Taking advantage of Functional Programming in TypeScript

The main concept of Functional Programming is: "*write programs by composing generic reusable functions".*

To do that, we can use the following three concepts:

- Use Functions Instead of Simple Values
- Model Data Transformations as a Pipeline
- Extract Generic Functions

Supposed model

Suppose that we have that classes model: A device have a name and a type. A Camera is a concrete type of device. Each camera has a type that can be movable or fixed.

Data Model

```
enum DeviceType {
  Camera, DMS
}
enum CameraType {
  Movable, Fixed
}
class Device {
  constructor(public name: string, public deviceType: DeviceType) {}
}
class Camera extends Device {
  constructor(public name: string, public cameraType: CameraType) {
    super(name, DeviceType.Camera);
  }
  isFixed(): boolean {
    return this.cameraType === CameraType.Fixed;
  }
}
let devices = [
    new Device("DMS1", DeviceType.DMS),
    new Camera("Camera1", CameraType.Movable),
    new Device("DMS2", DeviceType.DMS),
    new Camera("Camera2", CameraType.Fixed),
    new Camera("Camera3", CameraType.Movable),
    new Camera("Camera4", CameraType.Fixed),
    new Camera("Camera4", CameraType.Fixed)
  ];
```

Now assume that we need to create a function that counts the movable cameras from our devices list. That could be the code:

```
function totalMovableCameras(devices: Device[]): number {
  let totalCameras = 0;
   devices.forEach((e) => {

     if(e.deviceType === DeviceType.Camera && !<Camera>e.isFixed())
       totalCameras++;
   });
  return totalCameras;
}
```

If we execute console.log(totalMovableCameras(devices)), we get 2.

Use Functions Instead of Simple Value

At first, this concept can cause headaches, but is one of the most powerful techniques for generate and generalize code.

What does it mean? well, it means that we can treat conditions as a functions. In our case, we have two conditions:

`deviceType === DeviceType.Camera, and !camera.isFixed().`

Then, we can define a new type named Predicate, that receives a device and return a boolean value:

```
type Predicate = (d: Device) => boolean;
```

This is very powerful because we can refactor our function passing a list of predicates and check them directly:

```
function totalMovableCameras(devices: Device[], conditions: Predicate[]): number {
  let totalCameras = 0;
   devices.forEach((d) => {
       if (conditions.every(c => c(d)))   totalCameras++;
   });
  return totalCameras;
}
```

We can execute that function this way:

```
totalMovableCameras(devices, [(d) => d.deviceType === DeviceType.Camera, (d) =>
!<Camera>d.isFixed()]);
```

Wait a moment!! with that solution we can rename the function because with predicates is a more general function:

```
function getDevicesCountByConditions(devices: Device[], conditions: Predicate[]):
number {
  let count= 0;
   devices.forEach((d) => {
       if (conditions.every(c => c(d)))   count++;
   });
  return count;
}
```

Really cool, almost for me 😉!!

But we can improve that function a little bit more. We can say that a list of conditions is a composition of conditions, applying each one of them. Then, we can create a combinator, and use it into our function:

```
function and(predicates: Predicate[]): Predicate{
  return (e) => predicates.every(p => p(e));
}

function getDevicesCountByConditions(devices: Device[], conditions: Predicate[]):
number {
  let count= 0;
   devices.forEach((d) => {
       if (and(conditions)(d))   count++;
   });
  return count;
}
```

The and combinator applies conditions over passed device, using the currying concept, this is use that function doing partial application of the provided arguments. You can read more about currying here.

Model Data Transformations as a Pipeline

Can we improve that? Yes, we can. We can think as a pipeline: get first desired data and then count:

```
function getDevicesCountByConditions(devices: Device[], conditions: Predicate[]):
number {
  const filteredDevices = devices.filter(and(conditions));
  return filteredDevices.length;
}
```

Goal: Make it more complicated (or not), we want get the movable cameras average of whole cameras total. Is easy:

```
function movableCamerasAverage(devices: Device[]){
  const totalCameras = getDevicesCountByConditions(devices, [(d) => d.deviceType
=== DeviceType.Camera]);
  const movableCamerasCount = getDevicesCountByConditions(devices, [(d) =>
d.deviceType === DeviceType.Camera, (d) => !<Camera>d.isFixed()]);
  return movableCamerasCount / totalCameras;
}
console.log(movableCamerasAverage(devices));
```

We get 0.4 is the correct result.

Extract Generic Functions

We can simplify that code extracting Generic Functions. We can create a function to filter devices and use into our getDevicesCountByConditions function:

```
//Set the conditions into constants
const movableCamerasConditions = [(d) => d.deviceType === DeviceType.Camera, (d) =>
!<Camera>d.isFixed()];
const totalCamerasConditions = [(d) => d.deviceType === DeviceType.Camera];



function filterDevices(conditions: Predicate[]): Device[] {
   return (devices) => devices.filter(and(conditions));
}

//Uses currying to filter the devices applying conditions.
function getDevicesCountByConditions(devices: Device[], conditions: Predicate[]):
number {
  return filterDevices(conditions)(devices).length;
}

console.log(getDevicesCountByConditions(devices, movableCamerasConditions));
```

We get 2, is the number of movable cameras.

So far so good. Now we can change the movableCamerasAverage function:

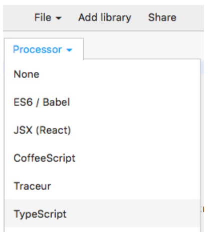
```
function movableCamerasAverage(devices: Device[]){
  const totalCameras = getDevicesCountByConditions(devices,
totalCamerasConditions);
  const movableCamerasCount = getDevicesCountByConditions(devices,
movableCamerasConditions);
  return movableCamerasCount / totalCameras;
}
console.log(movableCamerasAverage(devices));
```

It prints 0.4 as expected. Can be improved applying transformations as pipeline:

```
function movableCamerasAverage(devices: Device[]){
  const totalCameras = filterDevices(totalCamerasConditions)(devices);
  const movableCameras = filterDevices(movableCamerasConditions)(totalCameras);
  return movableCameras.length / totalCameras.length;
}
```

If we Compare the original and final solutions, obviously the last ones are better: more generic without breaking the interface of the function, and we are working with mutable states and if statements, which made the code more readable.

You can test the code here. To run you must set typescript as code processor:


File
Add library
Share
Processor
None
ES6 / Babel
JSX (React)
CoffeeScript
Traceur
TypeScript