

TypeScript 3.2 is now available. [Download \(/#download-links\)](#) our latest version today!

Documentation ▼

Classes

Introduction

Traditional JavaScript uses functions and prototype-based inheritance to build up reusable components, but this may feel a bit awkward to programmers more comfortable with an object-oriented approach, where classes inherit functionality and objects are built from these classes. Starting with ECMAScript 2015, also known as ECMAScript 6, JavaScript programmers will be able to build their applications using this object-oriented class-based approach. In TypeScript, we allow developers to use these techniques now, and compile them down to JavaScript that works across all major browsers and platforms, without having to wait for the next version of JavaScript.

Classes

Let's take a look at a simple class-based example:

```
class Greeter {
  greeting: string;
  constructor(message: string) {
    this.greeting = message;
  }
  greet() {
    return "Hello, " + this.greeting;
  }
}

let greeter = new Greeter("world");
```

The syntax should look familiar if you've used C# or Java before. We declare a new class `Greeter`. This class has three members: a property called `greeting`, a constructor, and a method `greet`.

You'll notice that in the class when we refer to one of the members of the class we prepend `this.`. This denotes that it's a member access.

In the last line we construct an instance of the `Greeter` class using `new`. This calls into the constructor we defined earlier, creating a new object with the `Greeter` shape, and running the constructor to initialize it.

Inheritance

In TypeScript, we can use common object-oriented patterns. One of the most fundamental patterns in class-based programming is being able to extend existing classes to create new ones using inheritance.

Let's take a look at an example:

```

class Animal {
    move(distanceInMeters: number = 0) {
        console.log(`Animal moved ${distanceInMeters}m.`);
    }
}

class Dog extends Animal {
    bark() {
        console.log('Woof! Woof!');
    }
}

const dog = new Dog();
dog.bark();
dog.move(10);
dog.bark();

```

This example shows the most basic inheritance feature: classes inherit properties and methods from base classes. Here, `Dog` is a *derived* class that derives from the `Animal` *base* class using the `extends` keyword. Derived classes are often called *subclasses*, and base classes are often called *superclasses*.

Because `Dog` extends the functionality from `Animal`, we were able to create an instance of `Dog` that could both `bark()` and `move()`.

Let's now look at a more complex example.

```

class Animal {
    name: string;
    constructor(theName: string) { this.name = theName; }
    move(distanceInMeters: number = 0) {
        console.log(`${this.name} moved ${distanceInMeters}m.`);
    }
}

class Snake extends Animal {
    constructor(name: string) { super(name); }
    move(distanceInMeters = 5) {
        console.log("Slithering...");
        super.move(distanceInMeters);
    }
}

class Horse extends Animal {
    constructor(name: string) { super(name); }
    move(distanceInMeters = 45) {
        console.log("Galloping...");
        super.move(distanceInMeters);
    }
}

let sam = new Snake("Sammy the Python");
let tom: Animal = new Horse("Tommy the Palomino");

sam.move();
tom.move(34);

```

This example covers a few other features we didn't previously mention. Again, we see the `extends` keywords used to create two new subclasses of `Animal`: `Horse` and `Snake`.

One difference from the prior example is that each derived class that contains a constructor function *must* call `super()` which will execute the constructor of the base class. What's more, before we *ever* access a property on `this` in a constructor body, we *have* to call `super()`. This is an important rule that TypeScript will enforce.

The example also shows how to override methods in the base class with methods that are specialized for the subclass. Here both `Snake` and `Horse` create a `move` method that overrides the `move` from `Animal`, giving it functionality specific to each class. Note that even though `tom` is declared as an `Animal`, since its value is a `Horse`, calling `tom.move(34)` will call the overriding method in `Horse`:

```
Slithering...
Sammy the Python moved 5m.
Gallop...
Tommy the Palomino moved 34m.
```

Public, private, and protected modifiers

Public by default

In our examples, we've been able to freely access the members that we declared throughout our programs. If you're familiar with classes in other languages, you may have noticed in the above examples we haven't had to use the word `public` to accomplish this; for instance, C# requires that each member be explicitly labeled `public` to be visible. In TypeScript, each member is `public` by default.

You may still mark a member `public` explicitly. We could have written the `Animal` class from the previous section in the following way:

```
class Animal {
  public name: string;
  public constructor(theName: string) { this.name = theName; }
  public move(distanceInMeters: number) {
    console.log(`${this.name} moved ${distanceInMeters}m.`);
  }
}
```

Understanding private

When a member is marked `private`, it cannot be accessed from outside of its containing class. For example:

```
class Animal {
  private name: string;
  constructor(theName: string) { this.name = theName; }
}

new Animal("Cat").name; // Error: 'name' is private;
```

TypeScript is a structural type system. When we compare two different types, regardless of where they came from, if the types of all members are compatible, then we say the types themselves are compatible.

However, when comparing types that have `private` and `protected` members, we treat these types differently. For two types to be considered compatible, if one of them has a `private` member, then the other must have a `private` member that originated in the same declaration. The same applies to `protected` members.

Let's look at an example to better see how this plays out in practice:

```

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

class Rhino extends Animal {
    constructor() { super("Rhino"); }
}

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

let animal = new Animal("Goat");
let rhino = new Rhino();
let employee = new Employee("Bob");

animal = rhino;
animal = employee; // Error: 'Animal' and 'Employee' are not compatible

```

In this example, we have an `Animal` and a `Rhino`, with `Rhino` being a subclass of `Animal`. We also have a new class `Employee` that looks identical to `Animal` in terms of shape. We create some instances of these classes and then try to assign them to each other to see what will happen. Because `Animal` and `Rhino` share the `private` side of their shape from the same declaration of `private name: string` in `Animal`, they are compatible. However, this is not the case for `Employee`. When we try to assign from an `Employee` to `Animal` we get an error that these types are not compatible. Even though `Employee` also has a `private` member called `name`, it's not the one we declared in `Animal`.

Understanding protected

The `protected` modifier acts much like the `private` modifier with the exception that members declared `protected` can also be accessed within deriving classes. For example,

```

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

class Employee extends Person {
    private department: string;

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

    public getElevatorPitch() {
        return `Hello, my name is ${this.name} and I work in ${this.department}.`;
    }
}

let howard = new Employee("Howard", "Sales");
console.log(howard.getElevatorPitch());
console.log(howard.name); // error

```

Notice that while we can't use `name` from outside of `Person`, we can still use it from within an instance method of `Employee` because `Employee` derives from `Person`.

A constructor may also be marked `protected`. This means that the class cannot be instantiated outside of its containing class, but can be extended. For example,

```
class Person {
  protected name: string;
  protected constructor(theName: string) { this.name = theName; }
}

// Employee can extend Person
class Employee extends Person {
  private department: string;

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

  public getElevatorPitch() {
    return `Hello, my name is ${this.name} and I work in ${this.department}.`;
  }
}

let howard = new Employee("Howard", "Sales");
let john = new Person("John"); // Error: The 'Person' constructor is protected
```

Readonly modifier

You can make properties readonly by using the `readonly` keyword. Readonly properties must be initialized at their declaration or in the constructor.

```
class Octopus {
  readonly name: string;
  readonly numberOfLegs: number = 8;
  constructor (theName: string) {
    this.name = theName;
  }
}

let dad = new Octopus("Man with the 8 strong legs");
dad.name = "Man with the 3-piece suit"; // error! name is readonly.
```

Parameter properties

In our last example, we had to declare a readonly member `name` and a constructor parameter `theName` in the `Octopus` class, and we then immediately set `name` to `theName`. This turns out to be a very common practice. *Parameter properties* let you create and initialize a member in one place. Here's a further revision of the previous `Octopus` class using a parameter property:

```
class Octopus {
  readonly numberOfLegs: number = 8;
  constructor(readonly name: string) {
  }
}
```

Notice how we dropped `theName` altogether and just use the shortened `readonly name: string` parameter on the constructor to create and initialize the `name` member. We've consolidated the declarations and assignment into one location.

Parameter properties are declared by prefixing a constructor parameter with an accessibility modifier or `readonly`, or both. Using `private` for a parameter property declares and initializes a private member; likewise, the same is done for `public`, `protected`, and `readonly`.

Accessors

TypeScript supports getters/setters as a way of intercepting accesses to a member of an object. This gives you a way of having finer-grained control over how a member is accessed on each object.

Let's convert a simple class to use `get` and `set`. First, let's start with an example without getters and setters.

```
class Employee {
    fullName: string;
}

let employee = new Employee();
employee.fullName = "Bob Smith";
if (employee.fullName) {
    console.log(employee.fullName);
}
```

While allowing people to randomly set `fullName` directly is pretty handy, this might get us in trouble if people can change names on a whim.

In this version, we check to make sure the user has a secret passcode available before we allow them to modify the employee. We do this by replacing the direct access to `fullName` with a `set` that will check the passcode. We add a corresponding `get` to allow the previous example to continue to work seamlessly.

```
let passcode = "secret passcode";

class Employee {
    private _fullName: string;

    get fullName(): string {
        return this._fullName;
    }

    set fullName(newName: string) {
        if (passcode && passcode == "secret passcode") {
            this._fullName = newName;
        }
        else {
            console.log("Error: Unauthorized update of employee!");
        }
    }
}

let employee = new Employee();
employee.fullName = "Bob Smith";
if (employee.fullName) {
    console.log(employee.fullName);
}
```

To prove to ourselves that our accessor is now checking the passcode, we can modify the passcode and see that when it doesn't match we instead get the message warning us we don't have access to update the employee.

A couple of things to note about accessors:

First, accessors require you to set the compiler to output ECMAScript 5 or higher. Downlevelling to ECMAScript 3 is not supported. Second, accessors with a `get` and no `set` are automatically inferred to be `readonly`. This is helpful when generating a `.d.ts` file from your code, because users of your property can see that they can't change it.

Static Properties

Up to this point, we've only talked about the *instance* members of the class, those that show up on the object when it's instantiated. We can also create *static* members of a class, those that are visible on the class itself rather than on the instances. In this example, we use `static` on the origin, as it's a general value for all grids. Each instance accesses this value through prepending the name of the class. Similarly to prepending `this.` in front of instance accesses, here we prepend `Grid.` in front of static accesses.

```
class Grid {
  static origin = {x: 0, y: 0};
  calculateDistanceFromOrigin(point: {x: number; y: number;}) {
    let xDist = (point.x - Grid.origin.x);
    let yDist = (point.y - Grid.origin.y);
    return Math.sqrt(xDist * xDist + yDist * yDist) / this.scale;
  }
  constructor (public scale: number) { }
}

let grid1 = new Grid(1.0); // 1x scale
let grid2 = new Grid(5.0); // 5x scale

console.log(grid1.calculateDistanceFromOrigin({x: 10, y: 10}));
console.log(grid2.calculateDistanceFromOrigin({x: 10, y: 10}));
```

Abstract Classes

Abstract classes are base classes from which other classes may be derived. They may not be instantiated directly. Unlike an interface, an abstract class may contain implementation details for its members. The `abstract` keyword is used to define abstract classes as well as abstract methods within an abstract class.

```
abstract class Animal {
  abstract makeSound(): void;
  move(): void {
    console.log("roaming the earth...");
  }
}
```

Methods within an abstract class that are marked as abstract do not contain an implementation and must be implemented in derived classes. Abstract methods share a similar syntax to interface methods. Both define the signature of a method without including a method body. However, abstract methods must include the `abstract` keyword and may optionally include access modifiers.

```

abstract class Department {

    constructor(public name: string) {
    }

    printName(): void {
        console.log("Department name: " + this.name);
    }

    abstract printMeeting(): void; // must be implemented in derived classes
}

class AccountingDepartment extends Department {

    constructor() {
        super("Accounting and Auditing"); // constructors in derived classes must call super()
    }

    printMeeting(): void {
        console.log("The Accounting Department meets each Monday at 10am.");
    }

    generateReports(): void {
        console.log("Generating accounting reports...");
    }
}

let department: Department; // ok to create a reference to an abstract type
department = new Department(); // error: cannot create an instance of an abstract class
department = new AccountingDepartment(); // ok to create and assign a non-abstract subclass
department.printName();
department.printMeeting();
department.generateReports(); // error: method doesn't exist on declared abstract type

```

Advanced Techniques

Constructor functions

When you declare a class in TypeScript, you are actually creating multiple declarations at the same time. The first is the type of the *instance* of the class.

```

class Greeter {
    greeting: string;
    constructor(message: string) {
        this.greeting = message;
    }
    greet() {
        return "Hello, " + this.greeting;
    }
}

let greeter: Greeter;
greeter = new Greeter("world");
console.log(greeter.greet());

```

Here, when we say `let greeter: Greeter`, we're using `Greeter` as the type of instances of the class `Greeter`. This is almost second nature to programmers from other object-oriented languages.

We're also creating another value that we call the *constructor function*. This is the function that is called when we `new` up instances of the class. To see what this looks like in practice, let's take a look at the JavaScript created by the above example:


```

let Greeter = (function () {
    function Greeter(message) {
        this.greeting = message;
    }
    Greeter.prototype.greet = function () {
        return "Hello, " + this.greeting;
    };
    return Greeter;
})();

let greeter;
greeter = new Greeter("world");
console.log(greeter.greet());

```

Here, `let Greeter` is going to be assigned the constructor function. When we call `new` and run this function, we get an instance of the class. The constructor function also contains all of the static members of the class. Another way to think of each class is that there is an *instance* side and a *static* side.

Let's modify the example a bit to show this difference:

```

class Greeter {
    static standardGreeting = "Hello, there";
    greeting: string;
    greet() {
        if (this.greeting) {
            return "Hello, " + this.greeting;
        }
        else {
            return Greeter.standardGreeting;
        }
    }
}

let greeter1: Greeter;
greeter1 = new Greeter();
console.log(greeter1.greet());

let greeterMaker: typeof Greeter = Greeter;
greeterMaker.standardGreeting = "Hey there!";

let greeter2: Greeter = new greeterMaker();
console.log(greeter2.greet());

```

In this example, `greeter1` works similarly to before. We instantiate the `Greeter` class, and use this object. This we have seen before.


Next, we then use the class directly. Here we create a new variable called `greeterMaker`. This variable will hold the class itself, or said another way its constructor function. Here we use `typeof Greeter`, that is "give me the type of the `Greeter` class itself" rather than the instance type. Or, more precisely, "give me the type of the symbol called `Greeter`," which is the type of the constructor function. This type will contain all of the static members of `Greeter` along with the constructor that creates instances of the `Greeter` class. We show this by using `new` on `greeterMaker`, creating new instances of `Greeter` and invoking them as before.

Using a class as an interface

As we said in the previous section, a class declaration creates two things: a type representing instances of the class and a constructor function. Because classes create types, you can use them in the same places you would be able to use interfaces.

```
class Point {  
    x: number;  
    y: number;  
}  
  
interface Point3d extends Point {  
    z: number;  
}  
  
let point3d: Point3d = {x: 1, y: 2, z: 3};
```

Made with ❤ in Redmond Follow @Typescriptlang (<https://twitter.com/typescriptlang>)

Privacy (<https://go.microsoft.com/fwlink/?LinkId=521839>) ©2012-2018 Microsoft  Microsoft