✓ 100 XP

Exercise - Define function types

5 minutes

You can define function types and then use them when creating your functions. This design is useful if you want to apply the same function type signature to more than one function.

You can define a function type using a type alias or an interface. Both approaches work essentially the same so it's up to you to decide which is best. An interface is better if you want to have the option of extending the function type. A type alias is better if you want to use unions or tuples.

Let's create a function that performs an add operation or a subtract operation depending on the value of a parameter that is passed to it. Both the add and subtract operations accept two numbers, `x` and `y`, and return the result as a number.

1. Open the Playground and remove any existing code.

2. Define a function type called `calculator` using a type alias. The type signature has a parameter list `(x: number, y: number)` and returns a `number`, separated by an arrow (`=>`) operator. (Notice that the syntax of the type signature is the same as an arrow function.)

   TypeScript

   ```typescript
   type calculator = (x: number, y: number) => number;
   ```

3. You can now use the function type as a type signature when declaring functions. Declare two variables of the function type `calculator`, one for the add operation and one for the subtract operation. Test the new functions by returning the result of each one to the console.

   TypeScript

```
let addNumbers: calculator = (x: number, y: number): number => x + y;
let subtractNumbers: calculator = (x: number, y: number): number => x - y;

console.log(addNumbers(1, 2));
console.log(subtractNumbers(1, 2));
```

4. You can also use the `calculator` function type to pass values from another function. Enter the `doCalculation` function, which returns the result of the `addNumbers` or `subtractNumbers` function based on the value passed to the `operation` parameter.

TypeScript

```
let doCalculation = (operation: 'add' | 'subtract'): calculator => {
    if (operation === 'add') {
        return addNumbers;
    } else {
        return subtractNumbers;
    }
}
```

5. Try running the function by entering `console.log(doCalculation('add')(1, 2))` and you'll notice that TypeScript is able to provide Intellisense help based on the types defined in `doCalculation` and `calculator`.

6. Now, comment out the `calculator` function type and declare a new one using an interface. Notice that the type signature is slightly different, with a colon (`:`) separating the parameter list and return type instead of the arrow operator. Replace the `calculator` function with the `Calculator` interface in the variable declarations. When you're done, the code should work the same.

TypeScript

```
// type calculator = (x: number, y: number) => number;
interface Calculator {
    (x: number, y: number): number;
}
```

Function type inference

When you define a function, the names of the function parameters don't need to match the names in the function type. While you're required to name the parameters in the type signature in both places, the names are ignored when checking if two function types are compatible.

You can also leave off the parameter types and return type because TypeScript will infer these types from the function type definition.

As far as TypeScript is concerned, these three statements are identical.

TypeScript

```typescript
let addNumbers: Calculator = (x: number, y: number): number => x + y;
let addNumbers: Calculator = (number1: number, number2: number): number => number1
+ number2;
let addNumbers: Calculator = (num1, num2) => num1 + num2;
```

Next unit: Lab - Use functions in TypeScript

Continue >

How are we doing? ☆ ☆ ☆ ☆ ☆