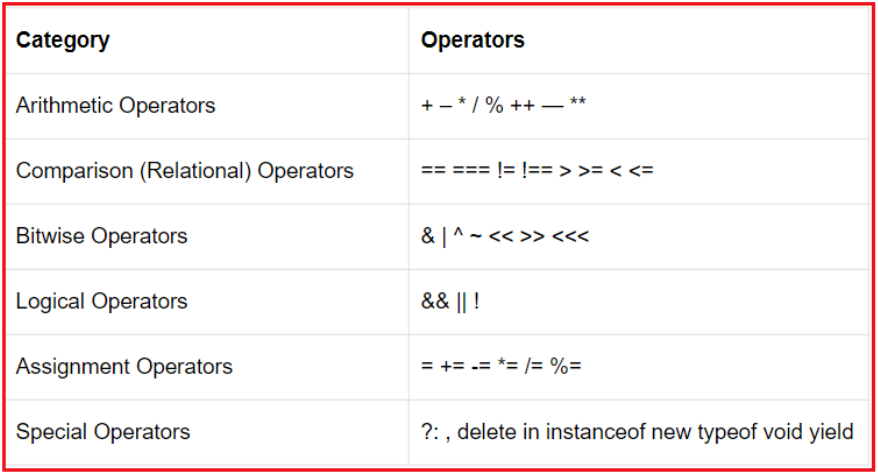
```
let hello = () => {
  console.log("hello");
};

hello(); // prints hello


const plusOne = (x) => {
    return x + 1;
};

plusOne(5); // returns 6
```

Operators



Category	Operators
Arithmetic Operators	+ – * / % ++ — **
Comparison (Relational) Operators	== === != !== > >= < <=
Bitwise Operators	& | ^ ~ << >> <<<
Logical Operators	&& || !
Assignment Operators	= += -= *= /= %=
Special Operators	?: , delete in instanceof new typeof void yield

Operators

Watch out!

Comparison operators and ”truthy”/”falsy” values

```
1 == '1'; // true
1 === '1'; // false

1 == true; // true
2 == true; // false
if (2) {console.log('true')}; // 2 evaluates "truthy" --
    prints "true"

0 == ''; // true
'' == false; // true

undefined == 0; // false
null == 0; // false
undefined == null; // true
```

Operators

Watch out: + is both an arithmetic operator and a string operator:

- if you try to add a number to a string, JS converts everything to a string
- if you try to concatenate lists/arrays using +, things get really weird

String operators and type coercion

```
1 + 1 // 2
1 + '1' // "11"

[1, 2, 3] + [4, 5, 6] // "1,2,34,5,6"
```

Collections

Arrays in JS act a lot like lists in Python.

Arrays

```
let l = [1, 5, 4, 7, 4];

// you can append to an array using push():
l.push(5);  // adds the 5 to the end of the list

// you can index into an array:
l[3]; // 7

// to slice an array, use slice():
l.slice(2,4); // equivalent to Python l[2:4]
l.slice(2);  // equivalent to Python l[2:]
l.slice(0,4); // equivalent to Python l[:4]
l.slice(-5, -1); // equivalent to Python l[-5:-1]
```

Collections

Objects in JS are very flexible, and similar to both dictionaries and class instances in Python

Objects

```
let food1 = {}; // empty object
let food2 = {"pizza": "margherita"};
// property quotes optional:
let food3 = {pizza: ["margherita", "funghi"]};

// common multiline format:
let prices= {
  "pizza": 150,
  "pasta": 120,
  "drink": "free"
};
```

Collections

Objects

```
// to access the properties (both formats work):
console.log(prices.pizza);      // prints 150
console.log(prices["pizza"]);   // prints 150

// you can add properties to the object, or change values
    of existing ones:
prices['coffee'] = 20
prices.pizza = 170

// objects can be nested:
let menu = {food: {pizza: ["margherita", "funghi"], pasta:
    ["carbonara"]}};
console.log(menu.food.pasta) // prints  ["carbonara"]
```

Collections

Watch out!

```
// if you try to access a property that doesn't exist, JS
    will not throw an error:

console.log(menu.food.cake); // prints "undefined"
```

Problems with JavaScript

- It fails silently: if you try to access a property that doesn't exist, it doesn't tell you (just returns `undefined`)
- It uses type coercion: silently converts data to same type before performing operations (e.g. `1 + '1' == '11'`)
- "truthy" and "falsy" values: things evaluate to `true` and `false` in ways that can give you unexpected results

TypeScript

A way to safeguard against some of the weird behaviors of JavaScript

JavaScript...

- is dynamically typed
- is prone to fail silently

TypeScript...

- is a language based on JavaScript
- does static typechecking
- throws errors where JavaScript would fail silently
- compiles to normal JavaScript (so it can still run in browsers)

TypeScript

Note:

All functioning JavaScript code is also TypeScript code!
The point of TypeScript is to add the missing typechecking/debugging functionality to JavaScript, not to be a separate language.

TypeScript

Type annotation

We can tell TypeScript which type of parameter a function wants (in this case, a number):

```
function addOne(x: number) {
  return x + 1;
}
let n = [0, 1, 2];

alert(addOne(n));
// error: Argument of type 'number[]' is not assignable to
    parameter of type 'number'.
```

This helps us avoid unexpected results! (Plain JavaScript would have silently given us "0,1,21")

TypeScript

Type annotation

You can also be very explicit and state what type of output you are expecting:

```
const double = (word: string): string => {
    return word + word;
};

alert(double('hello')) // 'hellohello'
```

Type annotation works for variable assignment as well:

```
let name: string = 'Bob';
// this way, TS will complain if we try to reassign 'name'
     as something other than a string
```

TypeScript

Types

```
// primitives:
string  // e.g. 'hello'
number // e.g. 5.4
boolean // e.g. false

// arrays:
string[] // ['apple', 'pear', 'banana']
number[] // [1,2,3]

// nested arrays:
number[][] // [[1,2,],[3,4],[5,6]]

// special type: any
any  // literally any type. will not cause type errors.

// literals: you can give an actual value as the type
```

TypeScript

Object types

To define an object type, simply list its properties and their types:

```
function printCoordinates(point: { x: number; y: number })
     {
  console.log("The coordinate's x value is " + point.x);
  console.log("The coordinate's y value is " + point.y);
};


let person: {name: string, age: number} = {
    name: 'Tom',
    age: 31,
};
console.log(`${person.name} is ${person.age} years old.`)
```

TypeScript

Union types

You can also give multiple alternative types, separated by `|`:

```
function printId(id: number | string) {
  console.log("Your ID is: " + id);
};
printId(101);
printId("202");

// this also works with literals:
let alignment: "left" | "right" | "center" = "left";
```

Note: any operations you do on a union type has to work for *every* member of the union (i.e. you can not perform string operations on something of type `number | string`)

TypeScript

Type aliases and interfaces

You can predefine types using `type` or `interface` (for our purposes they are pretty much equivalent):

```typescript
type ID = number | string;

interface Point {
  x: number;
  y: number;
};

// we can now use our defined types for type annotation:
function printId(id: ID) {
  console.log("Your ID is: " + id);
};
```

TypeScript

More examples

```
// define what the structure of a menu should be:
interface Menu {
    food: {
        pizza: string[];
        pasta: string[];
    };
    drinks: string[];
}
// define a specific menu of type Menu:
let myMenu: Menu = {
    food: {pizza: ["margherita", "funghi"], pasta: ["
        carbonara"]},
    drinks: ['coffee', 'milkshake'],
};
console.log(myMenu.drinks[1]) // 'milkshake'
```

Good luck!