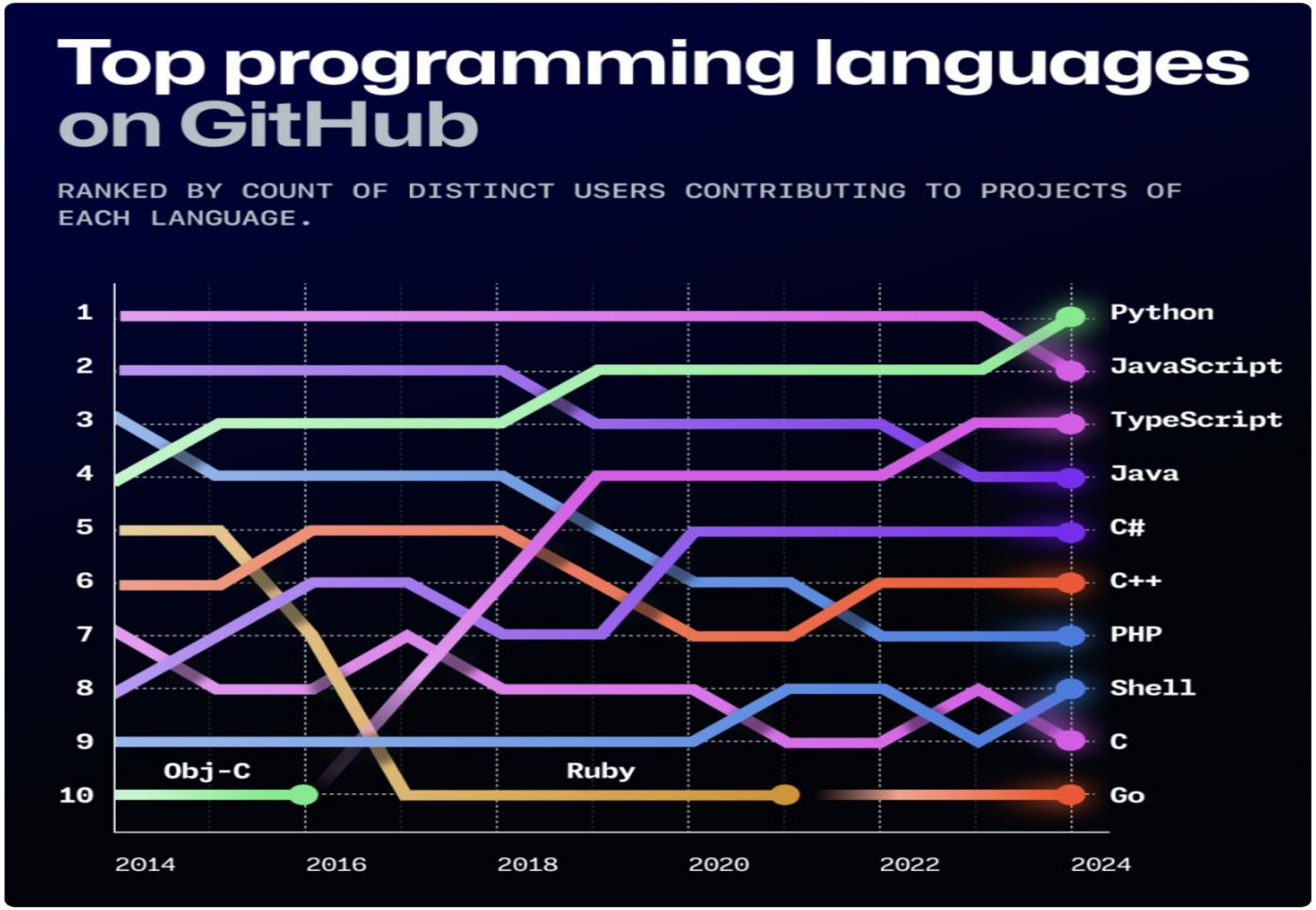
TypeScript

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.



Top programming languages on GitHub

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

1 introduction

File Extensions.

- .ts - source code file extension.
- .d.ts - declaration files.

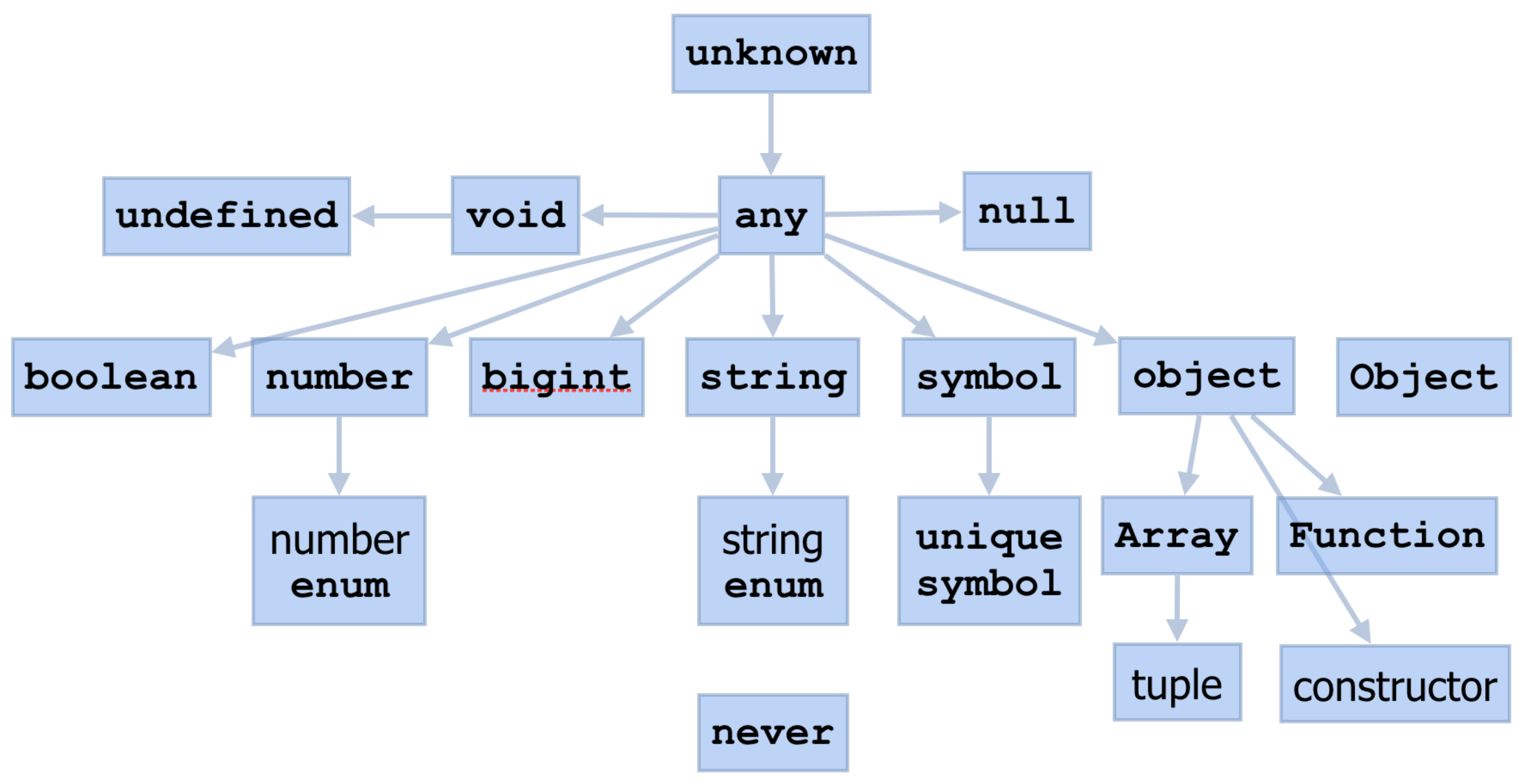
- Declaration source files:
  - Provide type definitions, separate from the source code.
  - Analogous to header files in C/C++.
  - Also used to describe the exported virtual types of a third-party JavaScript library, allowing TS developers to consume it.
  - Tooling - Gives type safety, intellisense and compiler error detection during development.

Types

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



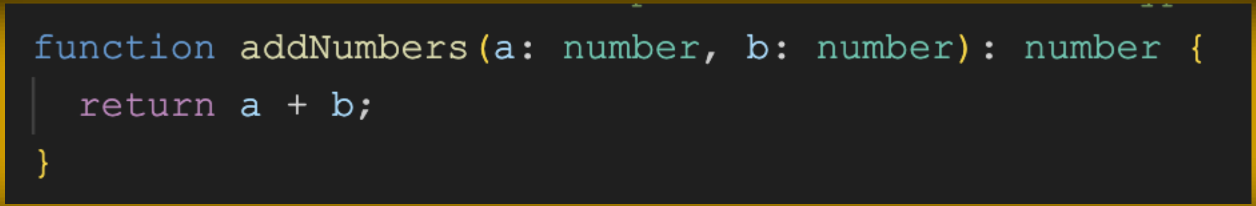
Type Annotations.

- (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:

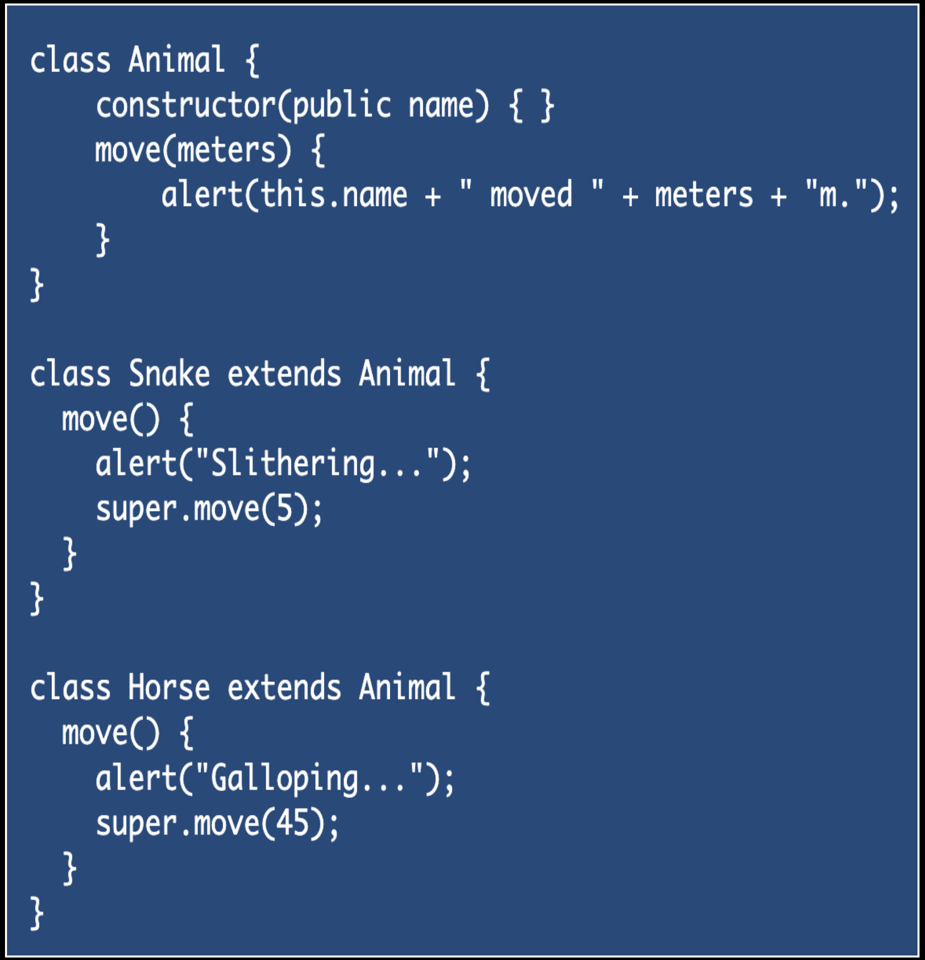
  e.g. let myNums: number[] = [1, 2, 3, 5]
- Can also apply annotations to function signature:



```
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);
  }
}
```

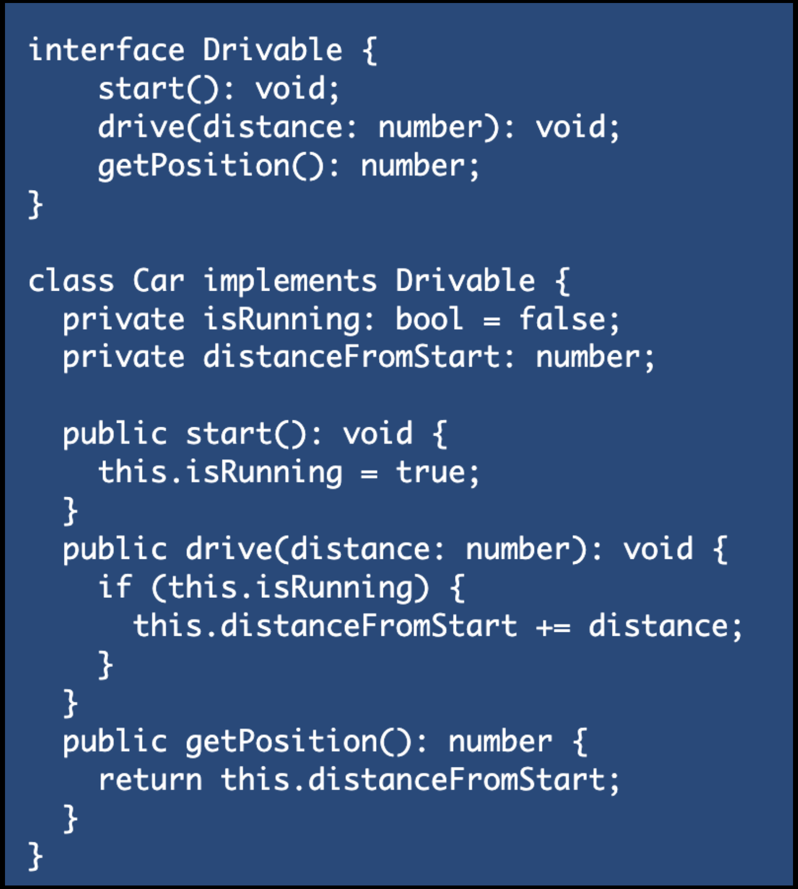
Interfaces

- 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;
  }
}
```

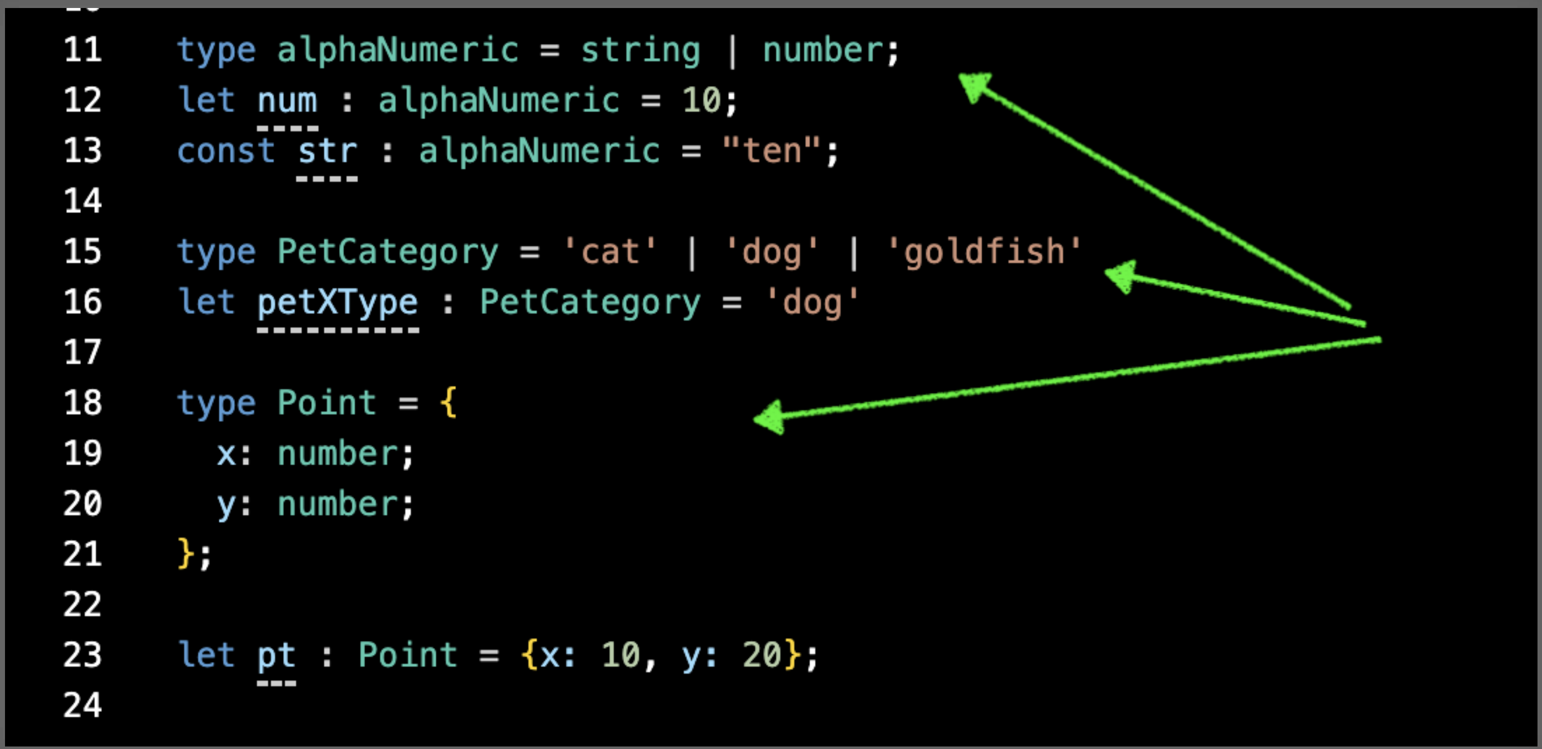
Interface Data Types (IDT).

- 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.

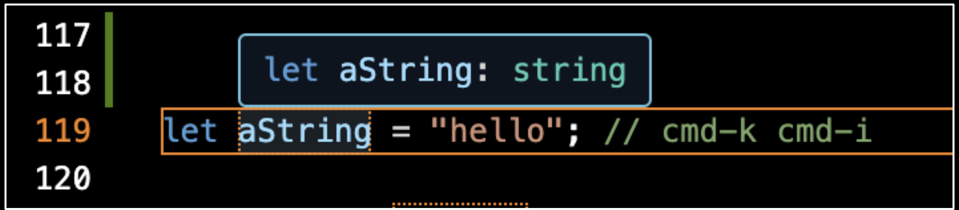
- Type aliases create a new name for a type. Type aliases are sometimes similar to interface data types, but can name primitives, unions, tuples, and any other types.



```
type alphaNumeric = string | number;
let num : alphaNumeric = 10;
const str : alphaNumeric = "ten";

type PetCategory = 'cat' | 'dog' | 'goldfish'
let petXType : PetCategory = 'dog'

type Point = {
  x: number;
  y: number;
};

let pt : Point = {x: 10, y: 20};
```

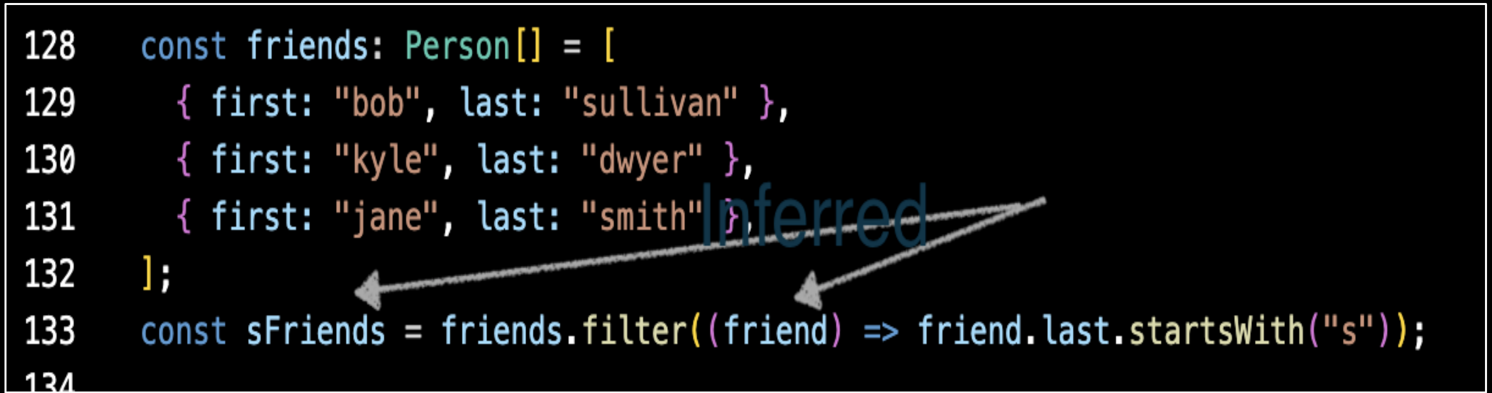
Type Inference.

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



```
let aString: string
```

```typescript
let aString = "hello"; // cmd-k cmd-i
```

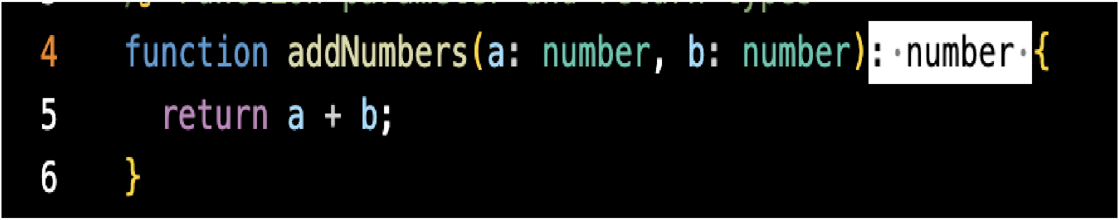


```typescript
const friends: Person[] = [
  { first: "bob", last: "sullivan" },
  { first: "kyle", last: "dwyer" },
  { first: "jane", last: "smith" },
];
const sFriends = friends.filter((friend) => friend.last.startsWith("s"));
```

- Inferencing increases developer productivity.

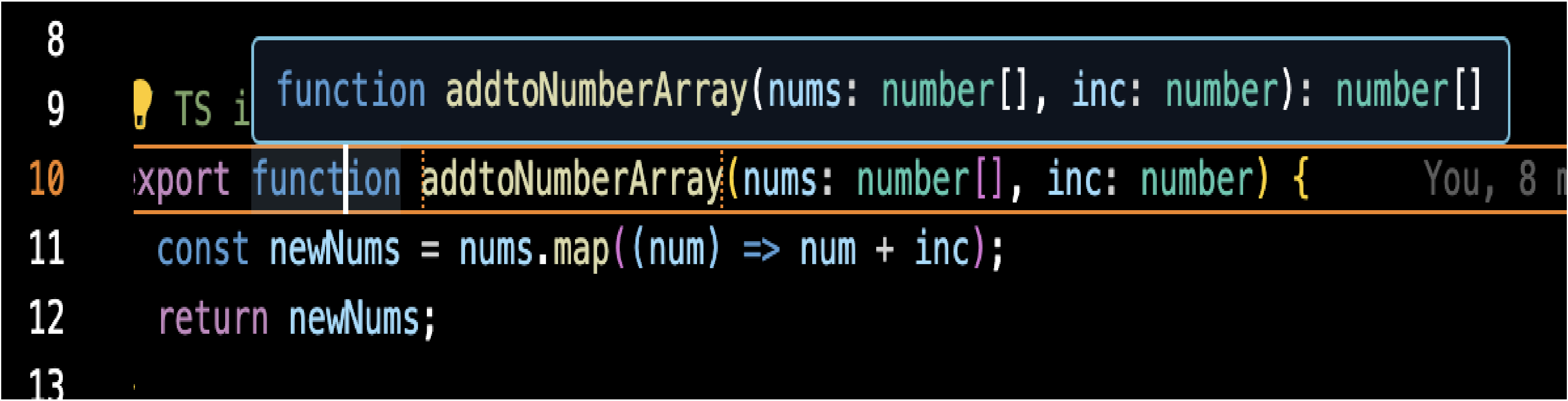
Functions

- Declaring the types in a function’s signature.



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

- Compiler can often infer the return type.

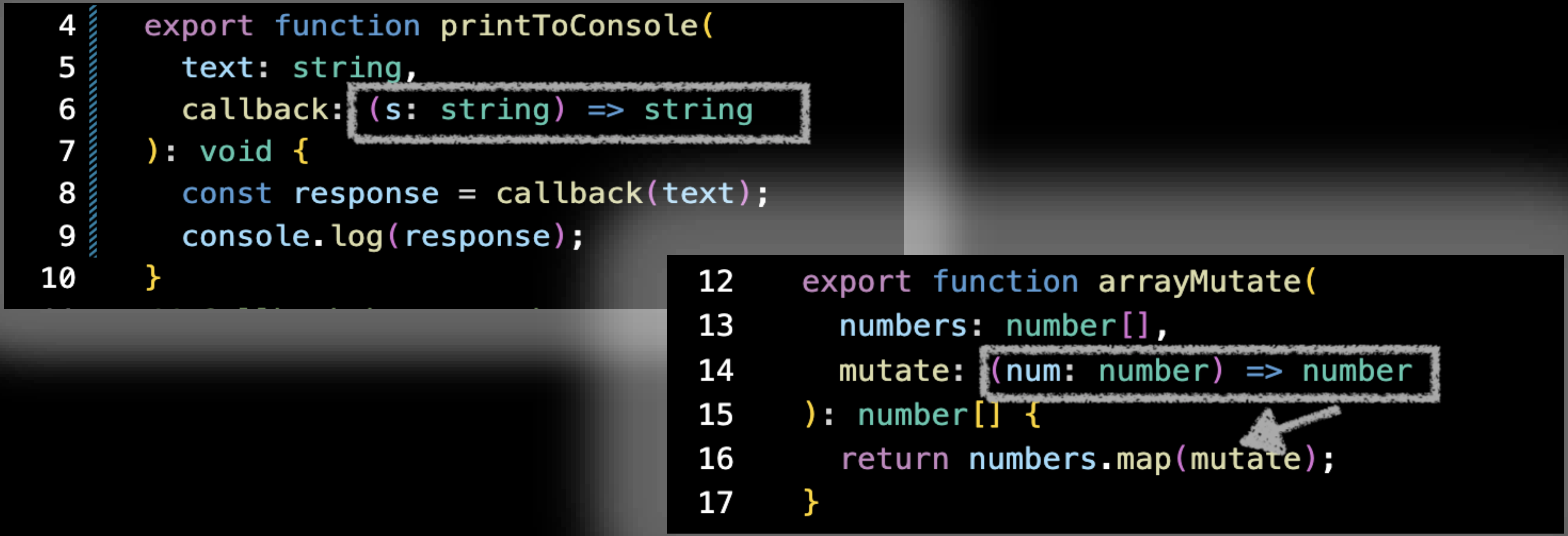


```
function addtoNumberArray(nums: number[], inc: number): number[]
```

```
TS i
xport function addtoNumberArray(nums: number[], inc: number) {      You, 8 m
  const newNums = nums.map((num) => num + inc);
  return newNums;
```

Higher Order Functions.

- Declaring the callback's type in a custom HOF.
  callback : (param1: type, param2: type, …) => return_type



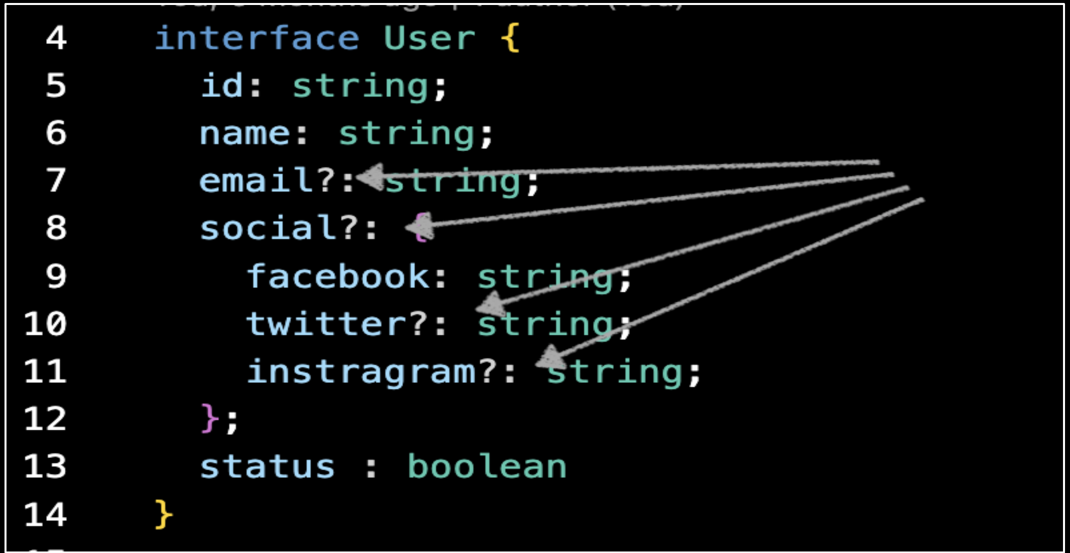
```
export function printToConsole(
  text: string,
  callback: (s: string) => string
): void {
  const response = callback(text);
  console.log(response);
}
```

```
export function arrayMutate(
  numbers: number[],
  mutate: (num: number) => number
): number[] {
  return numbers.map(mutate);
}
```

- 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.



```
interface User {
  id: string;
  name: string;
  email?: string;
  social?: {
    facebook: string;
    twitter?: string;
    instragram?: string;
  };
  status : boolean
}
```

- 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.

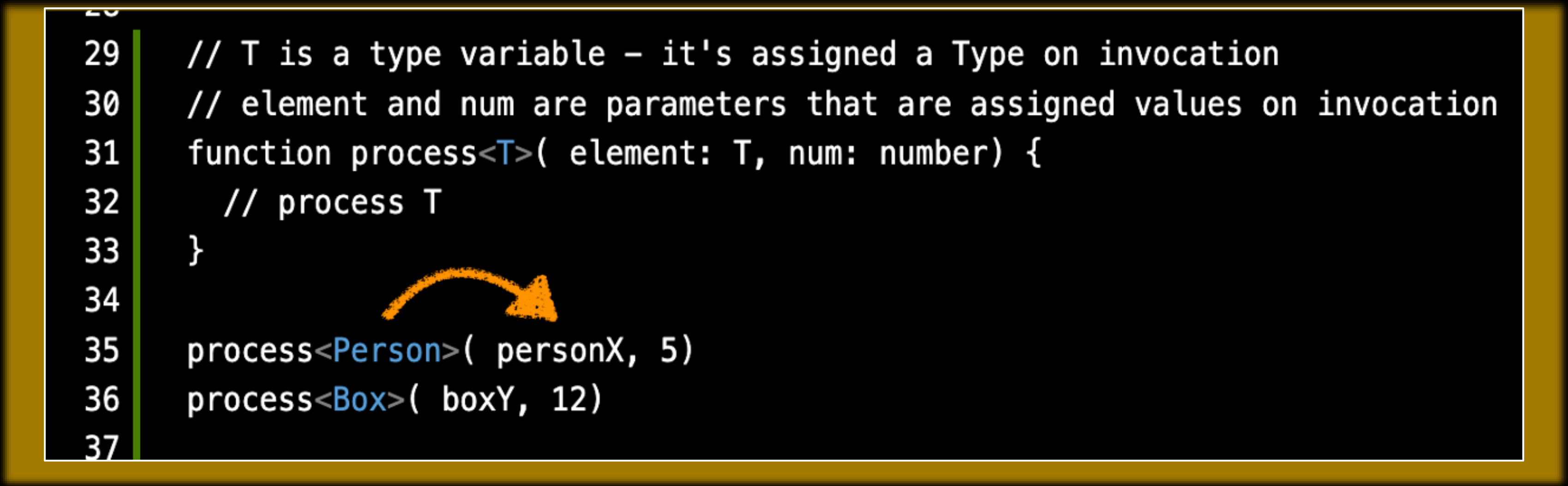
```
type Size = string | number. // Union type
let glassSz : Size = ‘medium’

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

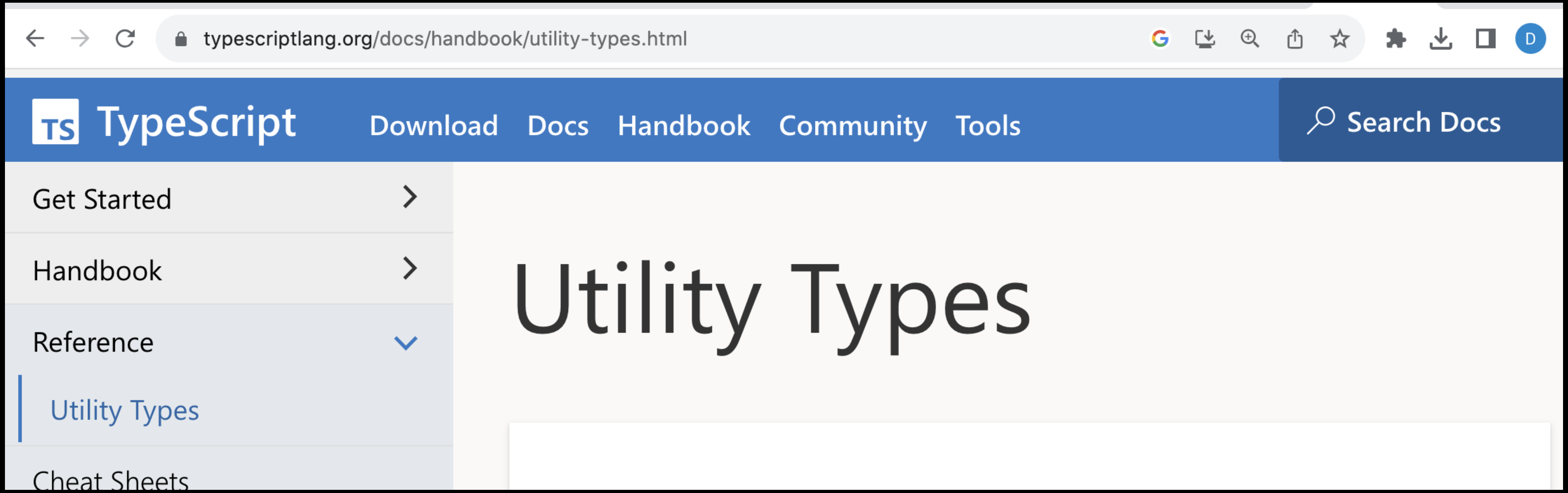
- A major part of software engineering is building components that not only have well-defined and consistent APIs, but are also reusable, i.e. can be used for multiple data types.
- Generics uses 'type variables' to create classes, functions & type aliases that don't need to explicitly define the data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



```
// T is a type variable - it's assigned a Type on invocation
// element and num are parameters that are assigned values on invocation
function process<T>( element: T, num: number) {
  // process T
}

process<Person>( personX, 5)
process<Box>( boxY, 12)
```

Utility types

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



typescriptlang.org/docs/handbook/utility-types.html

Utility Types