

Training TypeScript

Module: Generics



Peter Kassenaar

info@kassenaar.com

A generic is a "code template"
that relies on type variables:

`<T>`

Generics Features

Provide **reusable**
code templates

Provide more flexibility when
working with types

Compile-time only checks

Can be used in many scenarios
(classes, functions, etc.)

Can minimize the use of "any"

Why The Need for Generics?

```
class ListOfNumbers {  
    _items: number[] = [];  
  
    add(item: number) {  
        this._items.push(item);  
    }  
  
    getItems(): number[] {  
        return this._items;  
    }  
}
```

```
class ListOfString {  
    _items: string[] = [];  
  
    add(item: string) {  
        this._items.push(item);  
    }  
  
    getItems(): string[] {  
        return this._items;  
    }  
}
```

You find yourself writing duplicate code...

The Answer is Generics

```
class List<T> {  
    _items: T[] = [];  
  
    add(item: T) {  
        this._items.push(item);  
    }  
  
    getItems(): T[] {  
        return this._items;  
    }  
}
```

```
const nameList = new List<string>();
```

```
class List {  
    _items: string[] = [];  
  
    add(item: string) {  
        this._items.push(item);  
    }  
  
    getItems(): string[] {  
        return this._items;  
    }  
}
```



No Generics in transpiled Code

- Again: Generics benefits [only] the developer!
- No type information is found in transpiled code

```
// generics-list.js
var List = (function () {
    function List() {
        this._items = [];
    }
    List.prototype.add = function (item) {
        this._items.push(item);
    };
    List.prototype.getItems = function () {
        return this._items;
    };
    return List;
})();
var nameList = new List();
nameList.add('Peter');
nameList.add('Sandra');
...
```


Simple use case of generics

```
// function accepts generic parameter. Letter <T> is just a choice.  
// Could also have been S, or R, or any other character  
function echo<T>(arg: T): T {  
    return arg;  
}  
  
let msg = echo('Hello world'); // msg is of type 'string'  
console.log(msg);  
  
let myNumber = echo(123); // myNumber is of type 'number'  
console.log(myNumber);
```

Generics prevent development time errors

```
var numList = new List<number>();  
numList.add(1);  
numList.add(8);  
numList.add('Hello world'); // oops!  
console.log('numList: ', numList);
```

```
numList.add(2);  
numList.add(3);  
numList.add([ts] Argument of type '"Hello world"' is not assi  
numList.add(gnable to parameter of type 'number'.  
numList.add('Hello world'); // oops!  
console.log('numList: ', numList);
```



Generic Constraints

- Sometimes you need more information on a specific type **inside** a generic function
- AND: you want **type safety** on that type.
- This is where **generic constraints** come in.
- You can tell TypeScript that an incoming type `extends` some other type, so you have type safety on that.
- First, let's look at interfaces:
 - `../16-generic-interface.ts`

```
// 0. Set up some interfaces and some data.
```

```
interface Employee {  
  name: string;  
  age: number;  
}
```


```
let admin: Employee = {  
  name: 'Johan',  
  age: 34  
};
```

```
let manager: Employee = {  
  name: 'Bart',  
  age: 45  
};
```

1. No type information. This works, but gives you no type safety

```
function echoEmployees(person) {  
    return person;  
}  
  
let foo = echoEmployees(admin); // type of foo is now 'any'
```


```
}  
let foo: any  
let foo = echoEmployees(admin); // type of
```



2. With Generic: now we have at least type information on property 'person'.


```
function echoEmployees<T>(person: T): T {  
  return person;  
}  
  
let foo = echoEmployees(admin); // type of foo is now Employee (...)
```

```
}  
let foo: Employee  
let foo = echoEmployees(admin); //
```



3. With Generic and Type Constraint: we have type safety inside the function

```
function echoEmployees<T extends Employee>(person: T): T {  
  console.log('Name of the person: ', person.name);  
  return person;  
}  
  
let foo = echoEmployees(admin); //
```



```
function echoEmployees<T extends Employee>(person: T): T {  
  console.log('Name of the person: ', person.name);  
  return person;  
}
```



age
name (property) Employee.name: string ⓘ

Benefit of using a Generic Constraint

“When using an generic constraint, you are *absolutely sure* that the provided param at a minimum meets the fields/members of the constraint type”

```
function myFunction<T extends SomeType>(param: T): T {  
    // param meets minimal the fields of SomeType  
}
```

Same rule applies to classes

```
class Employee {  
    constructor(public name: string, public age: number) {}  
}  
  
class Admin extends Employee {}  
  
class Manager extends Employee {}  
  
let admin = new Admin('Johan', 33);  
let manager = new Manager('Bart', 45);
```

// 3. With Generic and Type Constraint: we have type safety inside the function

```
function echoEmployees<T extends Employee>(person: T): T {  
    console.log('Name of the person: ', person.name);  
    return person;  
}
```

Checkpoint

Generics are "code templates"

Generic templates rely on type variables: `<T>`

Generics templates are reusable

Generics provide more flexibility with types

Workshop

- Create a generic class `Queue` that has members:
 - `push()` - to add items to the queue
 - `pop()` - to remove items from the queue
 - `first()` - to return the first item in the queue
 - `last()` - to return the last item in the queue
 - Make sure that you can only add and remove items of the same type to a new `Queue()` instance.
 - Test this.
- Create a new class or interface (for instance `Person`)
- Make sure (by using a Type Constraint) that inside the function you can check for the `Person` properties that you are adding

