

TYPESCRIPT:

ausserdem:

S = Single Resp.

O = Open/Close (Open zu Erweiterungen, geschlossen für Anpassungen)

L = Leskov substitutionsprinzip

I = Interface Segregation Prinzip

D = Dependency Inversion Prinzip (Die Abhängigkeit umkehren, so dass Elemente auf höherer Ebene nicht abhängig sind detailelementen).

- **SOLID**

- Single responsibility principle
 - Klasse / Funktion dient einem gezieltem Zweck
- Open/Close principle (Open for extension, closed for modification)
 - Das Verhalten kann erweitert aber nicht modifiziert werden
- Liskov substitution principle
 - Es werden klare «Verträge» ausgehandelt die eingehalten werden müssen egal welche Implementation verwendet wird (Design by Contract)
- Interface segregation principle
 - Lieber mehrere spezifische Interfaces als ein riesiges generelles
- Dependency inversion principle

constructor function:

Hier muss eine function mitgegeben werden, (einfach return new T - geht nicht)



TypeScript

Share

Options

```
1
2 class B { }
3 class A extends B { }
4 class C extends B { }
5 class D { }
6
7
8 class Factory<T>
9 {
10     createNew(t : new() => T): T {
11         return new t();
12     }
13 }
14
15 let f = new Factory<A>();
16 let fa = f.createNew(A);
17
18 let f1 = new Factory<D>();
19 let d = f1.createNew(D);
```

typescript decorators:

2. Method decorators

A method decorator is a function that accepts 3 arguments: the object on which the method is defined, the key for the property (a string name or **symbol**) and a **property descriptor**. The function returns a property descriptor; returning undefined is equivalent to returning the descriptor passed in as argument.

```
1  const log = (target: Object, key: string | symbol, descriptor: TypedPropertyDescriptor<Function:
2      return {
3          value: function( ... args: any[]) {
4              console.log("Arguments: ", args.join(", "));
5              const result = descriptor.value.apply(target, args);
6              console.log("Result: ", result);
7              return result;
8          }
9      }
10 }
11
12 class Calculator {
13     @log
14     add(x: number, y: number) {
15         return x + y;
16     }
17 }
18
19 new Calculator().add(1, 3);
20 //Arguments: 1, 3
21 //Result: 4
```

ES gibt noch Constructor Binding und Property Binding

Async (auch im es6 möglich)

```
1
2 function delay(milliseconds: number) {
3     return new Promise<void>(resolve => {
4         setTimeout(resolve, milliseconds)
5     })
6 }
7
8 async function dramaticWelcome() {
9     console.log("Hello2");
10    for (let i = 1; i < 3; i++) {
11        await delay(1500);
12        console.log('.') + i);
13    }
14    console.log('World');
15 }
16
17 dramaticWelcome();
```

Output:(im browser)

```
Hello2
.1
.2
World
```


Typeguards (um anhand von classeninfos (properties) den Typen zurückzuliefert - und Frage beantworte ob pet is fish == true ist -> ein eigentlich schlechtes Beispiel um die Basisklasse generischer zu machen -> Leskov-Prinzip wird so umgangen)

- Type Guards können verwendet werden um dem Compiler zusätzliche Typeninformationen zu liefern um Casting zu verhindern

```
interface Bird {  
    fly();  
    layEggs();  
}  
  
interface Fish {  
    swim();  
    layEggs();  
}  
  
function getSmallPet(): Fish | Bird {  
    // ...  
}
```

```
function isFish(pet: Fish | Bird): pet is Fish {  
    return (<Fish>pet).swim !== undefined;  
}
```

```
let pet = getSmallPet();  
  
if ((<Fish>pet).swim) {  
    (<Fish>pet).swim();  
}  
else {  
    (<Bird>pet).fly();  
}
```



```
if (isFish(pet)) {  
    pet.swim();  
}  
else {  
    pet.fly();  
}
```