

TypeScript ❷

❶

-  _____
-  _____
 -  _____
 -  _____
 -  _____
 -  _____
-  _____
-  _____
-  _____
-  _____

❷



```
function getFirst(arr) {  
  return arr[0];  
}
```

 `getFirst()` 



```
function f(arr:any[]):any {  
  return arr[0];  
}
```



TypeScript generics “ ” type parameter

```
function getFirst<T>(arr:T[]):T {
    return arr[0];
}
```

getFirst()

T

<T>

<>

number[]

T

number

number

```
getFirst<number>([1, 2, 3])
```

getFirst()

T

<number>

TypeScript

```
getFirst([1, 2, 3])
```

TypeScript [1, 2, 3] T number

TypeScript

```
function comb<T>(arr1:T[], arr2:T[]):T[] {
    return arr1.concat(arr2);
}
```

arr1

arr2

```
comb([1, 2], ['a', 'b']) //
```

TypeScript

```
comb<number|string>([1, 2], ['a', 'b']) //
```

T

type

T

U

V

“,”

```
function map<T, U>(
  arr:T[],
  f:(arg:T) => U
):U[] {
  return arr.map(f);
}

// 测试
map<string, number>(
  ['1', '2', '3'],
  (n) => parseInt(n)
); // 返回 [1, 2, 3]
```

map()

T

U

T[]

f

T

U

U[]

function

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

```
// 関数型
let myId:<T>(arg:T) => T = id;

// オブジェクト型
let myId:{ <T>(arg:T): T } = id;
```

interface

```
interface Box<Type> {
    contents: Type;
}

let box:Box<string>;
```

--	--	--	--	--	--	--	--	--

```
interface Comparator<T> {
    compareTo(value:T): number;
}

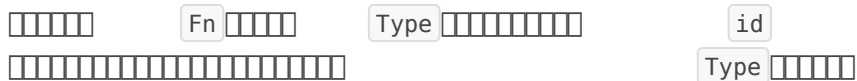
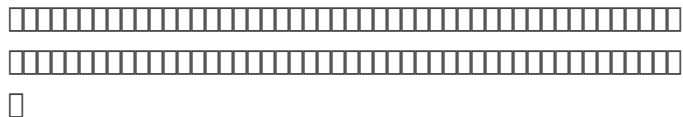
class Rectangle implements Comparator<Rectangle> {

    compareTo(value:Rectangle): number {
        // ...
    }
}
```

```
interface Fn {
    <Type>(arg:Type): Type;
}
```

```
function id<Type>(arg:Type): Type {
    return arg;
}
```

```
let myId:Fn = id;
```



```
class Pair<K, V> {
    key: K;
    value: V;
}
```



```
class A<T> {
    value: T;
}

class B extends A<any> {
}
```



```
const Container = class<T> {
    constructor(private readonly data:T) {}
};

const a = new Container<boolean>(true);
```

```
const b = new Container<number>(0);
```

data

```
class C<NumType> {
    value!: NumType;
    add!: (x: NumType, y: NumType) => NumType;
}

let foo = new C<number>();

foo.value = 0;

foo.add = function (x, y) {
    return x + y;
};
```

};

[illegible]

JavaScript

```
type MyClass<T> = new (...args: any[]) => T;

// []
interface MyClass<T> {
  new(...args: any[]): T;
}

// [][][]
function createInstance<T>(
  AnyClass: MyClass<T>,
  ...args: any[]
):T {
  return new AnyClass(...args);
}
```

```
function createInstance<T>(  
    AnyClass: MyClass<T>,  
    ...args: any[]  
) : T {  
    return new AnyClass(...args);  
}
```

```
T createInstance()
```



```
class C<T> {  
  static data: T; // []  
  constructor(public value:T) {}  
}
```



type []

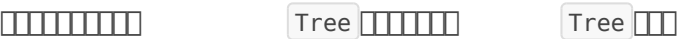
```
type Nullable<T> = T | undefined | null;
```



```
type Container<T> = { value: T };  
  
const a: Container<number> = { value: 0 };  
const b: Container<string> = { value: 'b' };
```



```
type Tree<T> = {  
  value: T;  
  left: Tree<T> | null;  
  right: Tree<T> | null;  
};
```



```
function getFirst<T = string>(
  arr:T[]
):T {
  return arr[0];
}
```

function getFirst T = string arr:T[] string getFirst() T
 TypeScript T string

function TypeScript T

```
getFirst([1, 2, 3]) // 
```

function [1, 2, 3] TypeScript T number string
function

```
class Generic<T = string> {
  list:T[] = []

  add(t:T) {
    this.list.push(t)
  }
}
```

function Generic T string list
function add()

```
const g = new Generic();

g.add(4) // 
g.add('hello') // 
```

function Generic g T T string add()
function

```
const g = new Generic<number>();

g.add(4) // 
g.add('hello') // 
```

g

T

number

add()

```

<T = boolean, U> // 

```

```

<T, U = boolean> // 

```

T U

T

U

T

Array<T>

Array TypeScript

T

```

let arr:Array<number> = [1, 2, 3];

```

Array<number>

number

Array<string>

TypeScript

number[] string[] Array<number> Array<string>

TypeScript

Array

```

interface Array<Type> {

    length: number;

    pop(): Type|undefined;

    push(...items:Type[]): number;

    // ...
}

```

push()

item

Type[]

Array()

Type

push()

TypeScript

TypeScript

Set<T> Promise<T>

Map Set Promise

Map<K, V>

TypeScript

ReadonlyArray<T>

```
function doStuff(  
  values:ReadonlyArray<string>  
) {  
  values.push('hello!');  //  
}
```

values ReadonlyArray<string>

ReadonlyArray<T>



```
function comp<Type>(a:Type, b:Type) {  
  if (a.length >= b.length) {  
    return a;  
  }  
  return b;  
}
```

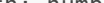
Type length

TypeScript



```
function comp<T extends { length: number }>(  
  a: T,  
  b: T  
) {  
  if (a.length >= b.length) {  
    return a;  
  }  
  return b;  
}
```

--	--	--	--	--	--

T extends { length: number } 

T				
---	--	--	--	--

```
{ length: number }
```

```
comp([1, 2], [1, 2, 3]) // □□
```

```
comp('ab', 'abc') // 1
```

```
comp(1, 2) // []
```



```
<TypeParameter extends ConstraintType>
```

--	--	--	--	--	--

TypeParameter

`extends`

ConstraintType

ConstraintType

```
type Fn<A extends string, B extends string = 'world'>
```

$$= [A, B];$$

```
type Result = Fn<'hello'> // ["hello", "world"]
```

[illegible][illegible]

B

Fn

A

--	--	--	--	--	--

B

[illegible]

$\langle T, U \text{ extends } T \rangle$

// 00

 $\langle T \text{ extends } U, U \rangle$

--	--	--	--	--	--

U

T			
---	--	--	--

T						
---	--	--	--	--	--	--

U

```
<T extends T>           // []
```

```
<T extends U, U extends T> // []
```

--	--	--	--	--	--

T								
---	--	--	--	--	--	--	--	--

[illegible]

T						
---	--	--	--	--	--	--

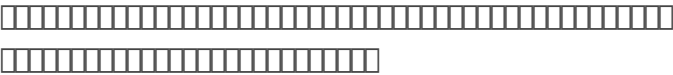
U U

--	--	--	--	--	--

[illegible]



1



2



```
function filter<
  T,
  Fn extends (arg:T) => boolean
>(
  arr:T[],
  func:Fn
): T[] {
  return arr.filter(func);
}
```



```
function filter<T>(
  arr:T[],
  func:(arg:T) => boolean
): T[] {
  return arr.filter(func);
}
```



3



```
function greet<Str extends string>(
  s:Str
) {
```

```
console.log('Hello, ' + s);
}
```

The diagram shows two horizontal registers. The top register is 16 bits wide and has a label 'Str' in a box above its right half. The bottom register is 32 bits wide.

```
function greet(s:string) {  
  console.log('Hello, ' + s);  
}
```

4

```
type OrNull<Type> = Type|null;

type OneOrMany<Type> = Type|Type[];

type OneOrManyOrNull<Type> = OrNull<OneOrMany<Type>>;
```

```
type OneOrMany<Type> = Type|Type[];
```

```
type OneOrManyOrNull<Type> = OrNull<OneOrMany<Type>>;
```

☐ ☒ OrNull ☐ OneOrMany

□□□ #2

2023-07-09 12:07:22 UTC

2023-08-18 09:20:31 UTC