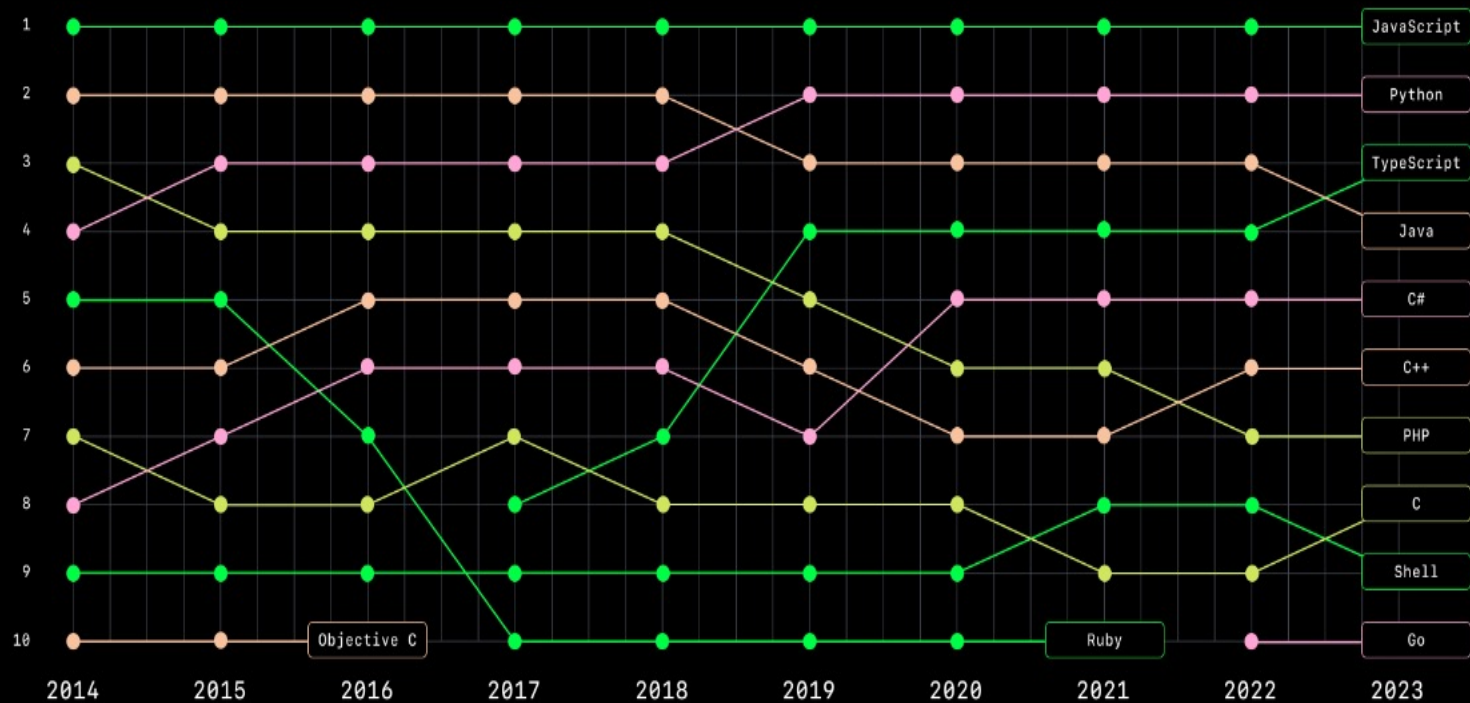


Typescript

Overview.

- **Open source language, developed by Microsoft (2010-12).**
 - *Anders Hejlsberg*, the creator of C# and Turbo Pascal
- **Based on ECMAScript 4 and 6.**
- **A superset of Javascript.**
- **We still write JS, but augmented by the class-based OOP of ES6, and the structural type system of ES4.**
- **TS is compiled to regular JS and runs in any browser, any host, and OS.**
- **“... and one thing TS got right: local type inference”** Bernard Eich
- **“What impressed me the most is what TS doesn't do; it does not output type checking into your JS code”** Nicholas C Zakas .
- **TS is a a language for application-scale JavaScript development.**

Top 10 programming languages on GitHub



File Extensions.

- **.ts - source file extension.**
- **.d.ts - declaration files.**
- **Declaration source files:**
 - **Provide type definitions, separate from the source code.**
 - **Analogous to header files in C/C++.**
 - **Also used to describe the exported virtual TypeScript types of a third-party JavaScript library, allowing TS developers to consume it.**
 - **Tooling - Gives type safety, intellisense and compile errors.**

Types

- **Primitive Types:**
 - number – represents integers. Floats, doubles.
 - booleans
 - string – single or double quote.
 - null.
 - undefined.
- **Object Types:**
 - Class, module, interface and literal types.
 - Supports typed arrays.
- **The Any type:**
 - All types are subtypes of a single top type called the Any type.
 - Represents any JavaScript value with no constraints.

Type Annotations.

- (Optional) static typing.
- Lightweight way to record the intended contract of a variable or function.
- Applied using a post-fix syntax.
 - e.g. `let me : string = “Diarmuid O; Connor”`
- Typed Array:
`let myNums: number[] = [1, 2, 3, 5];`
- Can also apply to function signature:

```
function add(a: number, b: number) {  
  return a + b;  
}
```

Classes

- **Support for ECMAScript 6 alike classes.**
- **public or private member accessibility.**
- **Parameter property declarations via constructor.**
- **Supports single-parent inheritance.**
- **Derived classes make use of super calls to parent methods..**

```
class Animal {  
    constructor(public name) { }  
    move(meters) {  
        alert(this.name + " moved " + meters + "m.");  
    }  
}  
  
class Snake extends Animal {  
    move() {  
        alert("Slithering...");  
        super.move(5);  
    }  
}  
  
class Horse extends Animal {  
    move() {  
        alert("Galloping...");  
        super.move(45);  
    }  
}
```

Interfaces.

- **Designed for development tooling support only.**
- **No output when compiled to JavaScript.**
- **Open for extension (may declare across multiple files).**
- **Supports implementing multiple interfaces.**

```
interface Drivable {
    start(): void;
    drive(distance: number): void;
    getPosition(): number;
}

class Car implements Drivable {
    private isRunning: bool = false;
    private distanceFromStart: number;

    public start(): void {
        this.isRunning = true;
    }
    public drive(distance: number): void {
        if (this.isRunning) {
            this.distanceFromStart += distance;
        }
    }
    public getPosition(): number {
        return this.distanceFromStart;
    }
}
```


Interface Data Types.

- An interface data type tells the TypeScript compiler about the property names an object can have and their corresponding value types. Therefore, interface is a type and is an abstract type since it is composed of primitive types.

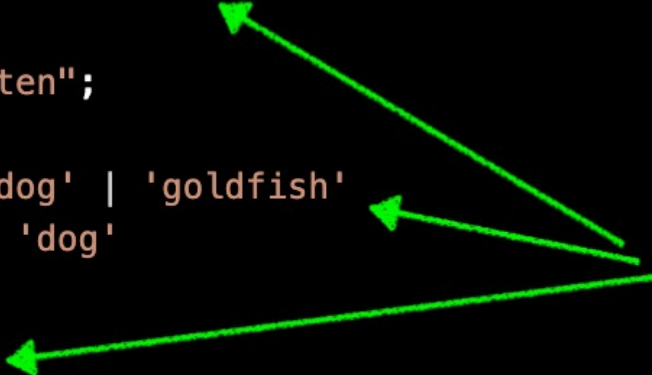
```
interface Person {  
    first: string;  
    last: string;  
}
```

```
const me: Person = {  
    first: "diarmuid",  
    last: "o connor",  
};
```

Type Aliases.

- **Type aliases create a new name for a type. Type aliases are sometimes similar to interface data types, but can name primitives, unions, tuples, and any other types.**

```
11  type alphaNumeric = string | number;
12  let num : alphaNumeric = 10;
13  const str : alphaNumeric = "ten";
14
15  type PetCategory = 'cat' | 'dog' | 'goldfish'
16  let petXType : PetCategory = 'dog'
17
18  type Point = {
19      x: number;
20      y: number;
21  };
22
23  let pt : Point = {x: 10, y: 20};
24
```



Type Inference.

- TS compiler can infer the types of variables based on their values.

```
117 |  
118 |  
119 | let aString = "hello"; // cmd-k cmd-i  
120 |  
128 | const friends: Person[] = [  
129 |   { first: "bob", last: "sullivan" },  
130 |   { first: "kyle", last: "dwyer" },  
131 |   { first: "jane", last: "smith" },  
132 | ];  
133 | const sFriends = friends.filter((friend) => friend.last.startsWith("s"));  
134 |
```

Inferred

- Inferencing increases developer productivity.

Functions.

- Declaring the types in a function's signature.

```
4 function addNumbers(a: number, b: number): number {  
5     return a + b;  
6 }
```

- Compiler can often infer the return type.

```
8  
9 TS i function addtoNumberArray(nums: number[], inc: number): number[]  
10 export function addtoNumberArray(nums: number[], inc: number) { You, 8 n  
11     const newNums = nums.map((num) => num + inc);  
12     return newNums;  
13 }
```

(Review) JavaScript functions.

- **Fundamental unit of encapsulation for logic (or BEHAVIOUR).**

- **Function syntax:**

- **ES5:**

- **Function declarations.**

function myName(parameters) { Body }

- **Function expressions.**

const myName = function(parameters) { Body }

- **Hoisting – all function declarations moved to the top of the current scope at runtime.**

- **ES6:**

- **Arrow functions.**

- **Shorthand version.**

- **Anonymous functions.**

(Review) Arrow functions

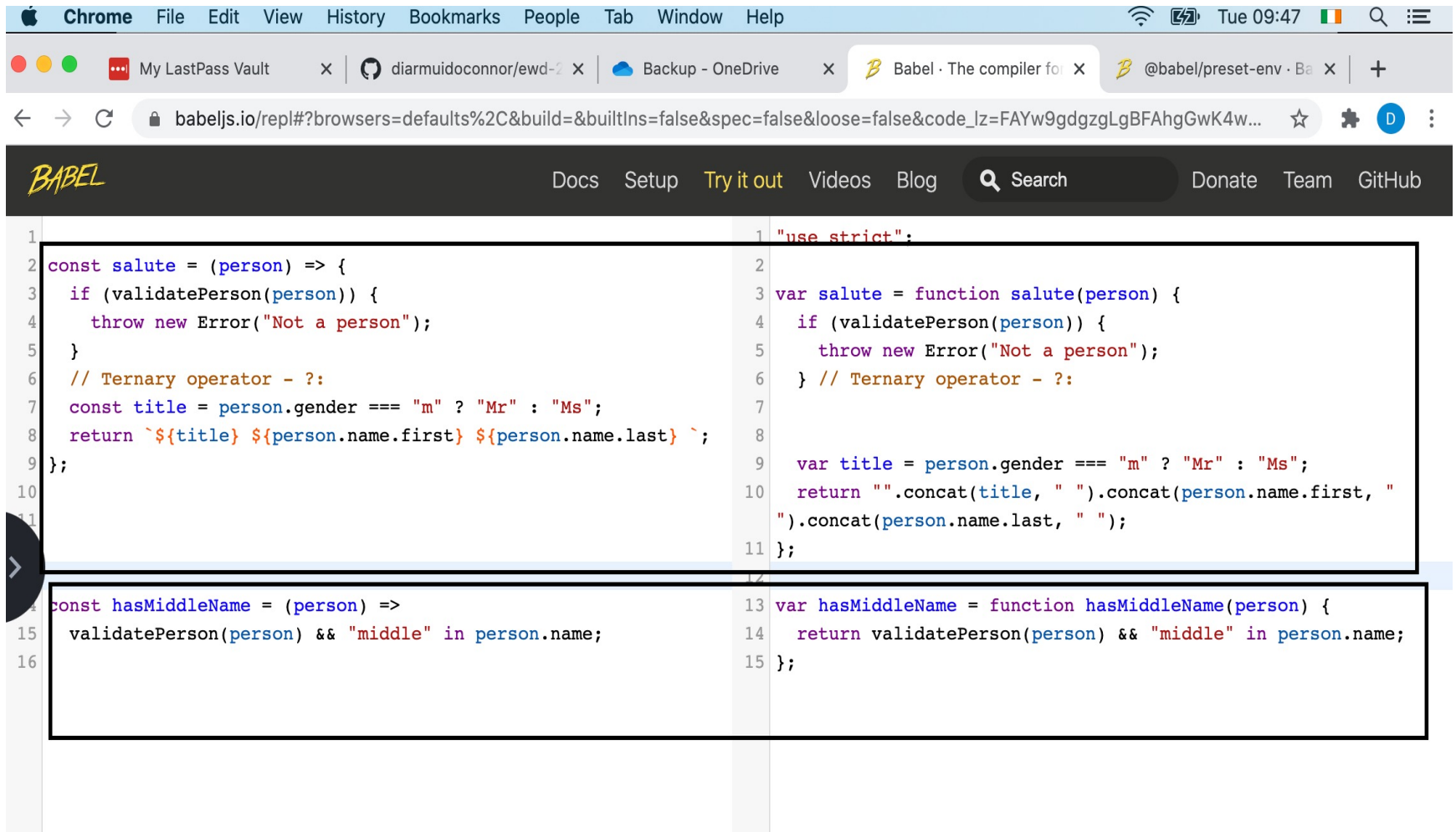
- **A cleaner syntax for creating functions.**

const myName = (parameters) => { Body }

- **The => (arrow) separates the function body from its parameters.**
- **Enclose the body with curly braces, { }.**
 - Unless it's a single expression (optional).
- **Enclose parameter list with parentheses, (...).**
 - Unless it's a single parameter (optional).
- **Omit the return token when it's a single-expression body (optional).**
- **Shorthand form, e.g.**

const isLeap = year => ((year % 4 === 0) && (year % 100 !== 0)) || (year % 400 === 0)

Arrow functions – ES6 → ES5



The screenshot shows the Babel REPL interface in a Chrome browser. The browser's address bar displays the URL `babeljs.io/repl#?browsers=defaults%2C&build=&builtIns=false&spec=false&loose=false&code_lz=FAYw9gdgzgLGbFAhgGwK4w...`. The Babel logo is in the top left, and navigation links for Docs, Setup, Try it out, Videos, Blog, Search, Donate, Team, and GitHub are in the top right. The main area is split into two panels: the left panel shows the input ES6 code, and the right panel shows the output ES5 code. The ES6 code defines a `salute` function using an arrow function and a ternary operator for a title, and a `hasMiddleName` function. The ES5 code shows the equivalent using `function` declarations and string concatenation.

```
1 const salute = (person) => {  
2   if (validatePerson(person)) {  
3     throw new Error("Not a person");  
4   }  
5   // Ternary operator - ?:  
6   const title = person.gender === "m" ? "Mr" : "Ms";  
7   return `${title} ${person.name.first} ${person.name.last}`;  
8 };  
9  
10  
11  
12  
13 const hasMiddleName = (person) =>  
14   validatePerson(person) && "middle" in person.name;  
15  
16
```

```
1 "use strict";  
2  
3 var salute = function salute(person) {  
4   if (validatePerson(person)) {  
5     throw new Error("Not a person");  
6   } // Ternary operator - ?:  
7  
8   var title = person.gender === "m" ? "Mr" : "Ms";  
9   return "".concat(title, " ").concat(person.name.first, "  
10 ").concat(person.name.last, " ");  
11 };  
12  
13 var hasMiddleName = function hasMiddleName(person) {  
14   return validatePerson(person) && "middle" in person.name;  
15 };  
16
```

(Review) Higher Order Functions (HOF).

- **Definition:** A function that takes a function as a parameter (and/or returns a function response).
 - Function parameter termed a callback.
function someHOF(. . ., callback) {..... Body}
 - Callback is usually an anonymous function.
- **Case study – The Array HOFs.**
 - `forEach()`
 - `filter()`
 - `map()`
 - `reduce()`

(Review) Array HOFs – `forEach()`.

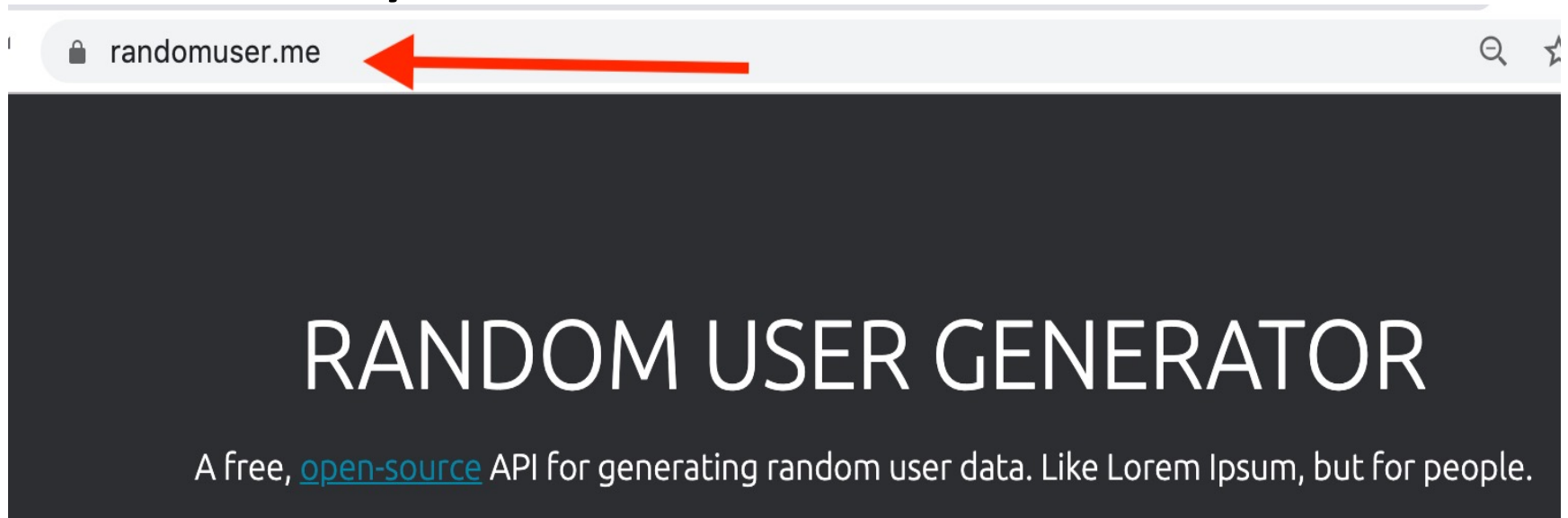
```
const sourceArray = [el1, el2, ....]
```

```
sourceArray.forEach(
```

```
  function(element, index, array) { ...body...} // Anonymous function  
)
```

- **`forEach` executes the callback for each array element.**
- **An alternative to a for-loop.**
- **Callback's index and array arguments are optional.**
- **Callback often coded using Arrow function style.**

Array HOF demos – Data source



- Open Web API.
- Accepts HTTP GET requests, e.g.
<https://randomuser.me/api/?results=10> - generate 10 user profiles and returns them in a JSON (Javascript Object Notation) structure.

Array HOF demos - Code.

- **Ref. `src/01-functions/reviewHOF/index.ts`**
- **`filter(callback)`.**
 - Select a subset of elements from a source array.
 - Selected element references added to a new array.
 - Source array unchanged (Pure).
- **`map(callback)`.**
 - Creates a new array based on the source – 1-for-1 mapping.
 - Source array unchanged (Pure).

Array HOF demos.

```
const accumulator = sourceArray.reduce(
  (acc, element, index, array) => { // Callback
    . . . . .
    return updatedAccumulator
  }, initialAccumulator ) // Note
```

- **reduce(.....)**
 - “reduces the source array to a single accumulated value.”
 - Source array unchanged (Pure)
 - The callback incrementally ‘builds’ the accumulator.
 - Accumulator is passed between callback invocations.

(Back to Typescript) Higher Order Functions.

- Declaring the callback's type in a custom HOF.

callback : (param1: type, param2: type, ...) => return_type

```
4 export function printToConsole(  
5   text: string,  
6   callback: (s: string) => string  
7 ): void {  
8   const response = callback(text);  
9   console.log(response);  
10 }
```

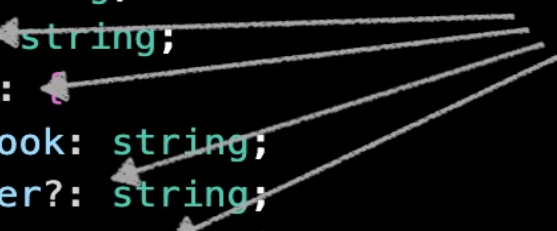
```
12 export function arrayMutate(  
13   numbers: number[],  
14   mutate: (num: number) => number  
15 ): number[] {  
16   return numbers.map(mutate);  
17 }
```

- Can use type aliases to improve readability of callback's signature.

Optionals. (?)

- Optional object properties are properties that can hold a value or be undefined.

```
4  interface User {  
5      id: string;  
6      name: string;  
7      email?: string;  
8      social?:  
9          facebook: string;  
10         twitter?: string;  
11         instagram?: string;  
12     };  
13     status : boolean  
14 }
```



- May also be used with function parameters.
 - An optional parameter cannot precede a required one.
 - Must accommodate undefined case in body – otherwise compiler errors may arise.

Union types & Type Literals

- **Union types** are used when a value can be more than a single type, e.g.

type Size = string | number

let glassSz : Size = 'medium'

let bottleSz: Size = 2. // litre

type Role = Student | Lecturer | Manager


- **Literal types** - three sets of literal types available in TS: strings, numbers, and booleans; by using literal types you can allow an exact value which a string, number, or boolean must have.

e.g. type DegreeNomination = 'BSc' | 'BEng' | 'BA' | 'BBs'

Generics

- *A major part of software engineering is building components that not only have well-defined and consistent APIs, but are also reusable, i.e. can be used for multiple data types.*
- *Generics allow creating 'type variables' to create classes, functions & type aliases that don't need to explicitly define the data types that they use.*

```
29 // T is a type variable - it's assigned a Type on invocation
30 // element and num are parameters that are assigned values on invocation
31 function process<T>( element: T, num: number) {
32     // process T
33 }
34
35 process<Person>( personX, 5)
36 process<Box>( boxY, 12)
37
```



Utility Types

- **TypeScript provides several utility types to facilitate common type transformations.**
- **These utilities are available globally.**

