

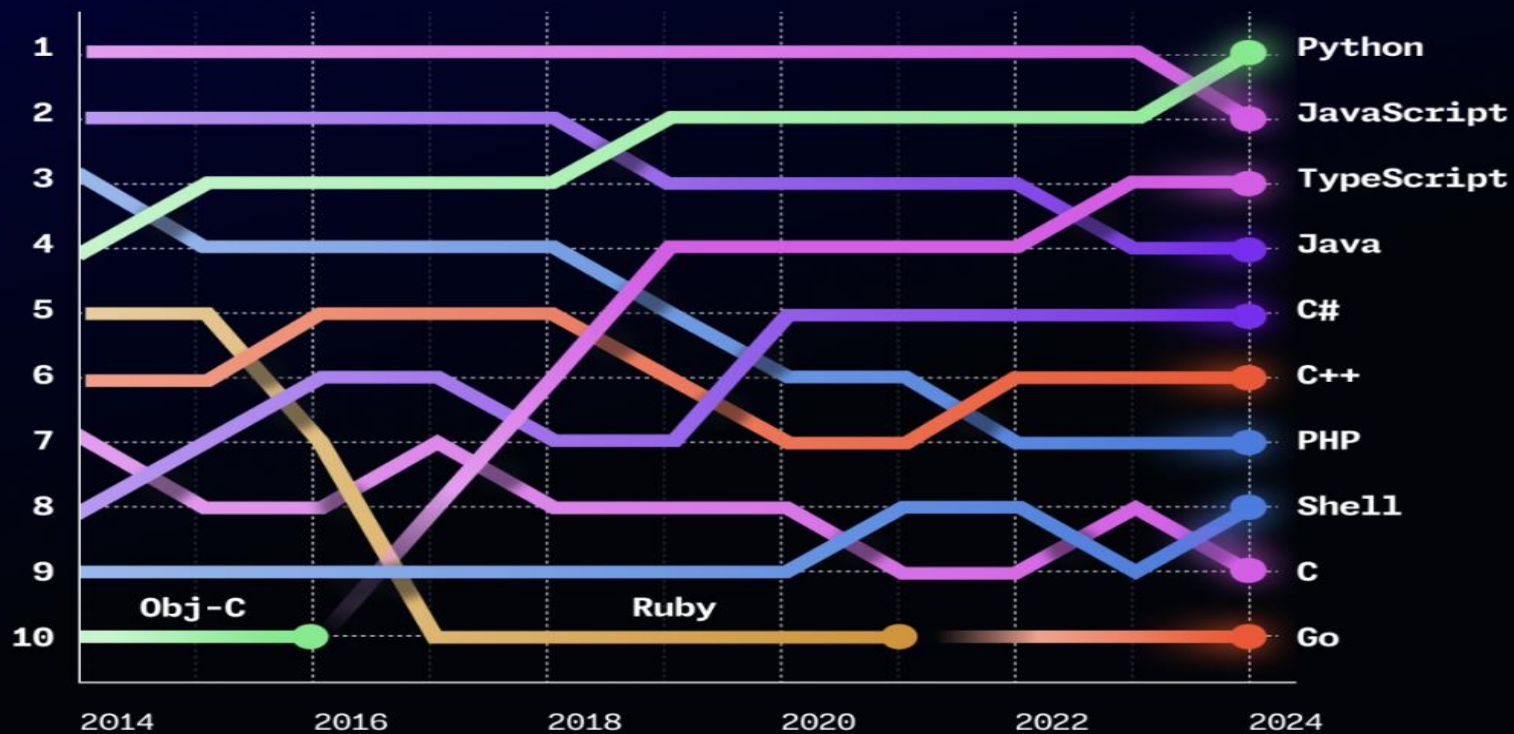
TypeScript

Source code - <https://github.com/diarmuidoconnor/typescript-demos>

Background

- Open-source language, developed by Microsoft (2010-12).
- Anders Hejlsberg - the creator of C# and Turbo Pascal
- Based on ECMAScript 4 (2000) and 6 (2015).
- A superset of JavaScript.
- We still write JS, but it's augmented by ES6 class-based OOP and the structural type system of ES4.
- TS is compiled to regular JS and runs in any browser, or OS.
- "... one thing TS got right: local type inference" Bernard Eich
- "What impressed me is what TS doesn't do; it does not output type-checking in the JS code" Nicholas C Zakas .
- TS is a a language for large-scale JavaScript development.

RANKED BY COUNT OF DISTINCT USERS CONTRIBUTING TO PROJECTS OF EACH LANGUAGE.



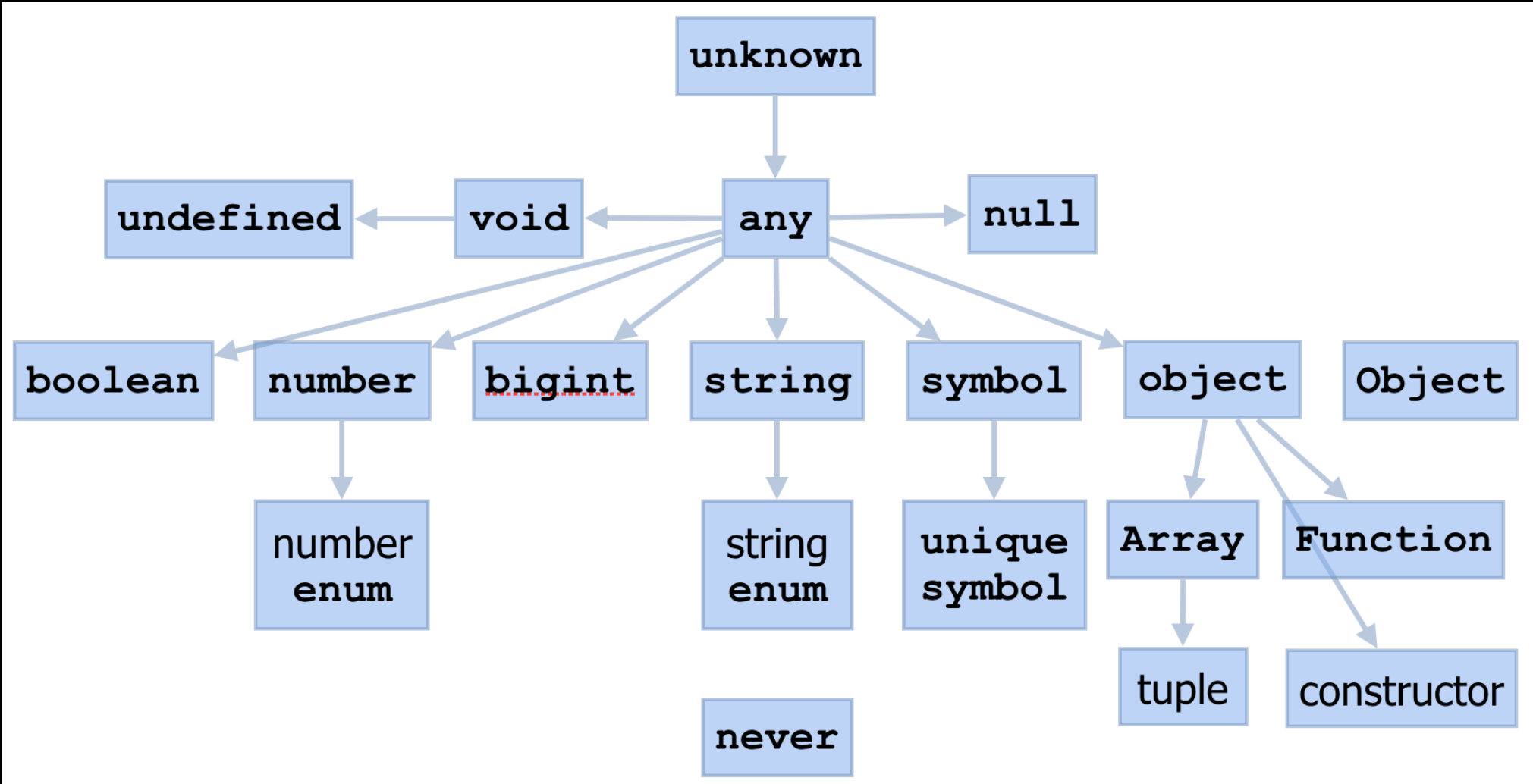
File Extensions.

- .ts - source code 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 types of a third-party JavaScript library, allowing TS developers to consume it.
 - Tooling - Gives type safety, intellisense and compiler error detection during development.

Types

- Primitive Types:
 - number – represents integers. Floats, doubles.
 - boolean
 - 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.

TypeScript Type Hierarchy



Type Annotations.

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

```
function addNumbers(a: number, b: number): 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 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 (IDT).

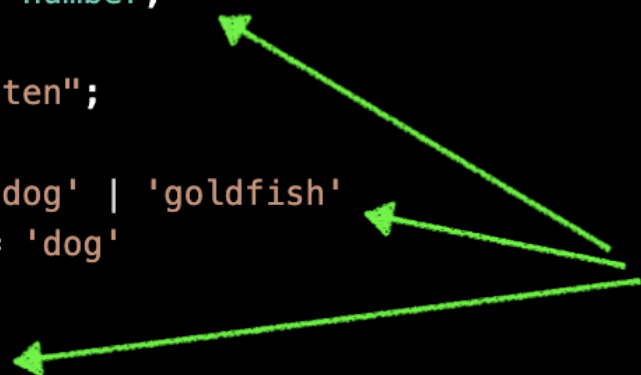
- An interface data type tells the TS compiler about the 'shape' of a data object.
 - property names and value types.
 - An IDT is a type.

```
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 |
```

let aString: string

```
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 }
```

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  }
```

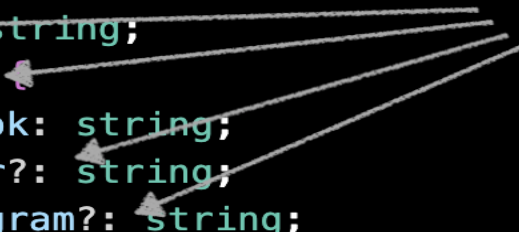
```
12  export function arrayMutate(  
13    numbers: number[],  
14    mutate: (num: number) => number  
15  ): number[] {  
16    return numbers.map(mutate);  
17  }
```

- Can use type aliases to improve the 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 the function body; otherwise, compiler errors may arise.

Union types & Type Literals

- Union types: When a value can be more than a single type.
- e.g.

```
type Size = string | number. // Union type
let glassSz : Size = 'medium'


let bottleSz: Size = 2.5 // liters
type Role = Student | Lecturer | Manager // Union type
const jane: Role = {... student properties ...}
```
- Literal types:
 - Three sets of literal types : strings, numbers, and Booleans.
 - They restrict a variable to specific set of values.e.g.

```
type DegreeNomination = 'BSc' | 'BEng' | 'BA' | 'BBs'
let myDegree : DegreeNomination = 'BEng'
```


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 uses 'type variables' to create classes, functions & type aliases that don't need to explicitly define the data types 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.

