Elliott Phillips

Data Scientist
BSc OR II

TypeScript

CPD log

Last updated: August 17, 2021

1.1 Why TypeScript?

Is it worth learning TypeScript?

TL;DR – yes, it definitely is. This book explains why TypeScript is such an obvious choice for writing the frontend in 2021 and demonstrate it's core concepts en route.

1.2 What is Typescript?

In short, Typescript is an open-source programming language created by Microsoft, and a superset of JavaScript which adds static typing to the language where in the end, Typescript compiles to JavaScript to run on any environment which supports JS. It can be both client-side (web browser) or server-side (Node.js).

Now you could ask: *"Ok but why do I need static typing at all? If Typescript compiles to JavaScript doesn't it mean that Javascript is enough?"*

As always in programming, the answer is "it depends". Of course, you can write everything in pure vanilla JS but why hurt yourself? It's always better to have information about types than not to.

1.3 Supported languages

1.3.1 Python

Suspendisse vel felis. Ut lorem lorem, interdum eu, tincidunt sit amet, laoreet vitae, arcu. Aenean faucibus pede eu ante. Praesent enim elit, rutrum at, molestie non, nonummy vel, nisl. Ut lectus eros, malesuada sit amet, fermentum eu, sodales cursus, magna. Donec eu purus. Quisque vehicula, urna sed ultricies auctor, pede lorem egestas dui, et convallis elit erat sed nulla. Donec luctus. Curabitur et nunc. Aliquam dolor odio, commodo pretium, ultricies non, pharetra in, velit. Integer arcu est, nonummy in, fermentum faucibus, egestas vel, odio.

```python
@property
def env_body(self) -> List[str]:
  r"""
  Return a str-type matrix of each row and column of the
  input dataframe. Rows end with "\\" to begin a new table
  row entry, and each cell is space-padded to the length
  of the longest cell's contents, separated with an "&".
  """
  export = []
  for i, row in self.dataframe.iterrows():
    export.append( # Test comment
      self.tab_space
      + " & ".join([
        self.pad_spaces(str(cell), col_index)
        for col_index, cell in enumerate(row.values)
        ])
      - f" {self.double_backslash}"
      )
  return "\n".join([item for item in export if item])
```

1.3.2 TypeScript

Sed commodo posuere pede. Mauris ut est. Ut quis purus. Sed ac odio. Sed vehicula hendrerit sem. Duis non odio. Morbi ut dui. Sed accumsan risus eget odio. In hac habitasse platea dictumst. Pellentesque non elit. Fusce sed justo eu urna porta tincidunt. Mauris felis odio, sollicitudin sed, volutpat a, ornare ac, erat. Morbi quis dolor. Donec pellentesque, erat ac sagittis semper, nunc dui lobortis purus, quis congue purus metus ultricies tellus. Proin et quam. Class aptent taciti sociosqu ad litora torquent per conubia nostra, per inceptos hymenaeos. Praesent sapien turpis, fermentum vel, eleifend faucibus, vehicula eu, lacus.

```typescript
function fn(x: string | number) {
  if (typeof x === "string") {
    69 // do something
  } else if (typeof x === "number") {
```

```
      "420" // do something else
    } else {
      `The answer is ${x}`; // has type 'never'!
    }
  }
```

2.1 Functions

2.1.1 Importing & exporting

Compile `.ts` files with `npx tsc ts_file_name.ts`
Then run the resultant `.js` file with `node js_file_name.js`

```ts
export function getName(user: { first: string; last: string }): string {
  return `${user?.first ?? "first"} ${user?.last ?? "last"}`;
}
```

./ts-func-export.ts

```ts
import getName from './ts-func-export'

console.log(getName({ first: "Elliott"; last: "Phillips" }))
```

./ts-func-import.ts

2.1.2 Functional parameters

Let's say you wanted to make a function that supports a callback, for example, printing to an external file:

```ts
export function printToFile(text: string, callback: () => void): void {
  console.log(text);
  callback();
}
```

Let's create an array mutation function that takes an array of numbers. This takes a function that is given each number and returns a new number.

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

```
7
8  console.log(arrayMutate([1, 2, 3], (v) => v % 10))
```

```
$ npx ts-node functional_parameters.ts                    node
[ 10, 20, 30 ]
```

2.1.3 Functions as types

Anything that follows a colon is a potential type specification that you can share with other objects. In the previous example, the `mutate` functional parameter is hard to read. Introducing the `type` keyword:

```
1  type MutationFunction = (v: number) => number;
```

This `MutationFunction` type definition can declare the type of the `mutate` parameter:

```
1  type MutationFunction = (v: number) => number;
2
3  export function arrayMutate(
4    numbers: number[],
5    mutate: MutationFunction
6  ): number[] {
7    return numbers.map(mutate)
8  }
9
10 const newMutateFunc: MutationFunction = (v: number) => v % 100;
```

2.1.4 Returning functions

Functions that return functions... This is building on the classic JavaScript closure syntax.

```
1  type AdderFunction = (v: number) => number;
2
```

```ts
export function createAdder(num: number): AdderFunction {
  return (val: number) => num + val;
}

const addOne = createAdder(1);

console.log(addOne(55));
```

```
$ npx ts-node returning_functions.ts
56
```

2.1.5 Function overloading

```ts
interface Coordinate {
  x: number;
  y: number;
}

function parseCoordinate(str: string): Coordinate;
function parseCoordinate(obj: Coordinate): Coordinate;
function parseCoordinate(x: number, y: number): Coordinate;
function parseCoordinate(argOne: unknown, argTwo?: unknown): Coordinate {
  let coord: Coordinate = {
    x: 0,
    y: 0,
  };

  if (typeof argOne === "string") {
    (arg1 as string).split(",").forEach((str) => {
      const [key, value] = str.split(":");
      coord[key as "x" | "y"] = parseInt(value, 10);
    });
  } else if (typeof argOne === "object") {
    coord = {
      ...(argOne as Coordinate),
    };
  } else {
    coord = {
      x: argOne as number,
      y: argTwo as number,
    };
  }

  return coord;
```

```
}

console.log(parseCoordinate(10, 20));
console.log(parseCoordinate({ x: 52, y: 35 }));
console.log(parseCoordinate("x:12, y:22"));
```

2.2 Optionals

2.2.1 Parameters

```ts
function printIngredient(quantity: string, ingredient: string, extra?: string) {
  console.log(`${quantity} ${ingredient} ${extra ? ` ${extra}` : ""}`);
}

printIngredient("1 cup", "sugar")
printIngredient("1 cup", "flour", "sifted")
```

2.2.2 Fields

Using the optional chaining feature to check if `user` exists, then if `info` exists, together with the coalescing operator `??` to return an empy string if the email field is undefined.

```ts
interface User {
  id: string;
  info?: {
    email?: string;
  }
}

function getEmail(user: User):string {
  return user?.info?.email ?? "";
}
```

2.2.3 Function callbacks

```ts
function addWithCallback(
  x: number,
  y: number,
  callback?: () => void
): void {
  console.log(x + y);
  callback?.();
}
```

2.3 Tuples

A tuple is an array, where each of its elements can be named and have different types.

2.3.1 Defining tuples

```ts
type ThreeDCoordinate = [x: number, y: number, z: number]

function addThreeDCoordinate(
  one: ThreeDCoordinate,
  two: ThreeDCoordinate
): ThreeDCoordinate {
  return [one[0] + two[0], one[1] + two[1], one[2] + two[2]];
}

console.log(addThreeDCoordinate([1, 2, 3], [4, 5, 6]))
```

```
$ npx ts-node defining_tuples.ts
[5, 7, 9]
```

2.3.2 Tuples with different types

If you're a React engineer, y0ou probably deal with one particular tuple all the time – the `useState()` tuple which returns a state and then a state setter. Let's implement our own simple string state that does something similar:

```ts
function simpleStringState(
  initial: string
): [() => string, (v: string) => void] {
  let str: string = initial;
  return [
    () => str,
    (v: string) => {
      str = v;
    },
  ];
}

const [strOneGetter, strOneSetter] = simpleStringState("hello");
const [strTwoGetter, strTwoSetter] = simpleStringState("unaffected");

console.log(strTwoGetter());
console.log(strOneGetter());

strOneSetter("goodbye");

console.log(strOneGetter());
console.log(strTwoGetter());
```

```
$ npx ts-node tuples_with_different_types.ts

"unaffected"
"hello"
"goodbye"
"unaffected"
```

2.4 Generics

2.4.1 Generic functions

Let's build on the previous example from the tuples section, replacing the `string` type keyword with some other type, say `T` ...

```ts
function simpleState<T>(initial: T): [() => string, (v: T) => void] {
  let val: T = initial;
  return [
    () => str,
    (v: T) => {
      val = v;
```

```
    },
  ];
}

const [strOneGetter, strOneSetter] = simpleState(10);

console.log(strOneGetter());
strOneSetter(42);
console.log(strOneGetter());
```

```
$ npx ts-node simple_state.ts
"unaffected"
"hello"
```

2.4.2 Overriding inferred generic types

In the previous example, we built a generic type `T` that inferred all instances of `T` should be of type number when passing `simpleState` a parameter of `10`.

What if we wanted this state function to accept, say, a value of type `string` or `null`? We need to override `T` by using the `<>` generic syntax, providing a type for the `simpleState` argument:

```
const [strTwoGetter, strTwoSetter] = simpleState<string | null>(null);

console.log(strTwoGetter());
strTwoSetter("some string");
console.log(strTwoGetter());
```

```
$ npx ts-node simple_state.ts
null
"some string"
```

2.4.3 Generic interfaces - ranking example

Given an array of objects containing information, we can build a function to sort them by some ranking method. Say, the array described a set of car models' top speeds and we wanted to return the list in order of speeds:

```ts
interface Car {
  name: string;
  topSpeed: number;
}

const cars: Car[] = [
  {
    name: "chiron",
    topSpeed: 261,
  },
  {
    name: "mondeo",
    topSpeed: 124,
  },
  {
    name: "regera",
    topSpeed: 249,
  },
  {
    name: "huayra",
    topSpeed: 231,
  },
];
```

Then an example of a ranking function is:

```ts
interface Rank<RankItem> {
  item: RankItem;
  rank: number;
}

function ranker<RankItem>(
  items: RankItem[],
  rank: (v: RankItem) => number
): RankItem[] {
  const ranks: Rank<RankItem>[] = items.map((item) => ({
    item,
    rank: rank(item),
  }));

  ranks.sort((a, b) => a.rank - b.rank);

  return ranks.map((rank) => rank.item);
}

const ranks = ranker(cars, ({ topSpeed }) => topSpeed);

console.log(ranks);
```

2.4.4 Generics with keyof

Imagine you have an array of objects, all sharing a key name, and you want one key in particular return as an array.

```ts
function pluck<DataType, KeyType extends keyof DataType>(
  items: DataType[],
  key: KeyType
): DataType[KeyType][] {
  return items.map((item) => item[key]);
}

const dogs = [
  { name: "Eva", age: 4 },
  { name: "Nala", age: 14 },
];

console.log(pluck(dogs, "age"));
console.log(pluck(dogs, "name"));
```

```
$ npx ts-node pluck.ts

[ 4, 14 ]
[ 'Eva', 'Nala' ]
```

```ts
interface BaseEvent {
  time: number;
  user: string;
}
interface EventMap {
  addToCart: BaseEvent & { quantity: number; productID: string };
  checkout: BaseEvent;
}

function sendEvent<Name extends keyof EventMap>(
  name: Name,
  data: EventMap[Name]
): void {
  console.log([name, data]);
}

sendEvent("addToCart", {
  productID: "foo",
  user: "baz",
  quantity: 1,
  time: 10,
});
sendEvent("checkout", { time: 20, user: "bob" });
```

```
$ npx ts-node generic-event-map.ts                                    node
[
  "addToCart",
  { productID: "foo", user: "baz", quantity: 1, time: 10 }
]
[ "checkout", { time: 20, user: "bob" } ]
```

3.1 Hooks

Hooks are functions that let us “hook into” state and lifecycle functionality in function components. Hooks allow us to:

- Reuse stateful logic between components
- Simplify and organize our code to separate concerns, rather allowing unrelated data to get tangled up together
- Avoid confusion around the behavior of the this keyword
- Avoid class constructors, binding methods, and related advanced JavaScript techniques

There are two main rules to keep in mind when using Hooks:

1. Only call Hooks from React functions
2. Only call Hooks at the top level, to be sure that Hooks are called in the same order each time a component renders

Common mistakes to avoid are calling Hooks inside of loops, conditions, or nested functions.

3.1.1 useState

The `useState()` Hook lets you add React state to function components. It should be called at the top level of a React function definition to manage its state.

`initialState` is an optional value that can be used to set the value of `currentState` for the first render. The `stateSetter` function is used to update the value of `currentState` and rerender our component with the next state value.

```JS
function Counter({ initialCount }) {
  const [count, setCount] = useState(initialCount);
  return (
    <div>
      <p>Count value is: {count}</p>
      <button onClick={() => setCount(initialCount)}>Reset</button>
      <button
        onClick={() => setCount(prevCount => prevCount - 1)}
      >
        Decrement
      </button>
      <button
        onClick={() => setCount(prevCount => prevCount + 1)}
      >
        Increment
      </button>
    </div>
```

```
18      );
19    }
```

When the previous state value is used to calculate the next state value, pass a function to the state setter. This function accepts the previous value as an argument and returns an updated value.

If the previous state is not used to compute the next state, just pass the next state value as the argument for the state setter.

`useState()` may be called more than once in a component. This gives us the freedom to separate concerns, simplify our state setter logic, and organize our code in whatever way makes the most sense to us!

3.1.2 useEffect

The primary purpose of a React component is to return some JSX to be rendered. Often, it is helpful for a component to execute some code that performs side effects in addition to rendering JSX.

In class components, side effects are managed with lifecycle methods. In function components, we manage side effects with the Effect Hook. Some common side effects include: fetching data from a server, subscribing to a data stream, logging values to the console, interval timers, and directly interacting with the DOM.

JS

```
function App(props) {
  const [title, setTitle] = useState('');
  useEffect(() => {
    document.title = title;
  }, [title]);

  const [time, setTime] = useState(0);
  useEffect(() => {
    const intervalId = setInterval(() => setTime((prev) => prev + 1), 1000);
    return () => clearInterval(intervalId);
  }, []);

  // ...
}
```

The dependency array is used to tell the useEffect() method when to call our effect.

By default, with no dependency array provided, our effect is called after every render. An empty dependency array signals that our effect never needs to be re-run. A non-empty dependency array signals to the Effect Hook that it only needs to call our effect again when the value of one of the listed dependencies has changed.

```js
useEffect(() => {
  alert('called after every render');
});

useEffect(() => {
  alert('called after first render');
}, []);

useEffect(() => {
  alert('called when value of `endpoint` or `id` changes');
}, [endpoint, id]);
```