In Typescript what does <T> mean?

Asked 1 year, 8 months ago Active 2 months ago Viewed 10k times

30

```
export declare function createEntityAdapter<T>(options?: {
    selectId?: IdSelector<T>;
    sortComparer?: false | Comparer<T>;
}): EntityAdapter<T>;
```

Can someone explain to me what the means? I know `<>` is type assertion but I don't know what `'T'` is. It'd also be helpful if someone could explain to me what this function is doing.

8

typescript

edited Jun 27 '18 at 17:12

Erik Philips
44.7k 6 104 133

asked Apr 3 '18 at 4:01

Snorlax
2,371 3 18 41

2 Answers

58

> Can someone explain to me what `<T>` the means?

That is typescripts Generics declaration.

Excerpt:

> A major part of software engineering is building components that not only have well-defined and consistent APIs, but are also reusable. Components that are capable of working on the data of today as well as the data of tomorrow will give you the most flexible capabilities for building up large software systems.
>
> In languages like C# and Java, one of the main tools in the toolbox for creating reusable components is generics, that is, being able to create a component that can work over a variety of types rather than a single one. This allows users to consume these components and use their own types.

.

> I don't know what 'T' is.

`'T'` is going to be a type declared at run-time instead of *compile* time. The `T` variable could be

`TVehicle` or `TAnimal` to help denote a valid type for future programmers (and could be considered best practice because just `T` is not intuitive). I prefer `TSomething` because I know that uppercase T means a generic type. `WSometing` or `ASomething` is also valid, but I just don't prefer it. (Microsofts APIs are almost always TContext or TEntity for example).

> It'd also be helpful if someone could explain to me what this function is doing.

Well the function isn't *doing* anything. This is more declaring a type of function that can have multiple run-time type values. Instead of explaining that, I'll include an excerpt taken directly from the link above.

```
function identity<T>(arg: T): T {
  return arg;
}
```

which can be used like:

```
// type of output will be 'string'
let output = identity<string>("myString");
```

or

```
// type of output will be 'string', the compiler will figure out `T`
// based on the value passed in
let output = identity("myString");
```

or

```
// type of output will be 'number'
let output = identity(8675309);
```

Which might beg the question:

> Why use generics

Well Javascript has arrays, but when you retrieve a value from the array, it literally could be anything (typescript: `any`). With typescript you get Type safety by declaring them like:

```
// Array<T>
let list: number[] = [1, 2, 3];
// or
let list: Array<number> = [1, 2, 3];
```

Now each value in the array has a type. Typescript will throw a compile-time error if you attempt to put a string into this array. And you get type-safety and intellisense (depending on your editor) when you retrieve a value:

```
class Person {
  FirstName: string;
}
```

```
// john is of type Person, the typescript compiler knows this
// because we've declared the people variable as an array of Person

console.log(john.FirstName);
```

Declaring type'd generic constraints. A very good example of Open - Closed Principle.

> In object-oriented programming, the open/closed principle states "software entities (classes, modules, functions, etc.) should be open for extension, but closed for modification";[1] that is, such an entity can allow its behaviour to be extended without modifying its source code.

In the following example, anyone could extend Human or Cheeah or even create their own derived type and the Logger functionality would continue to work without any modification.

```
interface IAnimal {
  LegCount: number;
}

class Cheetah
  implements IAnimal {
  LegCount: number = 4;
}

class Human
  implements IAnimal {
  LegCount: number = 2;
}

public class Logger<TAnimal extends IAnimal> {
  public Log(animal: TAnimal) {
    console.log(animal.LegCount);
  }
}

var logger = new Logger();
var human = new Human();
logger.Log(human);
```

Working Example

In the previous example I used a Generic Constraint to limit the `TAnimal` type programmers can use to create a `Logger` instance to types that derive from the interface `IAnimal`. This allows the compiler to validate that the `Logger` class always assume the type has a property `LegCount`.

edited Oct 5 at 4:40

answered Apr 3 '18 at 4:02

Erik Philips
44.7k 6 104 133

4 Phillips I have also seen <R> and <U>. What do these mean or can I use any character inside the angled brackets? – Charles Robertson May 25 '18 at 9:44

1 @CharlesRobertson Updated answer, thanks that will be helpful for future readers, great question. – Erik Philips May 25 '18 at 13:00

6

The example you provide is a function with a generic parameter. `T` (which does not have to be `T`. You could call it `G`.) is called a generic template where actual type of the `T` is replaced at runtime.

Imagine EntityAdapter has following implementation:

```
interface EntityAdapter<T> {
   save(entity: T);
}
```

Below code returns an EntityAdapter which content is `any`. `any` could be number,string, object or anything.

```
let adapter1 = createEntityAdapter<any>(<parameters here>)
```

Below code returns an EntityAdapter which content is `Car`.

```
let adapter2 = createEntityAdapter<Car>(<parameters here>)
```

Basically `Car` is more specific than `any` so that you can get extra type safety. So how does this help?

In a nutshell, generic template can give you extra type safety. For example,

```
adapter1.save('I am string') // this works because `T` is `any`
adapter1.save(new Car()) // this also works because `T` is `any`

adapter2.save('I am string') // this wont works  because `T` is `Car`, typescript
compiler will complains
adapter2.save(new Car()) //this works because `T` is `Car`
```

Hope this helps.

answered Apr 3 '18 at 4:20

Emily Parker
181 4

I like the way you explained that T could be any character or set of characters. I have often see <R> & <U> and wondered if they had special meaning. I gues that means 'no'? – Charles Robertson May 25 '18 at 13:27

1 @CharlesRobertson Yep you are right. They do not have any special meaning as far as typescript concerns. – Emily Parker May 28 '18 at 20:09

1 To make it clear and if I am understanding this right, T is just a placeholder for later type uses. Is that correct? – Eray T Apr 2 at 22:41