57 - TypeScript: clases genéricas

TypeScript permite crear clases y funciones que administren distintos tipos de datos. Esta característica no existe en JavaScript

Se utilizan mucho para la administración de colecciones de datos (pilas, colas, listas etc.)

Para entender las ventajas de definir clases genéricas implementaremos los algoritmos para administrar una pila de enteros y una pila de string. Primero lo haremos utilizando clases tradicionales y luego mediante una clase genérica.

Para concentrarnos en la sintaxis plantearemos la pila utilizando un vector donde definiremos los dos métodos fundamentales de insertar y extraer (no haremos ningún tipo de validaciones por simplicidad)

```
class PilaEnteros {
  private vec: number[] = [];

  insertar(x: number) {
    this.vec.push(x);
  }

  extraer() {
    if (this.vec.length > 0)
      return this.vec.pop();
    else
      return null;
  }
}

class PilaStrings {
  private vec: string[] = [];

  insertar(x: string) {
    this.vec.push(x);
  }

  extraer() {
    if (this.vec.length > 0)
      return this.vec.pop();
    else
      return null;
  }
}

let pila1 = new PilaEnteros();
pila1.insertar(20);
pila1.insertar(43);
pila1.insertar(1);
console.log(pila1.extraer());

let pila2 = new PilaStrings();
pila2.insertar('juan');
pila2.insertar('ana');
pila2.insertar('luis');
console.log(pila2.extraer());
```

Como podemos analizar hemos planteado dos clases, una para administrar una pila con tipos de dato enteros:

```
class PilaEnteros {
  private vec: number[] = [];

  insertar(x: number) {
    this.vec.push(x);
  }

  extraer() {
    if (this.vec.length > 0)
      return this.vec.pop();
    else
      return null;
  }
}
```

Y por otro lado otra clase para administrar una pila de tipo de dato string:

```
class PilaStrings {
  private vec: string[] = [];

  insertar(x: string) {
    this.vec.push(x);
  }

  extraer() {
    if (this.vec.length > 0)
      return this.vec.pop();
    else
      return null;
  }
}
```

Para probar estas dos clases definimos un objeto de la clase PilaEnteros e insertamos tres valores y luego extraemos uno:

```
let pila1=new PilaEnteros();
pila1.insertar(20);
pila1.insertar(43);
pila1.insertar(1);
console.log(pila1.extraer()); //se imprime un 1 ya que el último en entrar es el primero en salir en una pila.
```

De forma similar probamos creando un objeto de la clase PilaStrings:

```
let pila2=new PilaStrings();
pila2.insertar('juan');
pila2.insertar('ana');
pila2.insertar('luis');
console.log(pila2.extraer()); //se imprime 'luis'
```

Hasta este momento no hemos presentado ninguna novedad con respecto a lo que conocemos. Veamos ahora como podemos resolver este problema pero empleando una clase genérica:

```
class PilaGenerica<T>{
  private vec: T[] = [];

  insertar(x: T) {
    this.vec.push(x);
  }

  extraer() {
    if (this.vec.length > 0)
      return this.vec.pop();
    else
      return null;
  }
}

let pila3: PilaGenerica<number>;
pila3 = new PilaGenerica<number>();
pila3.insertar(20);
pila3.insertar(42);
pila3.insertar(1);
console.log(pila3.extraer());

let pila4: PilaGenerica<string>;
pila4 = new PilaGenerica<string>();
pila4.insertar('juan');
pila4.insertar('ana');
pila4.insertar('luis');
console.log(pila4.extraer());
```

Como vemos hemos declarado una sola clase llamada PilaGenerica y hemos sustituido en los lugares donde hacíamos referencia a number o string por el tipo 'T' que también tenemos que hacer referencia en donde declaramos la clase:

```
class PilaGenerica<T>{
  private vec: T[] = [];

  insertar(x: T) {
    this.vec.push(x);
  }

  extraer() {
    if (this.vec.length > 0)
      return this.vec.pop();
    else
      return null;
  }
}
```

Luego cuando creamos un objeto de la clase PilaGenerica debemos indicar cuando la creamos el tipo de datos que administrará nuestra pila:

```
let pila4:PilaGenerica<string>;
pila4=new PilaGenerica<string>();
```

Podemos crear objetos de la clase PilaGenerica con cualquier tipo de dato primitivo (number, string, boolean etc.) o de otra clase.

Problemos de codificar otro programa que cree una pila de otra clase creada por nosotros:

```
class PilaGenerica<T>{
  private vec: T[] = [];

  insertar(x: T) {
    this.vec.push(x);
  }

  extraer() {
    if (this.vec.length > 0)
      return this.vec.pop();
    else
      return null;
  }
}

class Persona {
  constructor(public nombre: string, public edad: number) { }
}

let pila5: PilaGenerica<Persona>;
pila5 = new PilaGenerica<Persona>();
pila5.insertar(new Persona('pedro', 33));
pila5.insertar(new Persona('maria', 33));
pila5.insertar(new Persona('marcos', 33));
console.log(pila5.extraer());
```

Creamos un objeto de la clase PilaGenerica e indicamos que almacenará objetos de la clase Persona:

```
let pila5:PilaGenerica<Persona>;
pila5=new PilaGenerica<Persona>();
```

Creamos tres objetos de la clase Persona y los insertamos en la Pila:

```
pila5.insertar(new Persona('pedro', 33));
pila5.insertar(new Persona('maria', 33));
pila5.insertar(new Persona('marcos', 33));
```

Si extraemos un elemento de la pila tenemos la última persona ingresada:

```
console.log(pila5.extraer());
```

El planteo de clases genéricas nos reduce tener que crear múltiples clases para administrar distintos tipos de datos.

Retornar