Array Type Annotations

The TypeScript type annotation for array types is fairly straightforward: we put [] after the element type. In this code, names is an Array that can only contain strings:

```
let names: string[] = ['Danny', 'Samantha'];
```

An alternative method is to use the Array syntax, where T stands for the type.

```
let names: Array<string> = ['Danny', 'Samantha']
```

Multi-dimensional Arrays

In fact, we can make arrays of any type whatsoever. We can also declare multidimensional arrays: arrays of arrays (of some type).

```
let arr: string[][] = [['str1', 'str2'], ['more', 'strings']];
```

The empty array ([]) is compatible with any array type:

```
let names: string[] = []; // No type errors.
let numbers: number[] = []; // No type errors.
names.push('Isabella');
numbers.push(30);
```

Tuples

In TypeScript, when an array is typed with elements of specific types, it's called a tuple.

Tuple types specify both the lengths and the orders of compatible tuples, and will cause an error if either of these conditions are not met:

```
let ourTuple: [string, number, string, boolean] = ['Is', 7 , 'our favorite number?' , false];
```

Array Type Inference

TypeScript can infer variable types from initial values and return statements. Even still, we may not know exactly what type inference to expect when dealing with arrays. For example:

```
let examAnswers= [true, false, false];
examAnswers[3] = true; // No type error.
```

We also get the same kind of type inference when we use the .concat() method:

```
let tup: [number, number, number] = [1,2,3];
let concatResult = tup.concat([4,5,6]); // concatResult has the value
[1,2,3,4,5,6].
```

Rest Parameters

Assigning types to rest parameters is similar to assigning types to arrays. Here's a rest parameter example without types:

```
function smush(firstString, ...otherStrings){
  let output = firstString;
  for(let i = 0; i < otherStrings.length; i++){
    output = output.concat(otherStrings[i]);
  }
  return output;
}

smush('a','h','h','H','H','H','!','!'); // Returns: 'ahhHHH!!'.
```

The function with a correctly typed rest parameter is then:

```
function smush(firstString:string, ...otherStrings: string[]){
  let output = firstString;
  for(let i = 0; i < otherStrings.length; i++){
    output = output.concat(otherStrings[i]);
  }
  return output;
}
console.log(smush('a','h','h','H','H','H','!','!')); // Returns:
'ahhHHH!!'.
```

Spread Syntax

Like the finest wines and cheeses, TypeScript's tuples pair beautifully with JavaScript's spread syntax. This is most useful for function calls that use lots of arguments, like this:

```
function gpsNavigate(startLatitudeDegrees:number,
startLatitudeMinutes:number, startNorthOrSouth:string,
startLongitudeDegrees: number, startLongitudeMinutes: number,
startEastOrWest:string, endLatitudeDegrees:number,
endLatitudeMinutes:number , endNorthOrSouth:string, endLongitudeDegrees:
number, endLongitudeMinutes: number,  endEastOrWest:string) {
```

```
  /* navigation subroutine here */
}
```

Instead, we can use tuple variables that represent the starting and ending coordinates:

```
let codecademyCoordinates: [number, number, string, number, number,
string] = [40, 43.2, 'N', 73, 59.8, 'W'];
let bermudaTCoordinates: [number, number, string, number, number, string]
= [25, 0 , 'N' , 71, 0, 'W'];
```

Now, we use JavaScript's spread syntax to write a very readable function call:

```
gpsNavigate(...codecademyCoordinates, ...bermudaTCoordinates);
// And by the way, this makes the return trip really convenient to compute
too:
gpsNavigate(...bermudaTCoordinates, ...codecademyCoordinates);
// If there is a return trip . . .
```

CUSTOM TYPES

TypeScript can also be used to create custom types, rather than being limited to pre-defined types. Custom types are what make TypeScript really fun and useful, because they enable type checking that's tailored to your exact purposes.

Enums

Our first example of a complex type is also one of the most useful: enums. We use enums when we'd like to enumerate all the possible values that a variable could have. This is in contrast to most of the other types we have studied. A variable of the string type can have any string as a value; there are infinitely many possible strings, and it would be impossible to list them all. Similarly, a variable of the boolean[] type can have any array of booleans as its value; again, the possibilities are infinite.

```
enum Direction {
  North,
  South,
  East,
  West
}
```

An enum type can be used in a type annotation like any other type.

```
let whichWayToArcticOcean: Direction;
whichWayToArcticOcean = Direction.North; // No type error.
whichWayToArcticOcean = Direction.Southeast; // Type error: Southeast is
not a valid value for the Direction enum.
whichWayToArcticOcean = West; // Wrong syntax, we must use Direction.West
instead.
```

Under the hood, TypeScript processes these kinds of enum types using numbers. Enum values are assigned a numerical value according to their listed order. The first value is assigned a number of 0, the second a number of 1, and onwards

We can change the starting number, writing something like Here, Direction.North, Direction.South, Direction.East, and Direction.West are equal to 7, 8, 9, and 10, respectively.

```
enum Direction {
  North = 7,
  South,
  East,
  West
}
```

We can also specify all numbers separately, if needed:

```
enum Direction {
  North = 8,
  South = 2,
  East = 6,
  West = 4
}
```

String Enums vs. Numeric Enums

The enums we have studied so far are referred to as numeric enums, since they are based on numbers. TypeScript also allows us to use enums based on strings, referred to as string enums. They are defined very similarly:

With numeric enums, the numbers could be assigned automatically, but with string enums we must write the string explicitly, as shown above

```
enum DirectionNumber { North, South, East, West }
enum DirectionString { North = 'NORTH', South = 'SOUTH', East = 'EAST',
West = 'WEST' }
```

Object Types

TypeScript's object types are extremely useful, as they allow us extremely fine-level control over variable types in our programs. They're also the most common custom types, so we'll have to understand them if we want to read other people's programs.

`let aPerson: {name: string, age: number};`

The type annotation looks like an object literal, but instead of values appearing after properties, we have types. Notice that the variable aPerson has yet to be assigned a value.

```
aPerson = {name: 'Aisle Nevertell', age: "wouldn't you like to know"}; //
Type error: age property has the wrong type.
aPerson = {name: 'Kushim', yearsOld: 5000}; // Type error: no age
property.
aPerson = {name: 'User McCodecad', age: 22}; // Valid code.
```

TypeScript places no restrictions on the types of an object's properties. They can be enums, arrays, and even other object types!

```
let aCompany: {
  companyName: string,
  boss: {name: string, age: number},
  employees: {name: string, age: number}[],
  employeeOfTheMonth: {name: string, age: number},
  moneyEarned: number
};
```

Type Aliases

One great way to customize the types in our programs is to use type aliases. These are alternative type names that we choose for convenience. We use the format type `<alias name> = <type>`:

```
type MyString = string;
let myVar: MyString = 'Hi'; // Valid code.
```

Type aliases are truly useful for referring to complicated types that need to be repeated, especially object types and tuple types.

```
type Person = { name: string, age: number };
let aCompany: {
  companyName: string,
  boss: Person,
  employees: Person[],
```

```
  employeeOfTheMonth: Person,
  moneyEarned: number
};
```

Function Types

One of the neat things about JavaScript is that functions can be assigned to variables.

```
let myFavoriteFunction = console.log; // Note the lack of parentheses.
myFavoriteFunction('Hello World'); // Prints: Hello World
```

One of the neat things about TypeScript is that we can precisely control the kinds of functions assignable to a variable. We do this using function types, which specify the argument types and return type of a function. Here's an example of a function type that is only compatible with functions that take in two string arguments and return a number.

```
type StringsToNumberFunction = (arg0: string, arg1: string) => number;
```

This syntax is just like arrow notation for functions, except instead of the return value we put the return type. In this case, the return type is number. Because this is just a type, we did not write the function body at all. A variable of type StringsToNumberFunction can be assigned any compatible function:

```
let myFunc: StringsToNumberFunction;
myFunc = function(firstName: string, lastName: string) {
  return firstName.length + lastName.length;
};

myFunc = function(whatever: string, blah: string) {
  return whatever.length - blah.length;
};
// Neither of these assignments results in a type error.
```