



Casting to Change Type

In this lesson, you will see how to move from one type to another.

We'll cover the following



- How do we cast?
 - Casting constraints
- Type assertion
 - Casting restrictions

How do we cast?

TypeScript can cast using two different forms: `<>` or `as`. The former is not recommended because it conflicts with the JSX/TSX format which is now becoming popular because of React. The latter is just as good, and it works in all situations.

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

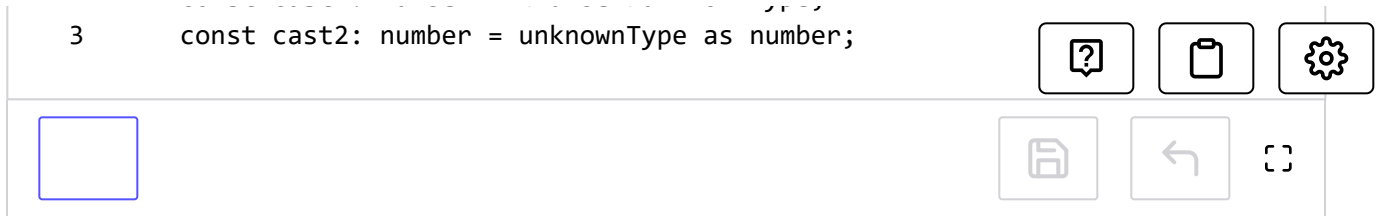
The second way is to use the keyword `as`. `as` is placed after the variable you want to cast and followed by the type you want to cast.

The following code demonstrates an `unknown` type cast to a `number`.

```
1    const unknownType: unknown = "123"  
2    const cast1: number = <number>unknownType;
```



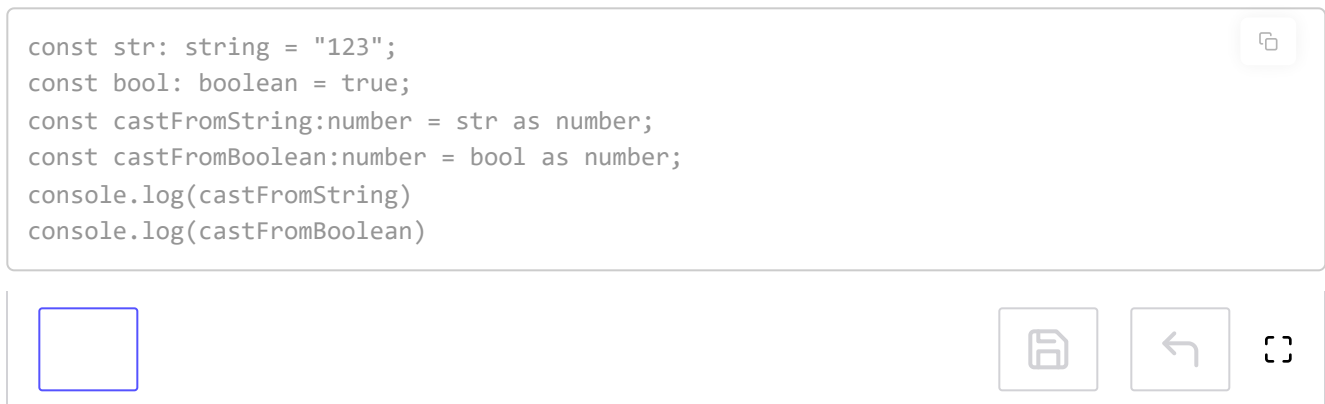
```
3    const cast2: number = unknownType as number;
```



Casting constraints#

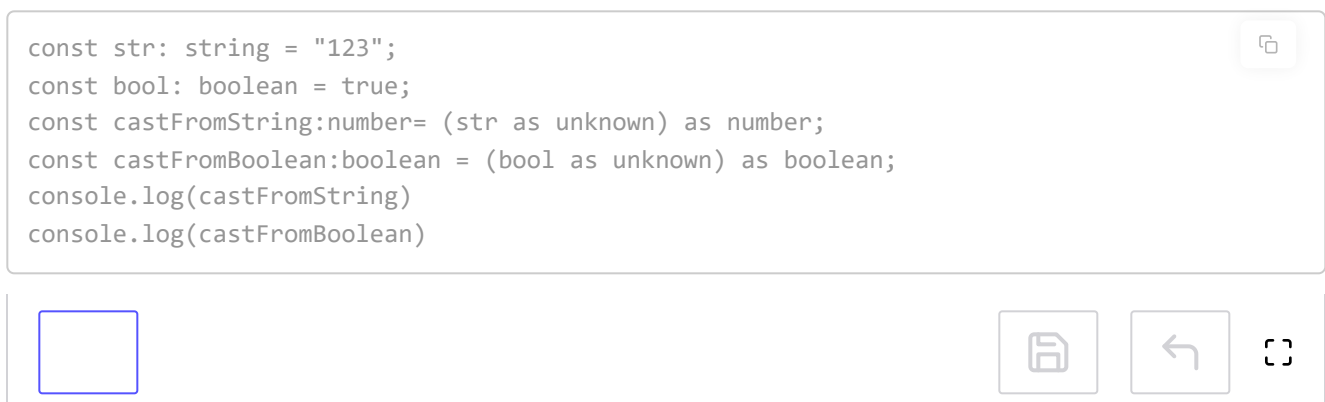
If you try to cast to a `string` directly without using an unknown type, TypeScript will warn that there are not sufficient overlaps. The TypeScript transpiler gives the solution: casting to `unknown`.

```
const str: string = "123";
const bool: boolean = true;
const castFromString:number = str as number;
const castFromBoolean:number = bool as number;
console.log(castFromString)
console.log(castFromBoolean)
```



In that case, a double cast is needed. First, you cast to `unknown` and then to the desired type.

```
const str: string = "123";
const bool: boolean = true;
const castFromString:number= (str as unknown) as number;
const castFromBoolean:boolean = (bool as unknown) as boolean;
console.log(castFromString)
console.log(castFromBoolean)
```



Type assertion



Casting is a delicate subject since you can cast any variable into anything

else without properly respecting its original type. A particular



assertion”. **Type assertion** is when you tell TypeScript what type an object is.

```
interface YourType {  
  m1: string;  
}  
  
let v1 = {m1: "ValueOfM1"} as YourType;  
console.log(v1);
```



Where it can be delicate is, for example, if you have an interface that requires many fields and you cast an empty object to it, it will compile even if you do not have the members.

```
interface IMyType {  
  m1: string;  
  m2: number;  
}  
  
let myVariable = {} as IMyType; //
```



The fallacy of the cast when the underlying object doesn't respect the type schema is one reason why it's better to 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 will be **any** by nature 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).



Casting restrictions#

Casting has some restrictions. For instance, you cannot cast a typed object into something that is not a subtype of the original type. If you have a TypeC that inherits TypeB which 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 with both cases. When going from a subtype to a type, without casting, the problem is that TypeScript will only validate access to 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; when casting, it only exposes (at design time) the first member, which is in TypeA. However, printing the object reveals that 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 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
console.log(icast1);
console.log(icast2);
console.log(icast3);

//icast2 = icast1; // doesn't work, miss a member
let icast4: ICast2 = icast1 as ICast2; // work but m2 undefined
console.log(icast4); // { m1: 'm1' } // m2 is missing even if not optional
```





The first example above shows an example of two different types (`ICast2` to `ICast1`) without casting (line 5). This is possible because they are the same structure. The second example, line 11, cast but is not fine. It coerces the type which causes `m2` to be `undefined` because it does not exist in `ICast1` . Line 10 is commented.

Uncommenting shows that without casting, when the structure is not similar, TypeScript blocks the transpilation.

The second issue is with casting. Since casting tells TypeScript that you know what you're doing, it won't complain. However, non-optional members that aren't present will be undefined even if the contract specifies that the type must have the member. You can see an example below of 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 it cannot be `undefined` . In fact, this can cause a runtime error if you try to access a function of the member. The example below has line 9 commented. It demonstrates that the member is not available at design time. However, when commented, you can see that it is present at runtime (JavaScript).

```
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!!!)
let typea2: ITypeA = typeb; // No cast needed
console.log(typea2); // { m1: 'm1', m2: 'm2' } However, only m1 is accessible at compile time
// let m2 = typea2.m2;
```



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

[← Back](#)[Symbol and Unique Symbol](#)[Next →](#)[Quiz](#) **Completed**[Report an Issue](#)