# Casting to Change Type

In this lesson, we will learn how to cast values from one type to another.

WE'LL COVER THE FOLLOWING



## Two ways to cast

TypeScript casts use two different forms: `<>` or `as`. The former enters into conflict with the JSX/TSX format which is now getting popular because of React, hence not recommended.

```
const cast1: number = <number>1;
```

The latter is as good, and it works in all situations.

```
const cast2: number = 1 as number;
```

The first way is to use the symbols `<` and `>` with the type desired in-between. The syntax requires the cast before the variable that we want to coerce.

The second way is to use the keyword `as`. `as` is placed after the variable we want to cast followed by the type to cast.

## The danger of casting

Casting is a delicate subject since you can cast every variable into something without completely respecting the contract into which we cast. For example, you can have an interface that requires many fields and cast an empty object

to that interface, and it will compile even if you do not have the members. The

fallacy of the cast when the underlying object doesn't respect the type schema is one reason why it's better if you can assign a type to the variable and not cast.

However, there is a situation where you must cast. For example, if you receive a JSON payload from an Ajax call, this one will be by nature `any` since the response of an Ajax is undetermined until the consumer does the call. In that case, you must cast to manipulate the data in a typed fashion in the rest of your application. The constraint here is that you must be sure that you are receiving the data in a format that provides all expected members. Otherwise, it would be wiser to define these members to be optional (undefined as well as the expected type).

```
interface ICast1 {
    m1: string;
}
interface ICast2 {
    m1: string;
    m2: string;
}
let icast1: ICast1 = { m1: "m1" };
let icast2: ICast2 = { m1: "m1", m2: "m2" };
let icast3: ICast1 = icast2; // work without cast because of the structure
//icast2 = icast1; // doesn't work, miss a member
let icast4: ICast2 = icast1 as ICast2; // work but m2 undefined
console.log(icast4); // m2 is missing even if not optional
```

## Casting restrictions #

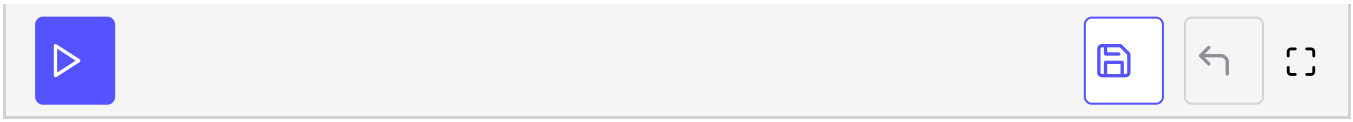
Casting has some restrictions. For instance, you cannot cast a typed object into something that is not a subtype. If you have `TypeC` that inherits `TypeB` that inherits `TypeA`, you can cast a `TypeC` to `TypeA` or `TypeB` without problem or `TypeB` to `TypeA` without casting. However, going the other way around requires a cast. Nevertheless, there are issues in both cases. When going from a subtype to a type, without casting, the problem is that TypeScript will validate that you can only access the public type from the desired interface. However, under the hood, the object still contains all the members.

For example (see below), `TypeB` has two members. **Line 5** defines `m2` and it

inherits `m1` from `ITypeA` at **line 2**. Thus when casting at **line 12** from A to B

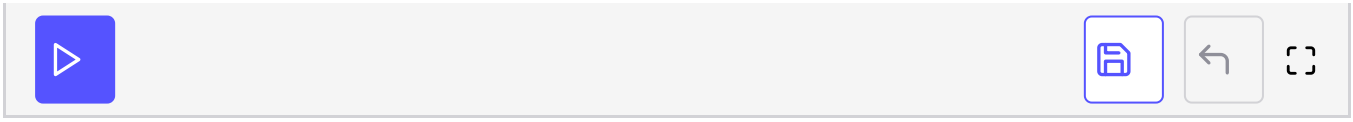
the variable `typeb2` has the field `m2` without value.

```
interface ITypeA {
    m1: string;
}
interface ITypeB extends ITypeA {
    m2: string;
}
interface ITypeC extends ITypeB {
    m3: string;
}
const typea: ITypeA = { m1: "m1" };
const typeb: ITypeB = { m1: "m1", m2: "m2" };
let typeb2: ITypeB = typea as ITypeB; // Work (m2 will be missing!!!)
console.log(typeb2); // No m2
```



When casting, it only exposes at design time the first member which is in `TypeA`. However, printing the object (**line 12**) reveals both members are still there. The lack of cohesion between the type's schema and the actual object structure is an important detail. For example, sending an object to an API without manually grooming the object may pass more information than anticipated.

```
interface ITypeA {
    m1: string;
}
interface ITypeB extends ITypeA {
    m2: string;
}
interface ITypeC extends ITypeB {
    m3: string;
}
const typeb: ITypeB = { m1: "m1", m2: "m2" };
let typea2: ITypeA = typeb; // No cast needed
console.log(typea2); // { m1: 'm1', m2: 'm2' } However, only m1 is accessible at compilation
```



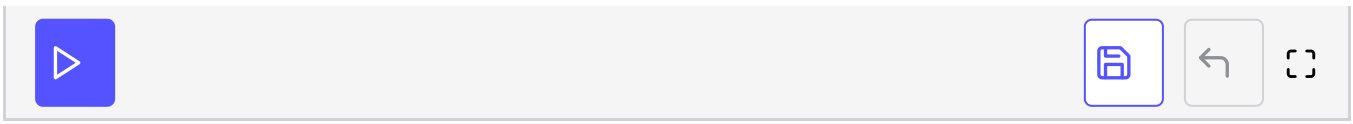
## Casting coercion #

The second issue is with casting. Since casting coerces by saying to TypeScript that you know what you are doing, it won't complain. However, non-optional

members not present will be undefined even if the contract specifies that the type must have the member. You can see an example below when an object of `TypeA` (base interface) is cast down to `TypeB`. The cast coerces the change of type, but `m2` is still not present. While it is good enough for TypeScript that you manually override the validation, it can be problematic if later in the code you try to access `m2` and believe that this one cannot be undefined. In fact, this can cause a runtime error if you try to access a function of the member.

However, casting to a string won't work as the commented **line 2** shows. But, casting to `any` and then casting to `string` will work, see **line 3**. Again, the sliding ground is with `any`.

```
let aNumberToCast: number = 1;
// let aString: string = aNumberToCast as string; // Doesn't work
let aString: string = aNumberToCast as any as string;
```



The best practice with casting is to try to cast as little as possible. To cast at a strategic area of your code like when getting untyped object is a smart and limited place. When you need to create a new type, it's better to assign a type (explicit declaration) than casting. Doing so, provide IntelliSense support and the compiler protection which keep the code stable with the expected type.