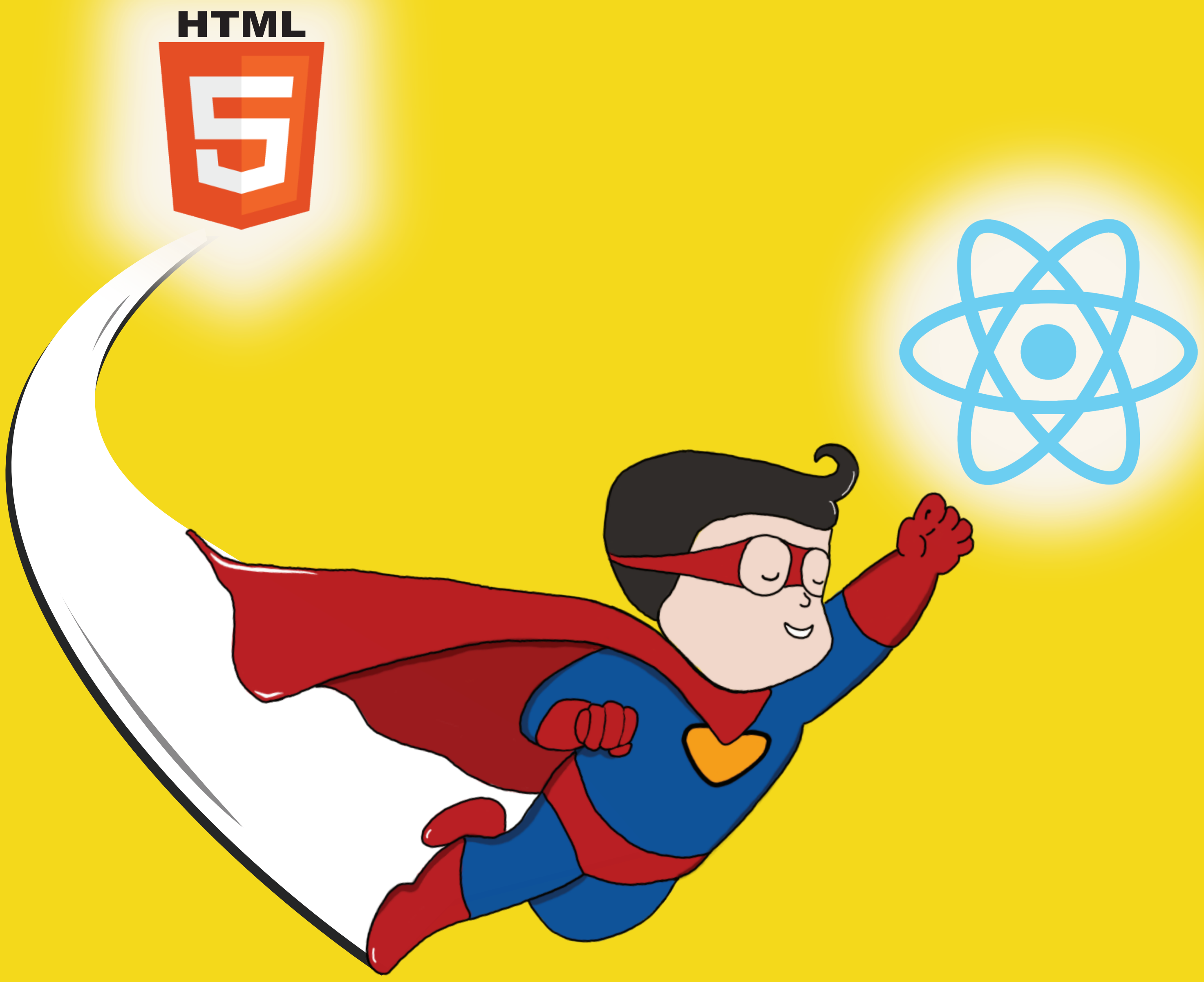

HTML TO REACT

The Ultimate Guide

NGNINJA ACADEMY

TypeScript

Table Of Content

Who created TypeScript?

- Designed by Anders Hejlsberg
- Who is also the designer of C# at Microsoft

What is TypeScript?

- Typed superset of JavaScript
  - Meaning JS plus additional features
- It compiles to plain JavaScript
- TypeScript uses static typing
  - You can give types to your variables now
  - JavaScript instead is super dynamic - doesn't care about types

More on TypeScript

- It is a pure Object oriented language
  - It has classes, interfaces, statically typed
  - Like C# or Java
- Supports all JS libraries and frameworks
- Javascript is basically Typescript
  - That means you can rename any valid `.js` file to `.ts`
- TypeScript is aligned with ES6
  - It has all features like modules, classes, etc

What more is offered by TypeScript?

- Generics
- Type annotations
- Above 2 features are not available in ES6

Overcomes JavaScript drawbacks

- Strong type checking
- Compile time error checks
- Enable IDEs to provide a richer environment

> Basically, TypeScript acts as your buddy programmer when you are pair programming with someone

Why use TypeScript?

- It is superior to its counterparts
  - Like Coffeescript and Dart
  - TypeScript extends JavaScript - they do not
  - They are different language altogether
  - They need language-specific execution env to run
- It gives compilation errors and syntax errors
- Strong static typing
- It supports OOP
  - Classes, interfaces, inheritance, etc.
  - Like Java, c#
- Reacher code hinting due to its typed nature
- Automated documentation

    - Due to its typed nature
    - Therefore, good readability

- No need of custom validation which is clunky for large apps

    - No boilerplate code

Install TypeScript

- Open terminal
- Run following command to install typescript globally
    - You can access it from any folder
- You will need `yarn` or `npm` installed already on your system

```
yarn add -g typescript

// if you are using npm
npm install -g typescript
```

> TIP: You can play around with TypeScript using their official playground

Configure TypeScript

- This allows us to define rule-sets for the typescript compiler
- `tsconfig` file is used for this
    - It is a JSON file
- Run following command to create the config file

```
tsc --init
```

- Open the `tsconfig.json` in IDE
    - My favorite IDE is VSCode
- It will look something like this
    - We will only focus on `compilerOptions` for now

```
{
  "compilerOptions": {
    "target": "es5",
    "noImplicitAny": true,
    "outDir": "public/js"
    "rootDir": "src",
  },
}
```

- `target` specifies the ECMAScript target version
    - The compiled JavaScript will follow `es5` version standards in this case
    - You can set it to `es6`, `es2020`, etc.
- `noImplicitAny` specifies strict typings
    - All variables must be explicitly given types or specify `any` type explicitly
- `outDir` specifies output directory for compiled JavaScript code
- `rootDir` specifies where your typescript files are

Compile TypeScript to JavaScript

- Run the following command from your project root to compile all the typescript files into JavaScript
- It will throw an error if any part of the file is not following the rule-set you defined under `compilerOptions`

```
tsc
```

- You can also compile single file by running the command like below

```
tsc ninjaProgram.ts
```

- You can run compile command on watch mode
- Meaning you don't have to compile files manually
- Typescript will watch your changes and compile them for you automatically

```
tsc -w
```

Data types

- `any`
  - Supertype of all data types
  - Its the dynamic type
  - Use it when you want to opt out of type checking
- Builtin types
  - Number, string, boolean, void, null, undefined
  - Number is double precision -> 64-bit floats
- User defined types
  - Arrays, enums, classes, interfaces

Examples

```
// Variable’s value is set to 'ninja' and type is string
let name:string = 'ninja'

// Variable’s value is set to undefined by default
let name:string

// Variable’s type is inferred from the data type of the value.
// Here, the variable is of the type string
let name = 'ninja'

// Variable’s data type is any.
// Its value is set to undefined by default.
let name
```

More Examples

```
// Variable is a number type with value 10
let num: number = 10

num = 50 // valid operation
num = false // type error
num = 'ninja' // type error
```

```
// Variable is a string array
let arr: string[] = ["My", "first", "string", "array"]

arr.push("add one more string") // valid operation
arr.push(1) // This will throw type error
```

Variable scopes

- Global scope
    - Declare outside the programming constructs
    - Accessed from anywhere within your code

```
const name = "sleepless yogi"

function printName() {
  console.log(name)
}
```

- Class scope
    - Also called fields
    - Accessed using object of class
    - Static fields are also available... accessed using class name

```
class User {
  name = "sleepless yogi"
}

const yogi = new User()

console.log(yogi.name)
```

- Local scope
  - Declared within methods, loops, etc ..
  - Accessible only withing the construct where they are declared
  - `vars` are function scoped
  - `let` are block scoped

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

Class inheritance

- Typescript does not support multiple inheritance
- It supports - single and multi-level inheritance
- `super` keyword is used to refer to the "immediate parent" of a class
- Method overriding is also supported

```
class PrinterClass {
  doPrint():void {
    console.log("doPrint() from Parent called…")
  }
}

class StringPrinter extends PrinterClass {
  doPrint():void {
    super.doPrint()
    console.log("doPrint() is printing a string…")
  }
}

var obj = new StringPrinter()
obj.doPrint()

// outputs
doPrint() from Parent called…
doPrint() is printing a string…
```

Data hiding

- Visibility of data members to members of the other classes
- It's called data hiding or encapsulation
- Access modifiers and access specifiers are used for this purpose
- Public
    - Member has universal accessibility
    - Member are public by default

```
class User {
  public name: string

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

  public getName(theName: string) {
    this.name = theName
  }
}

const yogi = new User("Sleepless Yogi")

console.log(yogi.name) // valid
console.log(yogi.getName()) // valid
```

- Private
    - Member are accessible only within its class

```
class User {
  // private member
  #name: string

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

const yogi = new User("Sleepless Yogi")

console.log(yogi.name) // invalid
```

- Protected
    - Similar to private members
    - But, members are accessible by members within same class and its child classes

```
class User {
  // protected member
  protected name: string

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

class Student extends User {
  private college: string

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

  public getDetails() {
    return `Hello, my name is ${this.name} and I study in
${this.college}.`;
  }
}


const yogiStudent = new Student("Yogi", "Some college")

console.log(yogi.name) // invalid
console.log(yogi.college) // invalid
console.log(yogi.getDetails()) // valid
```

Interface

- This is new in TypeScript that is not present in vanilla JavaScript
- It is a syntactical contract that entity should respect
- Interfaces contain only the declaration of members
    - Deriving class has to define them
- It helps in providing standard structure that deriving class should follow
- It defines a signature which can be reused across objects
- Interfaces are not to be converted to JavaScript
    - It's a typescript thing!

Example

- `IPerson` is an interface
- It defines two members
    - `name` as string
    - `sayHi` as a function

```
interface IPerson {
  name: string,
  sayHi?: () => string
}

// customer object is of the type IPerson
var customer1: IPerson = {
  name: "Ninja",
  sayHi: (): string => { return "Hi there" }
}
```

```
// customer object is of the type IPerson
// optional field sayHi omitted here
var customer2: IPerson = {
  name: "Ninja",
}
```

- We define `IPerson` interface in above example
- The interface declares two properties `name` and `sayHi`
- The object that implements this interface must define these two properties on them
- But notice `sayHi` is followed by `?` question mark - that means `sayHi` property is optional
    - So object can skip the definition for that property
- Hence `customer2` is a valid object that implements our interface

```
Optional members:

You can declass optional fields using the "?" operator

So, in the below example "sayHi" is an optional function

So, the objects that implements that interface need not define the "sayHi"
function
```

Namespaces

- Way to logically group related code
- This is inbuilt into typescript - unlike js
- Let's see an example

```
namespace SomeNameSpaceName {
  export interface ISomeInterfaceName { }

  export class SomeClassName { }
}
```

- Classes or interfaces which should be accessed outside the namespace should be marked with keyword `export`
- Can also define one namespace inside another namespace
- Let's see an example

```
//FileName : MyNameSpace.ts

namespace MyNameSpace {

  // nested namespace
  export namespace MyNestedNameSpace {

    // some class or interfaces
    export class MyClass {

    }
  }
}

// Access the above class somewhere else

/// <reference path = "MyNameSpace.ts" />

const myObject = new MyNameSpace.MyNestedNameSpace.MyClass();
```

Enums

- Allow us to define a set of named numeric constants
- Members have numeric value associated with them
- Let's see an example

```
enum Direction {
  Up = 1,
  Down,
  Left,
  Right
}


let directions = [Directions.Up, Directions.Down, Directions.Left,
Directions.Right]

// generated code will become - in JS

var directions = [0 /* Up */, 1 /* Down */, 2 /* Left */, 3 /* Right */];

let a = Enum.A;
let nameOfA = Enum[Enum.Up]; // "Up"
```

`never` and `unknown` primitive types

- they are both complementary
- `never` is for things that never happen
- ex: use never here because Promise never resolves
    - that is most specific type here
    - using `any` will be ambiguous
    - we would have lost benefits of type-checking
- using `unknown` is also not advised
    - because then we would not be able to do `stock.price`
    - `price` would be not available or known to the type checker

```
function timeout(ms: number): Promise<never> {
  return new Promise((_, reject) =>
    setTimeout(() => reject(new Error("Timeout elapsed")), ms)
  )
}

async function fetchPriceWithTimeout(tickerSymbol: string):
Promise<number> {
  const stock = await Promise.race([
    fetchStock(tickerSymbol), // returns `Promise<{ price: number }>`
    timeout(3000)
  ])
  return stock.price
}
```

- use `never` to prune conditional types
    - prune unwanted cases
- use `unknown` for values that could be anything
    - similar to `any` but not quite
    - it is the type-safe counterpart of `any`
- Anything is assignable to `unknown`, but `unknown` isn't assignable to anything but itself

`any` vs `unknown`

```
let vAny : any = 10 ; // We can assign anything to any
let vUnknown: unknown =  10; // We can assign anything to unknown just
like any


let s1: string = vAny; // Any is assignable to anything
let s2: string = vUnknown; // Invalid we can't assign vUnknown to any
```

```
other type (without an explicit assertion)

vAny.method(); // ok anything goes with any
vUnknown.method(); // not ok, we don't know anything about this variable
```

`void` vs `never`

- `void` return void, `never` never return
- `void` can be thought of as a type containing a single value
    - no means to consume this value though
    - but a `void` function can be thought of returning such value
- `never` is a type containing no values
    - meaning function with this return type can `never` return normally at all
    - either throw exception or reject promise or failing to terminate

Why static type checking

- Catch errors super early
- Increase confidence in code
    - Like adding unit tests
    - Or documentation
    - Or style guides
- Interfacing modules is made easy
    - If you know the types you can just start using things from other modules
    - Without worrying about breaking your code
- It can tell what you can do and what you cannot do
- BUT, beware sometimes it does not give error at all
- Check the example below

```
doSomething(m) {
  // m.count is undefined
  // and undefined not greater than 2
  // so returns "small"
  return m.count > 2 ? 'big' : 'small'
}

doSomething(5) // prints "small"... no error given
```

- Other alternatives to static type checking (flow, typescript)
    - Linters
        - But they are very rudimentary so not 100% sufficient
    - Custom runtime checking
        - It's manual validation, hard to maintain, looks clunky for large apps
        - Plus its only at runtime - not at compile time

```
// custom runtime checking

validate(arr) {
  if(!Array.isArray(arr)) {
    throw new Error('arr must be an Array')
  }
}
```

Type assertion

- You can change type of variables
- Possible only when types are compatible
- So, changing S to T succeed if
    - S is a subtype of T
    - OR...
    - T is a subtype of S
- This is NOT called type casting
    - Because type casting happens at runtime
    - Type assertion is purely compile time
- Example

```
var str = '1'

//str is now of type number
var str2:number = <number> <any> str
```

Generics

- Gives ability to create a component that can work over a variety of types
- Rather than only a single one
- Consumer can then consume these components and use their own types

Simple example

- Below we have two arrays
- `numArr` can take only numbers
- `strArr` can take only strings

```
type numArr = Array<number>;
type strArr = Array<string>;
```

- Basically we can use Generics to create one function to support multiple types
- So you don't have to create same function again and again
- You can use `any` type if you want
    - BUT - then you lose the type definition of your objects
    - That leads to buggy code

```
// you can use any type too
// but you don't get type definitions
// this can lead to buggy code
type anyArr = Array<any>;
```

Advanced example

- Below is a generic function
- It basically returns last element of `T type` of array that you pass to it
- The parameter `arr` accepts arguments of `array type T`
- It returns an element of type `T`

```
const getLastElement = <T>(arr: T[]): T => {
  return arr[arr.length - 1];
};


// T will number
// it will return 3
const lastNum = last([1, 2, 3]);

// T will be string
// it will return "c"
const lastString = last(["a", "b", "c"]);
```

Intersection Types

- It is a way to combine multiple types into one
- Merge `type A, type B, type C, and so on` and make a single type

```
type A = {
  id: number
  foo: string
}

type B = {
  id: number
  bar: string
}

// Merge type A and type B
type C = A & B


const myObject: C = {id: 1, foo: "test", bar: "test"}
```

- If you declare conflicting types it Typescript gives an error
- Example below

```
type A = {
  bar: number
}

type B = {
  bar: string
}

// Merge type A and type B
type C = A & B


const myObject: C = {bar: 1} // error
const myObject: C = {bar: "test"} // error
```

- To accept `bar` as number or string use Union types - explained below

Union Types

- Typescript gives the ability to combine types
- Union types are used to express a value that can be one of the several types
- Two or more data types are combined using the pipe symbol `(|)` to denote a Union Type

```
Type1 | Type2 | Type3
```

- Below example accepts both `strings and numbers` as a parameter

```
var val: string | number

val = 12
console.log("numeric value of val " + val)

val = "This is a string"
console.log("string value of val " + val)
```

Advanced example

- You can combine intersection types and union types to fix the above code snipped
- Like below

```
type A = {
  bar: string | number
}

type B = {
  bar: string | number
}

// Merge type A and type B
```

```
type C = A & B


const myObject: C = {bar: 1} // valid
const myObject: C = {bar: "test"} // valid
```

Partial Type

- It is a utility type
- Can be used to manipulate types easily
- Partial allows you to make all properties of the type T optional

```
Partial<T>
```

- Example

```
type Customer {
  id: string
  name: string
  age: number
}

function addCustomer(customer: Partial<Customer>) {
  // logic to add customer
}

// all of the below calls are valid

addCustomer({ id: "id-1" })
addCustomer({ id: "id-1", name: "NgNinja Academy" })
addCustomer({ id: "id-1", name: "NgNinja Academy", age: 10 })
```

Required Type

- It is also an utility type
- Required allows you to make all properties of the type T required

```
type Customer {
  id: string
  name?: string
  age?: number
}

function addCustomer(customer: Required<Customer>) {
  // logic to add customer
}


addCustomer({ id: "id-1" }) // type error
addCustomer({ id: "id-1", name: "NgNinja Academy" }) // type error
addCustomer({ id: "id-1", name: "NgNinja Academy", age: 10 }) // valid
```

Readonly

- To make the properties read only
- You cannot reassign values to the properties

```
type Customer {
  id: string
  name: string
}

function setCustomer(customer: Readonly<Customer>) {
  customer.name = "New Name"
}

setCustomer({ id: "id-1", name: "NgNinja Academy" })
```

```
// Error: Cannot assign to 'name' because it is a read-only property
```

- You can also set single properties as readonly
- In below example `name` property is set to `readonly`

```
type Customer {
  id: string
  readonly name: string
}

function setCustomer(customer: Customer) {
  customer.name = "New Name"
}

setCustomer({ id: "id-1", name: "NgNinja Academy" })
// Error: Cannot assign to 'name' because it is a read-only property
```

Pick

- Use this to create new type from an existing type `T`
- Select subset of properties from type `T`
- In the below example we create `NewCustomer` type by selecting `id` and `name` from type `Customer`

```
type Customer {
  id: string
  name: string
  age: number
}

type NewCustomer = Pick<Customer, "id" | "name">


function setCustomer(customer: Customer) {
  // logic
}


setCustomer({ id: "id-1", name: "NgNinja Academy" })
// valid call


setCustomer({ id: "id-1", name: "NgNinja Academy", age: 10 })
// Error: `age` does not exist on type NewCustomer
```

Omit

- This is opposite of `Pick` type
- It will omit (remove) the specified properties from type `T`
- In the below example we create `NewCustomer` type by removing `name` and `age` from type `Customer`
- It will only have `id` property

```
type Customer {
  id: string
  name: string
  age: number
}

type NewCustomer = Omit<Customer, "name" | "age">


function setCustomer(customer: Customer) {
  // logic
}


setCustomer({ id: "id-1" })
// valid call

setCustomer({ id: "id-1", name: "NgNinja Academy", age: 10 })
// Error: `name` and `age` does not exist on type NewCustomer
```

Extract

- It will extract common properties from two different types
- In the below example we extract type from `Customer` and `Employee` types
- Common properties are `id` and `name`
- So the new `Person` type will have `id` and `name`

```
type Customer {
  id: string
  name: string
  age: number
}

type Employee {
  id: string
  name: string
  salary: number
}

type Person = Extract<keyof Customer, keyof Employee>
// id and name property
```

Exclude

- This is opposite of Extract
- It will exclude the common properties from the two types specified
- In the below example we exclude type from `Customer` and `Employee` types
- Common properties are `id` and `name`
- So the new `Person` type will have `age` and `salary`

```
type Customer {
  id: string
  name: string
  age: number
}
```

```
type Employee {
  id: string
  name: string
  salary: number
}

type Person = Exclude<keyof Customer, keyof Employee>
// age and salary property
```

Record

- Handy when mapping properties of one type to another
- In below example it creates record of `number: Customer`
- Keys should be numbers only - any other type will throw error
- Value should be Customer type

```
type Customer {
  id: string
  name: string
  age: number
}

const customers: Record<number, Customer> = {
  0: { id: 1, name: "Jan", age: 12 },
  1: { id: 2, name: "Dan", age: 22 },
  2: { id: 3, name: "Joe", age: 32 },
}
```

NonNullable

- It allows you to remove `null` and `undefined` from a type
- Suppose you have a type `MyTask` which accepts `null` and `undefined`
- But you have a use case which should not allow `null` and `undefined`
- You can use `NonNullable` to exclude `null` and `undefined` from the type

- Then if you pass `null` or `undefined` to the type your app linter will throw an error

```
type MyTask = string | number | null | undefined

function addTask(task: NonNullable<MyTask>) {
  // logic
}

addTask("task-1")
// valid call

addTask(1)
// valid call

addTask(null)
// Error: Argument of type 'null' is not assignable

addTask(undefined)
// Error: Argument of type 'undefined' is not assignable
```

Type guards

- They allow you to check the type of variables
- You can check it using different operators mentioned below

`typeof`

- Checks if the type of the argument is of the expected type

```
if (typeof x === "number") {
  // YES! number logic
}
```

```
else {
  // No! not a number logic
}
```

instanceof

- Checks if the variable is instance of the given object/class
- This is mostly used with non-primitive objects

```
class Task {
  // class members
}

class Person {
  // class members
}

const myTasks = new Task()
const myPerson = new Person()

console.log(myTasks instanceof Task) // true
console.log(myPerson instanceof Person) // false
```

in

- Allows you to check if property is present in the given type

```
type Task {
  taskId: string
  description: string
}

console.log("taskId" in Task) // true
console.log("dueDate" in Task) // false
```