# Nominal Types

This lesson talks about nominal types, a concept that is not part of TypeScript, but can be emulated easily.

WE'LL COVER THE FOLLOWING



## Structural vs nominal typing

The majority of programming languages traditionally associated with static typing (Java, C#, C++) follow an approach called *nominal typing*. In a nominal type system every type is unique. Imagine that you have created two interfaces with exactly the same properties. In TypeScript, it's perfectly fine to assign an instance of one of these interfaces to a variable typed using the other one. However, in languages with nominal typing, that wouldn't be possible. TypeScript doesn't use nominal typing, instead is uses *structural typing*.

```
interface Cat {
    name: string;
    breed: string;
}

interface Dog {
    name: string;
    breed: string;
}

const cat: Cat = { name: 'Filemon', breed: 'Chartreux' };
let dog: Dog = cat; // ✅ No error
```

Run the code to verify that no error is produced.

The TypeScript is different because of the need for compatibility with JavaScript. With nominal typing, even the line below would result in an error:

```
const cat: Cat = { name: 'Filemon', breed: 'Chartreux' };
```

This is because the right-hand side of the assignment is an object of object literal type `{ name: string; breed: string; }`, a type that is nominally different from type `Cat`. Nominally typed language would therefore not allow such an assignment.

## Branded types #

However, nominal typing is sometimes useful. Imagine writing an application that loads many entities from a database. Each entity has some identifier (`id`) represented as a `string`. It might happen that you accidentally pass a `userId` where an `articleId` is needed and there is nothing in the type system that would prevent this.

With nominal typing, you could create `UserId` and `ArticleId` types that are effectively aliases for `string`, but since every type is unique in nominal typing, the type checker would let you pass an instance of `ArticleId` where `UserId` is required.

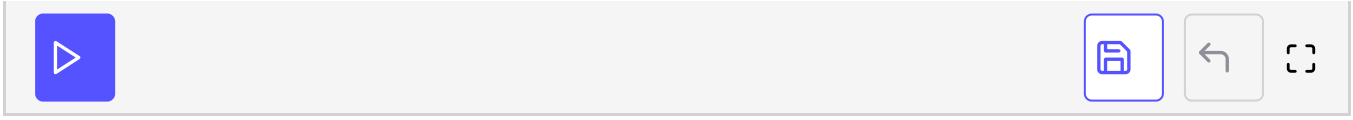
In TypeScript, it is possible to achieve suthish behavior with *branded types*.

```
type UserId = string & { __brand: "UserId" };
type ArticleId = string & { __brand: "ArticleId" };

declare const userId: UserId;

function getArticle(articleId: ArticleId) {}

getArticle(userId); // Error!
```



Run the code to see the error.

You can create a branded type by intersecting it with an object literal with one string literal property, `__brand`. The name of the property is just a convention,

any name should work. `UserId` is not assignable to `ArticleId` because the types of their `__brand` properties don't match. Therefore, it's very important to make sure that string literals used as `__brand` are unique across the whole application.

## Generalizing nominal types #

It's possible to generalize this mechanism. Below you can see the `Nominal` generic type which takes any type `T` and makes it a branded type by intersecting it with an object literal with `__brand` typed as `K`.

```
type Nominal<T, K extends string> = T & { __brand: K };

type UserId = Nominal<string, "UserId">;
type ArticleId = Nominal<string, "ArticleId">;
```

## Example: non-empty strings #

An example of using nominal types is making the type system help you detect a situation when someone is passing an empty string where a non-empty string is expected.

```
type Nominal<T, K extends string> = T & { __brand: K };

type NonEmptyString = Nominal<string, "nonempty">;

const isNonEmpty = (s: string): s is NonEmptyString => s.length > 0;

const foo = (s: NonEmptyString) => console.log(s);

const bar = "abcd";
foo(bar); // error!

if (isNonEmpty(bar)) {
  foo(bar); // OK
}
```

Run the code to see the error.

`NonEmptyString` is a branded `string`. The only way to create an instance of `NonEmptyString` (aside from type assertion) is by using a custom type guard, `isNonEmpty`. The type guard checks whether the string is indeed empty, in

which case, it marks the string as `NonEmptyString`. `foo` is a function that requires a `NonEmptyString`. The only way to call this function is to wrap it in the `isNonEmpty` type guard, which effectively forces the caller to only call the function if the string is indeed non-empty.