Prog. Orientada a Objetos

Carrera Programador full-stack

Herencia

Agenda

- Noción de generalización
- Ejemplos de la vida real
- Herencia en TypeScript
- Uso de *super*
- Uso de *protected*
- Clase Televisor usando Herencia
- Herencia vs. Composición
- Recomendaciones
- Ejercicios

Noción de Generalización

- Normalmente generalizamos las cosas para poder entenderlas mejor
- Generalizar es asignar una serie de características a un objeto
  - Por ejemplo nos muestran un dispositivo raro y nos dicen que es un teléfono
  - Automáticamente suponemos que por tratarse de un teléfono, va a hacer llamadas
  - Estamos generalizando que los teléfonos hacen llamadas
- La Programación Orientada a Objetos, por inspirarse en la vida real, también refleja este concepto de generalización → *Herencia*

Herencia en la Vida Real

- Clase Televisor
  - Televisor de Tubo
  - Televisor Plasma
  - Televisor LCD
  - Televisor LED
- Clase Teléfono
  - Teléfono Celular (los primeros)
  - Smartphone
- En ambos casos, por el hecho de tratarse de un Televisor o de un Teléfono, sabemos automáticamente que tienen una series de características
  - En el caso de Televisor → muestra una imagen
  - En el caso de Teléfono → hace llamadas

Herencia en la Vida Real (2)

- Clase Vehiculo
  - Automovil
  - Camioneta
  - Camion
- Clase Perro
  - Doberman
  - Chihuahua
- En ambos casos, por el hecho de tratarse de un Vehiculo o de un Perro, sabemos automáticamente que tienen una series de características
  - En el caso de Vehiculo→ arranca/frena
  - En el caso de Perro→ ladra/corre

Herencia en TypeScript (1)

```
class Televisor {
    private canalActual: number;
    private volumenActual: number;
    private estaPrendido: boolean;

    public constructor() {
        this.canalActual = 0;
        this.volumenActual = 10;
        this.estaPrendido = false;
    }

    ...
}
```

```
class SmartTV extends Televisor {
    public constructor() {

    }
}
```

```
let tele = new SmartTV();

console.log(tele);
```

```
PS C:\Users\Francisco\Documents\CFP\3. POO\Ejercicios\poo\herencia> tsc televisor.ts; node televisor.js
SmartTV { canalActual: 0, volumenActual: 10, estaPrendido: false }
```

SmartTV tiene lo mismo que Televisor

Herencia en TypeScript (1B)

```
class Vehiculo{
   private cantidadCilindros: number;
   private cantidadAsientos: number;
   private estaPrendido: boolean;

   public constructor() {
       this.cantidadCilindros = 8;
       this.cantidadAsientos = 5;
       this.estaPrendido = false;
   }

   ...
}
```

```
class Automovil extends Vehiculo{
   public constructor() {

   }
}
```

```
let ferrari = new Automovil();

console.log(ferrari);
```

Automovil tiene lo mismo que Vehculo

Herencia en TypeScript (2)

```typescript
class Televisor {
    private canalActual: number;
    private volumenActual: number;
    private estaPrendido: boolean;

    public constructor() {
        this.canalActual = 0;
        this.volumenActual = 10;
        this.estaPrendido = false;
    }

    public cambiarCanal(canal: number): void {
        this.canalActual = canal;
    }

    ...
}
```

```typescript
class SmartTV extends Televisor {
    public constructor() {
        super();
    }
}
```

Y esto?

```typescript
let tele = new SmartTV();

console.log(tele);

tele.cambiarCanal(30);

console.log(tele);
```

```
PS C:\Users\Francisco\Documents\CFP\3. POO\Ejercicios\poo\herencia> tsc televisor.ts; node televisor.js
SmartTV { canalActual: 0, volumenActual: 10, estaPrendido: false }
SmartTV { canalActual: 30, volumenActual: 10, estaPrendido: false }
```

También aplica a los métodos!

Herencia en TypeScript (2B)

```
class Vehiculo{
    private cantidadCilindros: number;
    private cantidadAsientos: number;
    private estaPrendido: boolean;

    public constructor() {
        this.cantidadCilindros = 8;
        this.cantidadAsientos = 5;
        this.estaPrendido = false;
    }


    public sumarAsientos(asientos: number): void {
        this.cantidadAsientos = asientos;
    }

    ...
}
```

```
class Automovil extends Vehiculo {
    public constructor() {
        super();
    }
}
```

Y esto?

```
let ferrari= new Automovil();

console.log(ferrari);

ferrari.sumarAsientos(1);

console.log(ferrari);
```

También aplica a los métodos!

Beneficios de usar TypeScript

```typescript
class Televisor {
    private canalActual: number;
    private volumenActual: number;
    private estaPrendido: boolean;

    constructor() {
        this.canalActual = 0;
        this.volumenActual = 10;
        this.estaPrendido = false;
    }

    public cambiarCanal(canal: number): void {
        this.canalActual = canal;
    }

    public cambiarVolumen(volumen: number): void {
        this.volumenActual = volumen;
    }

    public prenderApagar(): void {
        if (this.estaPrendido)
            this.estaPrendido = false;
        else
            this.estaPrendido = true;
    }
}
```

Código TypeScript

¿Cuál les parece que es más fácil de entender?

```javascript
var __extends = (this && this.__extends) || (function () {
    var extendStatics = function (d, b) {
        extendStatics = Object.setPrototypeOf ||
            ({ __proto__: [] } instanceof Array && function (d, b) { d.__proto__ = b; }) ||
            function (d, b) { for (var p in b) if (b.hasOwnProperty(p)) d[p] = b[p]; };
        return extendStatics(d, b);
    };
    return function (d, b) {
        extendStatics(d, b);
        function __() { this.constructor = d; }
        d.prototype = b === null ? Object.create(b) : (__.prototype = b.prototype, new __());
    };
})();
var Televisor = /** @class */ (function () {
    function Televisor() {
        this.canalActual = 0;
        this.volumenActual = 10;
        this.estaPrendido = false;
    }
    Televisor.prototype.cambiarCanal = function (canal) {
        this.canalActual = canal;
    };
    Televisor.prototype.cambiarVolumen = function (volumen) {
        this.volumenActual = volumen;
    };
    Televisor.prototype.prenderApagar = function () {
        if (this.estaPrendido)
            this.estaPrendido = false;
        else
            this.estaPrendido = true;
    };
    return Televisor;
}());
var SmartTV = /** @class */ (function (_super) {
    __extends(SmartTV, _super);
    function SmartTV() {
        return _super.call(this) || this;
    }
    return SmartTV;
}(Televisor));
```

Código JavaScript

Uso de *super*

- La restricción que tenemos al hacer herencia es que tenemos que invocar al constructor de la clase padre (o *superclase*)
- Si no se invoca → error al hacer “tsc …”
- Garantiza que las variables de la superclase se inicialicen de la manera esperada
- En la práctica van a utilizar una determinada librería en donde van a tener que heredar de una determinada clase para hacer algo determinado

Uso de *protected*

- Puede ocurrir que en una subclase, se quiera acceder/modificar el valor de una variable interna de la superclase
- Recordar que las variables internas se escriben con *private*, ya que las usa la misma clase
- Hay una forma de que las variables sigan siendo privadas para el exterior, pero que puedan ser accedidas desde una subclase → *protected*
- Si desde una subclase, queremos acceder a una variable privada de la superclase → Error al hacer “tsc …”

Uso de *protected* - Ejemplo

```
class Televisor {
    private canalActual: number;
    protected volumenActual: number;
    private estaPrendido: boolean;

    public constructor() {
        this.canalActual = 0;
        this.volumenActual = 10;
        this.estaPrendido = false;
    }

    ...
}
```

```
class SmartTV extends Televisor {
    public constructor() {
        super();
    }

    public subirVolumen(): void {
        this.volumenActual += 1;
    }
}


let tele = new SmartTV();

console.log(tele);

tele.subirVolumen();

console.log(tele);
```

```
PS C:\Users\Francisco\Documents\CFP\3. POO\Ejercicios\poo\herencia> tsc televisor.ts; node televisor.js
SmartTV { canalActual: 0, volumenActual: 10, estaPrendido: false }
SmartTV { canalActual: 0, volumenActual: 11, estaPrendido: false }
```

Desde la subclase accedemos a una variable de la superclase

Clase Televisor empleando Herencia

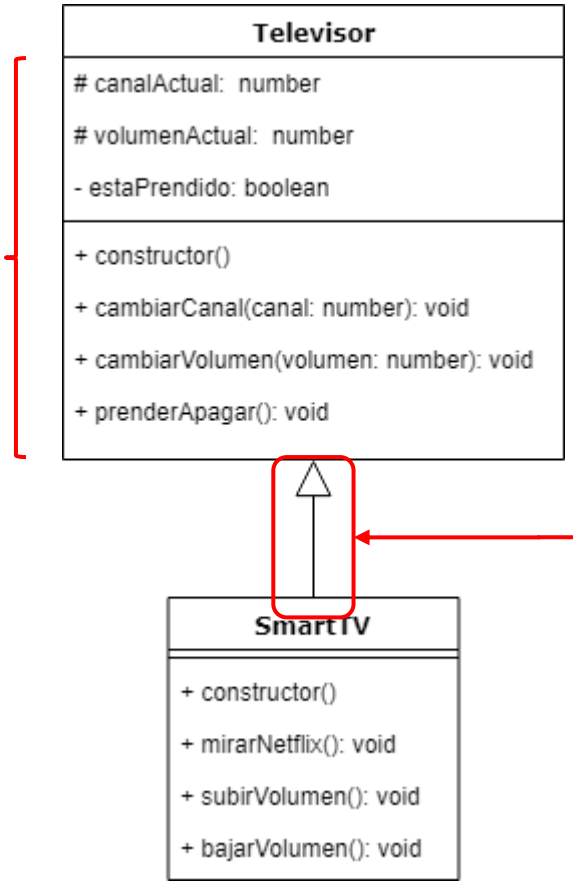
```
class Televisor {
    protected canalActual: number;
    protected volumenActual: number;
    private estaPrendido: boolean;

    public constructor() {
        this.canalActual = 0;
        this.volumenActual = 10;
        this.estaPrendido = false;
    }

    public cambiarCanal(canal: number): void {
        this.canalActual = canal;
    }

    public cambiarVolumen(volumen: number): void {
        this.volumenActual = volumen;
    }

    public prenderApagar(): void {
        if (this.estaPrendido)
            this.estaPrendido = false;
        else
            this.estaPrendido = true;
    }
}
```

```
class SmartTV extends Televisor {
    public constructor() {
        super();

        this.canalActual = 1;
    }

    public mirarNetflix(): void {
        console.log('Mirando Netflix...');
    }

    public subirVolumen(): void {
        this.volumenActual += 1;
    }

    public bajarVolumen(): void {
        this.volumenActual -= 1;
    }
}
```

***OPCIONAL*: Copiar el código y jugar con private/protected**

Representación Clase Televisor

 # → protected
- → private
+ → public



Indica de qué clase hereda

Repaso de Composición

```
class Televisor {
    private botonPrendido: Boton;

    private botonSubirVolumen: Boton;
    private botonBajarVolumen: Boton;

    private botonSubirCanal: Boton;
    private botonBajarCanal: Boton;

    private pantallaTelevisor: Pantalla;

    ...
}
```

```
class Auto {
    private ruedaDelanteraDerecha: Rueda;
    private ruedaDelanteraIzquierda: Rueda;
    private ruedaTraseraDerecha: Rueda;
    private ruedaTraseraIzquierda: Rueda;

    private volante: Volante;

    private palancaCambios: TransmisionManual;

    ...
}
```

Clases más *sencillas* componen una clase más *compleja*

Televisor y Auto están *compuestas* por clases más *simples*

Herencia vs. Composición (1)

Suponer que teniendo una clase ya implementada, queremos implementar *una nueva*

- Si hay cosas en común → Herencia
  - Tener cuidado → no abusar!
  - Las cosas en común tienen que tener sentido
- No necesito tener acceso a variables internas de una determinada clase → Composición

Herencia vs. Composición (2)

Suponer que teniendo una clase ya implementada, queremos implementar *una nueva*

- Ejemplo 1: quiero usar todos los métodos de la clase implementada, pero agregar *uno* más
  - La idea es escribir la menor cantidad posible de código
- Ejemplo 2: quiero usar solo algunos métodos de la clase que tengo implementada
  - La idea es exponer solamente la funcionalidad necesaria

Recomendaciones

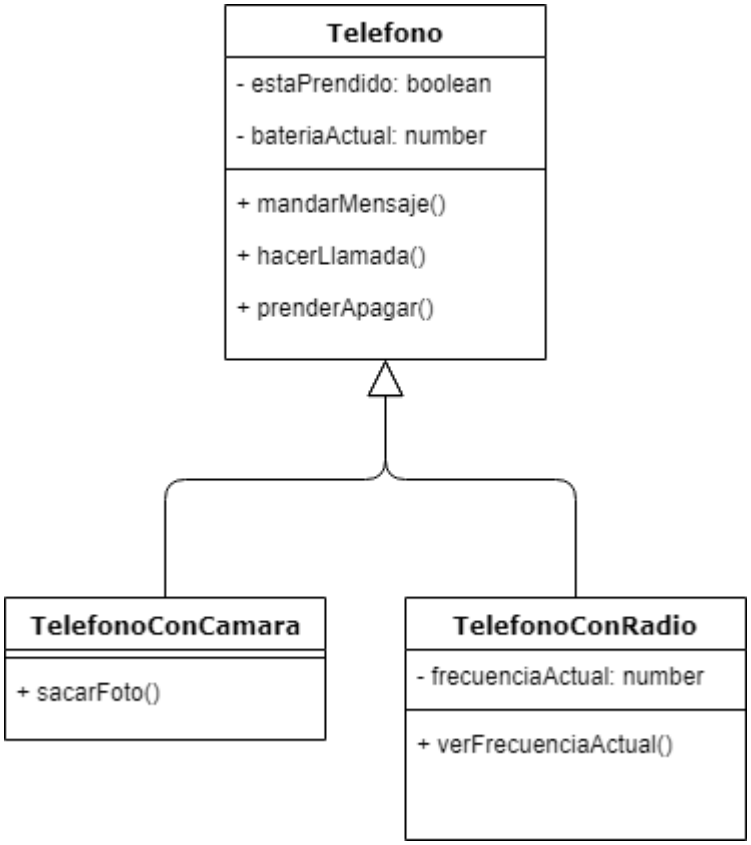
- La idea es evitar duplicar código *siempre que se pueda*
- En caso de aprovechar una clase empleando herencia o composición → siempre pensar en la forma de escribir el *mínimo código posible*
- No exponer funcionalidad de forma innecesaria
  - No se pueden heredar algunos métodos o variables → todo o nada
- Paciencia para entender bien → No son conceptos fáciles, con la práctica se van ajustando

Prog. Orientada a Objetos

Carrera Programador full-stack

Ejercicios

Ejercicios - En Clase



- Implementar las clases y métodos que se muestran
- Agregar variables/métodos adicionales
- Implementar cada clase en un archivo diferente
  - Ojo con la forma de hacer los *import*
- Subir las cosas a GitHub y avisar por Slack

Ejercicios - Fuera de Clase

- Crear proyecto NPM
- Subir proyecto a GitHub
- Implementar Registro Automotor visto anteriormente, pero agregando soporte de motos y camiones usando herencia
- Definir tarea NPM para compilar y correr los archivos necesarios
- Enviar por Slack el link al repositorio de GitHub

Ejercicios - En Clase

- Crear proyecto NPM
- Subir proyecto a GitHub
- Implementar una Selección de Fútbol, conformada por Futbolistas, Entrenador y Masajista.
- Aplicar herencia donde sea posible.