

TypeScript for JavaScript Programmers

- [Types by Inference](#)
- [Defining Types](#)
- [Composing Types](#)
- [Unions](#)
- [Generics](#)
- [Structural Type System](#)
- [Next Steps](#)

TypeScript stands in an unusual relationship to JavaScript. TypeScript offers all of JavaScript's features, and an additional layer on top of these: TypeScript's type system.

For example, JavaScript provides language primitives like `string` and `number`, but it doesn't check that you've consistently assigned these. TypeScript does.

This means that your existing working JavaScript code is also TypeScript code. The main benefit of TypeScript is that it can highlight unexpected behavior in your code, lowering the chance of bugs.

This tutorial provides a brief overview of TypeScript, focusing on its type system.

Types by Inference

TypeScript knows the JavaScript language and will generate types for you in many cases. For example in creating a variable and assigning it to a particular value, TypeScript will use the value as its type.

```
let helloWorld = "Hello World";  
let helloWorld: string
```

By understanding how JavaScript works, TypeScript can build a type-system that accepts JavaScript code but has types. This offers a type-system without needing to add extra characters to make types explicit in your code. That's how TypeScript knows that `helloWorld` is a `string` in the above example.

You may have written JavaScript in Visual Studio Code, and had editor auto-completion. Visual Studio Code uses TypeScript under the hood to make it easier to work with JavaScript.

Defining Types

You can use a wide variety of design patterns in JavaScript. However, some design patterns make it difficult for types to be inferred automatically (for example, patterns that use dynamic programming). To cover these cases, TypeScript supports an extension of the JavaScript language, which offers places for you to tell TypeScript what the types should be.

For example, to create an object with an inferred type which includes `name: string` and `id: number`, you can write:

```
const user = {  
  name: "Hayes",  
  id: 0,  
};
```

You can explicitly describe this object's shape using an `interface` declaration:

```
interface User {  
  name: string;  
  id: number;  
}
```

You can then declare that a JavaScript object conforms to the shape of your new `interface` by using syntax like `: TypeName` after a variable declaration:

```
const user: User = {  
  name: "Hayes",  
  id: 0,  
};
```

```
};
```

If you provide an object that doesn't match the interface you have provided, TypeScript will warn you:

```
interface User {  
  name: string;  
  id: number;  
}
```

```
const user: User = {  
  username: "Hayes",
```

```
Type '{ username: string; id: number; }' is not assignable to type 'User'.  
  Object literal may only specify known properties, and 'username' does not exist in type 'User'.
```

```
  id: 0,  
};
```

Since JavaScript supports classes and object-oriented programming, so does TypeScript. You can use an interface declaration with classes:

```
interface User {  
  name: string;  
  id: number;  
}
```

```
class UserAccount {  
  name: string;  
  id: number;  
  
  constructor(name: string, id: number) {  
    this.name = name;  
    this.id = id;  
  }  
}
```

```
const user: User = new UserAccount("Murphy", 1);
```

You can use interfaces to annotate parameters and return values to functions:

```
function getAdminUser(): User {  
  //...  
}  
  
function deleteUser(user: User) {  
  // ...  
}
```

There are already a small set of primitive types available in JavaScript: `boolean`, `bigint`, `null`, `number`, `string`, `symbol`, and `undefined`, which you can use in an interface. TypeScript extends this list with a few more, such as `any` (allow anything), `unknown` (ensure someone using this type declares what the type is), `never` (it's not possible that this type could happen), and `void` (a function which returns `undefined` or has no return value).

You'll see that there are two syntaxes for building types: [Interfaces and Types](#). You should prefer `interface`. Use `type` when you need specific features.

Composing Types

With TypeScript, you can create complex types by combining simple ones. There are two popular ways to do so: with Unions, and with Generics.

Unions

With a union, you can declare that a type could be one of many types. For example, you can describe a `boolean` type as being either `true` or `false`:

```
type MyBool = true | false;
```

Note: If you hover over `MyBool` above, you'll see that it is classed as `boolean`. That's a property of the Structural Type System. More on this below.

A popular use-case for union types is to describe the set of `string` or `number` [literals](#) that a value is allowed to be:

```
type WindowStates = "open" | "closed" | "minimized";
```

```
type LockStates = "locked" | "unlocked";
type OddNumbersUnderTen = 1 | 3 | 5 | 7 | 9;
```

Unions provide a way to handle different types too. For example, you may have a function that takes an `array` or a `string`:

```
function getLength(obj: string | string[]) {
  return obj.length;
}
```

To learn the type of a variable, use `typeof`:

Type	Predicate
string	typeof s === "string"
number	typeof n === "number"
boolean	typeof b === "boolean"
undefined	typeof undefined === "undefined"
function	typeof f === "function"
array	Array.isArray(a)

For example, you can make a function return different values depending on whether it is passed a string or an array:

```
function wrapInArray(obj: string | string[]) {
  if (typeof obj === "string") {
    return [obj];
  } else {
    return obj;
  }
}
```

(parameter) obj: string

Generics

Generics provide variables to types. A common example is an array. An array without generics could contain anything. An array with generics can describe the values that the array contains.

```
type StringArray = Array<string>;
type NumberArray = Array<number>;
type ObjectWithNameArray = Array<{ name: string }>;
```

You can declare your own types that use generics:

```
interface Backpack<Type> {
  add: (obj: Type) => void;
  get: () => Type;
}

// This line is a shortcut to tell TypeScript there is a
// constant called `backpack`, and to not worry about where it came from.
declare const backpack: Backpack<string>;

// object is a string, because we declared it above as the variable part of Backpack.
const object = backpack.get();

// Since the backpack variable is a string, you can't pass a number to the add function.
backpack.add(23);
```

Argument of type 'number' is not assignable to parameter of type 'string'.

Structural Type System

One of TypeScript’s core principles is that type checking focuses on the *shape* that values have. This is sometimes called “duck typing” or “structural typing”.

In a structural type system, if two objects have the same shape, they are considered to be of the same type.

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

function logPoint(p: Point) {
  console.log(`${p.x}, ${p.y}`);
}

// logs "12, 26"
const point = { x: 12, y: 26 };
logPoint(point);
```

The `point` variable is never declared to be a `Point` type. However, TypeScript compares the shape of `point` to the shape of `Point` in the type-check. They have the same shape, so the code passes.

The shape-matching only requires a subset of the object’s fields to match.

```
const point3 = { x: 12, y: 26, z: 89 };
logPoint(point3); // logs "12, 26"

const rect = { x: 33, y: 3, width: 30, height: 80 };
logPoint(rect); // logs "33, 3"

const color = { hex: "#187ABF" };
logPoint(color);
```

Argument of type '{ hex: string; }' is not assignable to parameter of type 'Point'.
Type '{ hex: string; }' is missing the following properties from type 'Point': x, y

There is no difference between how classes and objects conform to shapes:

```
class VirtualPoint {
  x: number;
  y: number;

  constructor(x: number, y: number) {
    this.x = x;
    this.y = y;
  }
}

const newVPoint = new VirtualPoint(13, 56);
logPoint(newVPoint); // logs "13, 56"
```

If the object or class has all the required properties, TypeScript will say they match, regardless of the implementation details.

Next Steps

This was a brief overview of the syntax and tools used in everyday TypeScript. From here, you can:

- Read the full Handbook [from start to finish](#) (30m)
- Explore the [Playground examples](#)