Generics

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.

Hello World of Generics

To start off, let's do the "hello world" of generics: the identity function. The identity function is a function that will return back whatever is passed in. You can think of this in a similar way to the `echo` command.

Without generics, we would either have to give the identity function a specific type:

```
function identity(arg: number): number {
  return arg;
}
```

Or, we could describe the identity function using the `any` type:

```
function identity(arg: any): any {
  return arg;
}
```

With generics we can define a functionality for a no of different data types

Lets try to read documentation further !!

While using `any` is certainly generic in that it will cause the function to accept any and all types for the type of `arg`, we actually are losing the information about what that type was when the function returns. If we passed in a number, the only information we have is that any type could be returned.

Instead, we need a way of capturing the type of the argument in such a way that we can also use it to denote what is being returned. Here, we will use a *type variable*, a special kind of variable that works on types rather than values.

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

Try

We've now added a type variable `Type` to the identity function. This `Type` allows us to capture the type the user provides (e.g. `number`), so that we can use that information later. Here, we use `Type` again as the return type. On inspection, we can now see the same type is used for the argument and the return type. This allows us to traffic that type information in one side of the function and out the other.

```
4  function identityOne(val: boolean | number): boolean | number{
5      return val
6  }
7
8  function identityTwo(val: any):any{
9      return val
10 }
11
12 function identityThree<Type>(val: Type): Type {
13     return val
14 }
15
```

→ Can accept boolean & number

→ Now this can Accept Any One soln was but Many Union lets see Generics soln

Generics Soln

Type of val

return type

Here is Any info about type is Gone

identityTwo Says take Any value & return Any Value

identityThree Says take Type Value & return that Type of Value only.

See fn calls with default data types

```
14 }
15 function identityThree<3>(val: 3): 3
16 identityThree(3)
```

```
14 }
15 function identityThree<"3">(val: "3"): "3"
16 identityThree("3")
```

```
14 }
15 function identityThree<true>(val: true): true
16 identityThree(true)
```

See Given type as a parameter is only one returned !!

```
// identityThree(true)

function identityFour<T>(val: T): T {
    return val
}
```

People usually write Type as T

```
function identityFour<T>(val: T): T {
    return val
}

interface Bootle{
    brand: string,
    type: number
}

class BottleImpl implements Bootle{
    constructor(
        public brand: string,
        public type: number,
        public mat: number,
    ){}

}
let b:BottleImpl=identityFour<BottleImpl>({brand:"abc",type:123,mat:456});
console.log(b);
```

Type Strict code

Quite understandable

<> we use when we send user defined data types

```
[Running] node "c:\Users\mohit\Desktop\programs\typescript\003.purets\src\generics.js"
{ brand: 'abc', type: 123, mat: 456 }
```

O/P

The generated JS code

```
// }
// identityThree(true)
function identityFour(val) {
    return val;
}
var BottleImpl = /** @class */ (function () {
    function BottleImpl(brand, type, mat) {
        this.brand = brand;
        this.type = type;
        this.mat = mat;
    }
    return BottleImpl;
}());
var b = identityFour({ brand: "abc", type: 123, mat: 456 });
console.log(b);
// function getSearchProducts<T>(products: T[]): T {
```

Generics in Array

What if we want to also log the length of the argument `arg` to the console with each call? We might be tempted to write this:

```
function loggingIdentity<Type>(arg: Type): Type {
  console.log(arg.length);
Property 'length' does not exist on type 'Type'.
  return arg;
}
```

We can use either of the way to define Generics

```
function loggingIdentity<Type>(arg: Type[]): Type[] {
  console.log(arg.length);
  return arg;
}
```

OR

```
function loggingIdentity<Type>(arg: Array<Type>): Array<Type> {
  console.log(arg.length); // Array has a .length, so no more error
  return arg;
}
```

Accepting Type T[] & returning T Type

```
39
40  function getSearchProducts<T>(products: T[]): T {
41      // do some database operations
42      const myIndex = 3
43      return products[myIndex]
44  }
45
```

Generics in Arrow fn

Tells its a Generic

parameters

Can use <T>

```ts
46
47  const getMoreSearchProducts = <T,>(products: T[]): T => {
48      //do some database operations
49      const myIndex = 4
50      return products[myIndex]
51  }
```

Return type

<T> } → Produce same result , tells ki ye hoi
<T,> } Normal type nhi thai

Using Type Parameters in Generic Constraints

You can declare a type parameter that is constrained by another type parameter. For example, here we'd like to get a property from an object given its name. We'd like to ensure that we're not accidentally grabbing a property that does not exist on the `obj`, so we'll place a constraint between the two types:

Just Read it

```ts
function getProperty<Type, Key extends keyof Type>(obj: Type, key:
  return obj[key];
}

let x = { a: 1, b: 2, c: 3, d: 4 };

getProperty(x, "a");
getProperty(x, "m");
```

```
Argument of type '"m"' is not assignable to parameter of type '"a"
| "b" | "c" | "d"'.
```

Documentation

```ts
42  function anotherFunction<T, U>(valOne:T, valTwo:U):object {
43      return{
44          valOne,
45          valTwo
46      }
47  }
```

→ Fn taking 2 type of values <T, U>

```ts
48
49  anotherFunction(3, "4")
```

```ts
interface Database {
    connection: string,
    username: string,
    password: string
}

function anotherFunction<T, U extends Database>(valOne:T, valTwo:U):object {
    return{
        valOne,
        valTwo
    }
}

anotherFunction(3, {})
```

U is a kind of Database
Any object Extending DB can use this

```ts
interface Quiz{
    name: string,
    type: string
}

interface Course{
    name: string,
    author: string,
    subject: string
}

class Sellable<T>{
    public cart: T[] = []

    addToCart(product: T){
        this.cart.push(product)
    }
}
```

Two types of Sellable Narrow
want something that can
work with both of these

a class that can work with
both Quize & Course

Class className <T> Tells class is Generic

Narrowing → Narrow down a data type

type of [1, 2, 3] → object

type of " " → object → not string

So type of Array in object is remembered !! ***

```
7
8   function provideId(id: string | null){
9       if(!id){
10          console.log("Please provide ID");
11          return
12      }
13      id.toLowerCase()
14  }
15
```

Provide null as parameter as id can be empty

Very Much Useful case

3 cases

- String
- String Array
- null

```
16
17  function printAll(strs: string | string[] | null) {
18
19    if (strs) {
20      if (typeof strs === "object") {
21        for (const s of strs) {
22          console.log(s);
23        }
24      } else if (typeof strs === "string") {
25        console.log(strs);
26      }
27    }
28  }
29
```

If String value is Empty " "

Here we are not doing anything with Empty String !!

The `in` operator narrowing

JavaScript has an operator for determining if an object or its prototype chain has a property with a name: the `in` operator. TypeScript takes this into account as a way to narrow down potential types.

For example, with the code: `"value" in x`. where `"value"` is a string literal and `x` is a union type. The "true" branch narrows `x`'s types which have either an optional or required property `value`, and the "false" branch narrows to types which have an optional or missing property `value`.

```
type Fish = { swim: () => void };
type Bird = { fly: () => void };

function move(animal: Fish | Bird) {
  if ("swim" in animal) {
    return animal.swim();
  }

  return animal.fly();
}
```

in operator

Ex

```ts
interface User {
    name: string,
    email: string
}

interface Admin{
    name: string,
    email: string,
    isAdmin: boolean
}

function isAdminAccount(account: User | Admin){
    if ("isAdmin" in account) {
        return account.isAdmin
    }
}
```

Quite understandable

In operator

Instance of narrowing

Narrowing ⇒ Narrowing of Types

```ts
function logValue(x: Date | string) {
    if (x instanceof Date) {
        console.log(x.toUTCString());
    } else {
        console.log(x.toUpperCase());
    }
}
```

instanceof narrowing

JavaScript has an operator for checking whether or not a value is an "instance" of another value. More specifically, in JavaScript `x instanceof Foo` checks whether the *prototype chain* of `x` contains `Foo.prototype`. While we won't dive deep here, and you'll see more of this when we get into classes, they can still be useful for most values that can be constructed with `new`. As you might have guessed, `instanceof` is also a type guard, and TypeScript narrows in branches guarded by `instanceof`s.

Predicates

Here on Right we check if pet is Fish or not But at bb TS is still confused it is Fish or Bird

```ts
type Fish = {swim: () => void};
type Bird = {fly: () => void};

function isFish(pet: Fish | Bird){
    return (pet as Fish).swim !== undefined
}

function getFood(pet: Fish | Bird){
    if (isFish(pet)) {
        pet
        return "fish food"
    } else {
        pet
        return "bird Food"
    }
}
```

```
64 function getFood(pet: Fish | Bird){
65     if ( (parameter) pet: Fish | Bird
66         pet
```

If you hover over that you see TS is still confused b/w Bird or Fish

```
55
56
57  type Fish = {swim: () => void};
58  type Bird = {fly: () => void};
59
60  function isFish(pet: Fish | Bird): pet is Fish{
61      return (pet as Fish).swim !== undefined
62  }
63
64  function getFood(pet: Fish | Bird){
65      if (isFish(pet)) {
66          pet
67          return "fish food"
68      } else {
69          pet
70          return "bird Food"
71      }
72  }
73
74
```

So The isFish return value is Fish if line 61 is true it returns Fish else no

Discriminated Unions

```
74
75  interface Circle {
76      kind: "circle",
77      radius: number
78  }
79
80  interface Square {
81      kind: "square"
82      side: number
83  }
84
85  interface Rectangle {
86      kind: "rectangle",
87      length: number,
88      width: number
89  }
90
91  type Shape = Circle | Square | Rectangle
92
```

```
93  function getTrueShape(shape: Shape){
94      if (shape.kind === "circle") {
95          return Math.PI * shape.radius ** 2
96      }
97      //return shape.side * shape.side
98  }
99
100
101 function getArea(shape: Shape){
102     switch(shape.kind){
103         case "circle":
104             return Math.PI * shape.radius ** 2
105
106         case "square":
107             return shape.side * shape.side
108         case "rectangle":
109             return shape.length * shape.width
110
111         default:
112             const _defaultforshape: never = shape
113             return _defaultforshape
114     }
115 }
```

have default case having variable of type never case if someone creates a new type of shape code says bhai case lgao yha pe

Suppose me created a shape Cube then default will say Rhai Cube ko handle kro

Dis Criminated main is just that a property named "Rid" !!