TYPESCRIPT ESSENTIALS

5 Critical Keys to Using TypeScript Effectively

By

Preston Wallace

Software Engineer, and All-Around Nice Guy

1 - TypeScript Type Annotations

Enhance Your Code's Safety and Readability

TypeScript is all about type safety, so it's important to understand how to annotate variables, functions, and classes with types.

Example:

```typescript
// Annotate variable with type
let message: string = "Hello, world!";

// Annotate function with argument types and return type
function add(a: number, b: number): number {
  return a + b;
}

// Annotate class properties and method arguments with types
class Person {
  name: string;
  age: number;

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

  greet(otherPerson: Person) {
    console.log(`Hello, ${otherPerson.name}!`);
  }
}
```

2 - Interfaces

Defining Contracts for More Robust Code

Interfaces are a key feature of TypeScript that allow you to define contracts for objects and classes.

Example:

```
// define interface
interface Person {
  name: string;
  age: number;
  school: string;
  greet(otherPerson: Person): void;
}

// use interface
class Student implements Person {
  name: string;
  age: number;
  school: string;

  constructor(name: string, age: number, school: string) {
    this.name = name;
    this.age = age;
    this.school = school;
  }

  greet(otherPerson: Person) {
    console.log(`Hello, ${otherPerson.name}!
    I'm a student at   ${this.school}.`);
  }
}
```

3 - Classes

Object-Oriented Programming Made Easy

TypeScript is an object-oriented language, so you'll need to understand how to create and use classes, as well as how inheritance and polymorphism work in TypeScript. Classes are not unique to TypeScript; they exist in JavaScript as well, but are more commonly-used in TypeScript.

Example:

```
// define a class
class Shape {
  private color: string;

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

  getArea(): number {
    return 0;
  }
}

// extend the above class, inheriting its properties
class Circle extends Shape {
  // private fields are only accessible from within the class
  private radius: number;

  constructor(color: string, radius: number) {
    // super allows us to pass arguments to the "parent" class
    super(color);
    this.radius = radius;
  }

  getArea(): number {
    return Math.PI * this.radius * this.radius;
  }
}

const myCircle = new Circle("red", 10);
console.log(myCircle.getArea()); // Output: 314.1592653589793
```

4 - Generics

Writing Flexible Code with TypeScript Generics

Generics allow you to write flexible and reusable code that can work with multiple data types. To use generics, we can define a type parameter by placing a type variable in angle brackets (<>) next to the name of the function or class. The type variable can then be used in place of specific types throughout the code. When we use the function or class, we can specify the type argument in place of the type variable.

In the below example, the reverse function has a type parameter T, which represents the type of the elements in the array. When we call the function, we specify the type argument as number to indicate that the array contains numbers. The function then returns a reversed array of the same type.

```typescript
// accepts an array of a generic type T. Returns an array of the same type.
function reverse<T>(items: T[]): T[] {
  return items.reverse();
}

const myArray = [1, 2, 3, 4];
const reversedArray = reverse(myArray); // Output: [4, 3, 2, 1]
```

5 - Decorators

Take Your Code to the Next Level

Decorators are a powerful feature of TypeScript that allow you to add functionality to classes and methods at runtime. Decorators are declared using the @ syntax

Here's an example of a simple decorator that logs the name of a method and its arguments:

```
function log(target: any, key: string, descriptor: PropertyDescriptor) {
  const originalMethod = descriptor.value;

  descriptor.value = function(...args: any[]) {
    console.log(`Calling ${key} with arguments: ${JSON.stringify(args)}`);
    const result = originalMethod.apply(this, args);
    console.log(`Result: ${JSON.stringify(result)}`);
    return result;
  };

  return descriptor;
}

class MyClass {
  @log
  myMethod(arg1: number, arg2: string): string {
    return `${arg1} ${arg2}`;
  }
}

const myObj = new MyClass();
console.log(myObj.myMethod(42, "My String"));
```