If you've ever written code in Java or C#, then you are probably familiar with class modifiers. These are keywords that we place before our property and method definitions which define how and when the fields on our class can be accessed. We can use modifiers to tell the TypeScript type checker how to treat our class members. The three class modifiers are `public`, `private`, and `protected`.

Public

By default, class fields are marked as `public`. This makes the fields accessible both inside and outside of the class.

```
class Fruit {
  name: string; // `public` modifier is implied
  public sweetness: number;
  constructor(name: string, sweetness: number) {
    this.name = name;
    this.sweetness = sweetness;
  }

  public sayName() {
    console.log(this.name);
  }
```

```
}
const apple = new Fruit("Apple", 80);
apple.name = "Banana";

apple.sayName(); // "Banana"
```

This is useful when we want to easily edit and modify properties or call methods outside of the class, but also gives any consumers of this class access to all of the class internals.

Protected

What if we were writing a class that will be used by other developers. Many of the parts of the class should be accessed by other developers, but other parts contain implementation details that we would prefer to keep to ourselves. We can use the `protected` and `private` modifiers to stop other developers from accessing certain properties and methods.

The `protected` modifier limits access to class properties and methods to the class itself and any classes that inherit the parent class. If we try to access a `protected` property or method, TypeScript will throw a type error.

```
class EdibleThing {
  protected name: string;
  constructor(name: string) {
    // We can set and modify the name from within thi
    this.name = name;
  }
}
class Fruit extends EdibleThing {
  protected sweetness: number;
  constructor(name: string, sweetness: number) {
    super(name);
    this.sweetness = sweetness;
  }
  public sayName() {
    console.log(this.name);
```

```
    }
}
const apple = new Fruit("Apple", 80);
apple.name = "Banana"; // Type Error: Property 'name'

apple.sayName(); // "Apple"
```

We can access the `public sayName` method, which accesses the `protected name` property on the parent class, but TypeScript warns us if we try to access `protected name`. However, if we were to ignore the warning and compile our TypeScript to JavaScript, it would still be able to access the property. TypeScript doesn't add any runtime protections to `protected` class fields, so JavaScript doesn't know that we shouldn't be allowed to access the property.

Private

`private` is the most restrictive of the class modifiers. It only allows access to properties on the class they are defined on. Trying to access the field in child classes and outside the class causes TypeScript to throw a type error.

```
class EdibleThing {
  private name: string; // Notice the modifier has be
  constructor(name: string) {
    this.name = name;
  }
}
class Fruit extends EdibleThing {
  protected sweetness: number;
  constructor(name: string, sweetness: number) {
    super(name);
    this.sweetness = sweetness;
  }
  public sayName() {
    console.log(this.name); // Type Error: Property '
  }
}
```

`readonly`

`readonly` is another modifier which is not limited to classes. You can use it on Interfaces, objects, arrays, or any other type as well. Adding this modifier to a property makes it so you cannot mutate or make changes to the property whatsoever. You can think of this as the `const` variable declaration, but for object properties.

```
type readOnlyFruit = { readonly name: string };
const apple: readOnlyFruit = { name: "Apple" };
apple.name = "Banana"; // Type Error: Cannot assign t
```

You can apply multiple layers of `readonly` modifiers to further limit what can be changed.

```
type fruitBasket = {
  readonly fruitList: readonly string[];
};
const basket: fruitBasket = { fruitList: ["Apple"] };
basket.fruitList[0] = "Banana"; // Type Error: Index
```

Property Shorthand

TypeScript gives us a shorthand when our constructor parameters are the same as our properties. By assigning a modifier to constructor parameters, TypeScript will automatically assign them to the appropriate property without us even having to define the property on the class. This can save us a bit of typing.

```
class Fruit {
  constructor(public name: string, protected sweetnes
}
const apple = new Fruit("Apple", 80);
console.log(apple); // Fruit { name: "Apple", sweetne
```

A Note about JavaScript support

Everything we've talked about class modifiers so far only exists in TypeScript. If you try to use these features when writing JavaScript, you will get a syntax error. Everything else in this section is a feature of JavaScript, and you can use them so long as you don't include the class field modifiers.

Accessors (get and set)

JavaScript provides the ability to add "accessors" to classes. These define virtual properties which call functions whenever a class property is read or changed. These functions have full access to the class instance and can read, transform, and modify any other instance property. We define accessors using the `get` and `set` keyword.

Here, we'll create a class that ensures the name is always capitalized.

```
class Fruit {
  constructor(protected storedName: string) {}
  set name(nameInput: string) {
    this.storedName = nameInput;
  }
  get name() {
    return (
      this.storedName[0].toUpperCase() + this.storedN
    );
  }
}

const apple = new Fruit("apple");
apple.name; // "Apple"
apple.name = "banana";
apple.name; // "Banana"
```

Notice that I store the name on a different property. If I were to try mutating `this.name` within my `set name()` method, it would call `set name()`

again, creating an infinite recursive function.

Class accessors which only use `get` but not `set` are automatically marked as `readonly` by TypeScript.

```
class Fruit {
  protected storedName: string;
  constructor(name: string) {
    this.storedName = name;
  }
  get name() {
    return (
      this.storedName[0].toUpperCase() + this.storedN
    );
  }
}
const apple = new Fruit("apple");
apple.name = "banana"; // Type Error: Cannot assign t
```

Adding `set` to classes can be helpful when we want to validate the input before assigning it to the class.

```
class Fruit {
  constructor(
    protected name: string,
    protected storedSweetness: number,
  ) {}
  get sweetness() {
    return this.storedSweetness;
  }
  set sweetness(sweetnessValue: number) {
    if (sweetnessValue < 0) {
      throw new Error("Sweetness cannot be less than
    }
    if (sweetnessValue > 100) {
      throw new Error("Sweetness cannot be greater th
```

```
    }
    this.storedSweetness = sweetnessValue;
  }
}
```

Here, we throw an error if the provided value is too high or too low.

ES Private Fields

ES Private Fields are a relatively new addition to the JavaScript language, and are distinctly different from TypeScript private fields. Since ES Private Fields are part of the JavaScript language, they have runtime guarantees which TypeScript cannot provide. If you mark a field as an ES Private Field, that field *will not* be accessible outside of the class instance.

ES Private Fields are indicated with a ~~pound sign~~ ~~hashtag~~ ~~octothorpe~~ number sign (**#**) immediately before the property name. They are accessed in the same way, which means you can have getters and setters that publicly expose the ES Private Field with the same name.

```
class Fruit {
  #name: string;
  constructor(name: string) {
    this.#name = name;
  }
  get name() {
    return this.#name[0].toUpperCase() + this.#name.s
  }
  set name(name: string) {
    this.#name = name;
  }
}
const apple = new Fruit("apple");
console.log(apple.name); // "Apple"
apple.#name = "banana"; // Type Error: Property '#nam
```

`private` or `#private`?

There are a number of factors that go into whether you should use TypeScript's `private` modifier or the ES Private Field for a specific property.

ES Private Fields provide "hard privacy", which means an outside viewer can't see those properties even if they wanted to. This stops consumers of the class from relying on internal fields, making it easier for you to change your implementation without disrupting how users consume the API. Also, classes that extend your class still won't have access to the private fields, which can help you avoid field naming collisions.

By contrast, TypeScript's `private` modifier provides "soft privacy", allowing consumers of the class to read and edit those fields at runtime. This might be helpful for users that need to work around your API or dive in to see how the internals work.

As a rule of thumb, ES Private fields are generally what you want to use. However, if you can't target ES Next when you compile your code, or if you can't afford to have a lot of polyfill code, TypeScript's `private` modifier is a good alternative. Also, if you need classes that extend the base class to access a private property, you should use the `protected` modifier.

Next Lesson or discuss in the Community