Type Checking with Intersections

This lesson explains how to identify a type using an intersection method.

Using a similar aspect of uniqueness as found in structure before, it is possible to define a unique type using a generic class and intersection. The result is many new types that have a private, unique property which make the shared type unique in turn, depending on its implementation.

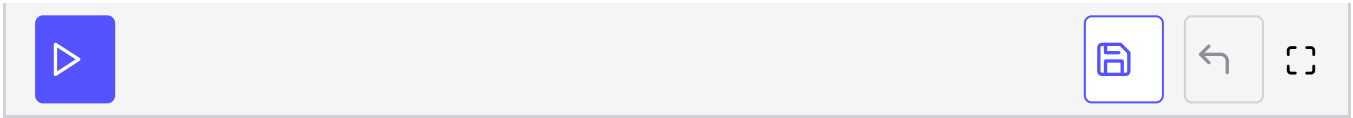
The epitome of this pattern of intersections is currencies. A currency is a number and many numbers can be of disparate currency. You want a single type for the money but many currencies. At some point, you may need to verify that you are using the same currency, hence determining the type.

```
interface Branded<TName extends string> {
  kind: TName;
}

type USD = number & Branded<"USD">
type CDN = number & Branded<"CDN">

let money1 = 50 as USD;
let money2 = 20 as CDN;

if (money1 === money2) {
  // Same
}
```



The example above does not transpile because TypeScript knows that they will always be different because of their structure. It is possible to cast since `50` and `20` are two valid numbers. Changing the `number & Branded<"USD">` to `string & Branded<"USD">` would not transpile at the casting line.

Once again, this methodology uses casting, which leans into bad practice in general. Nevertheless, TypeScript catches invalid casting with transpilation error guarding against forced casting to wrong value. A good practice is to

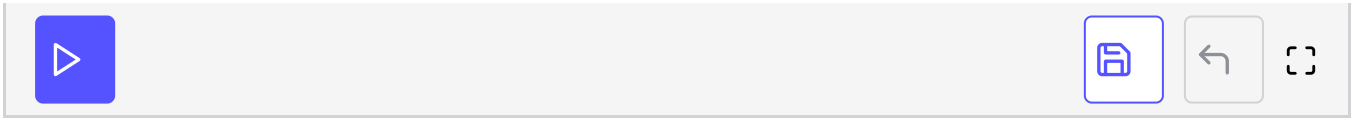
unify the casting at a unique place and limit the number of different places where casting occurs in the code.

```
type SupportedCurrencies = "USD" | "CDN"

interface Branded<TName extends SupportedCurrencies> {
  kind: TName;
}

function getMoney<T extends SupportedCurrencies>(amount: number) {
  return amount as number & Branded<T>;
}

let money1 = getMoney<"USD">(50);
let money2 = getMoney<"CDN">(20);
```



The modifications to the code leverage a type alias where all supported currency of the system can be added. Then, the `Branded` class is a generic with the constraint of needing a currency. After that, it is a matter of creating a function to get the money, which takes a number as an input parameter. The currency is provided and constrained by setting the type of the generic. Later, the type is used to cast