Creating an Exception

This lesson introduces how to work with exceptions.

Throwing exceptions

TypeScript follows ECMAScript in terms of the exception. You can throw a new exception by using the keyword `throw` followed by an instance of the exception you desire. It's always better to throw an object instead of a string, directly. The reason is that throwing an object comes with a whole execution stack since the object must inherit from the base interface `Error`. The interface `Error` defines a name, a message, and the stack. The only case where you do not need to throw an instance of an exception is when using the `ErrorConstructor` which lets you throw the error directly by using `throw Error("Message here that is optional")`.

```
function throw1() {
    throw "error in string";
}

function throw2() {
    throw Error("Message Here");
}

function throw3() {
    const err: Error = new Error("Message Here");
    throw err;
}

// throw1();
// throw2();
// throw3();
```

The code above has three lines commented. The first one, **line 14**, returns a `string`. When thrown the output is:

> /usercode/index.ts:3 throw “error in string”;

Comment again **line 14** and try **line 15** and then **line 16**. Both will return a richer output:

> /usercode/index.ts:6 throw Error(“Message Here”); ^ Error: Message Here at throw2 (/usercode/index.ts:6:11) at Object. (/usercode/index.ts:15:1) at Module._compile (internal/modules/cjs/loader.js:778:30) at Module.m._compile (/usr/lib/node_modules/ts-node/src/index.ts:473:23) at Module._extensions...js (internal/modules/cjs/loader.js:789:10) at Object.require.extensions.(anonymous function) [as .ts] (/usr/lib/node_modules/ts-node/src/index.ts:476:12) at Module.load (internal/modules/cjs/loader.js:653:32) at tryModuleLoad (internal/modules/cjs/loader.js:593:12) at Function.Module._load (internal/modules/cjs/loader.js:585:3) at Function.Module.runMain (internal/modules/cjs/loader.js:831:12)

It might be convoluted to read, but definitely more information using the `Error` object.

Custom typed error

TypeScript lets you inherit the Error class to create your own custom error. A custom error can be important because it lets you distinguish what error is being thrown later in order to handle the exception accordingly. It is possible to compare the messages but is good practice to rely on something more unique. A custom error gives you the option to compare the name of the exception as well as to provide additional information like error number, context data, etc.

context data, etc.

📜Note: the below code throws an error ❌

```
class MySpecificError extends Error {
    public data: string;
    constructor(data: string, message: string) {
        super(message);
        Object.setPrototypeOf(this, MySpecificError.prototype); // Restore prototype chain
        // OR:
        // Object.setPrototypeOf(this, new.target.prototype); // Restore prototype chain
        this.data = data;
    }
}

throw new MySpecificError("dataHere", "My Message");
```

The code above needed to have a call to `Object.setPrototypeOf` and you may ask why. The reason is that with the introduction of TypeScript version 2.1, when using EcmaScript 6 the `Error` constructor uses `new.target` to handle the prototype chain. In prior versions of EcmaScript, it did not. By manually calling it or by using the class's prototype to assign it, we restore the peace by being sure that the prototype chain is invoking `new.target`. Forgetting to do so will cause an issue when using `instanceOf` which we will use in an upcoming lesson to differentiate among exceptions.

Exception without error

It is best practice to always use `Error()` or `new Error()` which are identical solutions instead of directly throwing a string. You get more information with your error, like the stack.

Better to do:

```
throw Error("Message");
```

than:

```
throw "Message";
```

Be cautious

Like every language, an exception is only for an alternative path that occurs exceptionally. The reason is that an exception flow is harder to understand and is not performing.

There are alternatives possible:

For example, the function in error can return an error object, or undefined. However, the return option has the drawback of taking the spot of the potential returned value. It's possible to return the desired type and union with the exception type to mitigate this issue.

Indeed, the union solution brings more type checks on each invocation, which might not be convenient.

```
function withReturn(p1:number)
   : number | Error {
   if(wrong){
        return new Error("My error message");
   }
   return "String as expected when all good";
}
```

Another alternative to communicating exceptions is to use a parameter that is a function callback invoked in case of error.

```
function withCallback(
   p1:number,
   error:(errMsg: string) => void)
   :string {
   if(wrong){
        error("My error message");
   }
   return "String as expected when all good";
}
```

Finally, a possible choice, in the case of code using promises, is to reject the promise.

```
function withPromise(p1: number)
   => Promise<Error>{

   if(wrong){
        return Promise.reject("My error message);
   }
   return Promise.resolve("String as expected when all good");
}
```