# The TypeScript Handbook

## Classes

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 can 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");[Try](https://www.typescriptlang.org/play/#code/MYGwhgzhAEDiBOBTRAXR9oG8BQ1oHMlUBLAO3wC5oIV4z8BubXaYAe1JvgFdgU34ACgC2iKGHyIqXegEosLPCgAWxCADpCyFPWgBeaKPGSmeAL7M8W1IPk48eJCm7xS0AEQAJRCBBsANB7QANTQKmqaRDrkptAWFtggqARR6PrQpIgA7nCpQu5ZAiAAJu6yDEA)

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();[Try](https://www.typescriptlang.org/play/#code/MYGwhgzhAECCB2BLAtmE0DeAoa1kHsA3AUwAoATRCAFzHmGIEl4BZY64gJwgC5p4ArsgBGXaAF5oABgCUmHLmjB88CPhDEAdCHwBzUgAMEKNHiLFy0ACQZKNOg2ZsO3AL7JNBmQG4FrrP5YoJAwACJ60MQAHhzw5DDGqOjYuMJgnADWpHIpisqq6lo6+gBEAOr4+ABmAITQFdU1JT5+AVhBKjTQ5BGS8MQA7tDh+i09upppmdm+45oEJKQAjLKzepPpWT5AA)

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);[Try](https://www.typescriptlang.org/play/#code/MYGwhgzhAECCB2BLAtmE0DeAoa15mQFMAuaCAFwCdF4BzAbh2mAHt4LKBXYclygCnIALQgDkCJMlRq0AlJia5hiCADp8RaAF5owsRMa4Avk2QsAboX4ATFeTDxghAJLwAsoXKFKEUvE7IAEbe2tAADPLYuLis7CwghKogLLT8AAYAJBjKahqERtBmltbQWbYUDk6uHl4+RsiqabKG0CYmWKCQMADK+ADWhNCEAB5e8NYwCChoCjFsHNy8AnmkHDKRimScAA7e-HnNTCa4RVbl9o4u7p7eMDoArBvRzPPxicmpAETdIIh61HRVEDPodnhAdt5VKcbHZKlcardQa0sO1OlBoAAJPgQQYjMYTOBIVDoKIvdhURZ8fYSVbSOhPaLg3bLCRI46FCxnWGXao3HyhAAsj1m0ViEDeSRS-E+AHE0MltjIgaoQS1cEzIdDznDebUIGyUVgsAlyGQCKF4IQAO7QXpgAbS7oEZAAT10ImgAAUXcI2KrjZ5dCxkKQpsSLdbMdirJ8ACrB13uwaetDBmgsf1YCAEKGc-iHXgNaEAZgFzSAA)

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.

Galloping...

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.`);

}

}[Try](https://www.typescriptlang.org/play/#code/MYGwhgzhAECCB2BLAtmE0DeAoa0AOArgEYiLDTxjICmAXNBAC4BOi8A5gNxY77GnlgAe3hNmBYIyHMAFIwAW1AHJU6DFm3YBKTL1wLEEAHSUa0ALzQFy1d1wBfHrkIky0ZEIBu1GQBNDjGDwwNQAkvAAstSM1MwQ9PAEyESxOti4uMKiQiDURiBC7DIABgAkGAbGptT27l7UvtDl-kxBIeFRMXH2yEbFWnbQjvZAA)

## ECMAScript Private Fields

With TypeScript 3.8, TypeScript supports the new JavaScript syntax for private fields:

class Animal {

#name: string;

constructor(theName: string) {

this.#name = theName;

}

}

new Animal("Cat").#name;

Property '#name' is not accessible outside class 'Animal' because it has a private identifier.Property '#name' is not accessible outside class 'Animal' because it has a private identifier.[Try](https://www.typescriptlang.org/play/#code/PTAEAEFMCdoe2gZwFygIwA4AMaDMAoAYwBsBDRRUAQQDsBLAW1ONAG99RQBiG0hyVIgAu0OjQDmAbg6hCcGsOgBXQkIQAKIQAtIAOT4DQiseICUbGZ211EAOh4HQAXlDa9B6ZwC++H-hqQAO7U9EzE6gBEAMKkQhGm9rz8kkA)

This syntax is built into the JavaScript runtime and can have better guarantees about the isolation of each private field. Right now, the best documentation for these private fields is in the TypeScript 3.8 [release notes](https://devblogs.microsoft.com/typescript/announcing-typescript-3-8-beta/#ecmascript-private-fields).

## Understanding TypeScript’s private

TypeScript also has its own way to declare a member as being 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;

Property 'name' is private and only accessible within class 'Animal'.Property 'name' is private and only accessible within class 'Animal'.[Try](https://www.typescriptlang.org/play/#code/PTAEAEFMCdoe2gZwFygEwGYAsBGAUAMYA2AhooqAIIB2AlgLYlGgDeeooADtLQG4kAXSKGol6kVIgE9qAcwDcedqAJxqU6AFcCAhAAoBAC0gA5MRNAbacgJStlHI7UQA6UeNABeUEdPnFHAC+eMF41JAA7lR0jER6AEQAwoLxNm7+QA)

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;

Type 'Employee' is not assignable to type 'Animal'.

Types have separate declarations of a private property 'name'.Type 'Employee' is not assignable to type 'Animal'.

Types have separate declarations of a private property 'name'.[Try](https://www.typescriptlang.org/play/#code/PTAEAEFMCdoe2gZwFygEwGY1oFAGMAbAQ0UVAEEA7ASwFsiDQBvHUUAB2moDciAXSKEpFakVIj5dKAcwDcrUHjiUJ0AK54+CABR8AFpAByIsaFXUZASmYK2+6ogB0w0aAC8ofUZPy2AXxwA-GJSUAAlPQs4UEgADwFKABMyKjoGGzYlFUkNLWhtaxY2NkQ1dhhtACIIqMrLX1AAoMISMgBRWnYCOABPSEEiji5eASETcUkLOQUs1VydL2NRCalpQttPSKcXQQ9FnwUmnBwCSD5QIhp6Rg9KSAB3CiuGKoBxOH46+VPz6EjKaK3B7hf5wArfM4xTrdPq7ITAjpdXr9KoAITgACMvsdLmkbqA-lF5Ljru4oUjYbIgA)

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);

Property 'name' is protected and only accessible within class 'Person' and its subclasses.Property 'name' is protected and only accessible within class 'Person' and its subclasses.[Try](https://www.typescriptlang.org/play/#code/PTAEAEFMCdoe2gZwFygEwBYMFYBQBjAGwENFFQAFGROAO1AG9dRQAHeAF0ny4BNRaxALaRUiDtACWtAOYBuZqHx1x0AK48EACkEixE6TICUjRSw4ALSYgB0uyKAC8A4ZAUsAvri8ESZUACiQqyEcACekA6QAB5ctLzkVEh0pizskgBuxFygvJCsxNAcIrQc+lKyCorKtKoaHNr25YYANLn5hcWQpc2yJkwsLIhqrDA6rkbug5bWNnkFRSUcTu0LXaVTPmlqAEaEkvigMpAcAYSQWQ3QFJIc+BZa-Wag0Cdq0PQABgASkIShbSEYRcIlA1lAABIGDNbPYPKBiPFQABJUAAdwQAGswfQoTC5h1Ft0OB4bJ9Nt5cLhzssLHA0YV+M5aJA0YFgqEIpAtAAib70xk8to8gDKxHOiB5kwIKjg5xsoRkWjpDOgvBsx1O50uCBudweRmlNRo8sVyoFarsEzkQA)

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");

Constructor of class 'Person' is protected and only accessible within the class declaration.Constructor of class 'Person' is protected and only accessible within the class declaration.[Try](https://www.typescriptlang.org/play/#code/PTAEAEFMCdoe2gZwFygEwDYDsAWAUAMYA2AhooqAAoyJwB2oA3nqKAA7wAukB3AJqDokAtpFSJO0AJZ0A5gG4W7Lj36gC9CdACuvBAApOAC0gA5EWNBaZsgJRMlrY1MQA6IaNABeUMbMXFVgBfPBC8EFAAUWE2IjgAT0hIdRIGSAAPbjoBaiR6QlJyKJi4xOSMrL4KXNoGZlYOKQA3Em5QPkg2EmhOUTpOcUkbRSUNOi1dTgMPS2s5ABp2zu7eyH7B6Tl7etYrbTYYfRnbQN3nNw6unr7ObyWr1f7TsIbtACMiKQJQWUhOSKIkBaU2glCknAIRn020coGgf200AYAAMABKQIhxRbCeKCCygFygAAkjHO7gsQVAqQEAElQAB3BAAawJDBJZMuKxuQVcyOeoTweEBtyMcHp3QEPjokHpxViCSS+gARKixRKlYslQBlEiAxBKk5Cv6gABWcCMDClMqoNHoyoAUua6Ab5EA)

## 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";

Cannot assign to 'name' because it is a read-only property.Cannot assign to 'name' because it is a read-only property.[Try](https://www.typescriptlang.org/play/#code/PTAEAEFMCdoe2gZwFygEwFYAsAGAUAMYA2AhooqAPIEAucADgK4UDeeoo0kJAJnAHZEAnqH4kAtpFSIa0AJb8A5gG52nbn0Ej+jcQCMYlAGYAZSIpSjdB6KAC8oAByq1BATOiNaCABQ0AFpAAchJSoB4KigCUoGwcHAFyiAB0YpL2oAHBoaocAL54BXhEkDSgPLwZ-JAA7lTeTIg+AEQAsiT8oDVyAZmBTuGyAoqgJRbNUaoVPKmhGW0dXT3+fZCgAMwAtPRykARriIw9zcpAA)

## Parameter properties

In our last example, we had to declare a readonly member name and a constructor parameter theName in the Octopus class. This is needed in order to have the value of theName accessible after the Octopus constructor is executed. 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) {}

}

let dad = new Octopus("Man with the 8 strong legs");

dad.name;[Try](https://www.typescriptlang.org/play/#code/MYGwhgzhAEDywBcD2AHArjA3gKGtATgKZgAmSAdiAJ7TloC2ARofrAGYAyhA5hAFy0GzfNAC80ABwBuXNGAUICfGkRJ8ACiKkK1WmHqEBi-AEty3AJTRMAX2x3sIQgmglSY2oQDucVegjqAEQAsmDk0F4mCAAW0DGEktDGFNzQTryBFjJuJAB05PqEUkA)

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);

}[Try](https://www.typescriptlang.org/play/#code/PTAEAEGcBcCcEsDG0BcoBmBDANpApgFCLaaSSgCiAtgA7YD2AnnnqAN4GgYCu22Acpip40MBADsA5gG4CAXwIFseaKDy0GzVgF5Q4vAHdKGpiwAUASlnq6pvADp0vAUJ2gARACF6AI1ABlKnhoAAt3WQJ4dFAzG00WR2dBYQt2TlBEenFIemV7BklYky1EvmS8K3kgA)

While allowing people to randomly set fullName directly is pretty handy, we may also want enforce some constraints when fullName is set.

In this version, we add a setter that checks the length of the newName to make sure it’s compatible with the max-length of our backing database field. If it isn’t we throw an error notifying client code that something went wrong.

To preserve existing functionality, we also add a simple getter that retrieves fullName unmodified.

const fullNameMaxLength = 10;

class Employee {

private \_fullName: string = "";

get fullName(): string {

return this.\_fullName;

}

set fullName(newName: string) {

if (newName && newName.length > fullNameMaxLength) {

throw new Error("fullName has a max length of " + fullNameMaxLength);

}

this.\_fullName = newName;

}

}

let employee = new Employee();

employee.fullName = "Bob Smith";

if (employee.fullName) {

console.log(employee.fullName);

}[Try](https://www.typescriptlang.org/play/#code/PTAEAEGcBcCcEsDG0BcoBmBDANpApgFCID2AdjBgK7bYBymAtngLKYAeAMnqQObQAWoALygAjAAYA3ASLZMkSKACiDAA7ZiATzx5QAbwKhQqhADdM0XQH101OozxoYCXsNAAid9MOgeeaFQ09EwAFACUTnDwrgZGRrD+lLCkoALwkAB0NnbBeNJGAL4yRvgBtkEOIaR4AO65kS48Yfo+RvDooFW1uaAAZL2g1XUOGdjcfIIAfIH2TKyc4wLNsXFGArDENYO1yrAbsCHu5bO6-PKgmKAM7KBjvAKgxB3uoADUM7nzXPf8YflxRVaqX46Syxx6IiGuX+RUBYwCeDUGm0ukhOxU6i0OnC0kRmJRGXBDjc7gAQsQAEagADKDHgAi8MnanTxyJ0hJyDmWPhI5GIY1GxB4IVZWLwHIqTD+BAKQA)

To prove to ourselves that our accessor is now checking the length of values, we can attempt to assign a name longer than 10 characters and verify that we get an error.

A couple of things to note about accessors:

First, accessors require you to set the compiler to output ECMAScript 5 or higher. Downleveling 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 }));[Try](https://www.typescriptlang.org/play/#code/MYGwhgzhAEDiBOBLAJtA3gKGtCAXMuiw0A9kgOaIB20AvOtAB4Bc0ADADTQCerb0AXwDcGLNGBgQwAK7hcAUwAiiPGCrB5AMXgkAtgHkK1ABQAHEtVys0TVlWm6ARvPhCedh8-iCAlOjHYIPK4TMp4dNDmlgB0jNAAtHBIyNFkiJRUsSLYgcE8YSH0UVS40dwJSSipRpnc2TnwwdLwNACyBAAW0RAAjvC4xowF0ABUoSohANT5E6MzeH4A9NC4HSrdEkH1AqLYwCRUePDSwLhkZtKOIEQ4m-IeTi5+aDs7GEEh5MkAjBFU8gB3SrIYzfaJsHxuRbLb5xCB3d55L4oABMf0BwOMAFZwZDoNDoFi4QiMPtDiQgtEQCRyMZkcgwZsZHIlBM1BptHpDOkTDYWNBvpx3AL+AIfJDSQcIBT5FSaXTkijokzZARWap1FodAYasY+axBVxeCLfJCgA)

## 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...");

}

}[Try](https://www.typescriptlang.org/play/#code/IYIwzgLgTsDGEAJYBthjAgggOwJYFthkEBvAKAQVEhngUIGsBTAZQHsBXbAEwAoBKAFwIAbm1zcA3GQr02IpgOFiJpWZVhtsYNsiYA6ZGwDmvAERQ2wfLmzGEEABZMETYFCf6vZ-tMoBfMn8gA)

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

Cannot create an instance of an abstract class.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: department is not of type AccountingDepartment, cannot access generateReports

Property 'generateReports' does not exist on type 'Department'.Property 'generateReports' does not exist on type 'Department'.[Try](https://www.typescriptlang.org/play/#code/PTAEAEFMCdoe2gZwFygEwFYCMX0GY8BOAKAEMAjRAF2lIGMrQ6AbUxRUAEUgAdToqAW0gA7RgG9ioJnBHVoAVwYIAFDwXlmASzqgRpYanlaRAcwCUocQF9iU0D2gmqAOQOQV51ADc4WgCZW9tJ0sohwzJAAdMxwpioARNx8AsJieu6oCaAA1KBUABZaiFH6wuYA3Pa29hTy9IyOzgCykJBUJvFeoL4BFaAgoIIK1KDkkKBagjyRaVSQgSag-jBa3gtMrOyQiMQ1LGwcAIJ0oQpincn8QqKMkAAe8yL+HFept0EhYTRKVKqWkmk0kQCh4MESJzOFzMoFIz1ARwU-i0HTMCUqAzAoTkP2USEmImWq3WgQO2w4w1GdFIzGYoBBYOgnmqdmkTTErXanU8Pj8gUBQOx4UiMTiiQAKgUJpC4OdUaYuLxrnMhm0qBxIPQCqBmrJ-KQAJ6wxhYAAMBii6Kq0hq0lMohgpHmACVeAh1TyenzPoKwhForF4gkAOIO2jy2GnWXQhXQN0CEpRS2VFk1SKMFYpG5iVBvbNUfqDOAAa3ycCYcadE1IoDjADMYKI6BM-rDCXUaA18gawcRM8qPgBePSQADuiqzc08hbAMHg0FQ1JEIjgjDolfmbYJ1DhzdAcDrW47tAYm0OfaV73Sw5EY4RUbll0v+enmP3pdb681m7hgUOWlMds9FkABaY8uxBcgyV2fsryoKJ2VcdxpwvSdbgQpwOTVbkU1g-MontW9w0gV0eHdRBX0GOcEFQPCVWKYDGAPbswXvKF5TzOYABomDhFdGHoZt2FAQjHRdeN1SAA)

## 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()); // "Hello, world"[Try](https://www.typescriptlang.org/play/#code/MYGwhgzhAEDiBOBTRAXR9oG8BQ1oHMlUBLAO3wC5oIV4z8BubXaYAe1JvgFdgU34ACgC2iKGHyIqXegEosLPCgAWxCADpCyFPWgBeaKPGSmeAL7M8W1IPk48eJCm7xS0AEQAJRCBBsANB7QANTQKmqaRDrkptAWFtggqARR6FQI2uhM1mgYBqSIAO5wqULuhQIgACbuskzsnGxJ6n74gjnokdq2ddAA9H0e3r4B0BXw1e5AA)

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()); // "Hello, world"[Try](https://www.typescriptlang.org/play/#code/PTAEAEGcBcCcEsDG0BcoBmBDANpApgFDZ7SgDiseJesoAvKABToCuAdsvAPZtMCUoAN4FQGdpx7lK1WIwC2eSJEwBzPAOGjR0ABbxIAOhXTo8NivqgFS1XgDcI0AF8CjilWg0DAB1hdo-gCe3nhGJpasHKaSjBqOopTQLLC8AEQAEnjY2FwANKCpoADUoLr6YR5mKg6iTg6Oicm87jIOTnyx9cSkxh40Dr0ylmx4AO5SfbKpo1yw2AAmqXwOiDyQXMQGOSqMg56wFSSxy6AgBZnZeaAzc4tAA)

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()); // "Hello, there"

let greeterMaker: typeof Greeter = Greeter;

greeterMaker.standardGreeting = "Hey there!";

let greeter2: Greeter = new greeterMaker();

console.log(greeter2.greet()); // "Hey there!"

let greeter3: Greeter;

greeter3 = new Greeter();

console.log(greeter3.greet()); // "Hey there!"[Try](https://www.typescriptlang.org/play/#code/PTAEAEGcBcCcEsDG0BcoBmBDANpApgFCLaaSSgDisee0esoA3gaKDJtEm9JgHYAmmWPyo1OvAOagAvKABEACTzZsAewA0oaAAt6eOQG4WoCdVrxJaGAklHWpsQAoAlE2Ot46UI53xIAOgdzSVdmVnDQamgAV1heeSUVDXlQAGotbT9As3EJO3CAX1BlfDcI1ijY+NFaen92ASERHIs891AC407OgmxaExz6AEY0GrpYIyDxoZlQXjwAd0pB2BcjRFVeSFU+-zUJRynh7KdnZwNQEATlNU0dPTkCXv6j2ABZTABrejRoAE8AA54VReMb0WZgiYEV4fb6weo8RrCMatWaKPB-DJ6ACEhiefWgAzE9AATKMVrN5ksYV96GsiJttrt9ocViSTrQXOdLmB0Zj7tRcfiXisAMzk4lQ16iymLZaS+kbLY7PB7VQHaUc6Bci5XPlYwVyIA)

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. It is also good to mention that changing static property is frowned upon, here greeter3 has "Hey there!" instead of "Hello, there" on standardGreeting.

## 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 };