Branded Alias

This lesson introduces the pattern of branded aliases to enable different structures at design and runtime.

WE'LL COVER THE FOLLOWING

Limitations

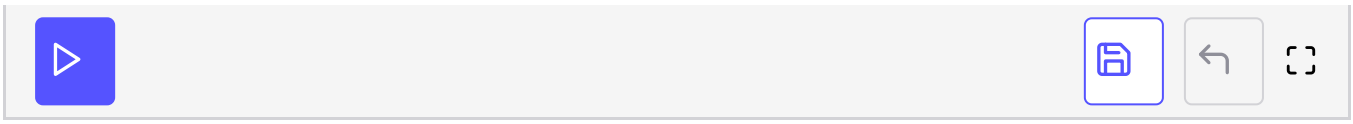
One strength of TypeScript is that relies on structure, similar to how JavaScript does not rely on type names. However, in some circumstances, it can be a limitation. The main case is when we want to know the type of a specific object, for example, in order to access specific type properties or to compare another object.

Branded type with a unique field

This method leverages the structure feature of TypeScript by using a unique field to prevent two different types from being similar. The following does not transpile because TypeScript figures out that it is impossible for the two types to be equal.

```
interface BaseClass {
  id: number;
}
interface Type1 extends BaseClass {
  _type1Brand: void;
}
interface Type2 extends BaseClass {
  _type2Brand: void;
}
```

```
let t1: Type1 = { id: 1 } as Type1;
let t2: Type2 = { id: 1 } as Type2;;

if (t1 === t2) {
  console.log("Same");
} else {
  console.log("Different");
}
```



The previous code returns the error `This condition will always return 'false' since the types 'Type1' and 'Type2' have no overlap.` .

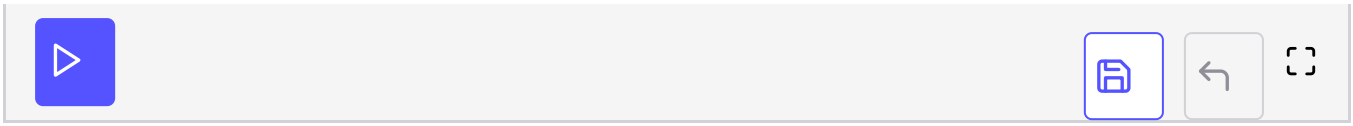
For many children inheriting (extending) a base class, branding with a unique field can be used to identify the specificity of the child. The downside of this technique is that it requires a cast that adds code at each assignment, though it does not add any runtime penalty since no value is assigned.

Branded type with literal type

The pattern of branded types means creating uniqueness in the structure itself. One way is to use a common name among types and to have a unique type. The unique type can be a literal type to avoid the creation of a new unique type each time

```
interface TypeA {
  kind: "TYPEA"
  name: string;
  id: number;
}

interface TypeB {
  kind: "TYPEB"
  error: boolean;
}

let var1: TypeA = {
  kind: "TYPEA",
  name: "Variable1",
  id: 1
};

let var2: TypeB = {
  kind: "TYPEB",
  error: true
};


if (var1 === var2) { // Always will be false
```

```
if (var1 === var2) { // Always will be false
}
```



The previous example uses the common property `_kind` which has a unique value for each type. When compared in the function `print`, TypeScript knows that because of its literal type that defines the interface, it can narrow down into the proper type, therefore allowing a developer to have proper Intellisense and successful compilation. Furthermore, similar to the `_brand` pattern, comparing two impossible objects will be caught by TypeScript similarly to the previous pattern.

Works with JavaScript

The two patterns in this lesson work well with JavaScript. The reason is that they force an instance of a type to have either a string defined on a unique property or to have a string literal assigned. In both cases, the generated JavaScript objects have a structural uniqueness available at runtime.