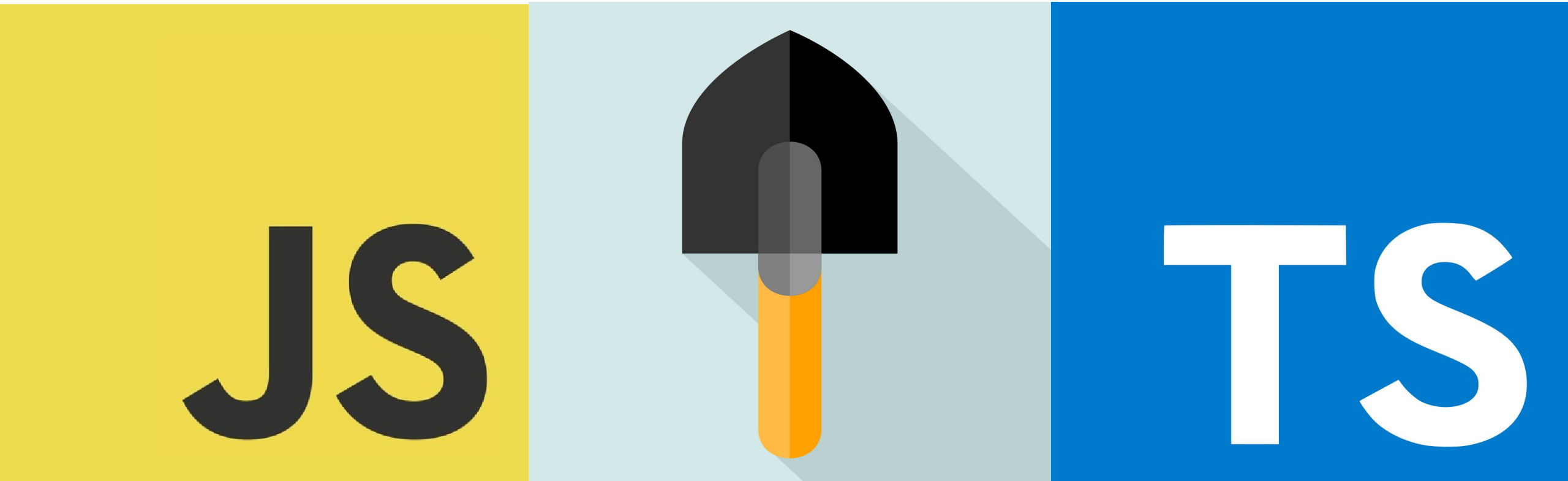
ADCenter

Agile Delivery Center

Capgemini

Digging(in) JavaScript and TypeScript

https://github.com/soyrochus/diggingjsandts



I have been working for nearly thirty years in the industry, working on the crosscut between Business, Technology, and Users; Designing, building & running teams, systems & services.

Currently CTO at Capgemini's European Industrial Center in Spain and Program Manager for devonfw, the standard Open Source software development platform for Capgemini Europe & India.

https://www.capgemini.com/experts/enterprise-architecture/iwan-van-der-kleijn/

https://twitter.com/soyrochus

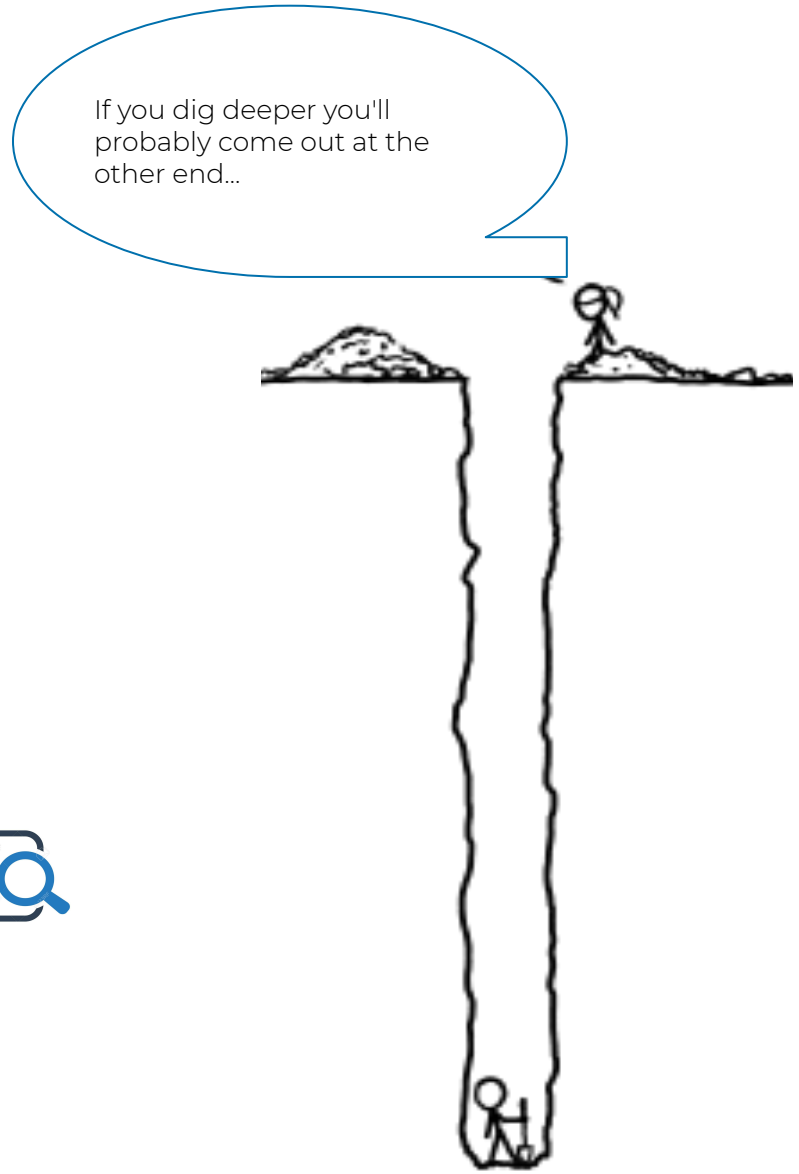
I am Computer Science Engineer with more than 20 years of experience currently focused on software architecture, industrialization processes and management of projects and teams, designing methodologies to ensure the delivery of projects of any kind: from monolith applications to microservices in cluster related technologies and cloud providers.

Currently Senior Software and Lead Architect at ADCenter Valencia (Spain) and Product Owner for devonfw, the standard Open Source software development platform for Capgemini Europe & India.

https://www.linkedin.com/in/santosjimenez/

https://twitter.com/sjimenez77

Caveat or Opinionated Opinion alert



- ✔ This is not a manual nor a book. It's the notes of a short course, basically an introduction.
- ✔ The text is not important. The accompanying code is, even though it's rather basic. Use the code, execute it, modify it, debug it, play with it.
- ✔ Neither this document nor the course itself do represent a well organised structure. It's a *narrative* which offers a thin thread through the labyrinth which is JavaScript/TypeScript.
- ✔ The narrative evolves, showing increasingly complex features, using topics shown previously. The point is to “dig in” and to let the matter grow own you.
- ✔ For that reason, the course only deals with a tiny bit of the fast expanse which is JavaScript/TypeScript. Use the icon seen down-left on many pages to explore more in depth.
- ✔ Finally, this course and document considers JavaScript and TypeScript to be the same language, the latter being a superset of the former. This has some small downsides but the advantages far outweigh these.

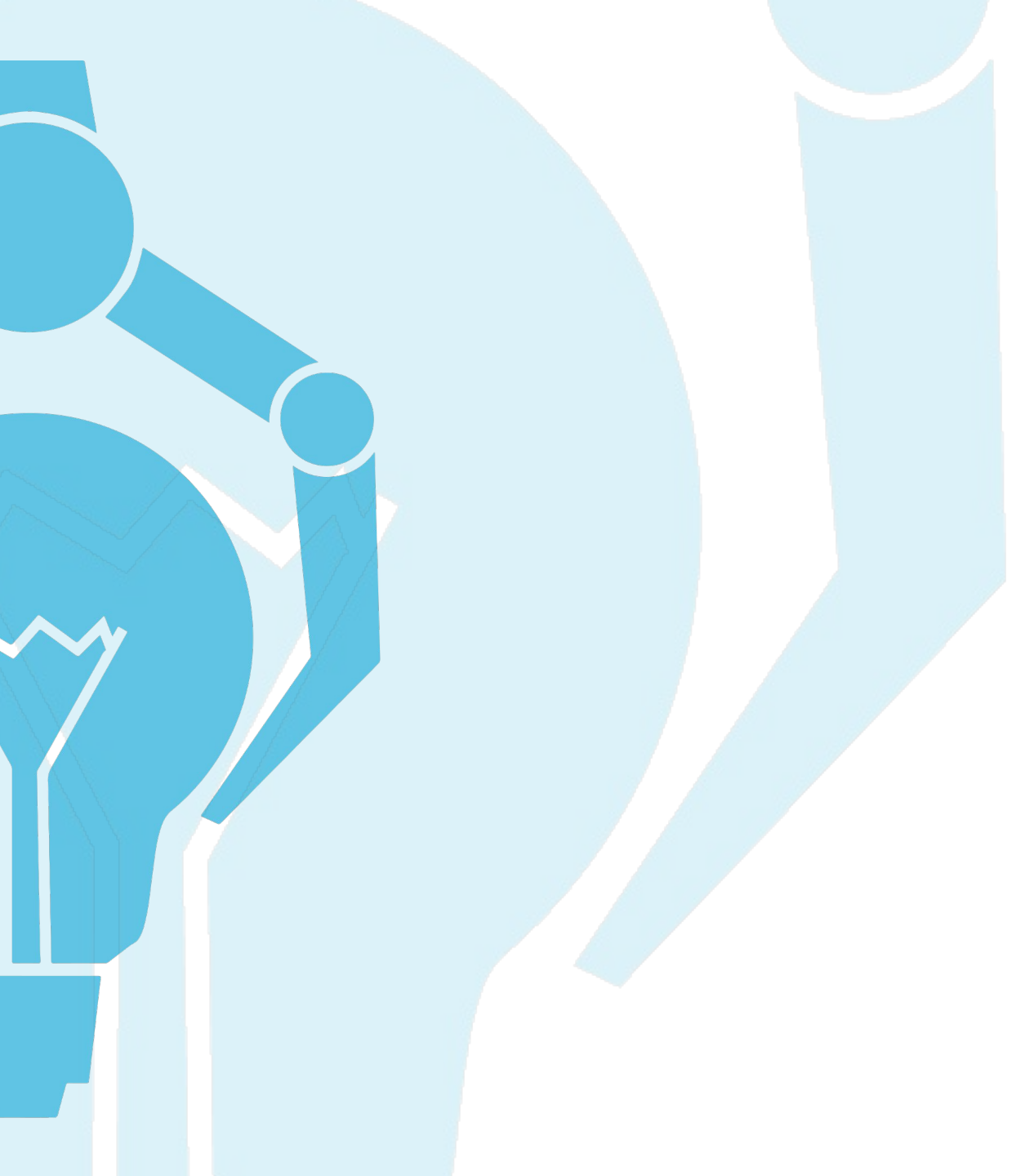
1 Digging In (History, background)

2 Prerequisites

3 Basics

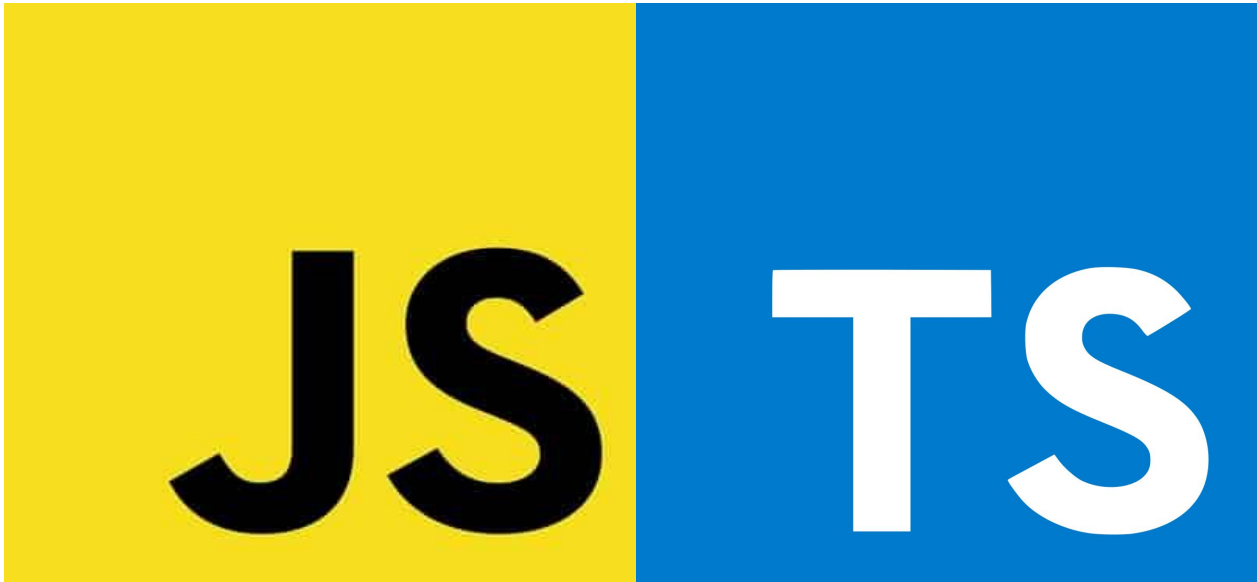
4 Functions and more ...

5 Digging Deeper



Digging In…

(History, background)



“JavaScript” - it's not Java

JavaScript History

1995 — **JavaScript** is born as LiveScript

1997 — **ECMAScript** standard is established

1999 — **ES3** comes out and IE5 is all the rage

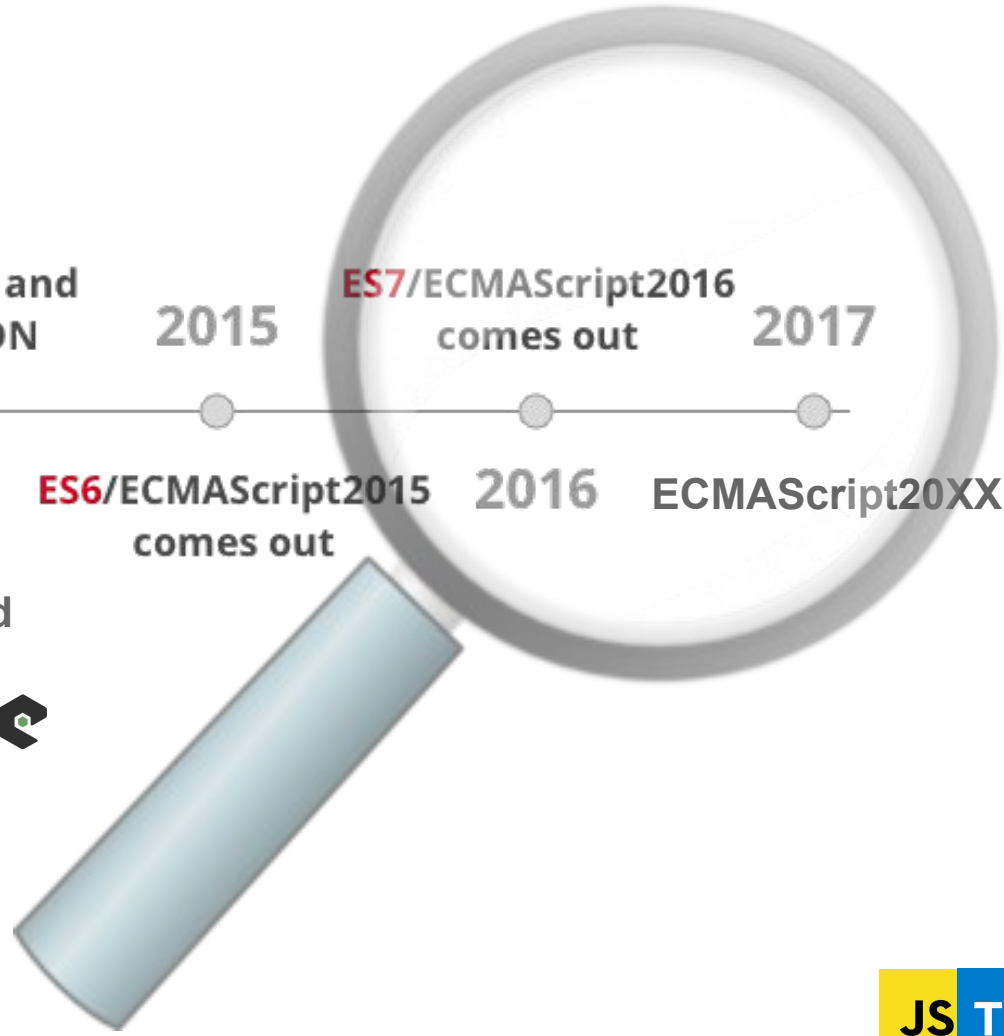
2000 — XMLHttpRequest, a.k.a. AJAX, gains popularity

2009 — Node.js released

2015 — **ES6**/ECMAScript2015 comes out

2016 — **ES7**/ECMAScript2016 comes out

2017 — ECMAScript20XX



Big changes in ES6: in effect a new language

What name to use?

ES6 or The **ECMAScript 2015** Language version was the last big release. Future updates to the specification will be smaller. New language features will be implemented in JavaScript engines before they are officially included in the specification.

You should talk about

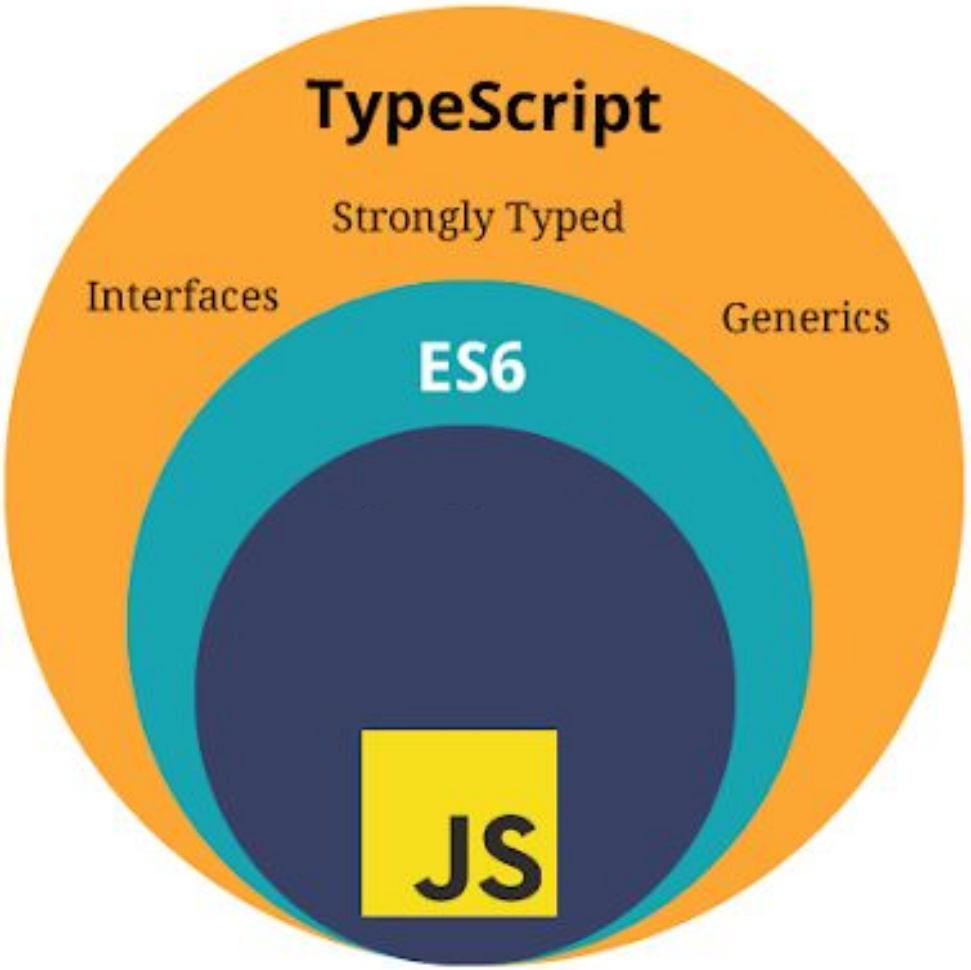
- **use ES6** to refer to *"ECMAScript 2015 Language"* (arrow functions, template strings, Promises), it's shorter than ES2015, and both are unofficial, ES6 was the last big release, and the name is in line with the previous big release ES5, things change after that
- **after ES6, use names of language features**, such as "globalThis" and "Array.prototype.flatMap", the specification is only updated after working implementations exist in JS engines, check TC39 Finished Proposals for a list of features to be included in the next specification
- for historically referring one year's updates to the ECMAScript specification use ES[year]

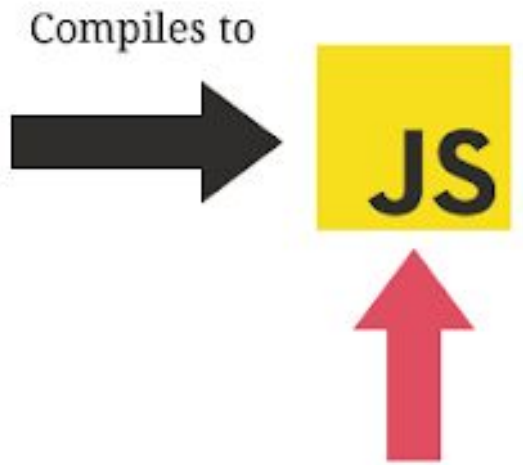
JS TS

TypeScript

"...**TypeScript is a typed superset of JavaScript that compiles to plain JavaScript**..."

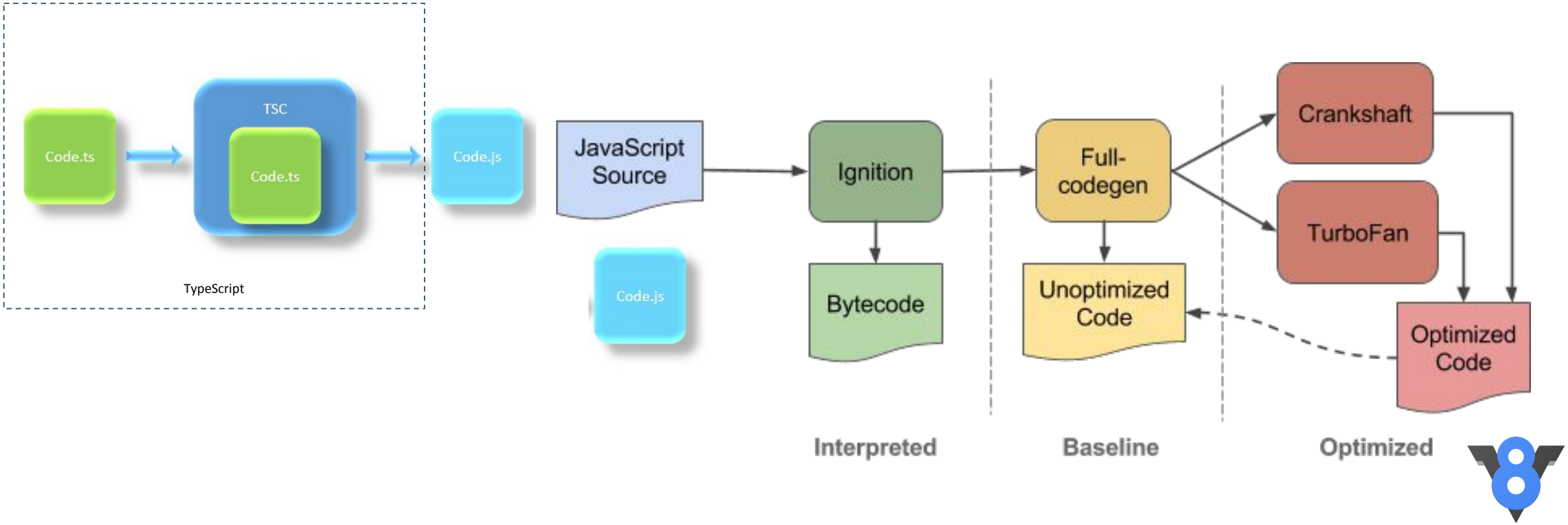
TypeScript was launched for public use in October 2012. It was a result of two years of development at Microsoft. One of the lead designers and developers is Anders Hejlsberg, who is very well known as being the lead architect of C#, as well as the creator of Delphi and Turbo Pascal.





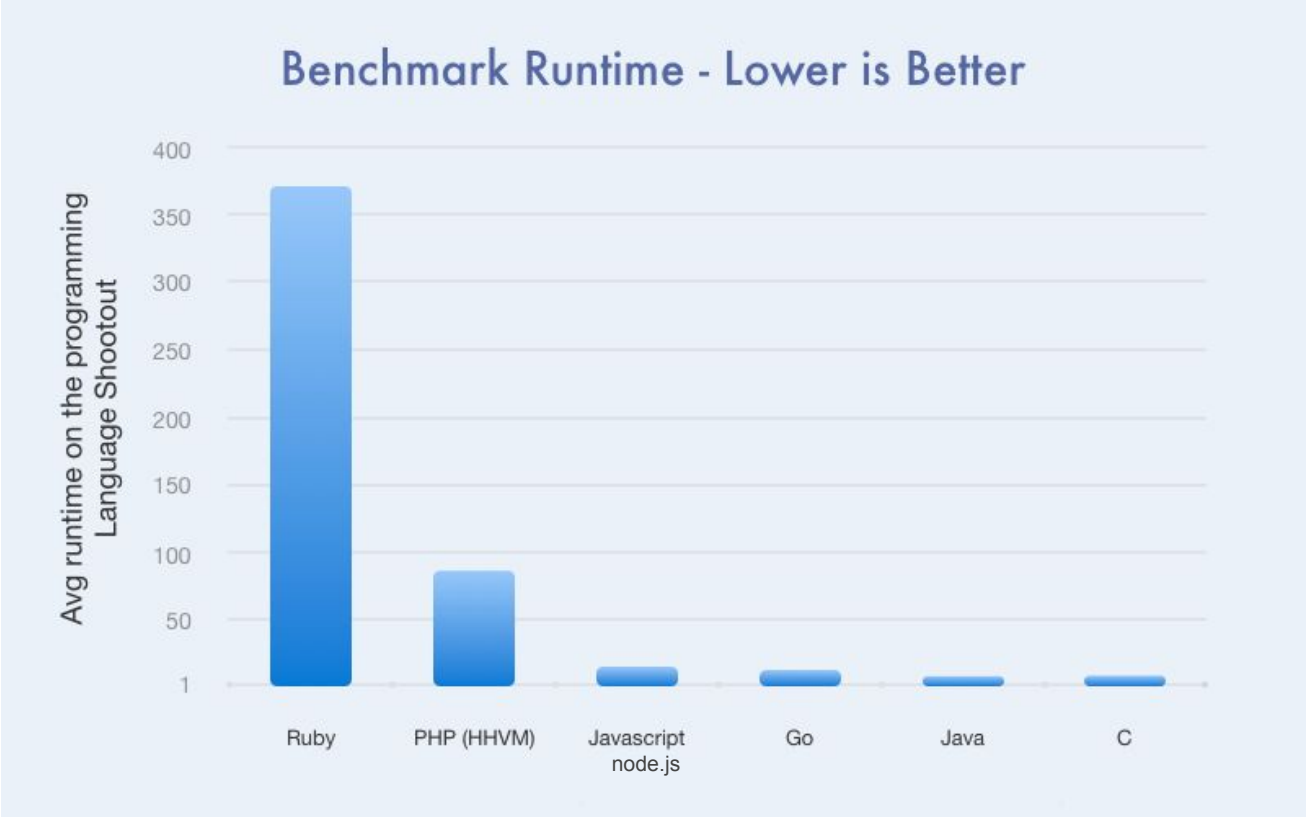
Neither the Browser nor Node.js can run TypeScript so it must be compiled ("transpiled") to JavaScript

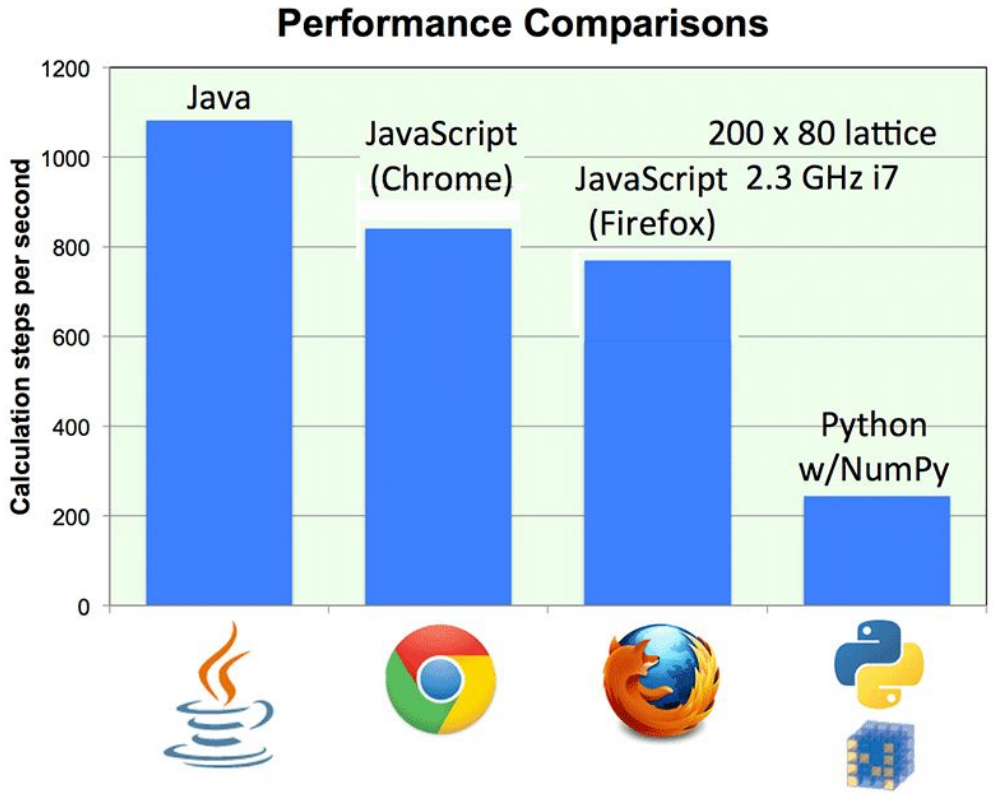
From source code to execution



V8 is a **JavaScript engine** built developed by Google. It is **open source** and written in **C++**. It is used for both client side (Google Chrome) and server side (node.js) JavaScript applications. V8 and other modern JavaScript engines get their speed via just-in-time (JIT) compilation of script to native machine code immediately prior to execution. Code is initially compiled by a baseline compiler, which can generate non-optimized machine code quickly. The compiled code is analyzed during runtime and optionally re-compiled dynamically with a more advanced optimizing compiler for peak performance. In V8, this script execution pipeline has a variety of special cases and conditions which require complex machinery to switch between the baseline compiler and two optimizing compilers, Crankshaft and TurboFan. Apart from this a JavaScript interpreter, called Ignition, can replace V8's baseline compiler, executing code with less memory overhead on some platforms.

Performance



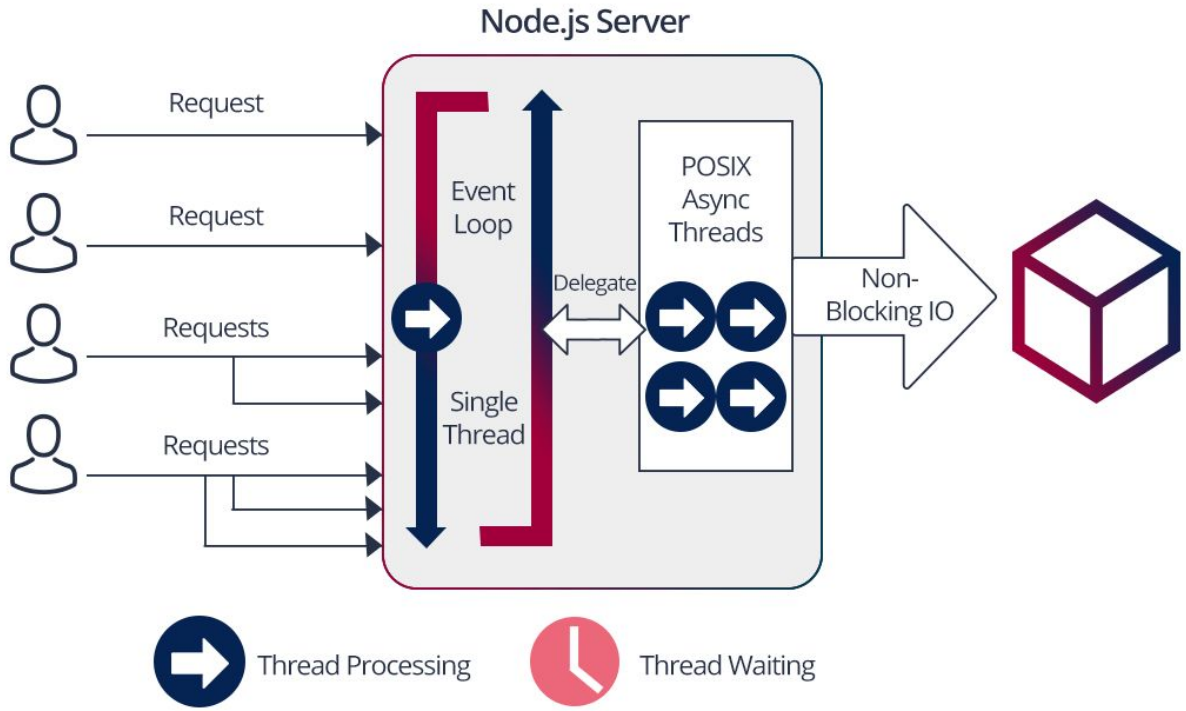


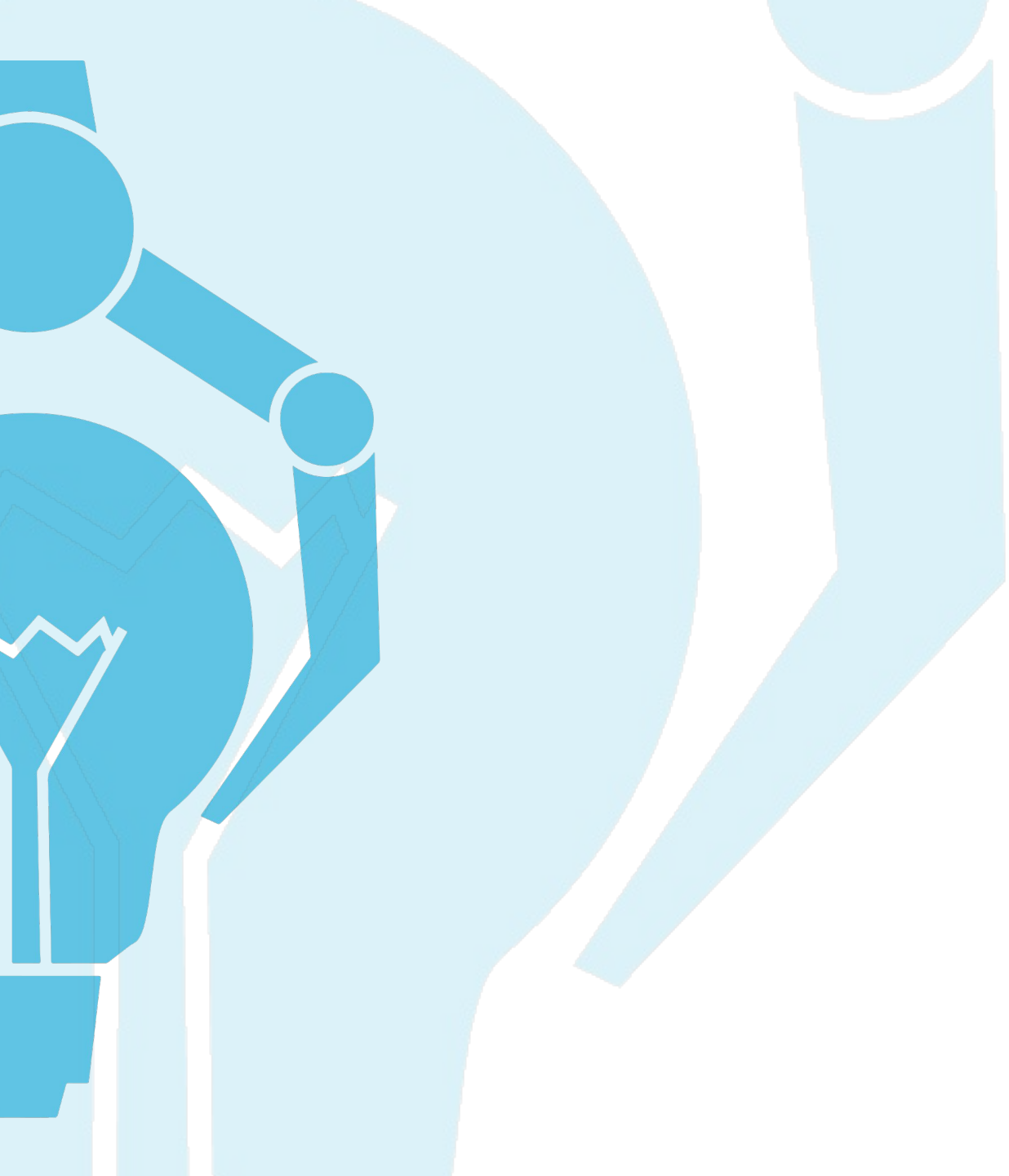
Single threaded

All Node JS applications uses the “Single Threaded Event Loop Model” architecture to handle multiple concurrent clients.

The main event loop is single-threaded but most of the I/O works run on separate threads, because the I/O APIs in Node. js are asynchronous/non-blocking by design, in order to accommodate the event loop.

So JavaScript code runs on **one single thread** but library code executes multi-threaded. This leads to **subtle but significant differences** with languages like Java or C# as related with the run-time model. For example: **you should never block a thread as no other requests can be served (!)**





Prerequisites

JS TS

Quick Installation

- Install **node.js** (see: https://nodejs.org/)
- Install **TypeScript and other tools** globally with the command:

```
npm install -g typescript ts-node prettier eslint
```

- Note that the executables will be installed to a path - typically something like `C:\Users\<<user>>\AppData\Roaming\npm\ -` which needs to be included in your PATH environment variable (see: https://www.architectryan.com/2018/03/17/add-to-the-path-on-windows-10/)
- Installed will be **ts-node**, which allows running a node shell with TypeScript without having a separate compile phase
- Installed will be **ESLint**, the primary linter for both JavaScript and TypeScript. A linter is a tool that analyzes source code to flag programming errors, bugs, stylistic errors, and suspicious constructs.
- Installed will be **Prettier**, which handles code formatting, important to guarantee a consistent coding style in a team
- For configuration option of both ESLint and Prettier see: https://www.robertcooper.me/using-eslint-and-prettier-in-a-typescript-project
- Clone the source code accompanying this presentation with

```
git clone https://github.com/soyrochus/diggingjsandts
```

- Install the remaining dependencies by running in the `diggingjsandts/src` directory

```
npm install
```

JS TS

Integrated Development Environment

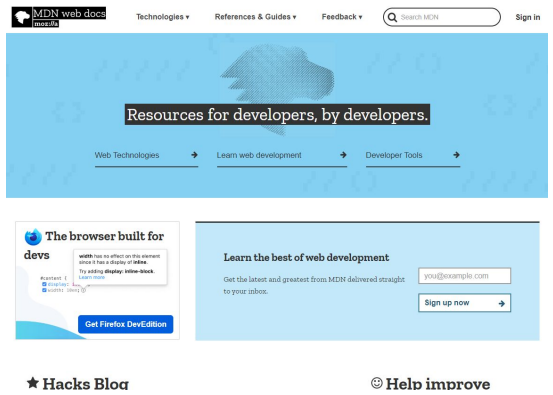
Use an **IDE** with full EcmaScript and TypeScript support:

- **Visual Studio Code**
  - Make sure the Code Runner extension is installed if using VS Code
- **SublimeText** with the TypeScript plugin https://github.com/Microsoft/TypeScript-Sublime-Plugin
- **Atom** with the atom-typescript plugin https://atom.io/packages/atom-typescript
- **WebStorm** https://www.jetbrains.com/webstorm/

Or just install devonfw which will provide you with a fully configured Visual Studio Code instance

JS TS

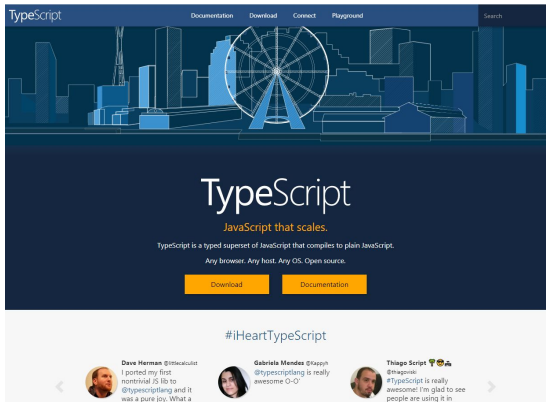
Resources



Mozilla Developer Network (MDN)

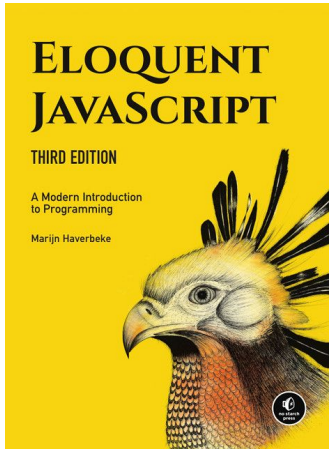
The foremost resource for web developers

https://developer.mozilla.org/



TypeScript website

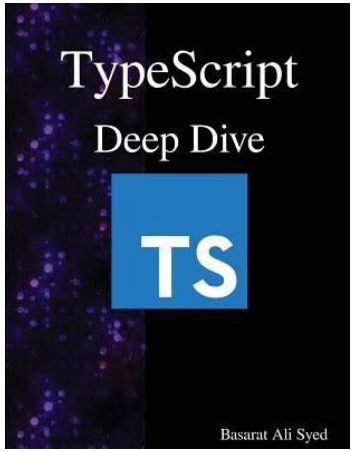
https://www.typescriptlang.org/



Eloquent JavaScript

(free e-book from basic to advanced JavaScript)

https://basarat.gitbook.io/typescript/



TypeScript Deep Dive

(free ebook about Advanced TypeScript)

https://basarat.gitbook.io/typescript/

Debugging node.js

A “step-through debugger” (also called an “interactive debugger” or just “debugger”) is a powerful tool that can be very handy when your application isn’t behaving the way you expect. You can use it to pause your application’s code execution to:

- Inspect or alter application state.
- See the code execution path (“call stack”) that leads to the currently-executing line of code.
- Inspect the application state at earlier points on the code execution path.

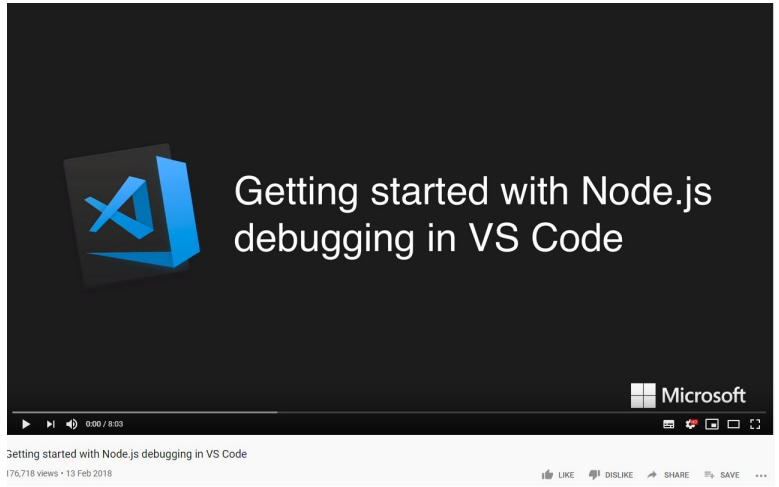
Interactive debuggers can help diagnose logic issues and are an indispensable part of your toolkit.

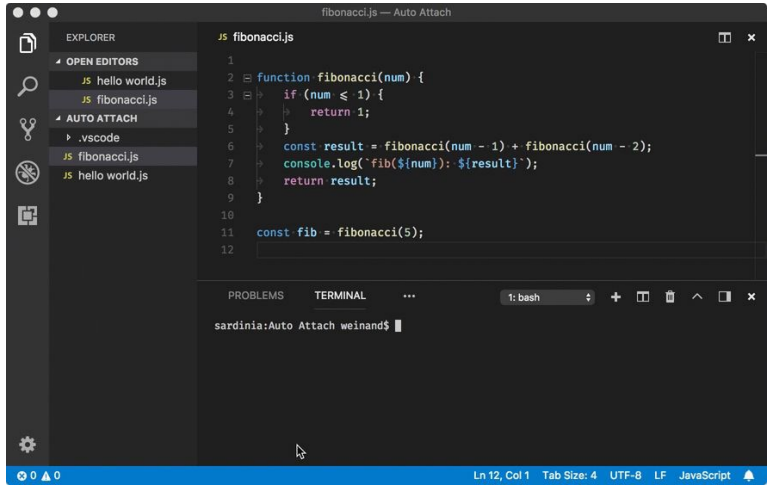
[Debugging Getting Started]

[Debugging using Visual Studio Code]

[Debugging using Google Chrome]

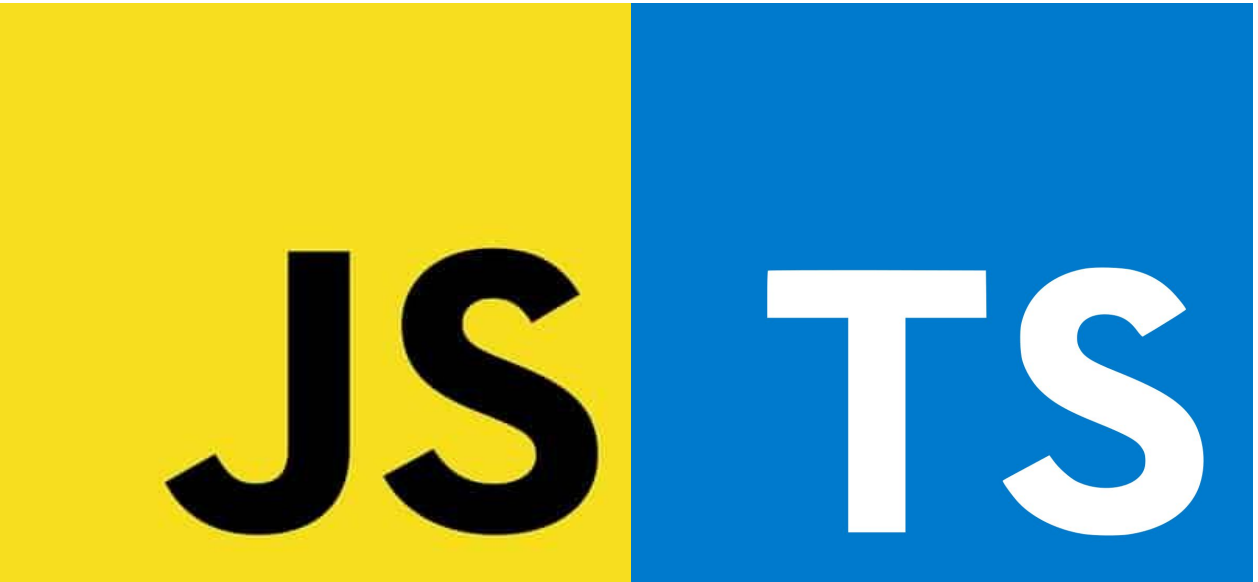
[Debugging using Webstorm]





JS TS

Basics



Basic syntax

```
// single line comment
/* multi-line
  comment */

let age = 15 // variable declaration
let status   // not initialized (set to 'undefined')

const planets = 8 // like let but cannot change


try {
   let o = JSON.parse("=)(/&");
   console.log(o)
} catch (e) {
   console.log(e)
}
```

> ! Before ES6, before the introduction of **let** and **const,** the only way to declare variables in JavaScript was with the keyword **var**. This is now obsolete. Don't use it.

```
// if statement
if ((age >= 14) && (age < 19)) {
    console.log("Eligible")
} else {
    console.log("Not eligible.")
}

// switch statement
switch (new Date().getDay()) {      // input is current day
   case 0:                          // if (day == 0)
       console.log("Sunday")
       break
   case 5:                          // if (day == 5)
       console.log("Friday")
       break
   case 6:                          // if (day == 6)
       console.log("Saturday")
       break
   default:                         // else...
       console.log("No Sabbath or equivalent for major three
religions")
}
```

Basic syntax intentionally (nearly) identical to Java, C, C++

JS TS

Of ";" and Holy Wars

```
let a = 1;
//semi-colon ";" as line separator
//(NOT terminator) is optional
```

Like many things in ES6 and TypeScript, this is one of those topics which is subject to Holy Wars. We will not pick a position during this course. For the subject of clarity, most ";" are omitted from the examples.

Basic syntax - loops

```
//for loop
for (let i = 0; i < 10; i++) {
   console.log(i)
}

// while loop
let j = 0;
while (j < 10) {
   console.log(j)
   j = j + 1
   //j++
}

// do while loop
let k = 0
do {
   k = k + 1
   console.log(k)
} while (k < 10)
```

```
// loop with break
for (let l = 0; l < 10; l++) {
   if (l == 5) { break; }           // stops and exits
the cycle
   console.log(l)
}
// loop with continue
for (let m = 0; m < 10; m++) {
   if (m == 5) { continue }       // skips the rest of
the cycle
   console.log(m)
}
// for-in loop ; iterate over a sequence or iterable
for (let n in [0, 1,2,3,4]) {
   console.log(n)
}
```

The basic loop constructs are error-prone. Prefered are `for-of` and sequence and stream operations (which will be shown later on)

JS TS

Basic types (not identical !)

JS

```
//Number
let decimal = 6
let hex = 0xf00d
let binary = 0b1010
let octal = 0o744

//bigInt
let big = 9007199254740992n //note ‘n’ at end
let bigger = 2n ** 153n
//1141798154164767904846628775559596109106197299 2n

//Boolean
let isDone = false

//String
let part= "Rubeus Hagrid"
let actor = "Robbie Coltrane"
let fact = `${part} is played by ${actor}`
```

≠

TS

```
//Number
let decimal = 6
let hex = 0xf00d
let binary = 0b1010
let octal = 0o744
// bigInt - NOTE do not work on TS
// when compiled to ES5 !!
let big = 9007199254740992n //note ‘n’ at end
let bigger = 2n ** 153n
//1141798154164767904846628775559596109106197299 2n

//Boolean
let isDone = false

//String
let part= "Rubeus Hagrid"
let actor = "Robbie Coltrane"
let fact = `$p{art} is played by ${actor}`
```

JS TS

Type inference

In modern statically typed programming languages like TypeScript, it is not necessary to always declare the type of a variable. The compiler can ***infer*** the type of the expression.

TS

```
//Number
let decimal    : number = 6
let hex        : number = 0xf00d
let binary     : number = 0b1010
let octal      : number = 0o744

//bigInt
let big        : BigInt = 9007199254740992n //note ‘n’ at end
let bigger     : BigInt = 2n ** 153n
//11417981541647679048466287755595961091061972992n

//Boolean
let isDone    : boolean = false

//String
let part      : string = "Rubeus Hagrid"
let actor     : string = "Robbie Coltrane"
let fact      : string = `$p{art} is played by ${actor}`
```

JS TS

Type inference

In modern statically typed programming languages like TypeScript, it is not necessary to always declare the type of a variable. The compiler can ***infer*** the type of the expression.

TS

```
//Number
let decimal           = 6
let hex               = 0xf00d
let binary            = 0b1010
let octal             = 0o744

//bigInt
let big               = 9007199254740992n //note ‘n’ at end
let bigger            = 2n ** 153n
//1141798154164767904846628775559596109106197299 2n

//Boolean
let isDone            = false

//String
let part              = "Rubeus Hagrid"
let actor             = "Robbie Coltrane"
let fact              = `$p{art} is played by ${actor}`
```

JS TS

Static vs Dynamic typing
Strong vs Weak typing

With JavaScript variable declaration & initialisation has the form of:

JS

```
let <<name>> = <<value>>

let decimal = 6
let isDone = false
```

JavaScript is characterized by:
dynamic typing:

- a variable can be assigned to a valueof a different type

```
decimal = "Robbie Coltrane"
```

weak typing:

- operations of different types are valid (and can give unexpected results)

```
console.log(decimal + isDone)
//'Robbie Coltranefalse'
```

≠

With TypeScript variable declaration & initialisation has the form of:

TS

```
let <<name>> : <<type>> = <<value>>

let decimal : number = 6
let isDone : boolean = false
```

TypeScript is characterized by:
static typing:

- a variable **cannot** be assigned to a valueof a different type

```
decimal = "Robbie Coltrane"
// error TS2322: Type '"Robbie Coltrane"' is not assignable to type 'number'.
```

strong typing:

- operations of different types are **invalid**

```
console.log(decimal + isDone)
//error TS2365: Operator '+' cannot be applied to types 'number' and 'boolean'.
```

JS TS

Equality comparisons and sameness

Strict equality using ===

Strict equality compares two values for equality. Neither value is implicitly converted to some other value before being compared. If the values have different types, the values are considered unequal. If the values have the same type, are not numbers, and have the same value, they're considered equal.**Strict equality is almost always the correct comparison operation to use.** For all values except numbers, it uses the obvious semantics: a value is only equal to itself.

```
var num = 0;
var obj = new String('0');
var str = '0';

console.log(num === num)// true
console.log(obj === obj) // true
console.log(str === str) // true

console.log(num === obj) // false
console.log(num === str) // false
console.log(obj === str) // false (!)
console.log(null === undefined) // false
console.log(obj === null) // false
console.log(obj === undefined) // false
```

Loose equality using ==

Loose equality compares two values for equality, *after* converting both values to a common type. After conversions (one or both sides may undergo conversions), the final equality comparison is performed exactly as === performs it. Loose equality is *symmetric*: A == B always has identical semantics to B == A for any values of A and B (except for the order of applied conversions).

The equality comparison is performed as follows for operands of the various types:

		Operand B					
		Undefined	Null	Number	String	Boolean	Object
Operand A	Undefined	`true`	`true`	`false`	`false`	`false`	`false`
	Null	`true`	`true`	`false`	`false`	`false`	`false`
	Number	`false`	`false`	`A === B`	`A === ToNumber(B)`	`A === ToNumber(B)`	`A == ToPrimitive(B)`
	String	`false`	`false`	`ToNumber(A) === B`	`A === B`	`ToNumber(A) === ToNumber(B)`	`A == ToPrimitive(B)`
	Boolean	`false`	`false`	`ToNumber(A) === B`	`ToNumber(A) === ToNumber(B)`	`A === B`	`ToNumber(A) == ToPrimitive(B)`
	Object	`false`	`false`	`ToPrimitive(A) == B`	`ToPrimitive(A) == B`	`ToPrimitive(A) == ToNumber(B)`	`A === B`

JS TS

The *"any"* type in TypeScript

As a superset of JavaScript, TypeScript offers the "any" type. This allows the programmer to opt-out of type checking and **let a variable have the same compile and run-time behaviour as in JavaScript.** So any call to an non-existing method on an object is compiled although it will generate a run-time error.

```
let notSure: any = 4
// same as
let notSure
notSure = 4
notSure.ifItExists()  // okay, ifItExists might exist at runtime
notSure.toFixed()     // okay, toFixed exists (but the compiler doesn't check)
```

The `Object` type plays a similar role, as it does in other languages. However, variables of type `Object` only allow assignment of properties. You can't call arbitrary methods on them, even ones that actually exist.

```
let prettySure: Object = 4
prettySure.toFixed()  // Error: Property 'toFixed' doesn't exist on type 'Object'.
```

JS TS

String type extra

```
let part= "Rubeus Hagrid"
let actor = "Robbie Coltrane"
let fact = `The role as ${part} is played by ${actor}`
```

This is equivalent to:

```
let fact = "The role as " + part + " is played by " + actor
```

```
//which in TypeScript can be written as
let fact : string = "The role as " + part + " is played by " + actor
```

JS TS

Basic types: Array

An **array** is the basic s**equence type** in both JavaScript and TypeScript

```
let list = [1, 2, 3]
let names = ["Harry", "Hermione", "Ron"]
```

Arrays are accessed by using square brackets and putting the position of the element (0 based)

```
console.log(names[0]) // -> "Harry"
```

Neither the length of a JavaScript array nor the types of its elements are fixed.

```
let shoppingItem = ["orange", 10, false]
shoppingItem.push("fruit") // add an element to the back of the array
console.log(shoppingItem) // -> [ 'orange', 10, false, 'fruit' ]
```

In TypeScript an array which behaves like a JavaScript array is an array which all elements of type **any**.

```
let shoppingItem : any[]= ["orange", 10, false]
```

Arrays typically are typed in TypeSCript. There are two notations: the "<*typename*>**[]**" variant and the "**Array<***typename***>**". The former is a specific notation for arrays while the latter is an array expressed as a *Generic Type*.

```
let list1: number[] = [1, 2, 3]
let list2: Array<number> = [1, 2, 3]
```

JS TS

For...of loops

The **for...of statement** creates a loop iterating over iterable objects, including: built-in String, Array, array-like objects (e.g., arguments or NodeList), TypedArray, Map, Set, and user-defined iterables. It invokes a custom iteration hook with statements to be executed for the value of each distinct property of the object.

Iterating over an Array

```
const iterable = [10, 20, 30]
for (const value of iterable) {
  console.log(value)
}
```

You can use let instead of const too, if you re-assign the variable inside the block.

```
const iterable = [10, 20, 30]

for (let value of iterable) {
 value += 1
 console.log(value)
}
```

There are many other examples, like Iterating over a String

```
const iterable = 'boo'

for (const value of iterable) {
 console.log(value)
}
// -> "b"
// -> "o"
// -> "o"
```

See MDN for a full list of options

Difference between for...of and for...in

Both for...in and for...of statements iterate over something. The main difference between them is in what they iterate over.

The for...in statement iterates over the enumerable properties of an object, in an arbitrary order.

The for...of statement iterates over values that the iterable object defines to be iterated over.

JS TS

Basic types: Tuple & type aliases

TS

In TypeScript a **tuple** represent an array where the type of a determined number of items is known and does not have to be the same.

```
// Declare a tuple type
let x: [string, number]
// Initialize it
x = ["hi", 10] // OK
console.log(x[0]) // -> 'hi'

// Initialize it incorrectly
// error TS2322: Type 'number' is not assignable to
type 'string'.
// error TS2322: Type 'string' is not assignable to
type 'number'.
x = [10, "hi"]
```

TS

A **type alias** can be created with the **type** keyword to simplify type notation.

```
type seat = [string, number]

let john : seat = ["John", 15]
```

Note that a type alias is not a “real” new type. And type aliases cannot be inferred. So the following are not equivalent:

```
let karen : seat = ["Karen", 15]
let karen        = ["Karen", 15]
```

JS TS

Basic types: enum

TS

Enum

An **Enum** allows us to assign names to sets of numeric variables.

```
enum Color {Red, Green, Blue}
let c: Color = Color.Green
```

By default **enums** start to initialize the indexes by 0. But we can initialize manually:

```
enum Color {Red = 1, Green, Blue}
let c: Color = Color.Green
```

Or define them completely:

```
enum Color {Red = 1, Green = 2, Blue = 4}
let c: Color = Color.Green
```

If we are not sure of the assigned mapping, **enums** allows us the following:

```
enum Color {Red = 1, Green, Blue}
let colorName: string = Color[2]
console.log(colorName)
```

JS TS

Basic types: null, undefined (and void)

Null and Undefined

Both are types in shared by JavaScript and TypeScript. They seem similar but have specific meanings.

Undefined is a value set to a variable if no value has been assigned.

```
let u // undefined
```

Null can be used to explicitly state that “no value” is returned.

```
let n = null
```

By default, both are subtypes of all the other, therefore we can assign **null** or **undefined** to any variable of any other type like **number** or **string**.

Both values evaluate to “false”. See “Falsy Values”.

Void

Basically the opposite of any. It is used in functions that do not return any type.

```
function warnUser() {
    alert("This is my warning message")
}
or in TypeScript
function warnUser(): void {
    alert("This is my warning message")
}

let nothing = warnUser()
//nothing is undefined
```

JS TS

Falsy Values

In JavaScript and TypeScript there is a specific **`boolean`** type which can has the value of **`true`** and **`false`**.

Values of this type can be used for boolean expressions, expressions which evaluate to **`true`** and **`false`**. But not just values of the boolean type can be used. It is possible to use a series of values from other types. These evaluate to false when coerced by JavaScript's typing engine into a boolean value, but they are not necessarily equal to each other. It is said that these values are “falsy”. Not equal to **`false`** but evaluating to it.

The falsy values in JavaScript are **`0`** , **`0n`** , **`null`** , **`undefined`** , **`false`** , **`NaN`** , and the empty string **`“”`** .

TS

```ts
let realfalse   : boolean = false
let falsyNumber : number = 0
let falseNaN    : number = NaN
let falsyBigInt : bigint = 0n
let nullValue: any = null
let undefinedVar  // undefined
let falseString: string = ""
```

JS TS

Type assertions

When we end up in a situation where you'll know more about a value than TypeScript does, we can tell the compiler "trust me, I know what I'm doing".

Usually this will happen when you know the type of some entity could be more specific than its current type.

There are two ways to do it:

```
let someValue: any = "this is a string"
// angle-bracket syntax
let strLength: number = (<string>someValue).length
// as syntax
let strLength: number = (someValue as string).length
```


Functions
and more...
JS
TS

Functions

Functions are one of the fundamental building blocks in JavaScript. A function is a JavaScript procedure—a set of statements that performs a task or calculates a value. The basic definition shared with all version of JavaScript and TypeScript is:

```
function multiply(x, y) {
    return x * y
}
```

Note that the function is a value (the executable code) assigned to a variable (the function name). So the above definition can be written as :

```
let multiply = function(x, y) {
    return x * y
}
```

In ES6 and TypeScript a new syntax has been introduced, the so-called Arrow functions (“fat arrow functions”). It's a shorter, more concise way to write functions and the arrow function behave differently (very subtle, this will be treated later), solving an issue with “function”.

```
let multiply = (x, y) => x * y  //single line, without { .. } and with implicit return


let multiply = (x, y) => {
   return x * y
}
```

JS TS

Function parameters

Functions can have zero or more parameters. A parameter is used to pass information to the code inside the function. JavaScript does not support Optional parameters through syntax. But if a parameter is not passed a value, it will evaluate to “undefined”.

ES6 also supports **default parameters**. These can be assigned default values which are set once a parameter is not passed to the function.

```
let buildName = (firstName, lastName, title= 'Mr./Ms.') =>{
 if (lastName)
   return `${title} ${firstName} ${lastName}`
 else
   return  `${firstName}`
}

// Note that this will not work in TypeScript with strict options set. It will fail to compile with
// error TS2554: Expected 2-3 arguments, but got 1.
console.log(buildName("John"))  // -> ‘John’
console.log(buildName("Rupert", "Grint")) // -> ‘Mr./Ms. Rupert Grint’
console.log(buildName("Emma", "Watson", "Miss")) // -> ‘Miss Emma Watson’
```

The **rest parameter syntax (with the spread operator: “...”)** allows us to represent an indefinite number of arguments as an array.

```
let cast = (...castmembers) =>{
   console.log(`Harry Potter cast members are: ${castmembers} `)
}
cast("Daniel Radcliffe", "Emma Watson", "Rupter Grint", "Alan Rickman", "Michael Gambon")
// -> Harry Potter cast members are: Daniel Radcliffe,Emma Watson,Rupter Grint,Alan Rickman,Michael Gambon
```

JS TS

Optional parameters in TypeScript

Note that in TypeScript with strict compile options set, if a parameter is not passed a value, it will NOT evaluate to "undefined" but rather generate an error.

Instead of the implicit behaviour, in TypeScript a parameter can be explicitly defined to be **optional** by putting a question mark behind its name: *"typename?"*

```
let buildName = (firstName: string, lastName?: string, title: string = 'Mr./Ms.') =>{
 if (lastName)
   return `${title} ${firstName} ${lastName}`
 else
   return  `${firstName}`
}

console.log(buildName("John"))  // -> 'John'
console.log(buildName("Rupert", "Grint")) // -> 'Mr./Ms. Rupert Grint'
console.log(buildName("Emma", "Watson", "Miss")) // -> 'Miss Emma Watson'
```

Values, references and parameters

In JavaScript there exist so-called value-types and reference types. **Value types** are simple atomic values like numbers but also strings. They are **immutable**: they cannot be changed. A variable points to a value type. It is not the value itself.

```
let name2 = "Harry" // variable “name” can be
re-assigned but the value “Harry” cannot be changed
```

A **reference type** on the other hand is a value which can refer to, point at, other types: both value and reference types. Both arrays and objects are examples of reference types. The elements **IN** a reference type can be changed or rather: the elements in a reference type can “point at” other type, i.e. behave like variables.

```
let shoppingItem = ["orange", 10, false]
shoppingItem[1] = 500
console.log(shoppingItem) // ->[“orange”, 500, false]
```

Now function parameters which are of a value type are “passed by value”. That means that an assignment on the parameter inside the function does not change the value of the variable which passed the value to the function.

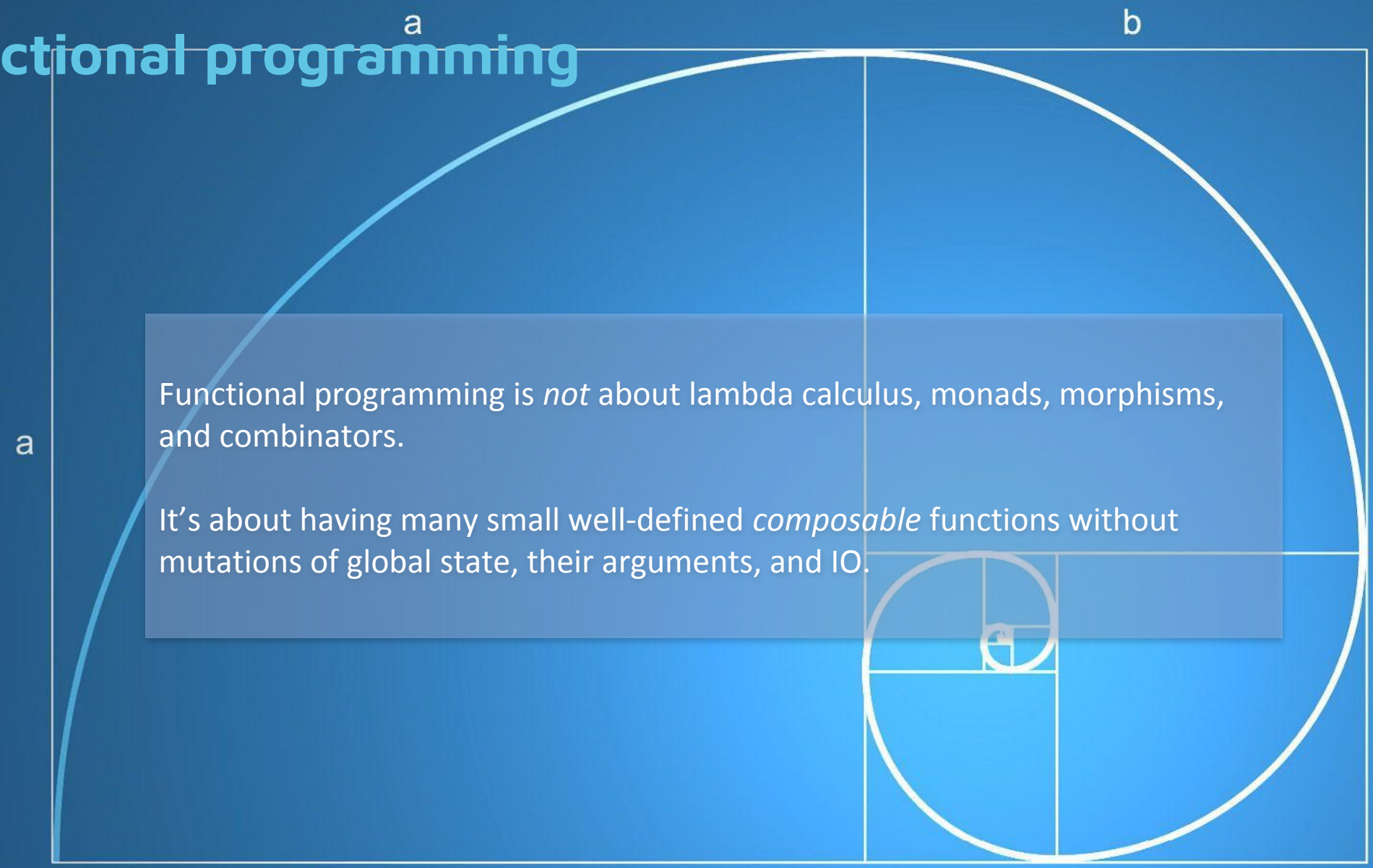
```
let reset = (x) => {
 x = null
}
let a = 100
reset(a)
console.log(a) // -> 100
```

When a reference type is passed, the original variable cannot be changed. But the members of the reference type it points to can be changed!

```
let resetArr = (x) => {
 x[0] = null
}
let a = [100, 200, 300]
resetArr(a)
console.log(a) // -> [null, 200, 300]
```

JS TS

Functional programming



Functional programming is *not* about lambda calculus, monads, morphisms, and combinators.

It's about having many small well-defined *composable* functions without mutations of global state, their arguments, and IO.

$$\frac{a+b}{a} = \frac{a}{b} = \varphi \approx 1{,}61803$$

Recursion

JavaScript and TypeScript functions support recursion. Which is a critical feature where a function can call itself, multiple times if necessary. Many algorithms are recursively defined so a language which support recursion is a great help in implementing such algorithms, even if for a first, naive, version (as recursion is “expensive” a more optimal, but more complex function are typically possible). For example:

Given a number *N* return the index value of the **Fibonacci sequence**, where the sequence is:

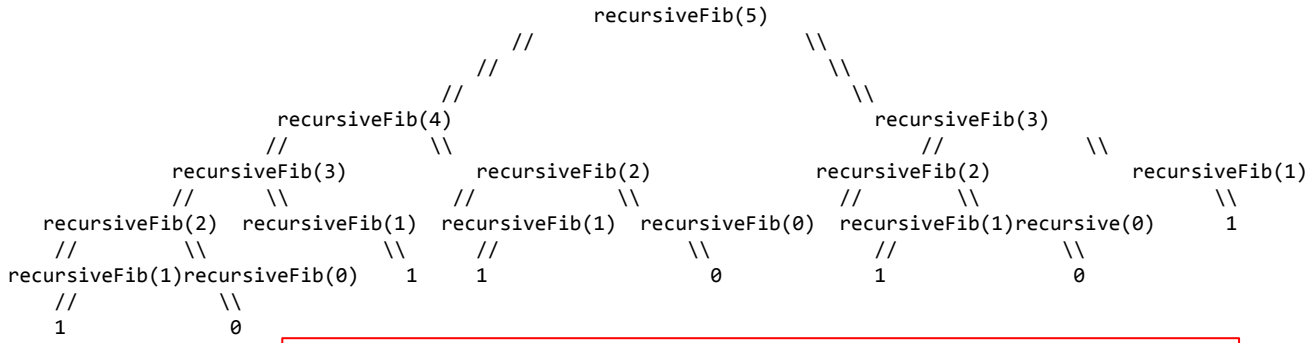
```
1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, ...
```

After a quick look, you can easily notice that the pattern of the sequence is that each value is the sum of the **2 previous values**, that means that for N=5 → 2+3 or in maths:

```
F(n) = F(n-1) + F(n-2)
```

The implementation in JavaScript is:

```
let fibonacci = (n) => {
 console.log(`fibonacci(${n})`)
 if (n === 0 || n === 1)
   return n;
 else
   return fibonacci(n - 1) + fibonacci(n - 2);
}

console.log(fibonacci(20))  // -> 6765
```



This diagram shows the inefficiency of a naive recursive implementation. Calculation the 5th fibonacci number leads to 15 calls to the function. Calculation of the 50th Fibonacci number leads to calling the function over **40 billion times.**

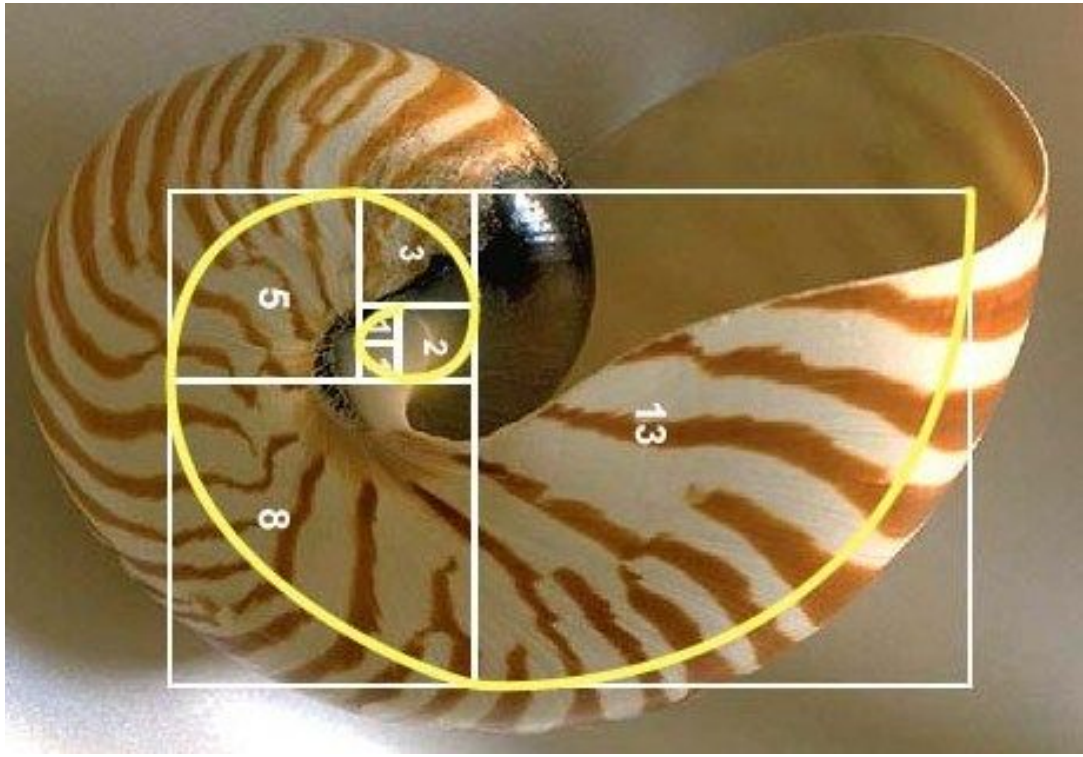
JS TS

Memoization

Recursion can be inefficient. For example, when calculation the 50th Fibonacci number, the function is called over **40 billion times.**

Through the usage of reference types (array, objects) it is possible to implement **memoization or caching** of values in recursive functions. In case of calculation of the 50th Fibonacci number, this brings the **number of calls to the fibonacci function back to a mere 99(!)**.

```
let fibonacci2 = (n, seq = []) => {
 console.log(`fibonacci2(${n})`)
 if (seq[n]) {
   return seq[n]
 } else {
   if (n === 0 || n === 1)
     {
       seq[n] = n
     } else {
       seq[n] = fibonacci2(n - 1, seq) + fibonacci2(n - 2, seq)
     }
   return seq[n]
 }
}
let seq = []
console.log(fibonacci2(20, seq)) //-> 6765
console.log(seq) // -> [0,1,1,2,3,5,8,13,21,34,55,89,144,233,377,610,987,1597,2584,4181,6765]
```



JS TS

Function as values

In JavaScript functions are not only syntax but also values, which means they can be assigned to variables, stored in the properties of objects or the elements of arrays, passed as arguments to functions, and so on. This is a critical part of JavaScript and a principal reason why it has been so successful.

The canonical example is to pass functions to the JavaScript sequence (array) methods like map, filter, reduce etc. The functions are operations which are executed by these generic functions (methods) on the array.

```
let shoppingList = [["orange", 10, "REGULAR"],
                    ["apple", 2, "SALE"],
                    ["orange", 15, "REGULAR"]]

let toSale = (e) => {
 if (e[2] == "SALE")
     e[2] = true
   else
     e[2] = false
 return e
}
let big = (e) => e[1]> 5

let products = shoppingList.map(toSale).filter(big)
console.log(products) // -> [ [ 'orange', 10, false ], [ 'orange', 15, false ] ]
```

The **map()** method **creates a new array** populated with the results of calling a provided function on every element in the calling array. It allows **transformation of a sequence**

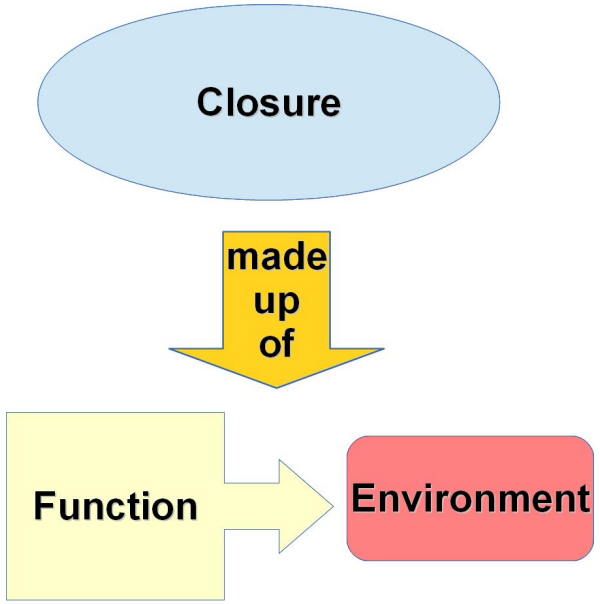
The **filter()** method **creates a new array** with all elements that pass the test implemented by the provided function. It allows **searching for particular conditions** (“filtering”) in a sequence

JS TS

Closure

A **closure** is the combination of a function bundled together (enclosed) with references to its surrounding state (the **lexical environment**). In other words, a closure gives you access to an outer function's scope from an inner function. In JavaScript, closures are created every time a function is created, at function creation time. With closures a function can be **parameterized**:

```
let toPower = (exponent) => {
 return (base) => base ** exponent
}

let square = toPower(2)
let cube = toPower(3)

console.log(square(3))  // -> 9
console.log(cube(3))    // -> 27
```



With closures **state can be passed between function calls** (without using global variables):

```
let counter = (step, init=0) => {
 let state = init
 return () => {
      state += step
      return state
 }
}
let one = counter(1)
let pair = counter(2)
let triple = counter(3)
let cuad = counter(4)

console.log(one())  // -> 1
console.log(one())  // -> 2
console.log(one())  // -> 3
console.log(cuad()) // -> 4
console.log(cuad()) // -> 8
console.log(cuad()) // -> 12
```

Parameter and Function types

TS

In TypeScript parameters and function return are typed as variables declared with let and const (and so are they inferred when no type declaration is given

```
let multiply = function(x: number, y: number): number
{
   return x * y
}
```

It should be noted that functions have their own types. Either the generic, primitive Function type (which should be avoided) or a syntax which defines the type as a function without function body (with the { .. }) . **The type alias keyword** is very useful for this purpose to make the code more readable.

In the example of the left that allows writing the definition of **mapmatrix** as

```
let mapMatrix = (matrix: matrix2d, operation: cellop)
```

rather than the more convoluted

```
let mapMatrix2 = (matrix: Array<Array<number>>,
operation: (_: number) => number)
```

TS

```
type matrix2d = Array<Array<number>>
type cellop = (_: number) => number
let mapMatrix = (matrix: matrix2d, operation: cellop)=> {

   let submatrix = (matrix : Array<number>)=>{
       return matrix.map(operation)
   }
   return matrix.map(submatrix)
}

let data = [[5, 9, -1], [100, 2], [-10, -8, 56]]
let floorzero : cellop = (e) => {
   if (e < 0)
       return 0
   else
       return e
}
console.log(mapMatrix(data, floorzero)) // -> [ [ 5, 9, 0 ],
[ 100, 2 ], [ 0, 0, 56 ] ]
```

JS TS

Generics

TS

Generics in Typescript give us the ability to pass in a range of types to a component, adding an extra layer of abstraction and re-usability to your code. Generics allow the usage of “type variables” which allow the creation of functions, interfaces and classes which get the real type inserted once using them. This avoid the usage of the **any** type which is **not type safe**.

In the example on the right, the type alias `Matrix2DG` is parametrized for type T (see the notation “**<T>**”) and so are the corresponding type `cellopG` and function `mapMatrixG`. Once used, by calling the function `mapMatrixG` with data structures and functions which have real types (in the example, `number` and `bigint` respectively), **two instance**s of `mapMatrixG` are used. One with a number type and the other with bigint. So the **type- variable T** is replaced by number in the one instance and by bigint in the other.

Generics give us great flexibility for assigning data to items in a type-safe way when *simplifying or minimising code where multiple types can be utilised. In* general there are two criteria we should meet when deciding whether to use generics:

1. When your function, interface or class will **work with a variety of data types**
2. When your function, interface or class **uses that data type in several places**

Note that generics only work at compile time. They do not exist in the generated Javascript (!)

TS

```
type Matrix2DG<T> = Array<Array<T>>
type cellopG<T> = (_: T) => T
function mapMatrixG<T>(matrix: Matrix2DG<T> , operation: cellopG<T>) {
   let submatrix = (matrix : Array<T>)=>{
      return matrix.map(operation)
  }
  return matrix.map(submatrix)
}

let examplenumber = [[5, 9, -1], [100, 2], [-10, -8, 56]]
let examplebigint = [[5n, 9n, -1n], [100n, 2n], [-10n, -8n, 56n]]

let floor_zero_number : cellopG<number> = (e) => (e<0) ? 0 : e
let floor_zero_numbigint : cellopG<number | bigint> = (e) => (e<0) ? 0
: e

console.log(mapMatrixG(examplenumber, floor_zero_number)) // -> [ [ 5,
9, 0 ], [ 100, 2 ], [ 0, 0, 56 ] ]

// compile error TS2345: Argument of type 'cellopG<number>' is not
assignable to parameter of type 'cellopG<bigint>'.
//console.log(mapMatrixG(examplebigint, floor_zero_number))

// This works
console.log(mapMatrixG(examplebigint, floor_zero_numbigint)) // -> [ [
5n, 9n, 0 ], [ 100n, 2n ], [ 0, 0, 56n ] ]
```

JS TS

Union Types and Type guards

TS

We can use **union types** in TypeScript to **combine multiple types into one type:**

let text: string | string[];

Variable **text** can now be either string or string array which can be pretty handy in some particular cases.

Use case in a function:

```
function print(text: string | string[]): string {
   if (typeof text === 'string') {
       return text
   }
   // compiler now knows that you can use join
   // and that variable type is definitely string[]
   return text.join(' ')
}

let x = print('hello text')
let y = print(['hello', 'text', 'array'])

// let z = print(5) // Error: Argument of type '5' is not
assignable to type 'string | string[]'
```

TS

Technique used inside of function ***print*** is known as t**ype guarding**. We first checked if parameter ***text*** is of type **'string'.** If the text is "string" that value is returned. Therefore in remaining part of the function it was known to compiler that parameter ***text*** is of type **'string[]'** – array of strings.

```
class Student {
   study() {}
}
class Professor {
   teach() {}
}
function getPerson(n: number): Student | Professor {
   if (n === 1) {
       return new Student()
   } else {
       return new Professor()
   }
}
let person: Student | Professor = getPerson(1);
if (person instanceof Student) {
   person.study() // OK
} else {
   // person.study()  // Error, person is of type Professor here.
   // so compiler recognizes we can call function teach()
   person.teach()
}
```

JS TS

Object literals - old style

Apart from functions, a powerful feature of JavaScript is the capability to **create objects without defining classes**. This unifies the concept of *hashmaps*, *structs* and *object singletons* of other program languages in one simple construct. **Object literals** make it easy to quickly create objects with properties inside the curly braces. To create an object, we simply notate a list of key: value pairs delimited by comma.

Note that properties in objects can be accessed with:

dot notation: `shoppingItem.name = "orange"`
square bracket notation: `shoppingItem["name"] = "orange"`

The latter allows for computed property name indexing. It is equivalent to how hashmaps work in other languages. However, in JavaScript there is some overlap between Arrays and Objects so care has to be taken the two are not mixed up.

```
let shoppingItem = {
    name: "orange",
    number: 10,
    sale: true,
    report: function() {
        let forSale
        if(this.sale){
            forSale = "for Sale! See the offer..."
        } else {
            forSale = "not for Sale. Normal price-quote in effect."
        }
        console.log(`The product ${this.name} is ${forSale}`)
    }
}

shoppingItem.report() // -> The product orange is for Sale! See
the offer...
```

JS TS

Methods: functions as properties - and `this`

Note that methods in JavaScript are simple functions assigned to properties. **The properties of the object can be read from within the function with the `this` keyword. "*this*" refers to the parent object.**

Due to the history of JavaScript this is one of the areas where the **function keyword** is preferred over the **arrow notation for functions**. The two differ in the value which **"this"** evaluates to:

function keyword: `this` refers to the the functions' context. This can be it's **parent object** when assigned to an object property or **global scope** if the function not assigned to property of object

So: in `shoppingItem.report()`, **`this`** refers to the object report is part of.

Just calling `report` (without the object), **`this`** refers to the global object.

arrow notation: `this` refers to the **lexical environment** at the time of the function definition (as in closures)

```
let shoppingItem = {
    name: "orange",
    number: 10,
    sale: true,
    report: function() {
        let forSale
        if(this.sale){
            forSale = "for Sale! See the offer..."
        } else {
            forSale = "not for Sale. Normal price-quote in effect."
        }
        console.log(`The product ${this.name} is ${forSale}`)
    }
}

shoppingItem.report() // -> The product orange is for Sale! See
the offer...
```

JS TS

Functions as objects

An important part of functions in JavaScript is the fact that they are objects themselves. Methods like `call`, `apply` and `bind` can be used to control the invocation of the function.

`apply(thisArg: any, argArray?: any): any`
The object to be used as the this object.

Calls the function, substituting the specified object for the this value of the function, and the specified array for the arguments of the function.

`call(thisArg: any, ...argArray: any[]): any`
Calls a method of an object, substituting another object for the current object, with a list of arguments to be passed to the method. While the syntax of this function is almost identical to that of apply(), the fundamental difference is that call() accepts an argument list, while apply() accepts a single array of arguments.

`bind (thisArg: any, ...argArray: any[]): any`
An object to which the this keyword can refer inside the new function.

For a given function, creates a bound function that has the same body as the original function. The this object of the bound function is associated with the specified object, and has the specified initial parameters.

```
let templ = {title: "Professor", school: "Hogwarts"}

let teacher1 = function(firstName,surName) {
   return `${this.title} ${firstName} ${surName} at ${this.school}`;
}
console.log(teacher1.apply(templ,["Minerva","McGonagall"])) // ->
Professor Minerva McGonagall at Hogwarts
console.log(teacher1.call(templ,"Minerva","McGonagall")) // -> Professor
Minerva McGonagall at Hogwarts
let boundteacher1 = teacher1.bind(templ)
console.log(boundteacher1("Minerva","McGonagall")) // -> Professor
Minerva McGonagall at Hogwarts

let teacher2 = (firstName,surName) =>{
   return `${this.title} ${firstName} ${surName} at ${this.school}`;
}
console.log(teacher2.apply(templ,["Minerva","McGonagall"])) // ->
undefined Minerva McGonagall at undefined
console.log(teacher2.call(templ,"Minerva","McGonagall")) // -> undefined
Minerva McGonagall at undefined
let boundteacher2 = teacher2.bind(templ)
console.log(boundteacher2("Minerva","McGonagall")) // -> undefined
Minerva McGonagall at undefined
```

JS TS

Object literals - ES6

ES6 makes the declaring of object literals concise and thus easier. Three major ways it does this are :

1. It provides a shorthand syntax for initializing properties from variables.
2. It provides a shorthand syntax for defining function methods.
3. It enables the ability to have computed property names in an object literal definition.

```
let getlang = (name)=>{
   return "EN"  // mock for demonstration purposes
}
let name =  "orange"
let number = 10

let shoppingItem = {
   number,
   sale: true,
   [getlang(name)]: name.toLowerCase(),

   report() {
       let forSale
       if(this.sale){
           forSale = "for Sale! See the offer..."
       } else {
           forSale = "not for Sale. Normal price-quote in effect."
       }
       console.log(`The product ${this.EN} ${forSale}`)
   }
}

shoppingItem.report() // -> The product orange is for Sale! See the
offer...
```

JS TS

Creating object instances

New instances of an object can be created by parameterizing them with a function. The new ES6 Object Literal syntax makes this concise and efficient. See the example below.

This should not be mistaken for **using a function as a Constructor**. This was the only way objects could be instantiated with the **new keyword** before ES6. See the example on the right. Best practice would be to **use new only with the class syntax for defining object classes.**

```
let createShoppingItem = (name, count, onsale) => {
   return {
        name,
        count,
        onSale,
        report() {
              if(this.onsale)
                    console.log(`The product ${this.name} is for Sale! See
the offer...`)
              else
                    console.log(`The product ${this.name} is for Sale! not
for Sale. Normal price-quote in effect."
        }
   }
}
let shoppingItem = createShoppingItem("orange", 10, true)
shoppingItem.report() // -> The product orange is for Sale! See the
offer...
```

```
function ShoppingItem(name, count, onsale) {
   this.name  = name
   this.count = count
   this.onsale = onsale
}
ShoppingItem.prototype.report = function() {
   if(this.onsale)
    console.log(`The product ${this.name} is for Sale! See the offer...`)
    else
    console.log(`The product ${this.name} is for Sale! not for Sale.
Normal price-quote in effect."
}

//The right way
let orange = new ShoppingItem("orange", 10, true)
orange.report() // -> The product orange is for Sale! See the offer...
// The wrong way
name = "Very Import Value in Global Variable"
orange = ShoppingItem("orange", 10, true)
console.log(name) // OOPS
orange.report() // -> TypeError: Cannot read property 'report' of
undefined
```

Principal risk with using **new** and **Constructor functions** is that “new” is forgotten and left out. That causes the **this variable** in the function scope to point at the global scope rather than at the object in the function context. Any global variable having the name of the properties set in this will then be overwritten. Probably causing hard to detect and serious bugs!

JS TS

Methods: closure vs this

```
const newStack_closure = () => {
   const items = []

   return {
       depth: () => items.length,
       top: () => items[0],
       push: newTop => {
           items.unshift(newTop)
       },
       pop: ()=> {
           items.shift()
       }
   }
}
```

Object literal returned from function. Using fat-arrow methods you cannot use “this” but you can use closures

```
const newStack_this = ()=>{

   return {
       items:[],
       // depth: function() ...
       depth() {
           return this.items.length
       },
       // top: function() ... etc, etc
       top() {
           return this.items[0]
       },
       push(newTop) {
           this.items.unshift(newTop)
       },
       pop() {
           this.items.shift()
       }
   }
}
```

Object literal returned from function. Using classical functions or method notation you can use “this” which refers to the object

```
const StackPrototype = function(){
   this.items = []
}
StackPrototype.prototype = {
   depth: function() {
       return this.items.length
   },
   top: function()  {
       return this.items[0]
   },
   push: function (newTop) {
       this.items.unshift(newTop)
   },
   pop: function() {
       this.items.shift()
   }
}
```

Classic style Constructor function with “this” referring to object created by using “new” function invocation

JS TS

Destructuring assignment

The **destructuring assignment** syntax is a JavaScript expression that makes it possible to unpack values from arrays, or properties from objects, into distinct variables.

It works with a simple form of pattern matching, which is the inverse of the array and object literal syntax. Instead of a structure with values in a array or object notation on the right-hand side of an assignment, **destructuring works with putting a structure with variables in an array or object notation on the *left-hand side* of an assignment**.

For example: array destructuring

```
[a, b] = [10, 20];
console.log(a); // -> 10
console.log(b); // -> 20
```

The **rest or spread operator** can be used to capture the tail of an array.

```
[a, b, ...rest] = [10, 20, 30, 40, 50];
console.log(a); // -> 10
console.log(b); // -> 20
console.log(rest); // -> [30, 40, 50]
```

For example: object destructuring

```
({ a, b } = { a: 10, b: 20 });
console.log(a); // 10
console.log(b); // 20
```

The parentheses (...) around the assignment statement are required when using object literal destructuring assignment without a declaration.

{a, b} = {a: 1, b: 2} is not valid stand-alone syntax, as the {a, b} on the left-hand side is considered a block and not an object literal.

However, `({a, b} = {a: 1, b: 2})` is valid, as is `const {a, b} = {a: 1, b: 2}`

Note: The (...) expression needs to be preceded by a semicolon or it may be used to execute a function on the previous line(!)

The **rest or spread operator** can be used to capture the remaining properties in a object destructuring.

```
({a, b, ...rest} = {a: 10, b: 20, c: 30, d: 40})
console.log(rest) // -> {c: 30, d: 40}
```

JS TS

Modules

TypeScript and ECMAScript 201x have a concept of **modules**. Any file containing a top-level import or export is considered a module.

Modules are executed within their own scope, not in the global scope; this means that variables, functions, classes, etc. declared in a module are not visible outside the module unless they are explicitly **exported** using one of the **export forms**.

Conversely, to consume an export (variables, etc.) from a different module, it has to be imported using one of the **import forms** (see: Import Statement)

Examples can be found on the right:

```
export function hello() {
   return 'Hello World!'
}
export function hola(){
   return 'Hola mundo!'
}
```

```
export default class {
   public hello() {
       return 'Hello World!';
   }

   public hola() {
       return 'Hola mundo!';
   }
}
```

```
import { hello } from './app/services';
import { hola as spanishhello } from './app/services';
import * as greetings from './app/services';

import Msg from './app/defaultservices'

console.log(hello())
console.log(spanishhello())
console.log(greetings.hello())
let msg = new Msg()
console.log(msg.hola())
```

JS TS

Import statement

The static **import** statement is used to import bindings which are exported by a module. This allows the developer to import and use modules which are provided by many thousands of packages in the Javascript & TypeScript ecosystem, primarily with the npm package manager.

The definition of the import statement unfortunately is unwieldy and not that easy to fully grasp. It takes some trial and error to get it right. The MDN page is required reading The following basic forms are important to know:

Import an entire module contents

```
import * as myModule from '/modules/my-module.js';
myModule.doAllTheAmazingThings();
```

Import a single export from a module

```
import {myExport} from '/modules/my-module.js';
```

Import multiple exports from module

```
import {foo, bar} from '/modules/my-module.js';
```

Import an export with a more convenient alias

```
import {reallyReallyLongModuleExportName as shortName}
 from '/modules/my-module.js';
```

Rename multiple exports during import

```
import {
 reallyReallyLongModuleExportName as shortName,
 anotherLongModuleName as short
} from '/modules/my-module.js';
```

Import a module for its **side effects only.** This runs the module's global code, but doesn't actually import any values.

```
import '/modules/my-module.js';
```

It is possible to have a **default export** (whether it is an object, a function, a class, etc.). The import statement may then be used to import such defaults.

```
import myDefault from '/modules/my-module.js';
import myDefault, * as myModule from '/modules/my-module.js';
import myDefault, {foo, bar} from '/modules/my-module.js';
// specific, named imports
```

In node.js the support for ES6 import is only available through optional command line options and the standard is not fully implemented. Node.js uses its own “**require**” function (Common js). Using TypeScript allows skipping that problem as ES6 imports are compiled to “require” statements

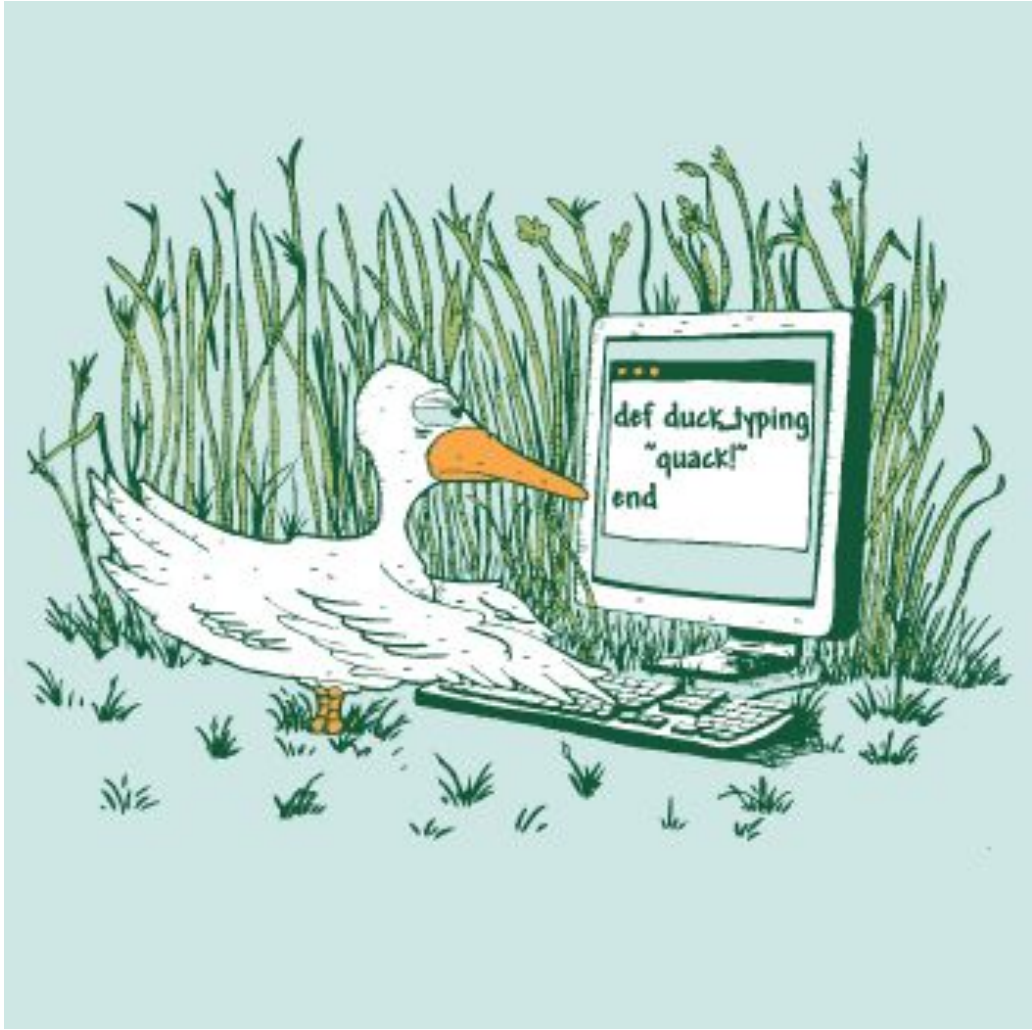
JS TS

Structural subtyping

One of TypeScript's core principles is that type-checking focuses on the shape that values have. This is sometimes called "***duck typing***" or "structural subtyping".

The concept name refers to the **duck test**, a joke of inductive reasoning attributed to the writer James Whitcomb Riley, that could be as it follows:

"When I see a bird that walks like a duck, swims like a duck and quacks like a duck, I call that bird a duck."



Interfaces

TS

Structural subtyping can be demonstrated by giving an object type definition to a function parameter. An object passed to the parameter can have more properties than the ones mentioned in the type declaration, but it must implement

```
// Any object passed to printLaberl needs to have the “label”
// property
let printLabel = (labeledObj: { label: string }) =>{
   console.log(labeledObj.label)
}

let goodObj = {size: 10, label: "Size 10 Object"}
printLabel(goodObj) // -> Size 10 Object

let badObj = {size: 10, text: "Size 10 Object"}
// Does not compile with error TS2345: Argument of type '{ size:
number; text: string; }' is not assignable to parameter of type '{
label: string; }'.
// Property 'label' is missing in type '{ size: number; text: string;
}' but required in type '{ label: string; }'.

printLabel(badObj)
```

TS

In TypeScript, **interfaces** fill the role of naming these structural types and are a powerful way of defining contracts within your code as well as contracts with code outside of your project.

```
interface LabeledValue {
   label: string
}
```

With the LabelledValue interface we define the requirement of the function. It still represents having a single property called label that is of type string.

```
function printLabel(labeledObj: LabeledValue) {
   console.log(labeledObj.label)
}

let goodObj = {size: 10, label: "Size 10 Object"}
printLabel(goodObj)

let badObj = {size: 10, text: "Size 10 Object"};
// Does not compile with error TS2345: Argument of type '{ size:
number; text: string; }' is not assignable to parameter of type
'LabeledValue'.
//  Property 'label' is missing in type '{ size: number; text:
string; }' but required in type 'LabeledValue'.
printLabel(badObj)
```

JS TS

Interfaces extra

TS

Optional properties

Not always we will need all the properties of an interface. With the symbol ? we represent what is optional.

```
interface SquareConfig {
    color?: string
    width?: number
}
```

Read-only properties

We can define a read-only interface with readonly:

```
interface Point {
    readonly x: number
    readonly y: number
}
let p1: Point = { x: 10, y: 20 }
p1.x = 5 // error TS2540: Cannot assign to 'x' because it is a read-only property.
```

TS

Interfaces can also define a function. This ensures the function signature.

```
// Using Point Interface from the left column
interface SetPoint {
   (x: number, y: number) : Point
}

let setPoint: SetPoint = (x, y)=> ({x, y})
let p1 = setPoint(10, 20)
console.log(p1) // => { x: 10, y: 20 }
```

Note: in current versions of TypeScript the difference between a type alias with the **type keyword** and an Interface declaration is small and subtle.

The problem is that at the moment of writing this, the official documentation not correctly reflect the latest development in the language. See: Typescript: Interfaces vs Types for more information

JS TS

Advanced Use Case: Union types recursive definition

TS

Using union types and Type Guards allow for some advanced use cases. In this example a **recursive type definition** is used. That means that a data structure is **defined in terms of itself.** In this way it is possible to define a matrix of "n" dimension. Note that this works really well in TypeScript - it is impossible to pass a type incompatible with `Matrix<T>` to the function `matrix_map`, but that when using the function from JavaScript, this protection no longer exist.

```
import {Matrix, matrix_map} from "./matrix"

let matrix3d: Matrix<number> = [[[5, 9, -1], [100, 2],
[-10, -8, 56]],                          [[],[],[]],
                                 [],
                                 [[50, 90, -100], [-90,
200], [-150, -8, 569]]]

function flooratzero(e: number) :  number {
  if (e < 0)
      return 0
  else
      return e
}
```

TS

```
export type Matrix<T> = Array<T> | Array<Matrix<T>>
export interface Celloperation<T> {
   (_: T): T
}

export function matrix_map<T>(matrix: Matrix<T>, operation:
Celloperation<T>): Matrix<T> {
   if (matrix.length === 0) {
       return matrix
   }

   if (!Array.isArray(matrix[0])) {
       return (matrix as Array<T>).map(operation)
   }

   let new_matrix = []
   for (const e of matrix) {
       new_matrix.push(matrix_map(e as Matrix<T>, operation))
   }
   return new_matrix
}
```

JS TS

Make Illegal States Unrepresentable

> **Making illegal states unrepresentable is** all about statically proving that all runtime values (without exception) correspond to valid objects in the business domain. The effect of this technique on eliminating meaningless runtime states is astounding and cannot be overstated.
>
> — John A De Goes (@jdegoes) January 28, 2019

In programming and science, it's often impossible to prove that something is *correct*, but it's reasonable to prove that something is *correct enough*.

With types, we can achieve that *correct enough*, and that makes our code much easier to reason about. And that gives us confidence to know that our business rules are being respected.

In the previous example the compiler enforces the usage of a type `Matrix<T>` parametrized for the type `number`. The type system is powerful enough to support n levels of sub matrices. It is therefore and flexible and robust enough to support most use cases in a safe way.

JS TS

TypeScript Advanced Types

The TypeScript type system offers many advanced type system features such as generic types, mapped types, and lookup types. That is far beyond the scope of this tutorial. However, any usage of TypeScript with the intent to fully utilize its power requires understanding of the essence of the type system.

For more info see:

TypeScript magic types - Let's dive into TS type system

TypeScript Handbook - Advanced Types

JS TS

Intermezzo - "classical" Object oriented programming

The more classic approach to Object Oriented Programming introduced in ES6 ("class" keyword etc) is a double edged sword. On the one hand "class based OOP" is something everyone learns and making the syntax better is a good thing. Besides it's an optional feature and there are other ways to create objects like factory functions. Using it for limited purposes is fine.

However….

The concept of "Class" doesn't exist in JavaScript. It's underlying mechanism (called "Prototype-based programming") is still how everythings works under the hood. So to truly understand the run-time model of JavaScript it is needed to get to know how prototypes work. And it is well recognozed that the concept of classes with "deep" inheritance hierachies make things brittle. Prototypes are better and very flexible. And finally using "class" guides people away from the simplicity and power of functional programming

JS TS

Classes

From ES6 onwards JavaScript developers are able to use a more classic approach to Object Oriented Programming which resembles other programming languages.

In the sample there is a `Greeter` class with a `greeting` property, a **constructor** and a `greet` method.

With `this.` access to the class members is provided

In order to construct an instance of the class the reserved word `new` is used.

```
class Greeter {
    greeting: string
    constructor(message: string) {
        this.greeting = message
    }
    greet() {
        return "Hello, " + this.greeting
    }
}

let greeter = new Greeter("world")
```

JS TS

Inheritance

The **extends** **keyword** is used in *class declarations* or *class expressions* to create a class as a child of another class.

The **super** **keyword** allows calling the parent object that is being inherited. It is good advice to avoid this as this can cause an even tighter coupling between objects, but there are occasions where it is appropriate to use.

In the example on the right case, it is be used in the constructor to assign to the super constructor.

If a constructor is not defined on a child class the super class constructor will be invoked by default.

```
class Animal {
    name: string;
    constructor(theName: string) { this.name = theName; }
    move(distanceInMeters: number = 0) {
        console.log(`${this.name} moved ${distanceInMeters}m.`)
    }
}

class Snake extends Animal {
    constructor(name: string) { super(name) }
    move(distanceInMeters = 5) {
        console.log("Slithering...")
        super.move(distanceInMeters)
    }
}

class Horse extends Animal {
    constructor(name: string) { super(name); }
    move(distanceInMeters = 45) {
        console.log("Galloping...")
        super.move(distanceInMeters)
    }
}
```

JS TS

Properties and getters and setters

With the JavaScript field declaration syntax, class definitions become more self-documenting, and the fields are always present.

As seen above, the fields can be declared with or without a default value. In TypeScript with strict options enabled, declaring field variables is mandatory.

A **getter** is a method that gets the value of a specific property. A **setter** is a method that sets the value of a specific property. Getters and setters can be defined on any predefined core object or user-defined object that supports the addition of new properties.

The syntax for defining getters and setters uses the object literal syntax.

```
class Person {
   _name = ""
   constructor(name) {
     this._name = name
   }
    get name() {
     return this._name.toUpperCase()
   }
    set name(newName) {
     this._name = newName // validation could be checked
here such as only allowing non numerical values
   }
    walk() {
     console.log(this._name + ' is walking.')
   }
 }
  let bob = new Person('Bob')
 console.log(bob.name) // Outputs 'BOB'
```

JS TS

Public, private and protected modifiers

TS

By default, in TypeScript every member is marked as **public**. Every member without any modifier behaves as if marked by public.

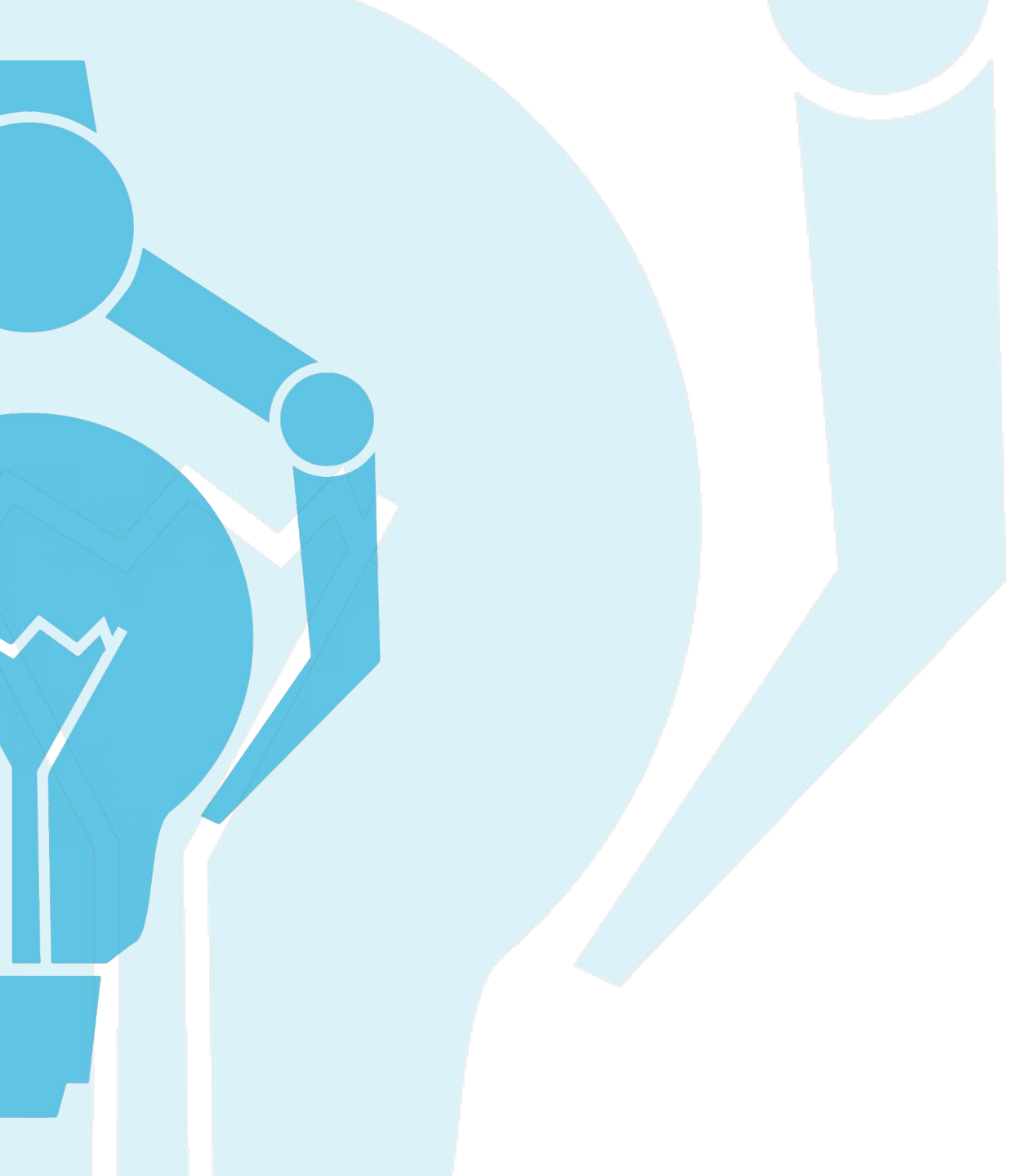
When a member is marked as **private** it is not possible access to it outside the class.

When a member is marked as **protected** it is only possible to access it from members from the class or its subclasses.

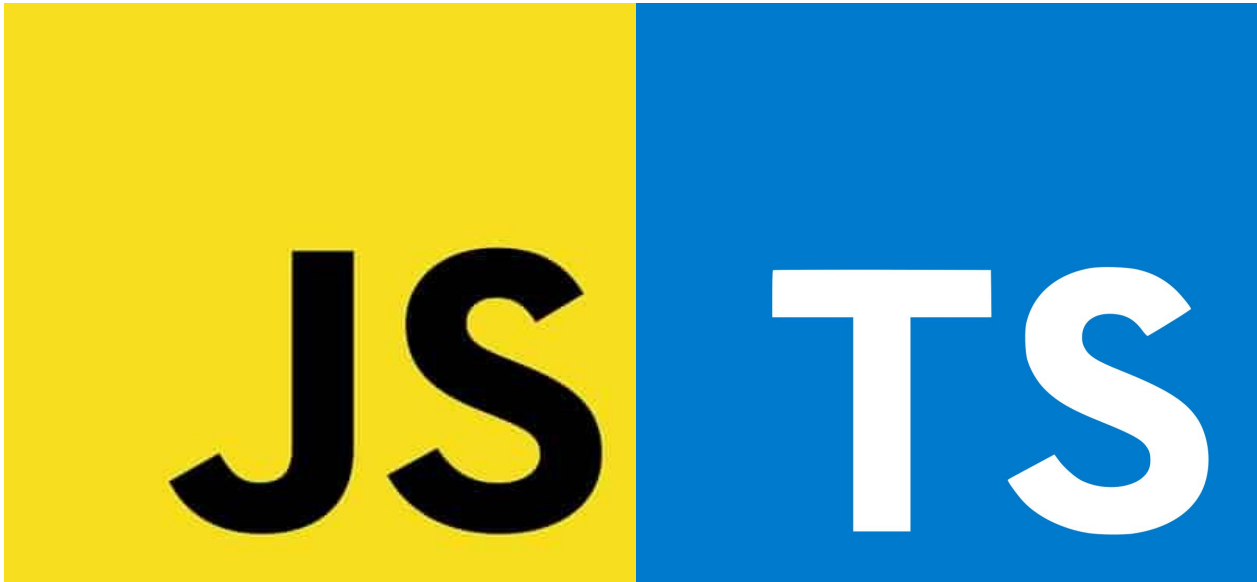
TS

```
class Person2 {
   private _name : string
   constructor(name: string) {
     this._name = name;
   }
   public get name() {
     return this._name.toUpperCase();
   }
   public set name(newName) {
     this._name = newName; // validation could be checked
here such as only allowing non numerical values
   }
   public walk() {
     console.log(this._name + ' is walking.');
   }
 }
  let bob = new Person2('Bob');
 console.log(bob.name); // Outputs 'BOB'
```

JS TS



Digging Deeper



Asynchronous functions and callbacks

In traditional programming practice, most I/O operations happen synchronously. In the example on the right, the function readFileSync is a synchronous and will perform its task until it returns the content of the file.

This is fine for simple scripts but **it is an anti-pattern in node.js**. As node.js is single-threaded, the JavaScript interpreter cannot execute any code while readFileSync is executed. Not so much an issue for a throw away script, but fatal for larger applications, for example web api's, who need to respond to many simultaneous requests.

So nearly all I/O functions in the run-time library in both node.js as well as the web browser are **asynchronous**. Asynchronous I/O is a form of input/output processing that permits other processing to continue before the transmission has finished.

An asynchronous functions take a function - a so-called call-back function - which will be executed once the operations is finished. As the asynchronous library functions are not written in Javascript but in C, C++ or Rust (etc), they are executed by multiple threads from the thread-pool of node.js or the browser. It's only the JavaScript which is executed on a single thread!

```
import {writeFile, readFile, readFileSync} from 'fs'

const data = readFileSync("ironman.txt", 'utf8')
for(let s of data.split('\n')){
  console.log(s)
}

console.log("you see me first")
readFile("ironman.txt", 'utf8', (error, data)=> {
   if (error) throw error
   console.log("you see me last")
   for(let s of data.split('\n')){
       console.log(s)
   }
   if (error) throw error

   writeFile("ironman.out.txt", data, "utf8", (err)=>{
               if (error) throw error
               console.log("file written")
       })
})
console.log("you see me second")
```

JS TS

Promises

A Promise is an object representing the eventual completion or failure of an asynchronous operation.

Essentially, a promise is a returned object to which you attach callbacks, instead of passing callbacks into a function.

Principal methods are `then` which receives a callback to be performed when the Promise is done with its work, or `catch`, which receives a callback to be performed when an error has happened.

Because `.then()` always returns a new promise, it's possible to chain promises with precise control over how and where errors are handled. Promises allow you to mimic normal synchronous code's try/catch behavior.

Like synchronous code, chaining will result in a sequence that runs in serial. In other words, you can do:

```
fetch(url)
 .then(process)
 .then(save)
 .catch(handleErrors)
```

```
import {promisify} from 'util'
import {readFile, writeFile} from 'fs'

let prnerror = (error: any)=> {
  console.log(error)
}
console.log("you see me first")
let readFp = promisify(readFile)
let writeFp = promisify(writeFile)

readFp("ironman.txt", 'utf8').then((data)=> {
 console.log("you see me last")
 for(let s of data.split('\n')){
     console.log(s)
 }
 return writeFp("ironman.out.txt", data, "utf8")

})
.then(()=>{
 console.log("file written")
})
.catch(prnerror)
```

JS TS

Async / Await

Using **async await** makes it possible to use Promises in a reliable and safe way. This method prevents chances of any programming errors.

Writing asynchronous functions is really easy. Just write a function and add the async keyword to it like in the example on the right.

Asynchronous functions can use the await operator in their bodies. The await operator can be attached to any variable. If that variable is not a Promise, the value returned for the await operator is the same as the variable.

But if the variable is a Promise, then the execution of the function is paused until it is clear whether the Promise is going to be resolved or rejected.

If the Promise resolves, the value of the await operator is the resolved value of Promise, and if the variable is a promise that gets rejected, the await operator throws an error in the body of the async function which we can catch with try/catch constructs.

```
import {promisify} from 'util'
import {writeFile, readFile} from 'fs'

let readFp = promisify(readFile)
let writeFp = promisify(writeFile)

console.log("you see me first")
async function printIronMan() {
  try {
      let data = await readFp("ironman.txt", 'utf8')
       console.log("you see me last")
       for(let s of data.split('\n')){
           console.log(s)
       }
       await writeFp("ironman.out.txt", data, "utf8")
   } catch(err) {
       console.log(err)
   }
}
printIronMan()
console.log("you see me last")
```

JS TS

Iterators

In JavaScript an **iterator** is an object which defines a sequence and potentially a return value upon its termination.

Specifically, an iterator is any object which implements the **Iterator protocol** by having a next() method returns an object with two properties:

`value`
The next value in the iteration sequence.

done
This is true if the last value in the sequence has already been consumed. If value is present alongside done, it is the iterator's return value.

Once created, an iterator object can be iterated explicitly by repeatedly calling next(). Iterating over an iterator is said to consume the iterator, because it is generally only possible to do once. After a terminating value has been yielded additional calls to next() should simply continue to return `{done: true}`.

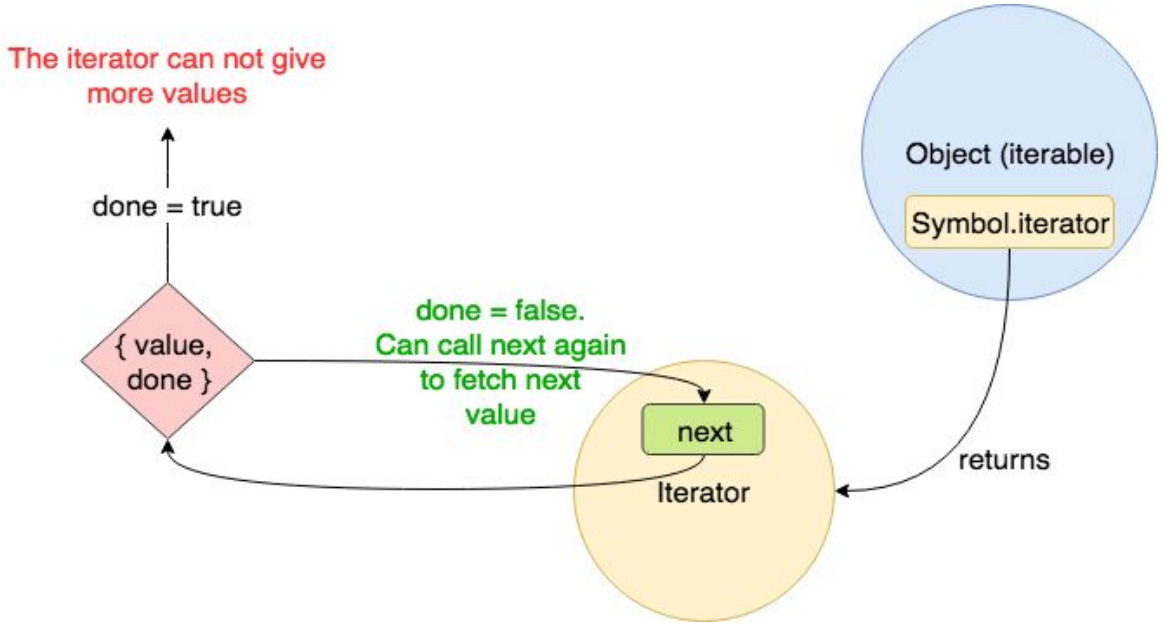
Iterators allow for ***lazy evaluation***, which means that an expression is only actually calculated on an as-needed basis. In the example on the right, lazy evaluation of the calculation of the fibonacci series saves on memory (no recursion) and memory (no need of saving of whole range)

```
function fibonacci_iterator(){
        return {
            n1: 0,
            n2: 1,
            next(): {value: number, done: boolean} {
                let res = this.n1
                this.n1 = this.n2
                this.n2 = res + this.n1 //fibonaci
                return {value: res, done: false}
            }
        }
}

let fibonaci_iter = (num: number)=>{
   let iterator = fibonacci_iterator()
   let fib = []
   for(let i = 0; i <= 20; i++){
       let {value, done} = iterator.next()
       fib.push(value)
   }
   return fib
}

console.log(fibonaci_iter(20)) //  // ->
[0,1,1,2,3,5,8,13,21,34,55,89,144,233,377,610,987,1597,2584,4181,6765]
```

JS TS

Iterables

As seen on the previous slide, an ***iterable*** is a data structure that wants to make its elements accessible to the public, i.e it is a pointer for traversing the elements of a data structure. In order to standardize the access to an iterator from ES6, the iteration protocol prescribes **implementing a method whose key is Symbol.iterator, written as [Symbol.iterator].** This is a zero-argument function that returns an object, conforming to the iterator protocol. The method is a **factory for *iterators***. That is, it will create *iterators*.



TS

```ts
import {take} from "./utils"

function fibonacci_iterable(){

    return {
        n1: 0,
        n2: 1,
        [Symbol.iterator](){
            return this
        },
        next(): {value: number, done: boolean} {
            let res = this.n1
            this.n1 = this.n2
            this.n2 = res + this.n1 //fibonaci
            return {value: res, done: false}
        }
    }
}
let i = 0
// the take utility function in the examples ("utils.ts") take the first n elements of any iterable and returns them in an array
console.log(take(fibonacci_iterable(), 21)) // _>
[0,1,1,2,3,5,8,13,21,34,55,89,144,233,377,610,987,1597,2584,4181,6765]
```

JS TS

Generators

A normal function cannot be stopped *before* it finishes its task i.e its last line is executed. It follows something called run-to-completion model.

The only way to exit the function is by returning from it, or throwing an error. If you call the function again, it will begin the execution from the top ***again***.

In contrast, a **generator** is a function (marked with a "*") that **can stop midway** using the **yield keyword** and then **continue *from where it stopped***.

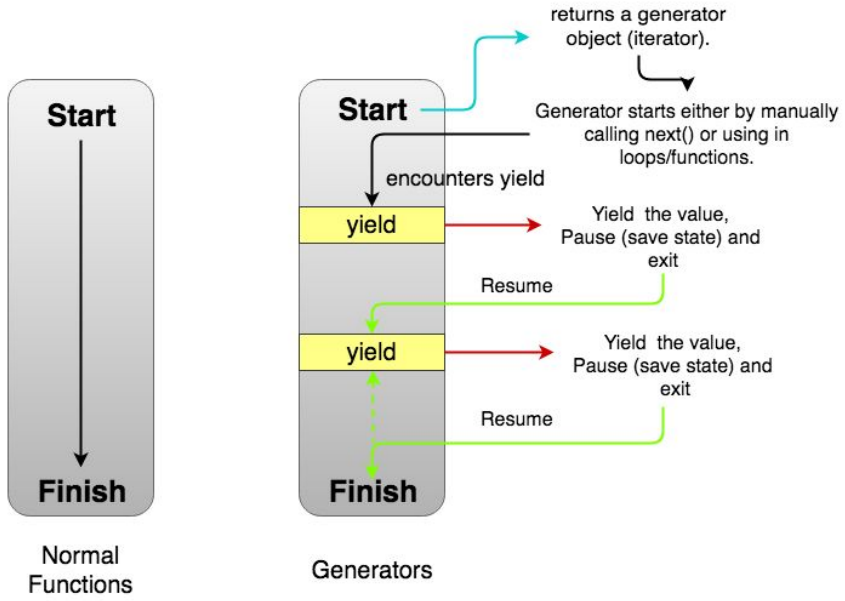
Each invocation of a generator function returns an object supporting the Iterable protocol, allowing it to produce an Iterator (for be used in for..of loops etc)

This allows generator functions to:

- simplify the task of writing iterators,
- produce a sequence of results instead of a single value, i.e you *generate* a series of values.

Note that each invocation of a generator function creates and returns a new instance of the iterable object. Which means that from the point of view of the innards of the generator function, local variables can be used to store state in between the different yield invocations, but that the local state is unique to the invocation of the generator function (like with a closure).

```
import {take} from "./utils"

function* fibonacci_gen() {
    yield 0
    yield 1
    let n1 = 0, n2 = 1
    while(true) {
        let res = n1 + n2 //fibonacci
        n1 = n2, n2 = res
        yield res
    }
}
// the take utility function in the examples ("utils.ts") take the first n
elements of any iterable and returns them in an array
console.log(take(fibonacci_gen(), 21)) // _>
[0,1,1,2,3,5,8,13,21,34,55,89,144,233,377,610,987,1597,2584,4181,6765]
```



Async Generators

ES6 recently introduced the concept of an async generator function. Now, JavaScript and TypeScript have 6 distinct types of functions:

- Normal functions `function() {}`
- Arrow functions `() => {}`
- Async functions `async function() {}`
- Async arrow functions `async () => {}`
- Generator functions `function*() {}`
- Async generator functions `async function*() {}`

Async generator functions are special because both await and yield can be used in an async generator function. Async generator functions differ from async functions and generator functions in that they don't return a promise or an iterator, but rather an async iterator. Async iterator can be thought of as an iterator who's `next()` function always returns a promise.

Async generator functions are a great solution for lazy evaluation of asynchronous streams. They are generic and, above all, composable, and therefore a great solution for many challenges to be encountered in node.js programming.

TS

```ts
import * as fs from "fs"
import  {join} from 'path'

async function * getFiles(path: string):
AsyncGenerator<[string, fs.Stats]>{
  const dir = await fs.promises.opendir(path);
 for await (const dirent of dir) {
     const f = join(path, dirent.name)
     yield [f, await fs.promises.stat(f)]
 }
}

async function main(){
  for await (const [name, stat] of
getFiles("/home/iwk/Downloads")){
     console.log(name, stat.isFile(), stat.size)
 }
}

main()
```

JS TS

Decorators

A **Decorator** is a special kind of declaration that can be attached to a *class declaration, method, accessor, property, or parameter*. Decorators use the form **@expression**, where expression must evaluate to a function that will be called at runtime with information about the decorated declaration.

It is a form of **meta-programming** which follows the model as set by *Decorators* in Python, i.e. they are actively executing functions, rather than the model followed by *Annotations* in Java (objects as passive meta-data)

The application of Decorators is an “experimental” feature of EcmaScript / TypeScript. It may change but it is unlikely to disappear (as many important libraries like Angular are already depending on it).

Decorators are an advanced topic and extremely powerful but also very “magical”, complex and therefore fraught with danger. Use with care.

TS

```
function log(target: any, key: string, value: any) {
   return {
     value(...args: any[]) {
        const result = value.value.apply(this, args)
        console.log(`method: '${key}' called with arguments
'${args}' and with result: '${result}'`)
         return result
      }
    }
 }
  class Demo {
   @log
   public say(...args: string[]) {
      console.log("Inside say with arguments: ", args)
      return 100
   }
 }
  let d = new Demo()
 d.say("Booh", "Lala")
 d.say("Bah")
```

JS TS

Decorator types (examples)

There are 4 types of objects which can decorated in ECMAScript2016 and Typescript: constructors, methods, properties and parameters.

The example on the previous slide shows a method decorator which takes 3 arguments:

- `target` - the method being decorated.
- `key` - the name of the method being decorated.
- `value` - a property descriptor of the given property if it exists on the object, undefined otherwise. The property descriptor is obtained by invoking the Object.getOwnPropertyDescriptor() function.

The example on the right shows a **class decorator.** This a function that accepts a **constructor function** and returns a constructor function. It should be noted that the class keyword in ES6 is nothing more than syntactic sugar for the creation of such a function. In order to be be able to successfully develop Decorators, one must:

- understand which JavaScript code is being generated by the ES6 or TypeScript compiler
- understand the run-time model of ES6 (how classes work, generators, iterators etc etc)

TS

```ts
import {take} from "./utils"

function Iterable(konstructor: Function){
   konstructor.prototype[Symbol.iterator] = function(){
       return this.__iter__()
   }
}

@Iterable //make any object iterable as long as it has a *__iter__
generator method (naming convention from Python)
class Test{
   private a:string; private b:string; private c:string

   constructor(a: string, b : string, c: string){
       this.a = a;this.b = b; this.c = c
   }
   * __iter__(){
       yield this.a
       yield this.b
       yield this.b
       yield this.c
       yield this.c
       yield this.c
   }
}
console.log(take(new Test("a", "b", "c") as any , 6))
```

JS TS

Tying it all together

Async Generators as (async) streams

see files:
`async-streams.ts` & `demo-streams.ts`

TS

Make a function getFiles which returns an AsyncGenerator. This allows the usage of asynchronous functions to obtain data and return the data as an (asynchronous) Iterable. Basically implementing asynchronous stream.

```
async function * getFiles(path: string): AsyncGenerator<[string, fs.Stats]>{
  const dir = await fs.promises.opendir(path);
 for await (const dirent of dir) {
     const f = join(path, dirent.name)
     yield [f, await fs.promises.stat(f)]
     if (dirent.isDirectory()){
       yield* getFiles(f)
     }
 }
}
```

Usage would be with a global async function:

```
async function main(rootDir: string){
    for await (const [name, stat] of getFiles(rootDir)){
       console.log(name, stat.isFile(), stat.size)
   }
}
main("/home/iwk/Music")
```

Now likewise, write the `take` and `map` functions, You should know the definition of the latter. `take(n)` should pick the first n items from the stream and ignore the rest.

TS

Now suppose that the stream can be filtered akin to Array.filter. This could lead to usage like:

```
   for await (const [name, stat] of filter(getFiles(rootDir), isDir)){
       console.log(name, stat.isFile(), stat.size)
   }
```

This can be implementing by creating a flter function encapsulating an existing stream:

```
    export async function* filter<T>(source: AsyncIterable<T>, predicate: (value: T) =>
boolean): AsyncIterable<T> {
   const It = source[Symbol.asyncIterator]

   //See
https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Function
/call

   let it: AsyncIterator<T> = It.call(source)
   while (true) {
       const { done, value } = await it.next()
       if (done) {
           return
       }
       if (predicate(value)) {
           yield value
       }
   }
}
```

JS TS

Chainable stream operators

see files:
`async-streams.ts` & `demo-streams.ts`

TS

Now consider a async_stream factory function taking an existing asynchronous stream and encapsulating it in an object which allows chaining functions like map, filter, take etc:

```
for await (const [name, stat] of
async_stream(getFiles(rootDir)).filter(isDir).map(upper).ta
ke(5) ){
        console.log(name, stat.isFile(), stat.size)
   }
```

The interface of AsyncStream is at follows:

```
export interface AsyncStream<T> {
   filter: (predicate: (value: T) => boolean) =>
AsyncStream<T>
   map: (mapper: (value: T) => T) => AsyncStream<T>

   all: () => AsyncIterable<T>
   take: (n: number | bigint) => AsyncIterable<T>
}
```

TS

The AsyncStream object allows for encapsulating of the original stream postponing execution of the iterator. Evaluation of the stream is done by calling the `all` or `take` methods:

```
export function async_stream<T>(source: AsyncIterable<T>):
AsyncStream<T> {
   let _source_ = source
   return {
       filter(condition) {
           _source_ = filter(_source_, condition)
           return this
       },
       map(mapper) {
           _source_ = map(_source_, mapper)
           return this
       },
       all() {
           return _source_
       },
       take(num) {
           _source_ = take(_source_, num)
           return _source_
       },
   }
}
```

JS TS

The last stretch

let **and** const **are the new var**

In earlier versions of JavaScript , a variable can be declared with the **var** keyword after it has been used. In other words; a variable can be used before it has been declared. This behaviour is called *"hoisting"* and is as if all variables are as if defined at the beginning of the function. It is said that these variables have **function scope**.

This causes all kinds of subtle errors and counterintuitive behaviours. WIth **let** and **const** a more conventional block scope is used for variables (no more "*hoisting*") which allows for more predictable and more advanced behaviour.

TS

```
//compiles fine
function functionscope(){

   for(i=0;i < 10; i++){
      console.log(i)
   }
   if (true){
     var i = 10000
   }
   console.log(i)
}
 //gives error TS2304: Cannot find name 'i'.
function blockscope(){
   //let i;

   for(i=0;i < 10; i++){
      console.log(i)
   }
   if (true){
     let i = 100000
   }
   console.log(i)
}
 functionscope()
blockscope()
```

JS TS

let **is the new** var**; fixes problem with closures**

Using **let** fixes a longstanding problem with **closures**

block scoped references captured in a closure will maintain the value at the moment of capture

That is **NOT** the case with **var** (the reference will point to the last value it was set to)

```
for(var j= 0; j < 10; j++){
   setTimeout(()=> console.log(`setTimeout with var: ${j}`),100)
}
// writes 10 times -> setTimeout with var: 10

for(let k= 0; k < 10; k++){
   setTimeout(()=> console.log(`setTimeout with let: ${k}`),100)
}
// writes  setTimeout with let: 2, setTimeout with let: 1, setTimeout
with let: 2 etc
```

JS TS

How constant is const

Use const when you want to denote immutability of a reference (the “variable”)

Notice that objects and arrays are not const. Only their references.

Use *Object.freeze* or Immutable.js (a library) for this.

No compile time, type level, support

TS

```
const co = 100
//ES6 gives error: Assignment to constant variable
//TS  gives error TS2540: Cannot assign to 'co'
because it is a constant or a read-only property.
//co = 1000

const co2 = {a: 100}
co2.a = 100000

console.log(co2)
Object.freeze(co2)
//RUN-TIME ERROR: TypeError: Cannot assign to read
only property 'a' of object '#<Object>'
co2.a = 0
```

JS TS

TypeScript´s Type System

The type system in TypeScript is designed to be **optional** so that your JavaScript is TypeScript.

TypeScript does not stop generating JavaScript code in the presence of type errors, allowing you to progressively update your JS to TS.

But that does mean that TypeScript is not “safe”

But `--noEmitOnError` flag prevents generating JavaScript code when there are Type errors

Therefore: **Types are annotations**; they don't “exist” run-time (**type erasure**)

TS

```ts
let foo = {};

foo.bar = 123; // Error: property 'bar' does not exist on `{}`
console.log(foo.bar)

foo.baz = "ABC" // Error: property 'baz' does not exist on `{}`
console.log(foo.baz)
```

JS TS

--strict (since TypeScript 2.3)

flag	Meaning
--alwaysStrict	Parse in strict mode and emit "use strict" for each source file
--noImplicitThis	Raise error on this expressions with an implied any type.
--strictNullChecks	Strict null checking mode, the null and undefined values are not in the domain of every type and are only assignable to themselves and any (the one exception being that undefined is also assignable to void).
--noImplicitAny	Raise error on expressions and declarations with an implied any type.
--noImplicitReturns	Report error when not all code paths in function return a value.
	… and more …

TypeScript supports a whole series of “compiler flags” which can be set in tsconfig.json. **From version 2.3 on, all flags related with “strict” compilation can be set in with the –strict flag.** In 2.2 and below the flags need to be set individually. **Setting –strict is considered best practice and highly recommended**

JS TS

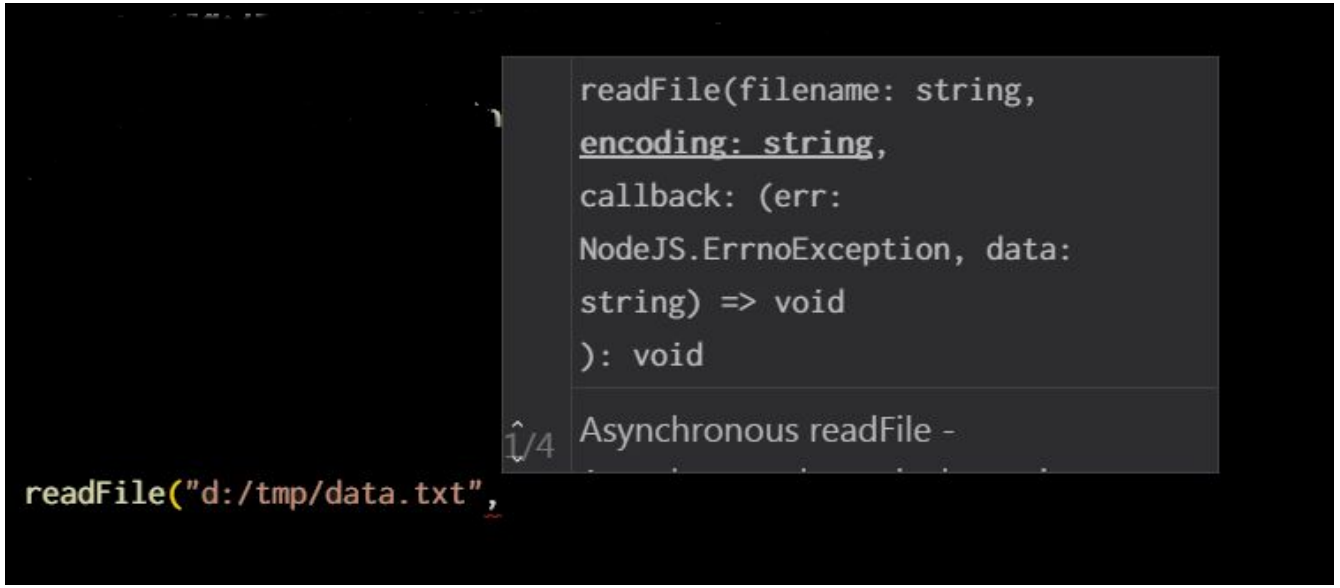
Contextual inference

Type inference also works in "the other direction" in some cases in TypeScript. Meaning: not from right-to left, impacting the variable declarations on the left-hand side.

It also work in callbacks. This is known as "contextual typing".

Contextual typing occurs when the type of an expression is implied by its location.

```
import { readFile } from "fs"


readFile("d:/tmp/data.txt", function(err,buf){

    console.log(buf.toString())
})
```



JS TS

Best practice

DRY and don't be verbose: use type inference

```
let s1 = "lala"
/*NOT*/
let s2: string = "lala"
```

Avoid the *implicit* usage of *any*

```
--noImplicitAny
```

Type the parameters of your own non call-back functions.
Optionally type the return value of the function

```
function translate(text:string) :string {
   return `¿Eh? ¿Que significue '${text}'?`
}
function translate2(text:string)  {
   return `¿Eh? ¿Que significue '${text}'?`
}
let s = translate("Blah")
let s2 = translate2("Blah")
```

JS TS

Type inference with assignment & declarations

There is a difference in the behaviour of type inference between **declarations** (“let”) with their corresponding *initializers* and **assignment.**

In the example assignment works but the variable initializer must exactly match the structure of Named, i.e. the Object literal may only specify known properties, and 'surname' does not exist in type 'Named'

TS

```ts
interface Named {
    name: string;
}
let p: Named;
 // also with anonymous objects
let o = { name: "Olaf" , surname: "Leifson"}
p = o;
console.log(p)

// compile error
//let n : Named = { name: "Olaf" , surname:
"Leifson"}
```

JS TS

Links

The Modern JavaScript Tutorial

https://mariusschulz.com/blog

ES6 Tutorial - Getting Started Quickly with the New Version of

ECMAScript 6: New Features: Overview and Comparison

Online Interactive JavaScript (JS)

JS TS

Capgemini

People matter, results count.

About Capgemini

A global leader in consulting, technology services and digital transformation, Capgemini is at the forefront of innovation to address the entire breadth of clients' opportunities in the evolving world of cloud, digital and platforms. Building on its strong 50-year heritage and deep industry-specific expertise, Capgemini enables organizations to realize their business ambitions through an array of services from strategy to operations. Capgemini is driven by the conviction that the business value of technology comes from and through people. It is a multicultural company of 200,000 team members in over 40 countries. The Group reported 2016 global revenues of EUR 12.5 billion.

Learn more about us at

www.capgemini.com