

# Generic Constraint

This lesson explains how to transform a generic type with a constraint to control what can be passed.

## WE'LL COVER THE FOLLOWING



- Function with constraints
- Accessing properties of the generic type

With generic, you can specify that the type passed into your generic class must extend a specific interface. Generic allows having code that can rely on a minimal interface without forcing a specific class.

For example, with a list, you do not need to force the list to hold a specific object. Instead, you can create a *generic* list where the generic type passed **must** extend a definition.

## Function with constraints #

In the following example, we are stating that any object that has at least a member named `id` of type `number` is allowed. By extending the generic type, you can perform logic on specific members; for example, access `id` regardless of fields not exposed to the generic code.

```
interface MyType { // Type that has a single field
  id: number;
}
interface AnotherType extends MyType {} // Another type that has all the field from MyType
function genericFunction<T extends MyType>(p1: T) {} // A function that take a generic type t

const arg: AnotherType = { id: 1 }; // Create an object that is not "MyType"
genericFunction(arg); // This is legit because AnotherType extend MyType, thus has all the r

genericFunction({ id: 123 }); // This is legit as well since id is the only required field fr

// genericFunction("doesn't compile") // Doest not compile, not legit.
```



**Line 12** is commented on purpose. The argument passed does not have a member `id`: it is a string. Hence, TypeScript knows it is not a legit type and error at transpilation time. On the contrary, at **line 8** the object has an `id` and at **line 10** as well.

## Accessing properties of the generic type #

The benefit is avoiding potential casting errors and accessing members that do not exist. It also allows access to the specified members inside the generic function.

```
interface MyType {
  id: number; // id is available
  id2: number;
}
interface AnotherType {
  id: number; // id is available
}
function genericFunction<T extends AnotherType>(p1: T) { // Any type that has all fields from AnotherType
  console.log("Inside generic:" + p1.id);
}

genericFunction({ id: 123, id2: 99999 }); // This is legit because we have id (and more)
```

The example passes an object with two members `id` and `id2` to a generic function that requires an `id` of type `number`. The object passed on line 12 does have `id` and `id2`. However, inside the generic function, only `id` is available. The reason is that the function expects to have the minimum and common understanding that `T` is respecting the type after `extends` and anything else is not guaranteed, hence it cannot be accessible.

Many use cases borrow the pattern of allowing a generic type and are concerned only by a subset of properties. For example, you may have a logging library that must have an object with `errorCode` and `errorDescription` and still allow any type as long as they have these two fields.

