Community

We've finally come to one of the most important sections in the entire course. What do you do when you have an `unknown` type and you want to use it in a meaningful way? What about a Union of several types? To do so safely requires us adding some runtime checks which prove to the TypeScript type checker that the value has a specific type. We call these runtime checks "type guards".

We can create type guards using any kind of conditional - `if` statements, `switch` statements, ternaries, and a few others. We put some kind of check against the value's type inside the `if` statement's condition. Then, inside the block, our value now has the type we matched it against.

Primitive types

The most straightforward check is strict equality.

```
function sayAlexsNameLoud(name: unknown) {
  if (name === "Alex") {
    // name is now definitely "Alex"
    console.log(`Hey, ${name.toUpperCase()}`); // "He
  }
}
```

We could pass literally any value to this function because of the type guard. It's still type safe, since TypeScript knows that inside the `if` block, name is a literal `"Alex"` type, which has the same methods and properties as a `string`.

That's a little tedious if we needed to do that for literally every name. Earlier in the course, we used the `typeof` operator to check what primitive type a value is. We can do that with any primitive type to narrow down a value to a particular type.

```
function sayNameLoud(name: unknown) {
  if (typeof name === "string") {
    // name is now definitely a string
    console.log(`Hey, ${name.toUpperCase()}`);
  }
}
```

We could also do the inverse of this. Instead of checking to see if `name` is a `string`, we can check if it is *not* a `string`, and then return early or throw an error. Then we know that everywhere else in our function, `name` is a `string`

```
function sayNameLoud(name: unknown) {
  if (typeof name !== "string") return;
  // name is now definitely a string
  console.log(`Hey, ${name.toUpperCase()}`);
}
```

We could use `switch` statements to check the type as well. In this case, we'll narrow a Union type of `number` and `string`, but we could do the same thing with `unknown`.

```
function calculateScore(score: number | string) {
  switch (typeof score) {
    case "string":
      return parseInt(score) + 10;
      break;
    case "number":
      return score + 10;

      break;
    default:
      throw new Error("Invalid type for score");
  }
}
```

`typeof` can cover us in a lot of cases, but once we get to referential types, like objects, arrays, and class instance, we can't use it anymore. Whenever we use `typeof` of any of these types, it will always return "object". We'll have to use more sophisticated methods to determine the types of our values.

Arrays

We can check if a value is an array using the `Array.isArray` method.

```
function combineList(list: unknown): any {
  if (Array.isArray(list)) {
    list; // (parameter) list: any[]
  }
}
```

Our list is now a list of `any` s, which is better than before, but we still don't know the type of the values inside the list. To do that, we'll have to loop through our array to narrow the type of each item. We can use the `.filter()` and `.map()` methods on our array to do that. We could also use a `for` loop. Which you choose depends on your circumstances.

```
function combineList(list: unknown): any {
  if (Array.isArray(list)) {
    // This will filter any items which are not numbe
    const filteredList: number[] = list.filter((item)
      if (typeof item !== "number") return false;
```

```
        return true;
      });

      // This will transform any items into numbers, an
      const mappedList: number[] = list.map((item) ⇒ {
        const numberValue = parseFloat(item);
        if (isNaN(numberValue)) return 0;
        return numberValue;
      });

      // This does the same thing as the filter, but wi
      let loopedList: number[] = [];
      for (let item of list) {
        if (typeof item === "number") {
          loopedList.push(item);
        }
      }
    }
  }
```

Classes

We can determine whether a value is an instance of a specific class using the `instanceof` operator.

```
class Fruit {
  constructor(public name: string) {}
  eat() {
    console.log(`Mmm. ${this.name}s.`);
  }
}

function eatFruit(fruit: unknown) {
  if (fruit instanceof Fruit) {
    fruit.eat();
  }
}
```

Narrowing a value to a class immediately lets us know all of the properties and methods which are on the value.

Objects

Objects are a little trickier to narrow, since they could have any combination of properties and types. Most of the time we don't care about the type of the entire object; all we want is to access one or two properties on the object.

We have already used the `in` operator to determine whether a property exists on an object. We can then use the `typeof` operator to determine that property's type.

The `in` operator only works to narrow union types, so we can't use it with `unknown`. Instead, we'll have to use another special type that comes with TypeScript: `object`. This type represents anything that isn't a `string`, `number`, `boolean`, or one of the other primitive types. Using `object` instead of `unknown` will tell TypeScript to let us attempt to access properties on this value. We can create a Union of the generic `object` type and an Interface with the property that we want to access.

```
interface Person {
  name: string;
}
function sayNameLoud(person: object | Person) {
  if ("name" in person) {
    console.log(`Hey, ${person.name.toUpperCase()}`);
  }
}
```

In this case, TypeScript doesn't care what other properties are on our `person` type. All that matters is that it has a `name` property which is a `string`. We could actually abuse this quirk in the type checker to create code that looks type safe but throws a runtime type error.

```
interface Person {
  name: string;
  age: number;
}
function sayNameLoud(person: object | Person) {
  if ("age" in person) {
    console.log(`Hey, ${person.name.toUpperCase()}`);
  }
}

sayNameLoud({ age: 27 });
```

The best way to avoid this is to check for all the properties that we actually use so we can guarantee that they are the type we want them to be.

Hopefully this illustrates that, while TypeScript is very sophisticated and will catch 98% of your type errors, there are still ways to unintentionally get around it and write unsafe code. In the next few sections, we'll see more ways to fool the type checker and how to avoid doing that.

Next Lesson or discuss in the Community