Marcin Brzeziński, 12.04.2022


# TypeScript Introduction

How to start with statically-typed JavaScript?

# 🕮 Table of contents

## Clone the repo

https://github.com/brzezinskimarcin/typescript-workshops

https://stackblitz.com/edit/typescript-lcbqt5?file=exercises/1-basics/index.ts

# 0. Why TypeScript?

# 0. Why TypeScript?

- JavaScript was initially developed as a simple scripting language
- Over time its use cases expanded significantly, from few lines of script on the webpage...
- ... to fully interactive web applications, web servers, etc.

Having language designed for quick uses with the intent to build full applications, results in some "quirks":

- `==` (and other comparison operators) can be problematic:

```
'0' == 0  // → true
0  == [] // → true
'0' == [] // → false, not transitive!
```

```
null == 0  // → false
null < 0 // → false
null ⩽ 0 // → true, different type conversions!
```

```
[] == ![]; // → true, array is not an array!
1 < 2 < 3 // → true
3 > 2 > 1 // → false
```

Easy! We can use `===` instead.

```
const a = '0'
const b = 0
if (a) { console.log('test') } // > 'test'
a == b // → true
if (b) { console.log('test') } // > nothing is printed!
```

# 0. Why TypeScript?

- but other operators, like `+` or `-` can also be problematic

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

```
'b' + 'a' + + 'a' + 'a' // → 'baNaNa'
```

```
 3   - 1   // → 2
 3   + 1   // → 4
'3' - 1   // → 2
'3' + 1   // → '31'
```

```
2 + 2 - 2 // → 2
2 - 2 + 2 // → 2
'2' + '2' - '2' // → 20
'2' - '2' + '2' // → '02'
```

```
[] + [] // → ''
{} + [] // → 0
[] + {} // → '[object Object]'
{} + {} // → '[object Object][object Object]'
(!+[]+[]+![]).length // → 9
```

```
(+{} + [] + +!![] / +![])[+!![] + [+[]]] +
([] + {})[+!![]] +
([][[]] + [])[!+[] + !![]] +
(![] + [])[+!![]]
// → 'yoda'
```

- what if we are using external libraries?

```
const date = faker.date.past() // is it javascript Date or is it a string in ISO format?
this.$cookies.set('termsAccepted', true, { expires: '3Y' }) // { expires: '3Y' } or just '3Y'?
```

# 0. Why TypeScript?

- accessing or writing to `undefined` properties

```
const rectangle = { width: 10, height: 15 };
const area = rectangle.widht * rectangle.height; // → NaN, there is a typo!
```

```
const product = {
  name: 'Shovel',
  price: 20,
  quantity: 20
};
const updated = { propertyName: 'price', value: 35 };
product[updated.propertyname] = updated.value; // → again typo!
// {
//   name: 'Shovel',
//   price: 20,
//   quantity: 20,
//   undefined: 35
// }
```

# 1. Basics

# 1. Basics

| | |
|---|---|
| TypeScript | superset of Javascript |
| Compiler | static type-checker |
| Compilation | type-checking and ereasing type annotations |

**TypeScript does not change the runtime behavior of our code!**

```
const rectangle = { width: 10, height: 15 };
const area = rectangle.widht * rectangle.height;
```

Property 'widht' does not exist on type '{ width: number; height: number; }'. Did you mean 'width'?

```
'2' - '2'
```

The left-hand side of an arithmetic operation must be of type 'any', 'number', 'bigint' or an enum type.
The right-hand side of an arithmetic operation must be of type 'any', 'number', 'bigint' or an enum type.

```
1 < 2 < 3
```

Operator '<' cannot be applied to types 'boolean' and 'number'.

```
[1, 2, 3] + [4, 5, 6]
```

Operator '+' cannot be applied to types 'number[]' and 'number[]'.

# 1. Basics

TypeScript introduces contextual, structural, gradual type system.

# 1. Basics

TypeScript introduces contextual, structural, gradual type system.

- you can explicitly put type annotations...

```
const x: number = 0;
```

- ... but compiler often is able to infer types from the context!

```
const x = 0; // x: number;
const arr = [1, 2, 3]; // x: number[];
const sum = arr.reduce((a, b) => a + b, 0); // sum, a, b: number;
const strings = arr.map((i) => i.toString(), 0); // i: number; strings: string[];
const todo = { value: 'Go to a gym', modified: new Date() } // todo: { value: string; modified: Date; };
window.onmousedown = function (mouseEvent) { // mouseEvent: MouseEvent;
  console.log(mouseEvent.button);
};
```

# 1. Basics

TypeScript introduces contextual, structural, gradual type system.

- types determined based on the structure
- duck typing...

*"If it walks like a duck and it quacks like a duck, then it must be a duck"*

```
class Duck {
  swim() {}
}

class Whale {
  swim() {}
}

let duck: Duck = new Whale(); // → no error!
duck.swim();
```

# 1. Basics

TypeScript introduces contextual, structural, gradual type system.

- types determined based on the structure
- duck typing...

*"If it walks like a duck and it quacks like a duck, then it must be a duck"*

```
class Duck {
  swim() {}
  fly() {}
}

class Whale {
  swim() {}
}

let duck: Duck = new Whale(); // → Property 'fly' is missing in type 'Whale' but required in type 'Duck'.
duck.swim();
```

- but static!

# 1. Basics

TypeScript introduces contextual, structural, gradual type system.

- whenever compiler is able to tell what the type of an expression should be, it performs type checking...
- ... but if not it uses a special type `any`, that turns off type checking

```
const arr = []; // arr: any[];
arr.push(1);
arr.push('oh no');
arr.push({ anything: null });

const x: any; // x: any;
x.push(1);
x + 2;
x.property;
```

- it means we can convert our non-type-safe javascript code to type-safe typescript code gradually

# 1. Basics

Javascript primitives:

- `number`
- `bigint`
- `string`
- `boolean`
- `symbol`
- `null`
- `undefined`

```
let decimal = 6;        // decimal: number;
let hex = 0×f00d;       // hex: number;
let big = 100n;         // big: bigint;
let align = 'left';     // align: string;
let isDone = false;     // isDone: boolean;
let symbol = Symbol(); // symbol: Symbol;
let undef = undefined; // undef: undefined;
let n = null;           // n: null;
```

# 1. Basics

Literal types:

```
let align = 'left'; // align: string;
```

But defining constant:

```
const align = 'left'; // align: 'left';
align = 'right' // → Cannot assign to 'align' because it is a constant.
const b = 0 // b: 0;
const a = false // a: false;
```

However:

```
const req = { url: 'https://example.com', method: 'GET' }; // { url: string; method: string }
req.method = 'RANDOM' // → OK
```

Solution? `as const`:

```
const req = { url: 'https://example.com', method: 'GET' } as const; // { url: 'https://example.com'; method: 'GET' }
req.method = 'RANDOM' // → Error
```

Right now, these types seem not to be valuable, but soon you'll find out they can be really useful!

# 1. Basics

## Functions

```
function logger(info: string) {
  console.log(`LOG ${new Date().toISOString()}: ${info}`);
}
```

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

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

const add = (a: number, b: number) ⇒ a + b;
```

```
const add: (arg1: number, arg2: number) ⇒ number = (a: number, b: number) ⇒ a + b;

function accumulate(arr: number[], adder: (a: number, b: number) ⇒ number) {
  return arr.reduce(adder, 0);
}
```

# 1. Basics

TypeScript specific types:

- `any` -> you can use whenever you don't want a particular value to cause typechecking errors
- `void` -> you can only assign `null` and `undefined`, typical use case: function returns nothing

```
function doNothing(): void {}
let a = doNothing();
a + 2; // → Operator '+' cannot be applied to types 'void' and 'number'.
```

- `unknown` -> works as "top" type, it basically means "I don't know anything about what type it is"

```
let a: unknown = { some_property: 'some_value' };
a = 2; // OK: a is 'unknown', so we can assign value of any type to it.
a - 2; // → Object is of type 'unknown'.
```

- `never` -> works as "bottom" type, it basically means "The value of such type should never occur"

```
let a: never = 5; // → Type 'number' is not assignable to type 'never'.
function throwError(): never { // OK: function immediately throws an error so it never reaches end point.
  throw new Error();
}
function doNothing(): never {} // → A function returning 'never' cannot have a reachable end point.
```

exercises/1-basics.ts

# 2. Complex types

# 2. Complex types

Objects

- object -> refers to any value that is not primitive

```
let obj: object = {}; // obj: object;
obj.hasOwnProperty('a'); // OK
obj.a; // → Property 'a' does not exist on type 'object'.
obj = 'asd' // → Type 'string' is not assignable to type 'object'.
```

- object literal -> simply list its properties and (optionally) their types

```
function printPoint(point: { x: number; y: number }) {
  console.log(`(${point.x}, ${point.y})`);
}
const point = { x: 3, y: 7 };
printPoint(point);
```

We should consider it as "constraint", so this code also works fine:

```
const point3d = { x: 3, y: 7, z: 5 };
printPoint(point3d);
```

Interesting consequence: **empty type**

```
const empty1: {} = { x: 3, y: 7, z: 5 };
const empty2: {} = { };
const empty3: {} = 123;
const empty4: {} = null;
// → Type 'null' is not assignable to type '{}'.
```

# 2. Complex types

Type aliases

Listing object properties each time can become inconvenient...

```
function distance(p1: { x: number; y: number }, p2: { x: number; y: number }) {
  const dx = p2.x - p1.x;
  const dy = p2.y - p1.y;
  return Math.sqrt(dx * dx + dy * dy);
}
```

So we can define type aliases...

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

function distance(p1: Point, p2: Point) {
  const dx = p2.x - p1.x;
  const dy = p2.y - p1.y;
  return Math.sqrt(dx * dx + dy * dy);
}
```

... with two possible ways

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

function distance(p1: Point, p2: Point) {
  const dx = p2.x - p1.x;
  const dy = p2.y - p1.y;
  return Math.sqrt(dx * dx + dy * dy);
}
```

What's the difference? Soon we will find out.

# 2. Complex types

Unions:

```
function printUserId(id: number | string) {
  console.log(`User id: ${id}`);
}
printUserId(101);
printUserId('202');
printUserId(null); // → Argument of type 'null' is not assignable to parameter of type 'string | number'.
```

```
interface Bird {
  fly(): void;
  layEggs(): void;
}

interface Fish {
  swim(): void;
  layEggs: () ⇒ void;
}

function foo(pet: Fish | Bird) {
  pet.layEggs(); // OK
  pet.swim(); // Property 'swim' does not exist on type 'Bird | Fish'. Property 'swim' does not exist on type 'Bird'.
};
```

# 2. Complex types

We can narrow the union type using javascript `typeof` operator:

```
function printId(id: number | string) {
  if (typeof id === 'string') {
    console.log(id.toUpperCase()); // id: string;
  } else {
    console.log(id); // id: number;
  }
}
```

Optional (`?`) operator `<type> | undefined`:

```
function printId(id?: number) {
  console.log(id); // id: number | undefined;
}

interface NetworkError {
  error?: string; // error: string | undefined;
}
```

Or we can finally use our string literals:

```
interface Loading {
  state: 'loading';
};
interface Failed {
  state: 'failed';
  code: number;
};
interface Success {
  state: 'success';
  response: { data: string; };
};
type NetworkState = Loading | Failed | Success;
function logger(state: NetworkState): string {
  switch (state.state) {
    case 'loading': // state: Loading;
      return 'Downloading...';
    case 'failed': // state: Failed;
      return `Error ${state.code} downloading`;
    case 'success': // state: Success;
      const { data } = state.response;
      return `Downloaded ${data}`;
  }
}
```

# 2. Complex types

## Intersections - symmetric to unions

```ts
interface CreateArtistBioBase {
  artistID: string;
  thirdParty?: boolean;
}
type BioFormat = { html: string } | { markdown: string };
type CreateArtistBioRequest = CreateArtistBioBase & BioFormat;

const correctRequest: CreateArtistBioRequest = {
  artistID: 'banksy',
  markdown: 'Banksy is an anonymous England-based graffiti artist ...',
}; // OK

const badRequest: CreateArtistBioRequest = {
  artistID: 'banksy',
}; // → Property 'markdown' is missing in type '{ artistID: string; }' but required in type '{ markdown: string; }'.

const badRequest2: CreateArtistBioRequest = {
  markdown: 'banksy',
}; // → Property 'artistID' is missing in type '{ markdown: string; }' but required in type 'CreateArtistBioBase'.
```

# 2. Complex types

## Tuples - fixed-length arrays

```
type Point = [number, number];
function print(coordinates: Point) {
  console.log(coordinates); // → 1,3
  const [x, y] = coordinates; // They behave just like arr
  console.log(`Provided coordinates - x: ${x}, y: ${y}`);
  const z = coordinates[3]; // → Tuple type 'Point' of le
}
```

## … and even use rest elements!

```
type Str_Num_Bools = [string, number,  ...boolean[]];
type Str_Bools_Num = [string,  ...boolean[], number];
type Bools_Str_Num = [ ...boolean[], string, number];
```

## Can be optional…

```
type Either2dOr3d = [number, number, number?];
function foo(coordinates: Either2dOr3d) {
  const [x, y, z] = coord; // z: number | undefined;
}
```

# 2. Complex types

Differences between `interface` and `type`:

| diff | interface | type |
| --- | --- | --- |
| primitives | `interface Name {}; // → ✗ Impossible to define!`<br>`interface Pair {}  // → ✗ Impossible to define!` | `type Name = string;  // → ✓`<br>`type Pair = [number, number]; // → ✓` |
| unions | `interface Info {} // → ✗ Impossible to define!`<br>`interface BioFormat { // → ✗ Impossible to define!`<br>`  html?: string;`<br>`  markdown?: string;`<br>`}`<br>`const bio: BioFormat = {};`<br>`// → OK` | `type Info = string \| { name: string }; // → ✓`<br>`type BioFormat = // → ✓`<br>`  { html: string } \|`<br>`  { markdown: string };`<br><br>`const bio: BioFormat = {};`<br>`// → Type '{}' is not assignable to type 'BioFormat'.` |

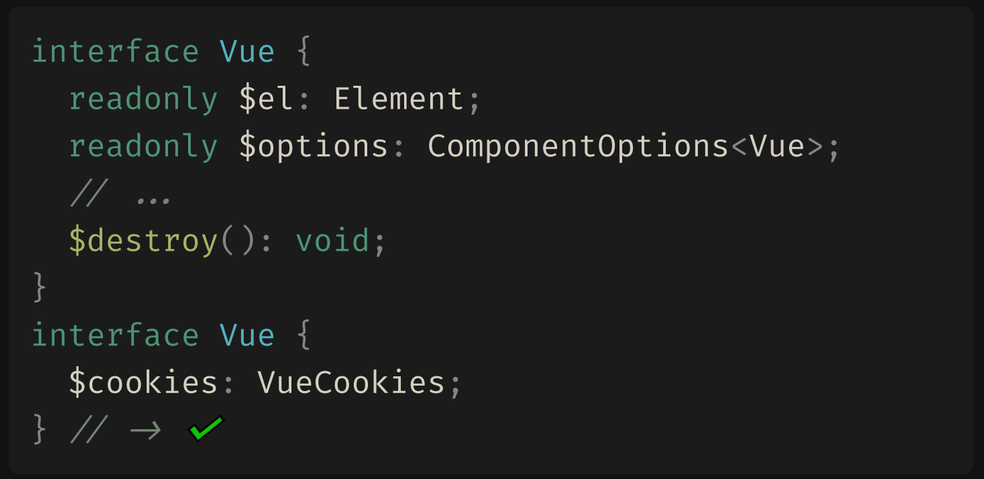
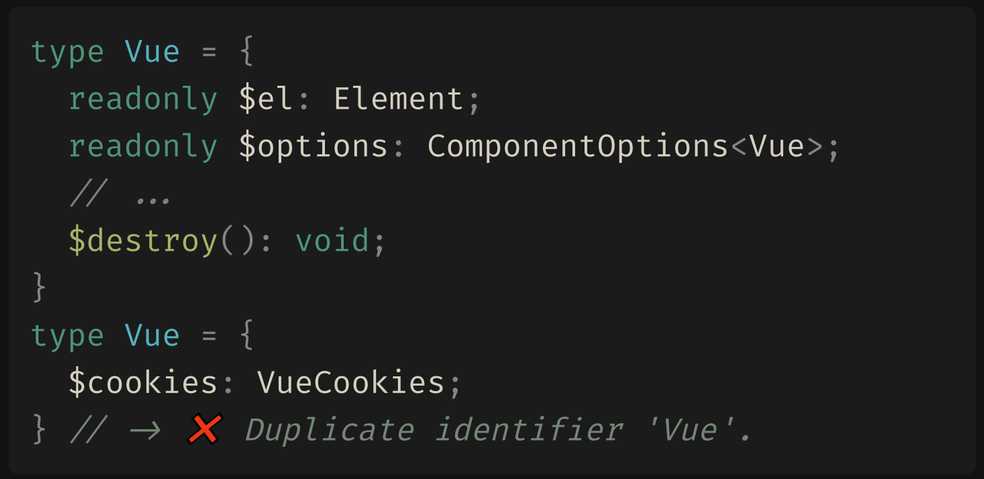
# 2. Complex types

Differences between `interface` and `type`:

| diff | interface | type |
| --- | --- | --- |
| extending | `interface Animal { name: string; } interface Bear extends Animal { honey: boolean; } // → ✔` | `type Animal = { name: string; } type Bear = Animal & { honey: boolean; } // → ✔` |
| errors | `interface Bird { wings: 2; } interface Owl extends Bird { nocturnal: true; } const owl: Owl = { wings: 2 }; // → Property 'nocturnal' is missing in type // '{ wings: 2; }' but required in type // 'Owl'. ✔` | `type Bird = { wings: 2; }; type Owl = Bird & { nocturnal: true }; const owl: Owl = { wings: 2 }; // → Property 'nocturnal' is missing in type // '{ wings: 2; }' but required in type // '{ nocturnal: true; }'. ✘` |

# 2. Complex types

Differences between `interface` and `type`:

| diff | interface | type |
| --- | --- | --- |
| merging |  |  |





There are also some differences related to generics which we will cover later.

Good practice:

Always use `interface` unless you need features from `type`.

# 2. Complex types

## Property modifiers

### optional modifier

```
interface Opt {
  a?: string;
}
const x1: Opt = { };
const x2: Opt = { a: 'string' };
```

### assignability

```
interface Person {
  name: string;
}
interface ReadonlyPerson {
  readonly name: string;
}
let wr: Person = {
  name: 'Person McPersonface'
};
let rd: ReadonlyPerson = wr;
console.log(rd.age); // prints '42'
wr.age++;
console.log(rd.age); // prints '43'
```

# 2. Complex types

## index signatures

often we don't want to name keys

```
interface StringDictionary {
  [index: string]: string;
}
const myDict: StringDictionary = { a: 'foo', b: 'bar' };
```

modifiers work on index signatures

```
interface ReadonlyStringArray {
  readonly [n: number]: string;
}
```

you can put multiple indexed signatures

```
interface Foo {
  [n: string]: string;
  [n: number]: number; // → 'number' index type 'number'
  // is not assignable to 'string' index type 'string'.
}
```

symbol and string do not have intersection

```
interface F { [n: string]: string; [n: symbol]: symbol; }
```

you can add named keys, but types must match!

```
interface NumberOrStringDictionary {
  [index: string]: number | string;
  length: number; // OK
  name: string; // OK
  valid: boolean; // → Property 'valid' of type
  // 'boolean' is not assignable to 'string' index type
  // 'string | number'.
}
```

in JS you can access numeric properties with string syntax...

```
interface Foo2 {
  [n: string]: string | number; [n: number]: number;
}
const foo: Foo2 = {
    a: 'foo', // OK
    0: 1, // OK
    1: 'a', // Error
    '2': 'a', // Error!
}


const a = foo[0]; // 'a';
```

exercises/2-complex-types.ts

# 3. Classes

# 3. Classes

we can declare and initialize fields

```
class Point {
  x = -1;
  y: number;
}

const pt = new Point();
pt.x = 0;
pt.y = 0;
```

strictPropertyInitialization

```
class BadGreeter {
  name: string; // → Property 'name' has no initializer
  // and is not definitely assigned in the constructor.
}
class OKGreeter {
  // Not initialized, but no error
  name!: string;
}
```

we can use modifiers

```
class Greeter {
  readonly name: string = 'world';

  constructor(name: string) {
    this.name = name;
  }

  err() {
    this.name = 'not ok'; // → Cannot assign to 'name'
    // because it is a read-only property.
  }
}
```

getters and setters

```
class Thing {
  _size = 0;

  get size(): number {
    return this._size;
  }

  set size(value: string | number | boolean) {
    let num = Number(value);
    this._size = num;
  }
}
```

# 3. Classes

we can implement interface(s)

```
interface Checkable {
  check(name: string): boolean;
}

class NameChecker implements Checkable {
  check(name) { // → s: any;
    return name.toLowerCase() === "ok";
  }
}
```

extend from base class

```
class Base {
  greet() {
    console.log('Hello, world!');
  }
}

class Derived extends Base {
  greet(name?: string) {
    if (name === undefined) {
      super.greet();
    } else {
      console.log(`Hello, ${name.toUpperCase()}`);
    }
  }
}

const d = new Derived();
d.greet();
d.greet('reader');
```

`implements` only type-checks, doesn't change the type!

```
interface A {
```

# 3. Classes

## type-only field declarations

```
interface Animal {
  dateOfBirth: any;
}

interface Dog extends Animal {
  breed: any;
}

class AnimalHouse {
  resident: Animal;
  constructor(animal: Animal) {
    this.resident = animal;
  }
}

class DogHouse extends AnimalHouse {
  // Does not emit JavaScript code,
  // only ensures the types are correct
  declare resident: Dog;
  constructor(dog: Dog) {
    super(dog);
  }
}
```

## member visibility - protected

```
class Greeter {
  public greet() {
    console.log(`Hello, ${this.getName()}`);
  }
  protected getName() { return 'name'; }
}
class SpecialGreeter extends Greeter {
  public howdy() {
    console.log(`Howdy, ${this.getName()}`);
  }
}
const g = new SpecialGreeter();
g.greet(); // OK
g.getName(); // → Error
```

## subclass can change visibility

```
class Base {
  protected m = 10;
}
class Derived extends Base {
  m = 15;

}
```

# 3. Classes

member visibility - private

```
class Base {
  private x = 0;
}
const b = new Base();
console.log(b.x); // → Property 'x' is private and only
// accessible within class 'Base'.
```

static fields - limitations on name

```
class S {
  static name = 'foo'; // → Static property 'name'
  // conflicts with built-in property 'Function.name' of
  // constructor function 'S'.
}
```

no static classes - they're not needed

```
class MyStaticClass {
  static doSomething() {}
}
const MyStaticObject = {


  dosomething() {}
```

private just on type level

```
class MySafe {
  private secretKey = 12345;
}
const s = new MySafe();
console.log(s.secretKey); // → Property 'secretKey' is
// private and only accessible within class 'MySafe'.
console.log(s['secretKey']); // OK
```

"private" vs "#"

```
class MySafe {
  #secretKey = 12345;
}
```

static fields

```
class MyClass {
  static x = 0;
  static printX() {
    console.log(MyClass.x);
  }


}
```

# 3. Classes

"this" can give a headache

```
class MyClass {
  name = 'MyClass';
  getName() { return this.name; }
}
const c = new MyClass();
const getName = c.getName;
console.log(getName()); // runtime error
```

"this" argument removed during compilation

```
class MyClass {
  name = 'MyClass';
  getName(this: MyClass) { return this.name; }
}
const c = new MyClass();
const getName = c.getName;
console.log(getName()); // The 'this' context of type
// 'void' is not assignable to method's 'this' of type
// 'MyClass'.
```

solution - arrow function?

```
class MyClass {
  name = 'MyClass';
  getName = () => { return this.name; };
}
const c = new MyClass();
const getName = c.getName;
console.log(getName()); // MyClass
```

can be used as any other type, in paremeters, returns, etc.

```
class Box {
  val: string = '';
  sameAs(b: this) { return b.val === this.val; }
}
```

shorthand for class fields in constructor

```
class Params {
  constructor(readonly x: number, private y: number) {}
}
const a = new Params(1, 2);


console.log(a.x);
```

# 3. Classes

## class expression

```
const someClass = class {
  content: string;
  constructor(value: string) {
    this.content = value;
  }
};
const m = new someClass('Hello, world');
```

## we have to implement abstract methods

```
class Derived extends Base {} // Non-abstract class
// 'Derived' does not implement inherited abstract member
// 'getName' from class 'Base'.
```

## solution - construct signature

```
function greet(ctor: new () => Base) {
  const instance = new ctor();
  instance.printName();
}
greet(Derived);

greet(Base); // → Cannot assign an abstract constructor
```

## abstract class

```
abstract class Base {
  abstract getName(): string;
  printName() {
    console.log(`Hello, ${this.getName()}`);
  }
}
const b = new Base(); // → Cannot create an instance of
// an abstract class.
```

## use abstract class with constructor

```
function greet(ctor: typeof Base) {
  const instance = new ctor(); // → Cannot create an
  // instance of an abstract class.
  instance.printName();
}
greet(Base);
```

## structural typing - again

```
class Person { name: string; }
class Employee { name: string; salary: number; }

const p: Person = new Employee(); // OK
```

exercises/3-classes.ts

# 4. Assertions, Narrowing

# 4. Assertions, Narrowing

Type assertions:

```
const input = document.querySelector('.input'); // input: Element | undefined;
if (input) { // input: Element;
  input.value = 'test'; // → Property 'value' does not exist on type 'Element'.
}
```

But we know it is input element!

```
const input = document.querySelector('.input') as HTMLInputElement; // input: HTMLInputElement;
input.value = 'test'; // → OK
```

"!" as shortcut for assertion of removing "undefined" and "null":

```
const input = document.querySelector('input'); // input: Element | undefined;
if (input) { // input: Element;
  const tagName1 = input.tagName;
}
const tagName2 = (document.querySelector('input') as Element).tagName;
const tagName3 = document.querySelector('input')!.tagName;
```

to allow "impossible" coercions, we can assert to `unknown` first:

```
const str = ('hello' as unknown) as number;
```

# 4. Assertions, Narrowing

## Type narrowing - "typeof"

```
function getId(id: number | string) {
  if (typeof id === 'string') { // id: string;
    console.log(id.toUpperCase());
  } else { // id: number;
    console.log(id);
  }
}
```

## Type narrowing - "truthiness"

```
function getUsersOnlineMessage(numUsersOnline?: number) {
  if (numUsersOnline) { // numUsersOnline: number;
    return `There are ${numUsersOnline} online now!`;
  }
  return 'Nobody is here. :(';
}
```

## Type narrowing - "equality"

```
function foo(x: string | number, y: string | boolean) {
  if (x === y) { // x: string, y: string;
    x.toUpperCase();
    y.toLowerCase();
  } else {
    console.log(x);
    console.log(y);
  }
}
```

## Type narrowing - "instanceof"

```
function logValue(x: Date | string) {
  if (x instanceof Date) { // x: Date;
    console.log(x.toUTCString());
  } else { // x: string;
    console.log(x.toUpperCase());
  }
}
```

# 4. Assertions, Narrowing

Type narrowing - "in"

```
interface Fish { swim(): void };
interface Bird { fly(): void };
function move(animal: Fish | Bird) {
  if ('swim' in animal) { // animal: Fish;
    return animal.swim();
  }
  return animal.fly(); // animal: Bird;
}
```

Type predicates also work on `this`

```
class BoxedNumber {
  constructor(public value?: number) {}
  hasValue(): this is { value: number } {
    return this.value !== undefined;
  }
}
const box = new BoxedNumber(2);
if (box.hasValue()) {
  box.value; // box.value: number;
}
```

Custom type narrowing - type predicates!

```
interface Fish {
  swim(): void
};
interface Bird {
  fly(): void
};
function isFish(animal: Fish | Bird): animal is Fish {
  return (animal as Fish).swim !== undefined;
}
function move(animal: Fish | Bird) {
  if (isFish(animal)) { // animal: Fish;
    return animal.swim();
  }
  return animal.fly(); // animal: Bird;
}
```

exercises/4-assertions-narrowing.ts

# 5. Generics

# 5. Generics

TS has generic types:

```
function identity<T>(x: T) {
  return x;
}
const identity = <T>(x: T) => x;
```

We can put generics on classes...

```
class Additive<T> {
  zero!: T;
  add!: (x: T, y: T) => T;
}

const num = new Additive<number>();
num.zero = 0;
num.add = (x, y) => x + y;

const str = new Additive<string>();
str.zero = '';
str.add = (x, y) => x + y;
```

We've already seen and used generics!

```
const arr1: number[] = [1, 2, 3];
const arr2: Array<number> = [4, 5, 6];
```

... and on type and interface too!

```
interface Array<T> {
  length: number;
  // ...
  push( ...items: T[]): number;
  pop(): T | undefined;
}

const arr: number[] = [1, 2, 3];
arr.push(4, 5, 6);
arr.pop();
console.log(arr.length);

type Ref<T> = { value: T };
const strRef: Ref<string> = { value: 'str' };
```

# 5. Generics

We can have multiple generics:

```
function map<T, U>(arr: T[], callbackfn: (value: T, index: number) ⇒ U): U[] {
  const res: U[] = [];
  arr.forEach((elm, i) ⇒ res.push(callbackfn(elm, i)));
  return res;
}
const arr1 = ['a', 'aa', 'aaa']; // arr1: string[];
const arr2 = map<string, number>(arr1, (i) ⇒ i.length); // arr2: number[];
```

and reuse generics inside interfaces or types:

```
interface Array<T> {
  reduce<U>(callbackfn: (previousValue: U, currentValue: T, currentIndex: number, array: T[]) ⇒ U, initial: U): U;
}
```

provide defaults...

```
type Maybe<T = number> = T | undefined;
const maybeNumber: Maybe = 5;
const maybeString: Maybe<string> = undefined;
```

type parameters in static members

```
class Box<Type> {
```

# 5. Generics

## ... and put constraints (extends)

```
interface Lengthwise {
  length: number;
}

function getLength<T extends Lengthwise>(arg: T) {
  return arg.length;
}

const arrayLength = getLength([1, 2, 3, 4]);
const strLength = getLength('Schneider');
getLength({}); // → Property 'length' is missing in type
```

```
type Constructor<T> = {
  new(): T;
}
function create<T extends Animal>(c: Constructor<T>): T {
  return new c();
}
create(Tiger).roar();
create(Bee).fly();
```

## Factory pattern (used in mixins)

```
class Animal {
  numLegs: number = 4;
}
class Bee extends Animal {
  fly() {};
}
class Tiger extends Animal {
  roar() {};
}
```

exercises/5-generics.ts

# 6. Types Manipulation

# 6. Types Manipulation

We have already seen one way of defining types from other types - generics. But there is more!

- `keyof` operator

```ts
type Point = { x: number; y: number };
type P = keyof Point; // 'x' | 'y';
```

```ts
type Arrayish = { [n: number]: unknown };
type A = keyof Arrayish; // number;
```

```ts
type Flags = { [k: string]: boolean };
type M = keyof Flags; // ?
```

- `typeof` operator

```ts
let s = "hello";
let n: typeof s; // n: string;
```

```ts
let f = () => ({ x: 10, y: 3 });
type P = typeof f; // () => { x: number, y: number }
```

- indexed access types

```ts
type P = { age: number; name: string; alive: boolean };
type Age = P['age']; // number;
```

```ts
type I1 = P['age' | 'name']; // string | number;
type I2 = P[keyof P]; // string | number | boolean;
```

```ts
const array = [ 1, true ];
type T = (typeof array)[number]; // number | boolean
```

```ts
type I1 = P['alve']; // → Property 'alve' does not
// exist on type 'P'.
```

# 6. Types Manipulation

- conditional types

"extends" keyword

```ts
interface Animal { live(): void; }
interface Dog extends Animal { woof(): void; }
type A = Dog extends Animal ? number : string; // number
type B = Date extends Animal ? number : string; // string
```

"infer" keyword

```ts
type Flatten1<T> = T extends any[] ? T[number] : T;
type Str = Flatten1<string[]>; // string
type Flatten2<T> = T extends Array<infer P> ? P : never;
type Str = Flatten2<string[]>; // string
```

distributive conditional types

```ts
type ToArr1<T> = T extends any ? T[] : never;
type A = ToArr1<string | number>; // string[] | number[]
type ToArr2<T> = [T] extends any ? T[] : never;
type B = ToArr2<string | number>; // (string | number)[]
type ToArr3<T> = T[];
type C = ToArr3<string | number>; // (string | number)[]
```

use with generics

```ts
type NotNullable<T> =
  T extends null | undefined ? never : T
type Payload = number | string | null | undefined;
type Value = NotNullable<Payload>; // number | string
```

we can produce really useful types

```ts
type Return<T> =
  T extends (...args: any) => infer R
    ? R
    : never;
type A = Return<() => number>; // number
type B = Return<(a: string) => string>; // string
type C = Return<(a: number) => string[]>; // string[]
type D = Return<string>; // never
```

# 6. Types Manipulation

- template literal types

based on string literals

```
type World = 'world'; // 'world';
type Greeting = `hello ${World}`; // 'hello world';
```

based on string literals

```
type EventsListener<T> = {
  on<K extends string & keyof T>(eventName: `${K}Changed`,
};
function makeWatchedObject<T>(obj: T): T & EventsListener<
const person = makeWatchedObject({ name: 'Saoirse', age: 2
person.on('notExistingChanged', () => {}); // Argument of
// parameter of type '"nameChanged" | "ageChanged"'.
person.on('nameChanged', newName => { // newName: string;
  console.log(`First letter of new name is: ${newName[0]}`
});
person.on('ageChanged', newAge => { // newAge: number;
  if (newAge < 0) {
    console.warn('warning! negative age');
  }
  newAge.slice(1); // Property 'slice' does not exist on t
});
```

distribution

```
type H = 'A' | 'B' | 'C' | 'D' | ...;
type V = '1' | '2' | '3' | '4' | ...;
type Chess = `${H}${V}`; //'A1' | 'B1' |... | 'A2' |...
```

# 6. Types Manipulation

- mapped types

"in" keyword

```ts
type OptionsFlags<T> = {
  [_ in keyof T]: boolean;
};
type AppHandlers = {
  darkMode: () => void;
  customLocale: () => void;
}
type AppOptions = OptionsFlags<AppHandlers>;
// { darkMode: boolean; customLocale: boolean; }
```

"-" modifier

```ts
type Writable<T> = {
  -readonly [Key in keyof T]: T[Key];
};
type LockedAccount = {
  readonly id: string;
  readonly name: string;
};
type UnlockedAccount = Writable<LockedAccount>;
// { id: string; name: string; }
```

"+" modifier

```ts
type Optional<T> = {
  [Key in keyof T]+?: T[Key]
};
type AccountDetails = {
  id: string;
  name: string;
};
type OptionalAccountDetails = Optional<AccountDetails>;
// { id?: string; name?: string; }
```

key remapping - "as" keyword

```ts
type Getters<T> = {
    [K in keyof T as `get_${K}`]: () => T[K]
};
type Person = {
  name: string;
  age: number;
}
type PersonGetters = Getters<Person>;
// { get_name: () => string; get_age: () => number; }
```

# 6. Types Manipulation

- mapped types

mapped types work well with other features regarding types manipulation

```ts
type FilterNotSensitive<T, K extends keyof T> = T[K] extends { sensitive: true } ? K : never;
type ExtractSensitiveKeys<T> = {
  [K in keyof T as FilterNotSensitive<T, K>]: T[K];
};
type DBFields = {
  id: { type: number; };
  name: { type: string; sensitive: true };
  location: { type: string; sensitive: true };
  gender: { type: string; };
};
type SensitiveKeys = ExtractSensitiveKeys<DBFields>;
// {
//   name: { type: string; sensitive: true };
//   location: { type: string; sensitive: true };
// }
```

# 6. Types Manipulation

## Utility types

`Partial<T>`:

```ts
interface Todo {
  title: string;
  description: string;
}
const payload: Partial<Todo> = {
  description: 'throw out trash'
};
```

`Required<T>`:

```ts
interface Props {
  a?: number;
  b?: string;
}
const obj1: Props = { a: 5 }; // OK
const obj2: Required<Props> = { a: 5, b: 'a' }; // OK
const obj3: Required<Props> = { a: 5 }; // Error
```

`Readonly<T>`:

```ts
interface Todo {
  title: string;
}
const todo: Readonly<Todo> = {
  title: "Delete inactive users",
};
todo.title = "Hello"; // Error
```

`Record<K, T>`:

```ts
type AppHandlers = {
  darkMode: () => void;
  customLocale: () => void;
}
const opts: Record<keyof AppHandlers, boolean> = {
  darkMode: true,
  customLocale: false,
};
```

# 6. Types Manipulation

## Utility types

`Pick<T, K>`:

```ts
interface Todo {
  title: string;
  description: string;
  done: boolean;
}
type TodoPreview = Pick<Todo, 'title' | 'done'>;
const todo: TodoPreview = { title: "TS", done: false };
```

`Omit<T, K>`:

```ts
interface Todo {
  title: string;
  description: string;
  done: boolean;
}
type TodoPreview = Omit<Todo, 'description'>;
const todo: TodoPreview = { title: "TS", done: false };
```

`Extract<T, K>`:

```ts
type T1 = Extract<'a' | 'b' | 'c', 'a' | 'b'>;
// 'a' | 'b'
type T2 = Extract<'a' | 'b' | 'c', 'a' | 'b' | 'x'>;
// 'a' | 'b'
type T3 = Extract<0 | 'a' | 'b' | '1', number>;
// 0
```

`Exclude<T, K>`:

```ts
type AB1 = Exclude<'a' | 'b' | 'c', 'a' | 'b'>;
// 'c'
type AB2 = Exclude<'a' | 'b' | 'c', 'a' | 'b' | 'x'>;
// 'c'
type Z = Exclude<0 | 'a' | 'b' | '1', number>;
// 'a' | 'b' | '1'
```

# 6. Types Manipulation

## Utility types

`Parameters<T>`:

```ts
type T0 = Parameters<() => string>;
// []
type T1 = Parameters<(a: number, b: string ) => string>;
// [a: number, b: string]
```

`ReturnType<T>`:

```ts
type T0 = ReturnType<() => string>;
// string
type T1 = ReturnType<(s: string) => void>;
// void
```

`ConstructorParameters<T, K>`:

```ts
class Reference {
  id: string;
  constructor(id: string) {
    this.id = id;
  }
}
type T0 = ConstructorParameters<typeof Reference>;
// [id: string]
type T1 = ConstructorParameters<ErrorConstructor>;
// [message?: string]
```

`InstanceType<T, K>`:

```ts
function factory<T extends new () => any>(
  c: T
): InstanceType<T> {
    return new c();
}

class A {
}
let a = factory(A);
// A;
```

# 6. Types Manipulation

## Utility types

`Uppercase<T>`:

```ts
type Greeting = 'Hello, world';
// 'Hello, world'
type ShoutyGreeting = Uppercase<Greeting>;
// 'HELLO, WORLD'
```

`Lowercase<T>`:

```ts
type Greeting = 'Hello, world';
// 'Hello, world'
type QuietGreeting = Lowercase<Greeting>;
// 'hello, world'
```

`Capitalize<T>`:

```ts
type LowercaseGreeting = 'hello, world';
// 'hello, world'
type Greeting = Capitalize<LowercaseGreeting>;
// 'Hello, world'
```

`Uncapitalize<T>`:

```ts
type UppercaseGreeting = 'HELLO, WORLD';
// 'HELLO, WORLD'
type Greeting = Uncapitalize<UppercaseGreeting>;
// 'hELLO, WORLD'
```

`NonNullable<T>`:

```ts
type T0 = NonNullable<string[] | null | undefined>;
// string[]
```

List of all utility types:
https://www.typescriptlang.org/docs/handbook/utility-types.html

exercises/6-types-manipulation.ts

# 7. More on functions

# 7. More on functions

call signatures

```
type DescribableFunction = {
  description: string;
  (someArg: number): boolean;
};
function doSomething(fn: DescribableFunction) {
  console.log(`${fn.description} returned ${fn(6)}`);
}
```

we can combine both

```
interface DateConstructor {
  new (): Date;
  (): string;
}
```

optional parameters in callbacks

```
function forEach(arr: any[], callback: (arg: any, index?:
  for (let i = 0; i < arr.length; i++) {
    callback(arr[i], i);
```

construct signatures

```
type SomeConstructor = {
  new (s: string): SomeObject;
};
function fn(ctor: SomeConstructor) {
  return new ctor('hello');
}
```

sometimes we have to specify type arguments

```
function concat<T>(arr1: T[], arr2: T[]): T[] {
  return arr1.concat(arr2);
}
const a1 = concat([1, 2, 3], ['a']); // → Type 'string'
// is not assignable to type 'number'.
const a2 = concat<string | number>([1, 2, 3], ['a']);
```

# 7. More on functions

## Guidlines on writing good generic functions

### If possible, use the type parameter itself rather than constraining it

```ts
function firstElement1<T extends any[]>(arr: T) {
  return arr[0];
}
function firstElement2<T>(arr: T[]) {
  return arr[0];
}
const a = firstElement1([1, 2, 3]); // a: any ✗
const b = firstElement2([1, 2, 3]); // b: number ✓
```

### If a type param appears in one place, reconsider if you really need it

```ts
function len1<T extends { length: number }>(item: T) {
  console.log(item.length);
}
function len2(item: { length: number }) {
  console.log(item.length);
}
len1([1, 2, 3]);
len1('12345');  // works, but len1 is overcomplicated ✗
len2([1, 2, 3]);
len2('12345'); // ✓
```

### Always use as few type parameters as possible

```ts
function filter1<T, F extends (arg: T) => boolean>(
  arr: T[],
  func: F
): T[] {
  return arr.filter(func);
}

function filter2<T>(
  arr: T[],
  func: (arg: T) => boolean
): T[] {
  return arr.filter(func);
}
const arr = [1, 2, 3, 4];
filter1<number>(arr, (a) => a % 2 === 0); // → Expected
// 2 type arguments, but got 1. ✗
filter2<number>(arr, (a) => a % 2 === 0); // ✓
```

# 7. More on functions

## function overloading

```ts
function makeDate(timestamp: number): Date;
function makeDate(m: number, d: number, y: number): Date;
function makeDate(mOrTimestamp: number, d?: number,
  y?: number
): Date {
  if (d !== undefined && y !== undefined) {
    return new Date(y, mOrTimestamp, d);
  } else {
    return new Date(mOrTimestamp);
  }
}
const d1 = makeDate(12345678);
const d2 = makeDate(5, 5, 5);
const d3 = makeDate(1, 3); // No overload expects 2
// arguments, but overloads do exist that expect either
// 1 or 3 arguments.
```

## types in implementation should match the signature

```ts
function fn(x: boolean): void;
function fn(x: string): void; // This overload signature

// is not compatible with its implementation signature
```

## if possible, use unions rather than overloads

```ts
function len1(s: string): number;
function len1(arr: any[]): number;
function len1(x: any) {
  return x.length;
}
len1(''); // OK
len1([0]); // OK
len1(Math.random() > 0.5 ? 'hello' : [0]); // Error!
function len2(x: any[] | string) {
  return x.length;
}
len2(Math.random() > 0.5 ? 'hello' : [0]); // OK
```

## implementation is not a signature

```ts
function fn(x: string): void;
function fn() {}
fn(); // Expected 1 arguments, but got 0.
```

## parameter destructuring

exercises/7-more-on-functions.ts

# 8. Mixins

# 8. Mixins

Let’s start with Sprite

```
class Sprite {
  name = '';
  x = 0;
  y = 0;
  setPos(x: number, y: number) {
    this.x = x;
    this.y = y;
  }
  constructor(name: string) {
    this.name = name;
  }
}
```

and create Scale mixin

```
type AnyClassConstructor = new (...args: any[]) => {};
function Scale<T extends AnyClassConstructor>(Base: T) {
  return class Scaling extends Base {
    scale = 1;
    setScale(scale: number) {
      this.scale = scale;
    }
  };
}
const EightBitSprite = Scale(Sprite);
const flappySprite = new EightBitSprite('Bird');
flappySprite.setScale(0.8);
console.log(flappySprite.scale);
```

generic constructor

```
type ConstrainedClassConstructor<T = {}> = new (...args: a
type Positionable = ConstrainedClassConstructor<{  setPos:
function Jumpable<T extends Positionable>(Base: T) {
  return class Jumpable extends Base {
    jump() { this.setPos(0, 20); }
  };
```

exercises/8-mixins.ts

# 9. Decorators

# 9. Decorators

Decorators have three primary capabilities:

- They can replace the value that is being decorated with a matching value that has the same semantics. (e.g. a decorator can replace a method with another method, a class with another class, and so on).
- They can provide access to the value that is being decorated via accessor functions which they can then choose to share.
- They can initialize the value that is being decorated, running additional code after the value has been fully defined. In cases where the value is a member of class, then initialization occurs once per instance.

Long story short, decorators can be used to do metaprogramming and add functionality to a value, without fundamentally changing its external behavior.

In TypeScript decorator can be attached to a class declaration, method, accessor, property, or parameter.

# 9. Decorators

decorators are essentialy functions...

```
function Prop(target) {
  // do something with 'target' ...
}
```

... or, if customization is a concern, functions factory

```
function Component(name: string) {
  return function (target) {
    // do something with 'name' and 'target' ...
  };
}
```

that can be used to annonate classes, methods, accessors, properties, or parameters

```
@Component('name')
class App extends Vue {
  @Prop
  name!: string;
}
```

composition - order of declaration and execution

```
function first() {
  console.log('first(): factory evaluated');  // 1
  return function(
    target: any,
    propertyKey: string,
    descriptor: PropertyDescriptor
  ) { console.log('first(): called'); };      // 4
}

function second() {
  console.log('second(): factory evaluated'); // 2
  return function(
    target: any,
    propertyKey: string,
    descriptor: PropertyDescriptor
  ) { console.log('second(): called'); };     // 3
}

class ExampleClass {
  @first()
  @second()

  method() {}
```

# 9. Decorators

Decorators execute only once, when a class definition is first evaluated at runtime:

```
function f(Base: any) {
  console.log('apply decorator');
  return Base;
} // logs, even though we don't have any instance of A
@f class A {}
```

```
evaluate:  Instance Method
evaluate:  Instance Method Parameter
call:  Instance Method Parameter
call:  Instance Method
evaluate:  Instance Property
call:  Instance Property
evaluate:  Static Property
call:  Static Property
evaluate:  Static Method
evaluate:  Static Method Parameter
call:  Static Method Parameter
call:  Static Method
evaluate:  Class Decorator
evaluate:  Constructor Parameter
call:  Constructor Parameter
call:  Class Decorator
```

The evaluation order of different types of decorators is well-defined:

```
function f(key: string): any {
  console.log('evaluate: ', key);
  return function () {
    console.log('call: ', key);
  };
}
@f('Class Decorator')
class C {
  @f('Static Property')
  static prop?: number;

  @f('Static Method')
  static method(@f('Static Method Parameter') foo) {}

  constructor(@f('Constructor Parameter') foo) {}

  @f('Instance Method')
  method(@f('Instance Method Parameter') foo) {}

  @f('Instance Property')
  prop?: number;
}
```

# 9. Decorators

## Class decorators

### Type declaration

```
type Constructor = new (...args: any[]) => {};
type ClassDecorator = <T extends Constructor>(target: T) =
```

It could be suitable for extending class with new properties...

```
function Serializable<T extends Constructor>(Base: T) {
  return class extends Base {
    toString() {
      return JSON.stringify(this);
    }
  };
}

@Serializable
class Foo {
  public foo = 'foo';
  public bar = 24;
}

console.log(new Foo().toString());
// → "{"foo":"foo","bar":24}"
```

... but decorators do not change type of the returned class!

```
type Constructor = new (...args: any[]) => {};
function Serializable<T extends Constructor>(Base: T) {
  return class extends Base {
    serialize() {
```

# 9. Decorators

## Class decorators - `Component`

```ts
type VClass<V> = new (...args: any[]) => V & Vue;
function Comp<V extends Vue>(opt: ComponentOptions<V>) {
  return function<VC extends VClass<V>>(Comp: VC): VC {
    const p = Comp.prototype;
    const data: any = {};
    opt.name = Comp.name; // name
    Object.getOwnPropertyNames(p).forEach((i) => {
      const de = Object.getOwnPropertyDescriptor(p, i)!;
      if (de.get || de.set) { // computed
        opt.computed[i] = { get: de.get, set: de.set};
      } else if (typeof de.value === 'function') {
        opt.methods[i] = de.value; // methods
      } else { data[i] = de.value; }
    });
    opt.data = function(this: Vue) {
      Object.getOwnPropertyNames(data).forEach((i) => {
        Object.defineProperty(this, i, {
          get: () => data[i],
          set: value => { data[i] = value },
        });
      });
    };
    return Vue.extend(opt);
  }
}
```

```ts
@Comp({
  components: {
    OtherComponent,
  }
})
export default class Counter extends Vue {
  count = 0;
  increment() {
    this.count++;
  }
  get label() {
    return `Count: ${this.count}`;
  }
}
// export default Vue.extend({
//   name: 'Counter',
//   components: { OtherComponent },
//   data() { return { count: 0 } },
//   computed: {
//     label() { return `Count: ${this.count}`; }
//   },
//   methods:  { increment() { this.count++; } }
// });
```

# 9. Decorators

## Method decorators

### Type declaration

```
type MethodDecorator = <T>(target: Object, propertyKey: string | symbol, descriptor: TypedPropertyDescriptor<T>) =>
  TypedPropertyDescriptor<T> | void;
```

Just like with class, you can alter the method behavior

```
function Logger(target: unknown, propertyKey: string, descriptor: TypedPropertyDescriptor<( ...args: any[]) => any>) {
  const originalMethod = descriptor.value!;
  descriptor.value = function ( ...args) {
    console.log('params: ', ...args);
    const result = originalMethod.call(this, ...args);
    console.log('result: ', result);
    return result;
  }
}
class C {
  @Logger
  add(x: number, y:number) { return x + y; }
}
const c = new C();
c.add(1, 2);
// → params: 1, 2
// → result: 3
```

# 9. Decorators

## Method decorators - `Watch`

```
function Comp<V extends Vue>(opt: ComponentOptions<V>) {
  return function<VC extends VClass<V>>(Comp: VC): VC {
    // ...
    Comp.__methodDecorators__.forEach(fn => fn(opt));
    return Vue.extend(opt);
  }
}
```

```
@Comp()
export default class Answer extends Vue {
  question = '';
  answer = '';
  @Watch('question')
  answer() {
    this.answer = 'Answer';
  }
}
// export default Vue.extend({
//   name: 'Answer ',
//   data() { return { answer: '', question: '' } },
//   watch: { question: [{ handler: 'answer' }] },


//   methods: { answer() { this.answer = 'Answer' } }
```

```
function Watch(name: string, opts: WatchOptions = {}) {
  return function(Comp: Vue, key: string) {
    Comp.__methodDecorators__.push((options) => {
      const watch = options.watch;
      if (
        typeof watch[path] === 'object'
        && !Array.isArray(watch[path])
      ) {
        watch[path] = [watch[path]];
      } else if (typeof watch[path] === 'undefined') {
        watch[path] = [];
      }
      watch[path].push({ handler: key, ...opts });
    });
  }
}
```

# 9. Decorators

## Accessor decorators

### Type declaration

```
type AccessorDecorator = <T>(target: Object, propertyKey: string | symbol, descriptor: TypedPropertyDescriptor<T>) ⇒
  TypedPropertyDescriptor<T> | void;
```

Just like with class, you can alter the method behavior

```
function immutable(target: unknown, propertyKey: string, descriptor: PropertyDescriptor) {
  const original = descriptor.set!;
  descriptor.set = function (value: any) {
    return original.call(this, { ...value })
  }
}
class C {
  private _point = { x: 0, y: 0 };
  @immutable
  set point(value: { x: number, y: number }) { this._point = value; }
  get point() { return this._point; }
}
const c = new C();
const point = { x: 1, y: 1 }
c.point = point;
console.log(c.point === point); // → false
```

# 9. Decorators

## Parameter Decorators

### Type declaration

```
type ParameterDecorator = (target: Object, methodName: string | symbol, parameterIndex: number) ⇒ void;
```

Parameter decorator itself can't do much. But it can save information about parameters to be used in other decorators:

```
function NotNull(target: unknown, methodName: string, parameterIndex: number) {
  Validator.registerNotNull(target, methodName, parameterIndex);
}
function Validate(target: unknown, methodName: string, descriptor: PropertyDescriptor) {
  const originalMethod = descriptor.value;
  descriptor.value = function (...args: any[]) {
    if (!Validator.performValidation(target, methodName, args)) {
      console.log(`Validation failed for method: ${methodName}`); return;
    }
    return originalMethod.call(this, ...args);
  }
}
class Task {
  @Validate
  run(@NotNull name: string): void {
    console.log("running task, name: " + name);
  }
}
```

# 9. Decorators

## Property Decorators

### Type declaration

```
type PropertyDecorator = (target: Object, propertyKey: string | symbol) ⇒ void;
```

Note the return type - the returned value is ignored, so we cannot modify the property in the decorator!

```
function Prop(options: PropOptions) {
  return function(Comp: Vue, key: string) {
    Comp.__propertyDecorators__.push((options) ⇒ {
      options.props[key] = options;
    });
  }
}

@Comp()
export default class Hello extends Vue {
  @Prop({ type: String })
  greeting!: string;
}
// export default Vue.extend({
//   name: 'Hello ',
//   props: { greeting: { type: String } },
// });
```

# 9. Decorators

## Metadata

Is there a way to avoid this type annotation duplication?

```
function Prop(options: PropOptions) {
  return function(Comp: Vue, key: string) {
    Comp.__propertyDecorators__.push((options) => {
      options.props[key] = options;
    });
  }
}

@Comp()
export default class Hello extends Vue {
  @Prop({ type: String })
  greeting!: string;
}
```

Yes, with reflect-metadata

```
import 'reflect-metadata';
function Prop(options: PropOptions) {
  return function(Comp: Vue, key: string) {
    Comp.__propertyDecorators__.push((options) => {
      const type = Reflect
        .getMetadata('design:type', Comp, key);
      options.props[key] = {
        type,
        ... options
      };
    });
  }
}

@Comp()
export default class Hello extends Vue {
  @Prop()
  greeting!: string;
}
```

`emitDecoratorMetadata` option adds the following information during compilation:

- `design:type`
- `design:paramtypes`
- `design:returntype`

exercises/9-decorators.ts

# 10. Modules and namespaces

# 10. Modules and namespaces

## Modules - way of organizing the code

`export` keyword

```ts
// @filename: ZipCodeValidator.ts
export const regex = /^[0-9]+$/;
export class ZipCodeValidator {
  isAcceptable(s: string) {
    return s.length === 5 && regex.test(s);
  }
}
```

export statements

```ts
// @filename: ZipCodeValidator.ts
export const regex = /^[0-9]+$/;
class ZipCodeValidator {
  isAcceptable(s: string) {
    return s.length === 5 && regex.test(s);
  }
}
export { ZipCodeValidator };
export { ZipCodeValidator as mainValidator };
```

`import` keyword

```ts
// @filename: main.ts
import { ZipCodeValidator } from './ZipCodeValidator';
let myValidator = new ZipCodeValidator();
```

we can also rename imports

```ts
// @filename: main.ts
import { mainValidator as v } from './ZipCodeValidator';
let myValidator = new v();
```

`*` for importing whole module

```ts
// @filename: main.ts
import * as validator from './ZipCodeValidator';
let myValidator = new validator.ZipCodeValidator();
```

importing for side effects only

```ts
// @filename: main.ts
import './my-module';
```

# 10. Modules and namespaces

## Modules - way of organizing the code

we can also export types

```ts
// @filename: test.ts
export const x = /^[0-9]+$/;
export interface Validator {
  isAcceptable(s: string): boolean;
}
export type Res = 404 | 500;
```

re-exporting shorthand - `export ... from ...`

```ts
// @filename: test2.ts
export { Validator } from './test';
// equivalent to:
// import { Validator } from './test';
// export { Validator }
export * from './test';
// equivalent to:
// import { x, Validator, Res } from './test';
// export { x, Validator, Res }
export * as mod from './test';
// equivalent to:
// import * as mod from './test';
// export mod;
```

and import them

```ts
// @filename: main.ts
import { Validator } from './test';
import type { Validator, Res } from './test';
import { x, type Validator, type Res } from './test';
```

`export default` keyword

```ts
// @filename: validator.ts
export default class ZipCodeValidator {
  static numberRegexp = /^[0-9]+$/;
  isAcceptable(s: string) {
    return ZipCodeValidator.numberRegexp.test(s);
  }
}
// @filename: main.ts
import myValidator from './validator';
let valid = new myValidator();
```

There can be one default export in a file.

# 10. Modules and namespaces

## Modules resolution

There are two module resolution strategies, that you can control with `moduleResolution` option:

- classic

Relative imports are resolved relative to the imported file.

```
//@file /root/src/A.ts
import { b } from './moduleB';
```

results in the following lookups:

```
/root/src/moduleB.ts
/root/src/moduleB.d.ts
```

In non-relative imports compiler walks up the directory tree.

```
//@file /root/src/A.ts
import { b } from 'moduleB';
```

results in the following lookups:

```
/root/src/moduleB.ts
/root/src/moduleB.d.ts
/root/moduleB.ts
/root/moduleB.d.ts
/moduleB.ts
/moduleB.d.ts
```

# 10. Modules and namespaces

## Modules resolution

There are two module resolution strategies, that you can control with `moduleResolution` option:

- node (tries to mimick node.js module resoluton, hence name)

Relative imports are resolved relative to the imported file.

```
//@file /root/src/A.ts
import { b } from './moduleB';
```

results in the following lookups:

```
/root/src/moduleB.ts
/root/src/moduleB.tsx
/root/src/moduleB.d.ts
/root/src/moduleB/package.json
(if it specifies a "types" property)
/root/src/moduleB/index.ts
/root/src/moduleB/index.tsx
/root/src/moduleB/index.d.ts
```

In non-relative imports compiler walks up the directory tree.

```
//@file /root/src/A.ts
import { b } from 'moduleB';
```

results in the following lookups:

```
/root/src/node_modules/moduleB.ts
/root/src/node_modules/moduleB.tsx
/root/src/node_modules/moduleB.d.ts
/root/src/node_modules/moduleB/package.json
(if it specifies a "types" property)
/root/src/node_modules/@types/moduleB.d.ts
/root/src/node_modules/moduleB/index.ts
/root/src/node_modules/moduleB/index.tsx
/root/src/node_modules/moduleB/index.d.ts
/root/node_modules/moduleB.ts
..
/node_modules/moduleB.ts
..
```

# 10. Modules and namespaces

## Modules resolution

There is `baseUrl` option that allows you to write relative imports as absolute relatively to the path defined in that option. For example, if `baseUrl: '.'` then with files: `src/views/example/components/hello.ts` and `src/utils/sort.ts`

we can replace the following import in `hello.ts`:

```
import sort from '../../../utils/sort';
```

with

```
import sort from 'src/utils/sort';
```

We can also provide custom path mappings with `paths` option. All paths all resolved relative to the `baseUrl`. Typical setup:

```
{
  "baseUrl": ".",
  "paths": {
    "@/*": ["src/*"],
  },
}
```

```
import App from '@/views/components/App';
// resolved to <root>/src/views/components/App.ts
import router from '@/router';
// resolved to <root>/src/router.ts
import store from '@/store';
// resolved to <root>/src/store/index.ts
```

There is also `rootDirs` option, that makes the compiler creating "virtual" directory from provided roots:

```
{

  "rootDirs": ["src/views", "generated"]
```

```
src

└── views
```

# 10. Modules and namespaces

## Namespaces

`namespace` keyword

```ts
namespace Validation {
  export interface Validator {
    isAcceptable(s: string): boolean;
  }
  const lettersRegexp = /^[A-Za-z]+$/;
  const numberRegexp = /^[0-9]+$/;
  export class LettersValidator implements Validator {
    isAcceptable(s: string) {
      return lettersRegexp.test(s);
    }
  }
  export class ZipCodeValidator implements Validator {
    isAcceptable(s: string) {
      return s.length === 5 && numberRegexp.test(s);
    }
  }
}
let strings = ['Hello', '98052', '101'];
let xs: Record<string, Validation.Validator> = {};
xs["ZIP"] = new Validation.ZipCodeValidator();
xs["Letters"] = new Validation.LettersValidator();
```

spread over multiple files with reference tags

```ts
// @filename: Validation.ts
namespace Validation {
  export interface Validator {
    isAcceptable(s: string): boolean;
  }
}
```

```ts
// @filename: LettersValidator.ts
/// <reference path="Validation.ts" />
namespace Validation {
  const re = /^[A-Za-z]+$/;
  export class LettersValidator implements Validator {
    isAcceptable(s: string) { return re.test(s); }
  }
}
```

```ts
/// <reference path="Validation.ts" />
/// <reference path="LettersValidator.ts" />
let strings = ['Hello', '98052', '101'];
let xs: Record<string, Validation.Validator> = {};
xs["ZIP"] = new Validation.ZipCodeValidator();
xs["Letters"] = new Validation.LettersValidator();
```

# 10. Modules and namespaces

## Namespaces

You can put aliases for namespaces

```
namespace Shapes {
  export namespace Polygons {
    export class Triangle {}
    export class Square {}
  }
}
import polygons = Shapes.Polygons;
let sq = new polygons.Square();
// Same as 'new Shapes.Polygons.Square()'
```

# 10. Modules and namespaces

- Modules provide for better code reuse, stronger isolation and better tooling support for bundling It is also worth noting that, for Node.js applications, modules are the default. Starting with ECMAScript 2015, modules are native part of the language, and should be supported by all compliant engine implementations.
- Namespaces are simply named JavaScript objects in the global namespace. This makes namespaces a very simple construct to use. Unlike modules, they can span multiple files, and can be concatenated using outFile.

It’s usually better to use modules over namespaces in modern code, especially for new projects. However there is one very good use case for namespaces - globally exposed type declaration files.

Needless Namespacing

```
export namespace Shapes {
  export class Triangle {  /* ... */ }
  export class Square { /* ... */ }
}
```

```
import * as shapes from "./shapes";
let t = new shapes.Shapes.Triangle(); // shapes.Shapes?
```

exercises/10-modules-and-namespaces.ts

# 11. Declaration files

# 11. Declaration files

Declaration files are files with `.d.ts` extension, that contain only type declarations, without any implementation.

```
declare namespace D3 {
  export interface Selectors {
    select: {
      (selector: string): Selection;
      (element: EventTarget): Selection;
    };
  }
  export interface Event {
    x: number;
    y: number;
  }
  export interface Base extends Selectors {
    event: Event;
  }
}
declare var d3: D3.Base;
```

# 11. Declaration files

With `declaration` option, type declaration file is automatically created and emitted by the compiler along with executable JavaScript.

```ts
//@file index.ts
function d((timestamp: number): Date;
function d((m: number, d: number, y: number): Date;
function d((mOrTimestamp: number, d?: number, y?: number
): Date {
  if (d !== undefined && y !== undefined) {
    return new Date(y, mOrTimestamp, d);
  } else { return new Date(mOrTimestamp); }
}
```

```ts
//@file index.js
function d((mOrTimestamp, d, y) {
  if (d !== undefined && y !== undefined) {
    return new Date(y, mOrTimestamp, d);
  } else { return new Date(mOrTimestamp); }
}
```

```ts
//@file index.d.ts
declare function d(timestamp: number): Date;
declare function d(m: number, d: number, y: number): Date
```

If type definitions file is provided in `"types"` field in package.json is automatically picked up by typescript module resolution algorithm.

```ts
{
  "name": "package-name",
  "version": "1.0.0",
  "types": "./index.d.ts",
  "typesVersions": {
    "≥3.2": { "*": ["ts3.2/*"] },
    "≥3.1": { "*": ["ts3.1/*"] }
  }
}
```

You can also write type definitions manually and publish them separately. It's not that uncommon, there are a lot of libraries written in pure javascript with separate type definition files. Usually you can recognize type definition packages by the name, which starts with `@types`.

```ts
{
  "@types/express": "^4.17.13",
}
```

# 11. Declaration files

module libraries

```
import * as path from 'path';
import { readFile } from 'fs';
```

"function" module

```
import assert from 'assert';
assert(7 > 5);
```

"class" module

```
import Vue from 'vue';
new Vue({
  render: (h) => h(App),
}).$mount('#app');
```

"plugin" module

```
import jest from 'jest';
import '@testing-library/jest-dom';
```

global library

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

// Web
window.createGreeting = function (s) {
  return "Hello, " + s;
};
// Node
global.createGreeting = function (s) {
  return "Hello, " + s;
};
// Potentially any runtime
globalThis.createGreeting = function (s) {
  return "Hello, " + s;
};
```

UMD libraries

```
// in module
import moment from 'moment';
console.log(moment.format());
```

# 11. Declaration files

module libraries

```
// You can define functions that can be consumed by the module
export function isPrime(x: number): boolean;
export function isEven(x: number): boolean;
export as namespace mathLib;
// You can declare types that are available via importing the module
export interface Scientific {
  mantissa: number;
  exponent: number;
}
// You can declare properties of the module using const, let, or var
export const PI: number;
```

we can consume it in another module:

```
import { isPrime, type Scientific } from 'math-lib';
const x: Scientific = { mantissa: 1.24, exponent: 12 };
isPrime(2);
mathLib.isPrime(2); // ERROR: can't use the global definition from inside a module
```

or using global variable in script:

# 11. Declaration files

module "function"

```ts
export as namespace isPrime;
export = IsPrime;
declare function IsPrime(x: number): boolean;
declare namespace IsPrime {
  export interface Scientific {
    mantissa: number;
    exponent: number;
  }
}
```

It can be consumed in another module:

```ts
import IsPrime from 'math-lib';
IsPrime(2);
const x: IsPrime.Scientific = {
  mantissa: 1.24, exponent: 12
};
```

or using global variable in script:

```ts
isPrime.IsPrime(2);
```

# 11. Declaration files

## module "class"

```
export as namespace mathLib;
export = MathLib;
declare class MathLib {
  locale = 'en-EN';
  constructor(customLocale?: string);
  isPrime(x: number): boolean;
  isEven(x: number): boolean;
}
declare namespace MathLib {
  export interface Scientific {
    mantissa: number;
    exponent: number;
  }
}
```

It can be consumed in another module:

```
import MathLib from 'math-lib';
const mathLib = new MathLib();
const x: MathLib.Scientific = {
  mantissa: 1.24, exponent: 12
};
```

or using global variable in script:

```
const lib = new mathLib.MathLib();
lib.isPrime(2);
```

# 11. Declaration files

module "plugin"

```
import { greeter } from 'super-greeter'; // Normal Greeter API
greeter(2);
greeter('Hello world');
import 'hyper-super-greeter'; // Now we extend the object with a new function at runtime
greeter.hyperGreet();
```

The definition for “super-greeter”:

```
export interface GreeterFunction {
  (name: string): void
  (time: number): void
}
export const greeter: GreeterFunction;
```

The definition for our plugin "hyper-super-greeter"

```
import { greeter } from 'super-greeter';
export module 'super-greeter' {
  export interface GreeterFunction {
    hyperGreet(): void;
  }
}
```

It uses the fact, that typescript merges the declaration for the modules. so with `export module some_existing_module` we can extend that module!

Declaration merging gives us also one more good use case - module augmenting.

# 11. Declaration files

## Module augmentation - `declare module`

```ts
// our code
export interface Vue {
  readonly $el: Element;
  // ...
  $emit(event: string, ...args: any[]): this;
}
```

```ts
// our code
declare module 'vue/types/vue' {
  export interface Vue {
    $customProperty: number;
  }
}
Vue.prototype.$customProperty = 5;
```

```ts
// component extending class implementing interface above
@Component
export default class Component extends Vue {
  handler() {
    this.$emit('change', this.$customProperty);
    // we can use `$customProperty` without errors!

  }
```

## Global augmentation - `declare global`

```ts
export class Observable<T> {
  // ...
}
declare global {
  interface Array<T> {
    toObservable(): Observable<T>;
  }
}
Array.prototype.toObservable = function () {
  // ...
};
```

## Wildcard module declarations

```ts
import App from './App.vue ';
// → module not found './App.vue' ts(2307)
```

```ts
declare module '*.vue' {
  import Vue from 'vue';
  export default Vue;
}
```

# 11. Declaration files

ambient namespaces in non-module code are globally exposed

```
declare namespace api {
  export interface User {
    name: string;
    surname: string;
    user_id: string;
  }
  // ...
}
```

We can use them in the code without any imports

```
class AuthModule {
  private user: api.User | null = null;

  async authenticate() {
    const { data } = await axios.get<api.User>('/user');
    this.user = data;
  }
}
```

# 11. Declaration files

## global module

```
declare namespace D3 {
  export interface Selectors {
    select: {
      (selector: string): Selection;
      (element: EventTarget): Selection;
    };
  }
  export interface Event {
    x: number;
    y: number;
  }
  export interface Base extends Selectors {
    event: Event;
  }
}
declare var d3: D3.Base; // "declare const" also works if it's read-only
```

It can be consumed without any imports:

```
d3.select('p');
```

exercises/11-declaration-files.ts

# 12. Build tools

# 12. Build tools

We have covered the language specifics, ways to organize and consume the types, but we did not say a word about how to actually build our code.

In order to have the compiler available we need to install npm package called `typescript`. Then we can compile our code using CLI tool called `tsc`.

Emit JS for just the index.ts with the compiler defaults

```
tsc index.ts
```

Emit JS for any .ts files in src, with the default settings

```
tsc src/*.ts
```

Emit JS for app.ts and util.ts files in src, with custom settings

```
tsc app.ts util.ts --target esnext --outfile index.js
```

Compile with look through the fs for `tsconfig.json`

```
tsc
```

Emit files referenced in with the compiler settings from tsconfig.production.json

```
tsc --project tsconfig.production.json
```

# 12. Build tools

The presence of a tsconfig.json file in a directory indicates that the directory is the root of a TypeScript project. The tsconfig.json file specifies the root files and the compiler options required to compile the project.

We can specify list of input files with `files` field:

```
{
  "compilerOptions": { ... },
  "files": [ "index.ts", "core.ts" ]
}
```

There is also `references` field, that allows you to refer to other configuration files in other projects. Then when importing something from that project it uses that project's configuration file. We won't cover it in these workshops though.

If you're interested you can read more about it here: https://www.typescriptlang.org/docs/handbook/project-references.html

or using the `include` and `exclude` properties

```
{
  "compilerOptions": { ... },
  "include": ["src/**/*"],
  "exclude": ["node_modules", "**/*.spec.ts"]
}
```

We can extend other configurations with `extends`

```
{
  "extends": "@vue/tsconfig/tsconfig.web.json",
  "compilerOptions": { ... },
  "include": [
    "src/**/*.ts",
    "src/**/*.vue"


  ]
```

# 12. Build tools

in `compilerOptions` you can provide more than 100 options for the compiler.

`allowUnreachableCode` - `true | false | undefined`

Raises errors (or warning) for unreachable code.

`noImplicitAny` - `true | false`

Raises an error whenever typescript infers any.

`strictNullChecks` - `true | false`

Controls if undefined and null values should be ignored.

`noUnusedLocals` - `true | false`

Report errors on unused local variables.

`noUnusedParameters` - `true | false`

Report errors on unused parameters in functions.

`noUncheckedIndexedAccess` - `true | false`

It will add undefined to any un-declared field in the type that uses index signatures.

`alwaysStrict` - `true | false`

Ensures that your files are parsed in the ECMAScript strict mode, and emit “use strict” for each source file.

`strict` - `true | false`

The strict flag enables a wide range of type checking (including, but no limited to noImplicitAny, strictNullChecks) behavior that results in stronger guarantees of program correctness.

# 12. Build tools

in `compilerOptions` you can provide more than 100 options for the compiler.

`allowJs` - `true | false`

Allows to import js code in the ts files.

`paths` - JSON

Sets the project paths, that can be used in imports.

`moduleResolution` - `node | node12 | classic`

Specify the module resolution strategy.

`declaration` - `true | false`

Generate .d.ts files for your project.

`experimentalDecorators` - `true | false`

Enables experimental support for decorators.

`baseUrl` - `string`

Sets the project base url.

`module` - `esnext | es2022 | umd | ... | commonjs`

Sets the module system for the program.

`types` - `string[]`

Includes listed type definitions in the global scope.

`noEmitOnError` - `true | false`

Do not emit code if any errors were reported.

`target` - `esnext | es2022 | ... | es2015 | es3`

Sets the compilation target. For older, some features might

## All tsconfig options

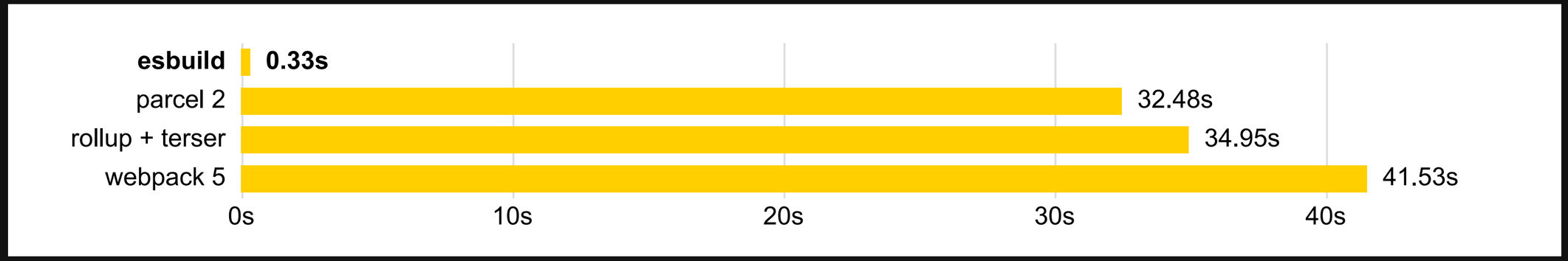
https://www.typescriptlang.org/tsconfig

# 12. Build tools

While you can use TypeScript compiler to produce JavaScript code from TypeScript code, it's also common to use other transpilers to do this.

The most common are:

- Babel - mainly used to convert ECMAScript 2015+ code into a backwards compatible version of JavaScript, but includes its own typescript transpiler
- ESBuild (used in Vite) - very fast compiler, **that does not perform typechecking**:




Most of these alternative transpilers compile file by file, so it's recommended to set previously mentioned `isolatedModules` option in typescript config file

# Bonus #1: TypeScript is unsound

```
const xs = [0, 1, 2];  // xs: number[];
const x = xs[3];  // x: number;
```

```
function messUpTheArray(arr: Array<string | number>): void {
    arr.push(3);
}

const strings = ['foo', 'bar']; // strings: string[];
messUpTheArray(strings);

const s = strings[2]; // s: string;
console.log(s.toLowerCase()); // → s.toLowerCase is not a function
```

There are JS type systems in which code above throws an error, for example Flow.

https://effectivetypescript.com/2021/05/06/unsoundness/

https://www.typescriptlang.org/docs/handbook/type-compatibility.html

https://github.com/JSMonk/hegel

# Bonus #2: TypeScript in browsers?

There is a proposal for TC39 to include type annotation syntax in browsers.

https://devblogs.microsoft.com/typescript/a-proposal-for-type-syntax-in-javascript/

https://github.com/giltayar/proposal-types-as-comments/

If it will become included in the standard most typescript applications could be running in the browser without compilation step!

# Bonus #3: Type checking at runtime

Adds type checking at run time!

https://github.com/fabiandev/ts-runtime

```
let num: number;
num = "Hello World!";
```

compiled into:

```
let _numType = t.number(), num;
num = _numType.assert("Hello World!")
```

- has some limitations
- affects performance (because before execution type assertions are checked)

# Bonus #4: TypeScript type system is really powerful...

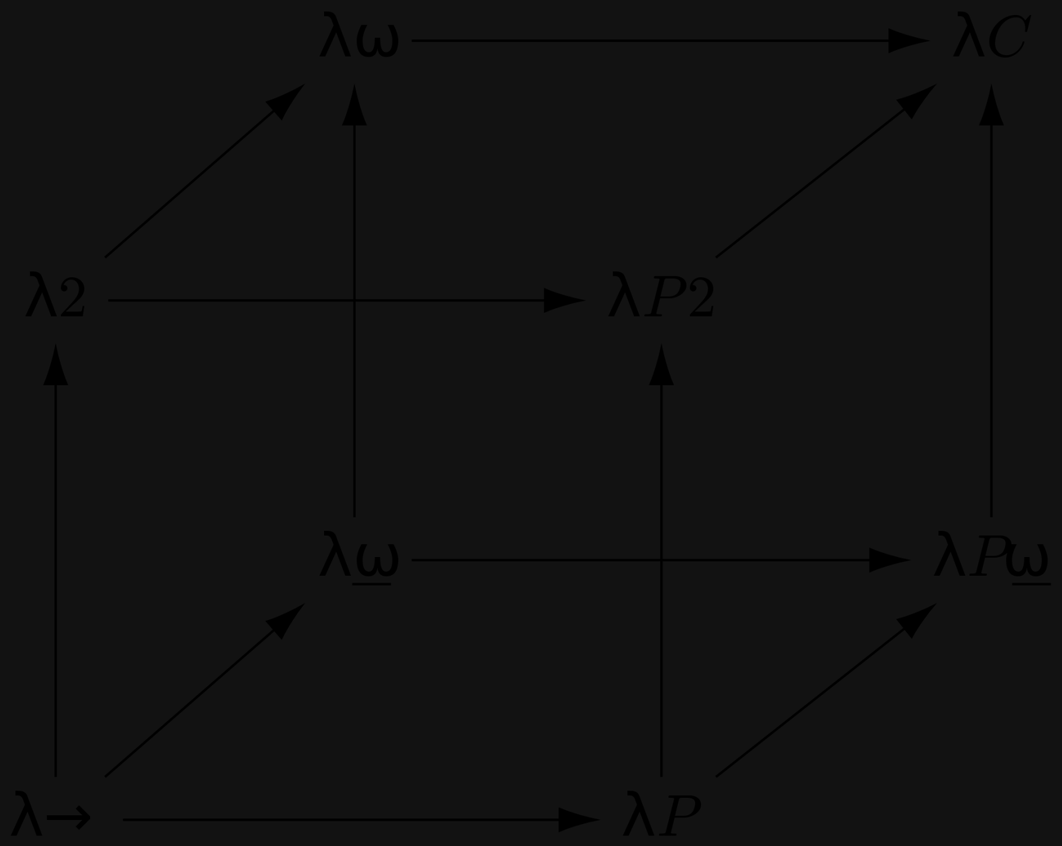
TypeScript is turing complete!

https://github.com/microsoft/TypeScript/issues/14833

https://gist.github.com/hediet/63f4844acf5ac330804801084f87a6d4

https://github.com/fc01/TypeGame

# Bonus #5: ... but not powerful enough!




- z-axis ($\nearrow$): types that can bind types, corresponding to type operators.

  Example: type manipulation techniques
- y-axis ($\uparrow$): terms that can bind types, corresponding to polymorphism.

  Example: overloading, overriding
- x-axis ($\rightarrow$): types that can bind terms, corresponding to dependent types.

  Example: not present in typescript!

```
type BoundedInt(n) = { i:Int | i⩽n }
```

If types can depend on the value, we can write types like "that variable is always smaller than 10", or "that function always returns 2", or "this program works according to specification". This means we can introduce the "formal correctness" of a

program. There are some languages that offer the third option, like Coq (which is $\lambda C$ in the graph). They are using types to