# 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;
}
```

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;
}
```

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.

We say that this version of the `identity` function is generic, as it works over a range of types. Unlike using `any`, it's also just as precise (i.e., it doesn't lose any information) as the first `identity` function that used numbers for the argument and return type.

Once we've written the generic identity function, we can call it in one of two ways. The first way is to pass all of the arguments, including the type argument, to the function:

```
let output = identity<string>("myString");
       let output: string
```

Here we explicitly set `Type` to be `string` as one of the arguments to the function call, denoted using the `<>` around the arguments rather than `()`.

The second way is also perhaps the most common. Here we use *type argument inference* — that is, we want the compiler to set the value of `Type` for us automatically based on the type of the argument we pass in:

```
let output = identity("myString");
       let output: string
```

Notice that we didn't have to explicitly pass the type in the angle brackets (`<>`); the compiler just looked at the value `"myString"`, and set `Type` to its type. While type argument inference can be a helpful tool to keep code shorter and more readable, you may need to explicitly pass in the type arguments as we did in the previous example when the compiler fails to infer the type, as may happen in more complex examples.

## Working with Generic Type Variables

When you begin to use generics, you'll notice that when you create generic functions like `identity`, the compiler will enforce that you use any generically typed parameters in the body of the function correctly. That is, that you actually treat these parameters as if they could be any and all types.

Let's take our `identity` function from earlier:

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

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;
}
```

When we do, the compiler will give us an error that we're using the `.length` member of `arg`, but nowhere have we said that `arg` has this member. Remember, we said earlier that these type variables stand in for any and all types, so someone using this function could have passed in a `number` instead, which does not have a `.length` member.

Let's say that we've actually intended this function to work on arrays of `Type` rather than `Type` directly. Since we're working with arrays, the `.length` member should be available. We can describe this just like we would create arrays of other types:

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

You can read the type of `loggingIdentity` as "the generic function `loggingIdentity` takes a type parameter `Type`, and an argument `arg` which is an array of `Type`s, and returns an array of `Type`s." If we passed in an array of numbers, we'd get an array of numbers back out, as `Type` would bind to `number`. This allows us to use our generic type variable `Type` as part of the types we're working with, rather than the whole type, giving us greater flexibility.

We can alternatively write the sample example this way:

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

You may already be familiar with this style of type from other languages. In the next section, we'll cover how you can create your own generic types like `Array<Type>`.

## Generic Types

In previous sections, we created generic identity functions that worked over a range of types. In this section, we'll explore the type of the functions themselves and how to create generic interfaces.

The type of generic functions is just like those of non-generic functions, with the type parameters listed first, similarly to function declarations:

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

let myIdentity: <Type>(arg: Type) => Type = identity;
```

We could also have used a different name for the generic type parameter in the type, so long as the number of type variables and how the type variables are used line up.

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

let myIdentity: <Input>(arg: Input) => Input = identity;
```

We can also write the generic type as a call signature of an object literal type:

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

let myIdentity: { <Type>(arg: Type): Type } = identity;
```

Which leads us to writing our first generic interface. Let's take the object literal from the previous example and move it to an interface:

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

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

let myIdentity: GenericIdentityFn = identity;
```

In a similar example, we may want to move the generic parameter to be a parameter of the whole interface. This lets us see what type(s) we're generic over (e.g. `Dictionary<string>` rather than just `Dictionary`). This makes the type parameter visible to all the other members of the interface.

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

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

let myIdentity: GenericIdentityFn<number> = identity;
```

Notice that our example has changed to be something slightly different. Instead of describing a generic function, we now have a non-generic function signature that is a part of a generic type. When we use `GenericIdentityFn`, we now will also need to specify the corresponding type argument (here: `number`), effectively locking in what the underlying call signature will use. Understanding when to put the type parameter directly on the call signature and when to put it on the interface itself will be helpful in describing what aspects of a type are generic.

In addition to generic interfaces, we can also create generic classes. Note that it is not possible to create generic enums and namespaces.

# Generic Classes

A generic class has a similar shape to a generic interface. Generic classes have a generic type parameter list in angle brackets ( <> ) following the name of the class.

```
class GenericNumber<NumType> {
  zeroValue: NumType;
  add: (x: NumType, y: NumType) => NumType;
}

let myGenericNumber = new GenericNumber<number>();
myGenericNumber.zeroValue = 0;
myGenericNumber.add = function (x, y) {
  return x + y;
};
```

This is a pretty literal use of the `GenericNumber` class, but you may have noticed that nothing is restricting it to only use the `number` type. We could have instead used `string` or even more complex objects.

```
let stringNumeric = new GenericNumber<string>();
stringNumeric.zeroValue = "";
stringNumeric.add = function (x, y) {
  return x + y;
};

console.log(stringNumeric.add(stringNumeric.zeroValue, "test"));
```

Just as with interface, putting the type parameter on the class itself lets us make sure all of the properties of the class are working with the same type.

As we cover in our section on classes, a class has two sides to its type: the static side and the instance side. Generic classes are only generic over their instance side rather than their static side, so when working with classes, static members can not use the class's type parameter.

# Generic Constraints

If you remember from an earlier example, you may sometimes want to write a generic function that works on a set of types where you have *some* knowledge about what capabilities that set of types will have. In our `loggingIdentity` example, we wanted to be able to access the `.length` property of `arg`, but the compiler could not prove that every type had a `.length` property, so it warns us that we can't make this assumption.

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

Instead of working with any and all types, we'd like to constrain this function to work with any and all types that *also* have the `.length` property. As long as the type has this member, we'll allow it, but it's required to have at least this member. To do so, we must list our requirement as a constraint on what `Type` can be.

To do so, we'll create an interface that describes our constraint. Here, we'll create an interface that has a single `.length` property and then we'll use this interface and the `extends` keyword to denote our constraint:

```
interface Lengthwise {
  length: number;
}

function loggingIdentity<Type extends Lengthwise>(arg: Type): Type {
  console.log(arg.length); // Now we know it has a .length property, so no more error
  return arg;
}
```

Because the generic function is now constrained, it will no longer work over any and all types:

```
loggingIdentity(3);
```

```
Argument of type 'number' is not assignable to parameter of type 'Lengthwise'.
```

Instead, we need to pass in values whose type has all the required properties:

```
loggingIdentity({ length: 10, value: 3 });
```

## 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:

```
function getProperty<Type, Key extends keyof Type>(obj: Type, key: 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"'.
```

Try

## Using Class Types in Generics

When creating factories in TypeScript using generics, it is necessary to refer to class types by their constructor functions. For example,

```
function create<Type>(c: { new (): Type }): Type {
  return new c();
}
```

A more advanced example uses the prototype property to infer and constrain relationships between the constructor function and the instance side of class types.

```
class BeeKeeper {
  hasMask: boolean = true;
}

class ZooKeeper {
  nametag: string = "Mikle";
}

class Animal {
  numLegs: number = 4;
}

class Bee extends Animal {
  numLegs = 6;
  keeper: BeeKeeper = new BeeKeeper();
}

class Lion extends Animal {
  keeper: ZooKeeper = new ZooKeeper();
}

function createInstance<A extends Animal>(c: new () => A): A {
  return new c();
}

createInstance(Lion).keeper.nametag;
createInstance(Bee).keeper.hasMask;
```

This pattern is used to power the mixins design pattern.

## Generic Parameter Defaults

By declaring a default for a generic type parameter, you make it optional to specify the corresponding type argument. For example, a function which creates a new `HTMLElement`. Calling the function with no arguments generates a `HTMLDivElement`; calling the function with an element as the first argument generates an element of the argument's type. You can optionally pass a list of children as well. Previously you would have to define the function as:

```
declare function create(): Container<HTMLDivElement, HTMLDivElement[]>;
declare function create<T extends HTMLElement>(element: T): Container<T, T[]>;
declare function create<T extends HTMLElement, U extends HTMLElement>(
  element: T,
  children: U[]
): Container<T, U[]>;
```

With generic parameter defaults we can reduce it to:

```
declare function create<T extends HTMLElement = HTMLDivElement, U extends HTMLElement[] = T[]>(
  element?: T,
  children?: U
): Container<T, U>;

const div = create();
```

```
const div: Container<HTMLDivElement, HTMLDivElement[]>

const p = create(new HTMLParagraphElement());
      const p: Container<HTMLParagraphElement, HTMLParagraphElement[]>
```

A generic parameter default follows the following rules:

- A type parameter is deemed optional if it has a default.
- Required type parameters must not follow optional type parameters.
- Default types for a type parameter must satisfy the constraint for the type parameter, if it exists.
- When specifying type arguments, you are only required to specify type arguments for the required type parameters. Unspecified type parameters will resolve to their default types.
- If a default type is specified and inference cannot choose a candidate, the default type is inferred.
- A class or interface declaration that merges with an existing class or interface declaration may introduce a default for an existing type parameter.
- A class or interface declaration that merges with an existing class or interface declaration may introduce a new type parameter as long as it specifies a default.

## Variance Annotations

> This is an advanced feature for solving a very specific problem, and should only be used in situations where you've identified a reason to use it

Covariance and contravariance are type theory terms that describe what the relationship between two generic types is. Here's a brief primer on the concept.

For example, if you have an interface representing an object that can `make` a certain type:

```
interface Producer<T> {
  make(): T;
}
```

We can use a `Producer<Cat>` where a `Producer<Animal>` is expected, because a `Cat` is an `Animal`. This relationship is called *covariance*: the relationship from `Producer<T>` to `Producer<U>` is the same as the relationship from `T` to `U`.

Conversely, if you have an interface that can `consume` a certain type:

```
interface Consumer<T> {
  consume: (arg: T) => void;
}
```

Then we can use a `Consumer<Animal>` where a `Consumer<Cat>` is expected, because any function that is capable of accepting an `Animal` must also be capable of accepting a `Cat`. This relationship is called *contravariance*: the relationship from `Consumer<T>` to `Consumer<U>` is the same as the relationship from `U` to `T`. Note the reversal of direction as compared to covariance! This is why contravariance "cancels itself out" but covariance doesn't.

In a structural type system like TypeScript's, covariance and contravariance are naturally emergent behaviors that follow from the definition of types. Even in the absence of generics, we would see covariant (and contravariant) relationships:

```
interface AnimalProducer {
  make(): Animal;
}

// A CatProducer can be used anywhere an
// Animal producer is expected
interface CatProducer {
  make(): Cat;
}
```

TypeScript has a structural type system, so when comparing two types, e.g. to see if a `Producer<Cat>` can be used where a `Producer<Animal>` is expected, the usual algorithm would be structurally expand both of those definitions, and compare their structures. However, variance allows for an extremely useful optimization: if `Producer<T>` is covariant on `T`, then we can simply check `Cat` and `Animal` instead, as we know they'll have the same relationship as `Producer<Cat>` and `Producer<Animal>`.

Note that this logic can only be used when we're examining two instantiations of the same type. If we have a `Producer<T>` and a `FastProducer<U>`, there's no guarantee that `T` and `U` necessarily refer to the same positions in these types, so this check will always be performed structurally.

Because variance is a naturally emergent property of structural types, TypeScript automatically *infers* the variance of every generic type. **In extremely rare cases** involving certain kinds of circular types, this measurement can be inaccurate. If this happens, you can add a variance annotation to a type parameter to force a particular variance:

```
// Contravariant annotation
interface Consumer<in T> {
  consume: (arg: T) => void;
}

// Covariant annotation
interface Producer<out T> {
  make(): T;
}

// Invariant annotation
interface ProducerConsumer<in out T> {
  consume: (arg: T) => void;
  make(): T;
}
```

Only do this if you are writing the same variance that *should* occur structurally.

> Never write a variance annotation that doesn't match the structural variance!

It's critical to reinforce that variance annotations are only in effect during an instantiation-based comparison. They have no effect during a structural comparison. For example, you can't use variance annotations to "force" a type to be actually invariant:

```
// DON'T DO THIS - variance annotation
// does not match structural behavior
interface Producer<in out T> {
  make(): T;
```

```
}

// Not a type error -- this is a structural
// comparison, so variance annotations are
// not in effect
const p: Producer<string | number> = {
    make(): number {
        return 42;
    }
}
```

Here, the object literal's `make` function returns `number`, which we might expect to cause an error because `number` isn't `string | number`. However, this isn't an instantiation-based comparison, because the object literal is an anonymous type, not a `Producer<string | number>`.

> Variance annotations don't change structural behavior and are only consulted in specific situations

It's very important to only write variance annotations if you absolutely know why you're doing it, what their limitations are, and when they aren't in effect. Whether TypeScript uses an instantiation-based comparison or structural comparison is not a specified behavior and may change from version to version for correctness or performance reasons, so you should only ever write variance annotations when they match the structural behavior of a type. Don't use variance annotations to try to "force" a particular variance; this will cause unpredictable behavior in your code.

> Do NOT write variance annotations unless they match the structural behavior of a type

Remember, TypeScript can automatically infer variance from your generic types. It's almost never necessary to write a variance annotation, and you should only do so when you've identified a specific need. Variance annotations *do not* change the structural behavior of a type, and depending on the situation, you might see a structural comparison made when you expected an instantiation-based comparison. Variance annotations can't be used to modify how types behave in these structural contexts, and shouldn't be written unless the annotation is the same as the structural definition. Because this is difficult to get right, and TypeScript can correctly infer variance in the vast majority of cases, you should not find yourself writing variance annotations in normal code.

> Don't try to use variance annotations to change typechecking behavior; this is not what they are for

You *may* find temporary variance annotations useful in a "type debugging" situation, because variance annotations are checked. TypeScript will issue an error if the annotated variance is identifiably wrong:

```
// Error, this interface is definitely contravariant on T
interface Foo<out T> {
  consume: (arg: T) => void;
}
```

However, variance annotations are allowed to be stricter (e.g. `in out` is valid if the actual variance is covariant). Be sure to remove your variance annotations once you're done debugging.

Lastly, if you're trying to maximize your typechecking performance, *and* have run a profiler, *and* have identified a specific type that's slow, *and* have identified variance inference specifically is slow, *and* have carefully validated the variance annotation you want to write, you *may* see a small performance benefit in extraordinarily complex types by adding variance annotations.

> Don't try to use variance annotations to change typechecking behavior; this is not what they are for