Understanding type compatibility

In this lesson, we'll learn how TypeScript decides whether an item can be assigned to another, i.e. how TypeScript decides whether types are compatible.

WE'LL COVER THE FOLLOWING

Basic type compatibility

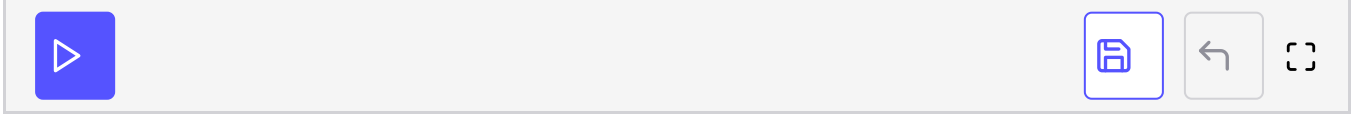
Consider the code below. Hopefully, it is no surprise that TypeScript isn't happy with the assignment on the last line because the types aren't compatible.

```
let firstName: string = "Fred";
let age: number = 30;
firstName = age;  // Type 'number' is not assignable to type 'string'.
```

What about the code below. Will TypeScript be happy with the assignment on the last line?

</> TypeScript

```
let jones: "Tom" | "Bob" = "Tom";
let jane: string = "Jane";
jane = jones;
```



Show Answer

What if we switch the assignment around:

```
jones = jane;
```

Will TypeScript be happy with the assignment on the last time now?

Show Answer

So, if a type is a subset of another type, it can be assigned to it. However, if a type is a superset of another type, it can't be assigned to it.

Object type compatibility

TypeScript is a structurally-typed language which means types are based only on their members.

The types `Person` and `IPerson` below are equivalent because the type members are the same:

```
type Person = {
  name: string;
}
interface IPerson {
  name: string;
}

const bob: Person = {
  name: "Bob"
}
const fred: IPerson = {
  name: "Fred"
}
let person = bob;
person = fred;  // okay
```

So, the type names aren't important, it is the structure that is important.

If we add an `age` property to `IPerson`, are the types compatible now? Will

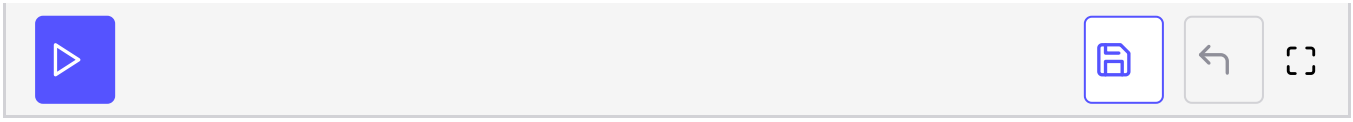
TypeScript be happy with the statement `person = fred;` on the last line?

</> TypeScript

```
type Person = {
  name: string;
}
interface IPerson {
  name: string;
  age: number;
}

const bob: Person = {
  name: "Bob",
}
const fred: IPerson = {
  name: "Fred",
  age: 30
}

let person = bob;
person = fred;  // okay?
```



Show Answer

If we move the `age` property to `Person`, are the types compatible now? Will TypeScript be happy with the statement `person = fred;` on the last line?

</> TypeScript

```
type Person = {
  name: string;
  age: number;
}
interface IPerson {
  name: string;
}

const bob: Person = {
  name: "Bob",
  age: 30
}
const fred: IPerson = {
  name: "Fred",
}

let person: Person = bob;
person = fred;  // okay?
```

Show Answer

What about the following example, are types `Dog` and `Shape` compatible?

</> TypeScript

```
type Dog = {
  name: string;
}
type Shape = {
  name: "Circle" | "Square"
}
let ben: Dog = {
  name: "Ben",
}
let circle: Shape = {
  name: "Circle",
}
circle = ben;  // okay?
```

Show Answer

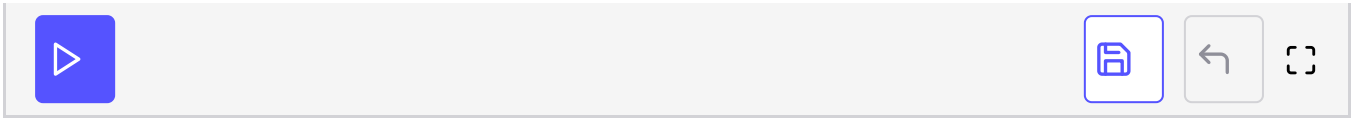
So, the types of the members of the objects are essential. Each member type has to be compatible in order for the object to be compatible.

What if we switch the assignment around? Is the assignment going to be ok now?

</> TypeScript

```
type Dog = {
  name: string;
}
type Shape = {
  name: "Circle" | "Square"
}
let ben: Dog = {
  name: "Ben",
}
let circle: Shape = {
  name: "Circle",
```

```
}
ben = circle;  // okay?
```



Show Answer

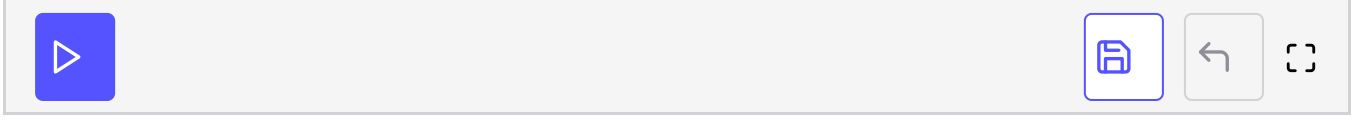
Function type compatibility

Type compatibility for functions is based on structure as well. TypeScript checks that the function parameter types and the return type are compatible.

Consider the code where we have two functions:

</> TypeScript

```
let add = (a: number, b: number): number => a + b;
let sum = (x: number, y: number): number => x + y;
sum = add; // okay?
```



Are these functions compatible? Is the assignment on the last line okay?

Show Answer

So, the parameter names aren't important, it is only the types of parameters that are checked.

If we introduce an additional parameter into the `add` function, is the assignment okay now?

</> TypeScript

```
let add = (a: number, b: number, c:number): number => a + b + c;
let sum = (x: number, y: number): number => x + y;
sum = add; // okay?
```

Show Answer

So, the number of parameters is important.

Let's make the `c` parameter optional. Is the assignment ok now?

</> TypeScript

```
let add = (a: number, b: number, c?: number): number => a + b + (c || 0);
let sum = (x: number, y: number): number => x + y;
sum = add; // okay?
```

Show Answer

Let's make `c` required again and switch the assignment around:

</> TypeScript

```
let add = (a: number, b: number, c: number): number => a + b + c;
let sum = (x: number, y: number): number => x + y;
add = sum; // okay?
```

Is the assignment ok now?

Show Answer

So, if function parameters are a subset of the parameters of another function, it can be assigned to it.

Wrap up

TypeScript uses structural typing, which means that variables with different types can be assigned to one another if the types are compatible. Here are some rules we can use to determine whether types are compatible:

- A variable, `a`, can be assigned to another value, `b`, if the type of `b` is narrower than the type of `a`.
- An object, `a`, can be assigned to another object, `b`, if `b` has at least the same members as `a`.
- A function, `a`, can be assigned to another function, `b`, if each parameter in `a` has a corresponding parameter in `b` with a compatible type.

More information on type compatibility can be found in the TypeScript handbook.

Well done! We now understand how to create a range of different types in TypeScript.

Next, let's check what we have learned with a quiz.