Chapter 7. Handling Errors

A physicist, a structural engineer, and a programmer were in a car driving over a steep alpine pass when the brakes failed. The car went faster and faster, they were struggling to get around the corners, and once or twice the flimsy crash barrier saved them from tumbling down the side of the mountain. They were sure they were all going to die, when suddenly they spotted an escape lane. They pulled into the escape lane, and came safely to a halt.

The physicist said, "We need to model the friction in the brake pads and the resultant temperature rise, and see if we can work out why they failed."

The structural engineer said, "I think I've got a few spanners in the back. I'll take a look and see if I can work out what's wrong."

The programmer said, "Why don't we see if it's reproducible?"

—Anonymous

TypeScript does everything it can to move runtime exceptions to compile time: from the rich type system it provides to the powerful static and symbolic analyses it performs, it works hard so you don't have to spend your Friday nights debugging misspelled variables and null pointer exceptions (and so your on-call coworker doesn't have to be late to their great aunt's birthday party because of it).

Unfortunately, regardless of what language you write in, sometimes runtime exceptions do sneak through. TypeScript is really good about preventing them, but even it can't prevent things like network and filesystem failures, errors parsing user input, stack overflows, and out of memory errors. What it does do —thanks to its lush type system—is give you lots of ways to deal with the runtime errors that end up making it through.

In this chapter I'll walk you through the most common patterns for representing and handling errors in TypeScript:

- Returning `null`
- Throwing exceptions
- Returning exceptions
- The `Option` type

Which mechanism you use is up to you, and depends on your application. As I cover each error-handling mechanism, I'll talk about its pros and cons so you can make the right choice for yourself.

Returning null

We're going to write a program that asks a user for their birthday, which we will then parse into a `Date` object:

```
function ask() {
  return prompt('When is your birthday?')
}

function parse(birthday: string): Date {
  return new Date(birthday)
}

let date = parse(ask())
console.info('Date is', date.toISOString())
```

We should probably validate the date the user entered—it's just a text prompt, after all:

```
// ...
function parse(birthday: string): Date | null {
  let date = new Date(birthday)
  if (!isValid(date)) {
    return null
  }
  return date
}

// Checks if the given date is valid
```

```
function isValid(date: Date) {
  return Object.prototype.toString.call(date) === '[ob
      && !Number.isNaN(date.getTime())
}
```

When we consume this, we're forced to first check if the result is `null` before we can use it:

```
// ...
let date = parse(ask())
if (date) {
  console.info('Date is', date.toISOString())
} else {
  console.error('Error parsing date for some reason')
}
```

Returning `null` is the most lightweight way to handle errors in a typesafe way. Valid user input results in a `Date`, invalid user input in a `null`, and the type system checks for us that we handled both.

However, we lose some information doing it this way `parse` doesn't tell us why exactly the operation failed, which stinks for whatever engineer has to comb through our logs to debug this, as well as the user who gets a pop up saying that there was an "Error parsing date for some reason" rather than a specific, actionable error like "Enter a date in the form YYYY/MM/DD."

Returning `null` is also difficult to compose: having to check for `null` after every operation can become verbose as you start to nest and chain operations.

Throwing Exceptions

Let's throw an exception instead of returning `null`, so that we can handle specific failure modes and have some metadata about the failure so we can debug it more easily.

```
// ...
function parse(birthday: string): Date {
  let date = new Date(birthday)
  if (!isValid(date)) {
    throw new RangeError('Enter a date in the form YYY
  }
  return date
}
```

Now when we consume this code, we need to be careful to catch the exception so that we can handle it gracefully without crashing our whole application:

```
  // ...
  try {
    let date = parse(ask())
    console.info('Date is', date.toISOString())
  } catch (e) {
    console.error(e.message)
  }
```

We probably want to be careful to rethrow other exceptions, so we don't silently swallow every possible error:

```
  // ...
  try {
    let date = parse(ask())
    console.info('Date is', date.toISOString())
  } catch (e) {
    if (e instanceof RangeError) {
      console.error(e.message)
    } else {
      throw e
    }
  }
```

We might want to subclass the error for something more specific, so that when another engineer changes `parse` or `ask` to throw other `RangeError` s, we can differentiate between our error and the one they added:

```
  // ...

  // Custom error types
  class InvalidDateFormatError extends RangeError {}
  class DateIsInTheFutureError extends RangeError {}

  function parse(birthday: string): Date {
    let date = new Date(birthday)
    if (!isValid(date)) {
      throw new InvalidDateFormatError('Enter a date in t
    }
    if (date.getTime() > Date.now()) {
      throw new DateIsInTheFutureError('Are you a timelor
    }
    return date
  }

  try {
    let date = parse(ask())
    console.info('Date is', date.toISOString())
  } catch (e) {
    if (e instanceof InvalidDateFormatError) {
      console.error(e.message)
    } else if (e instanceof DateIsInTheFutureError) {
      console.info(e.message)
    } else {
      throw e
    }
  }
```

Looking good. We can now do more than just signal that something failed: we can use a custom error to indicate *why* it failed. These errors might come in handy when combing through our server logs to debug an issue, or we can map them to specific error dialogs to give our users actionable feedback about what they did wrong and how they can fix it. We can also effectively chain and nest operations by wrapping any number of operations in a single `try` / `catch` (we don't have to check each operation for failure, like we did when returning `null`).

What does it feel like to use this code? Say the big `try` / `catch` is in one file, and the rest of the code is in a library being imported from somewhere else. How would an engineer know to catch those specific types of errors (`InvalidDateFormatError` and `DateIsInTheFutureError`), or to even just check for a regular old `RangeError`? (Remember that TypeScript doesn't encode exceptions as part of a function's signature.) We could indicate it in our function's name (`parseThrows`), or include it in a docblock:

```
  /**
   * @throws {InvalidDateFormatError} The user entered th
   * @throws {DateIsInTheFutureError} The user entered a
```

```
 */
function parse(birthday: string): Date {
  // ...
```

But in practice, an engineer probably wouldn't wrap this code in a `try` / `catch` and check for exceptions at all, because engineers are lazy (at least, I am), and the type system isn't telling them that they missed a case and should handle it. Sometimes, though—like in this example—errors are so expected that downstream code really should handle them, lest they cause the program to crash.

How else can we indicate to consumers that they should handle both the success and the error cases?

Returning Exceptions

TypeScript isn't Java, and doesn't support `throws` clauses.[1] But we can achieve something similar with union types:

```
// ...
function parse(
  birthday: string
): Date | InvalidDateFormatError | DateIsInTheFutureErr
  let date = new Date(birthday)
  if (!isValid(date)) {
    return new InvalidDateFormatError('Enter a date in
  }
  if (date.getTime() > Date.now()) {
    return new DateIsInTheFutureError('Are you a timelc
  }
  return date
}
```

Now a consumer is forced to handle all three cases—`InvalidDateFormatError`, `DateIsInTheFutureError`, and successful parse—or they'll get a `TypeError` at compile time:

```
// ...
let result = parse(ask()) // Either a date or an error
if (result instanceof InvalidDateFormatError) {
  console.error(result.message)
} else if (result instanceof DateIsInTheFutureError) {
  console.info(result.message)
} else {
  console.info('Date is', result.toISOString())
}
```

Here, we successfully took advantage of TypeScript's type system to:

- Encode likely exceptions in `parse`'s signature.
- Communicate to consumers which specific exceptions might be thrown.
- Force consumers to handle (or rethrow) each of the exceptions.

A lazy consumer can avoid handling each error individually. But they have to do so explicitly:

```
  // ...
  let result = parse(ask()) // Either a date or an error
  if (result instanceof Error) {
    console.error(result.message)
  } else {
    console.info('Date is', result.toISOString())
  }
```

Of course, your program might still crash due to an out of memory error or a stack overflow exception, but there's not much we can do to recover from those.

This approach is lightweight and doesn't require fancy data structures, but it's also informative enough that consumers will know what type of failure an error represents and what to search for to find more information.

A downside is that chaining and nesting error-giving operations can quickly get verbose. If a function returns `T | Error1`, then any function that consumes that function has two options:

1. Explicitly handle `Error1`.
2. Handle `T` (the success case) and pass `Error1` through to its consumers to handle. If you do this enough, the list of possible errors that a consumer has to handle grows quickly:

```
function x(): T | Error1 {
  // ...
}
function y(): U | Error1 | Error2 {
  let a = x()
  if (a instanceof Error) {
    return a
  }
  // Do something with a
}
function z(): U | Error1 | Error2 | Error3 {
  let a = y()
  if (a instanceof Error) {
    return a
  }
  // Do something with a
}
```

This approach is verbose, but gives us excellent safety.

The Option Type

You can also describe exceptions using special-purpose data types. This approach has some downsides compared to returning unions of values and errors (notably, interoperability with code that doesn't use these data types), but it does give you the ability to chain operations over possibly errored computations. Three of the most popular options (heh!) are the `Try`, `Option`,[2] and `Either` types. In this chapter, we'll just cover the `Option` type;[3] the other two are similar in spirit.

NOTE

Note that the `Try`, `Option`, and `Either` data types don't come built into JavaScript environments the same way that `Array`, `Error`, `Map`, or `Promise` are. If you want to use these types, you'll have to find implementations on NPM, or write them yourself.

The `Option` type comes to us from languages like Haskell, OCaml, Scala, and Rust. The idea is that instead of returning a value, you return a *container* that may or may not have a value in it. The container has a few methods defined on it, which lets you chain operations even though there may not actually be a value inside. The container can be pretty much any data structure, so long as it can hold a value. For example, you could use an array as the container:

```
// ...
function parse(birthday: string): Date[] {
  let date = new Date(birthday)
  if (!isValid(date)) {
    return []
  }
  return [date]
}

let date = parse(ask())
date
  .map(_ => _.toISOString())
  .forEach(_ => console.info('Date is', _))
```

NOTE

As you may have noticed, a downside of `Option` is that, much like our original `null` -returning approach, it doesn't tell the consumer why the error happened; it just signals that something went wrong.

Where `Option` really shines is when you need to do multiple operations in a row, each of which might fail.

For example, before we assumed that `prompt` always succeeds, and `parse` might fail. But what if `prompt` can fail too? That might happen if the user cancelled out of the birthday prompt—that's an error and we shouldn't continue our computation. We can model that with… another `Option` !

```
function ask() {
  let result = prompt('When is your birthday?')
  if (result === null) {
    return []
  }
  return [result]
}
// ...
```

```
ask()
  .map(parse)
  .map(date => date.toISOString())
    // Error TS2339: Property 'toISOString' does not ex
  .forEach(date => console.info('Date is', date))
```

Yikes—that didn't work. Since we mapped an array of `Date` s (`Date[]`) to an array of arrays of `Date` s (`Date[][]`), we need to flatten it back to an array of `Date` s before we can keep going:

```
flatten(ask()
  .map(parse))
  .map(date => date.toISOString())
  .forEach(date => console.info('Date is', date))

// Flattens an array of arrays into an array
function flatten<T>(array: T[][]): T[] {
  return Array.prototype.concat.apply([], array)
}
```

This is all getting a bit unwieldy. Because the types don't tell you much (everything is a regular array), it's hard to understand what's going on in that code at a glance. To fix this, let's wrap what we're doing—putting a value in a container, exposing a way to operate on that value, and exposing a way to get a result back from the container—in a special data type that helps document our approach. Once we're done implementing it, you'll be able to use the data type like this:

```
ask()
  .flatMap(parse)
  .flatMap(date => new Some(date.toISOString()))
  .flatMap(date => new Some('Date is ' + date))
  .getOrElse('Error parsing date for some reason')
```

We'll define our `Option` type like this:

- `Option` is an interface that's implemented by two classes: `Some<T>` and `None` (see Figure 7-1). They are the two kinds of `Option` s. `Some<T>` is an `Option` that contains a value of type `T` , and `None` is an `Option` without a value, which represents a failure.

- `Option` is both a type and a function. Its type is an interface that simply serves as the supertype of `Some` and `None`. Its function is the way to create a new value of type `Option`.

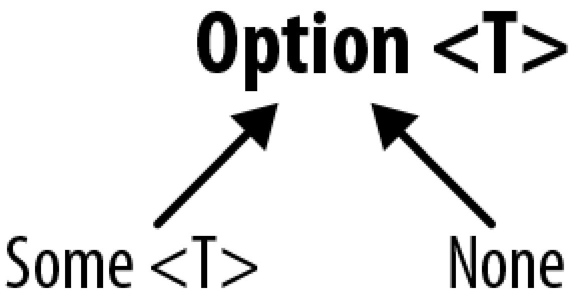

Figure 7-1. Option<T> has two cases: Some<T> and None

Let’s start by sketching out the types:

```
interface Option<T> {}                    ❶

class Some<T> implements Option<T> {      ❷

  constructor(private value: T) {}
}
class None implements Option<never> {}    ❸
```

❶ `Option<T>` is an interface that we’ll share between `Some<T>` and `None`.

❷ `Some<T>` represents a successful operation that resulted in a value. Like the array we used before, `Some<T>` is a container for that value.

❸ `None` represents an operation that failed, and does not contain a value.

These types are equivalent to the following in our array-based `Option` implementation:

- `Option<T>` is `[T] | []`.
- `Some<T>` is `[T]`.
- `None` is `[]`.

What can you do with an `Option`? For our bare-bones implementation, we’ll define just two operations:

`flatMap`

A way to chain operations on a possibly empty `Option`

`getOrElse`

A way to retrieve a value from an `Option`

We’ll start by defining these operations on our `Option` interface, meaning that `Some<T>` and `None` will need to provide concrete implementations for them:

```
interface Option<T> {
  flatMap<U>(f: (value: T) => Option<U>): Option<U>
  getOrElse(value: T): T
}
class Some<T> extends Option<T> {
  constructor(private value: T) {}
}
class None extends Option<never> {}
```

That is:

- `flatMap` takes a function `f` that takes a value of type `T` (the type of the value the `Option` contains) and returns an `Option` of `U`. `flatMap` calls `f` with the `Option`’s value, and returns a new `Option<U>`.
- `getOrElse` takes a default value of the same type `T` as the value that the `Option` contains, and returns either that default value (if the `Option` is an empty `None`) or the `Option`’s value (if the `Option` is a `Some<T>`).

Guided by the types, let’s implement these methods for `Some<T>` and `None`:

```
interface Option<T> {
  flatMap<U>(f: (value: T) => Option<U>): Option<U>
  getOrElse(value: T): T
}
class Some<T> implements Option<T> {
  constructor(private value: T) {}
  flatMap<U>(f: (value: T) => Option<U>): Option<U> {
                                         ❶

    return f(this.value)
  }
  getOrElse(): T {
                              ❷

    return this.value
  }
}
class None implements Option<never> {
  flatMap<U>(): Option<U> {
                              ❸

    return this
  }
  getOrElse<U>(value: U): U {
                              ❹

    return value
  }
}
```

When we call `flatMap` on a `Some<T>`, we pass in a function `f`, which `flatMap` calls with the `Some<T>`'s value to yield a new `Option` of a new type.

Calling `getOrElse` on a `Some<T>` just returns the `Some<T>`'s value.

❸ Since a `None` represents a failed computation, calling `flatMap` on it always returns a `None`: once a computation fails, we can't recover from that failure (at least not with our particular `Option` implementation).

Calling `getOrElse` on a `None` always returns the value we passed into `getOrElse`.

We can actually go a step beyond this naive implementation, and specify our types better. If all you know is that you have an `Option` and a function from `T` to `Option<U>`, then an `Option<T>` always `flatMap`s to an `Option<U>`. But when you know you have a `Some<T>` or a `None`, you can be more specific.

Table 7-1 shows the result types we want when calling `flatMap` on the two types of `Option`s.

Table 7-1. Result of calling .flatMap(f) on Some<T> and None

	From `Some<T>`	**From** `None`
To `Some<U>`	`Some<U>`	`None`
To `None`	`None`	`None`

That is, we know that mapping over a `None` always results in a `None`, and mapping over a `Some<T>` results in either a `Some<T>` or a `None`, depending on what calling `f` returns. We'll exploit this and use overloaded signatures to give `flatMap` more specific types:

```
interface Option<T> {
  flatMap<U>(f: (value: T) => None): None
  flatMap<U>(f: (value: T) => Option<U>): Option<U>
  getOrElse(value: T): T
}
class Some<T> implements Option<T> {
  constructor(private value: T) {}
  flatMap<U>(f: (value: T) => None): None
  flatMap<U>(f: (value: T) => Some<U>): Some<U>
  flatMap<U>(f: (value: T) => Option<U>): Option<U> {
    return f(this.value)
  }
  getOrElse(): T {
    return this.value
  }
}
class None implements Option<never> {
  flatMap(): None {
    return this
  }
  getOrElse<U>(value: U): U {
    return value
  }
}
```

We're almost done. All that's left to do is implement the `Option` function, which we'll use to create new `Option` s. We already implemented the `Option` *type* as an interface; now we're going to implement a function with the same name (remember that TypeScript has two separate namespaces for types and for values) as a way to create a new `Option`, similar to what we did in "Companion Object Pattern". If a user passes in `null` or `undefined`, we'll give them back a `None`; otherwise, we'll return a `Some`. Once again, we'll overload the signature to do that:

```
function Option<T>(value: null | undefined): None   ❶
function Option<T>(value: T): Some<T>   ❷
function Option<T>(value: T): Option<T> {   ❸
  if (value == null) {
    return new None
  }
  return new Some(value)
}
```

❶ If the consumer calls `Option` with `null` or `undefined`, we return a `None`.

❷ Otherwise, we return a `Some<T>`, where `T` is the type of value the user passed in.

❸ Finally, we manually calculate an upper bound for the two overloaded signatures. The upper bound of `null | undefined` and `T` is `T | null | undefined`, which simplifies to `T`. The upper bound of `None` and `Some<T>` is `None | Some<T>`, which we already have a name for: `Option<T>`.

That's it. We've derived a fully working, minimal `Option` type that lets us safely perform operations over maybe-`null` values. We use it like this:

```
let result = Option(6)              // Some<number>
  .flatMap(n => Option(n * 3))      // Some<number>
  .flatMap(n => new None)           // None
  .getOrElse(7)                     // 7
```

Getting back to our birthday prompt example, our code now works as we'd expect:

```
ask()                                                // (
  .flatMap(parse)                                    // (
  .flatMap(date => new Some(date.toISOString()))     // (
  .flatMap(date => new Some('Date is ' + date))      // (
  .getOrElse('Error parsing date for some reason') // s
```

Option s are a powerful way to work with series of operations that may or may not succeed. They give you excellent type safety, and signal to consumers via the type system that a given operation might fail.

However, Option s aren't without their downsides. They signal failure with a None , so you don't get more details about what failed and why. They also don't interoperate with code that doesn't use Option s (you'll have to explicitly wrap those APIs to return Option s).

Still, what you did there was pretty neat. The overloads you added let you do something that you can't express in most languages, even those that rely on the Option type for working with nullable values. By restricting Option to just Some or None where possible via overloaded call signatures, you made your code a whole lot safer, and a whole lot of Haskell programmers very jealous. Now go grab yourself a cold one—you deserve it.

Summary

In this chapter we covered the different ways to signal and recover from errors in TypeScript: returning null , throwing exceptions, returning exceptions, and the Option type. You now have an arsenal of approaches for safely working with things that might fail. Which approach you choose is up to you, and depends on:

- Whether you want to simply signal that something failed (null , Option), or give more information about why it failed (throwing and returning exceptions).
- Whether you want to force consumers to explicitly handle every possible exception (returning exceptions), or write less error-handling boilerplate (throwing exceptions).
- Whether you need a way to compose errors (Option), or simply handle them when they come up (null , exceptions).

Exercises

1. Design a way to handle errors for the following API, using one of the patterns from this chapter. In this API, every operation might fail—feel free to update the API's method signatures to allow for failures (or don't, if you prefer). Think about how you might perform a sequence of actions while handling errors that come up (e.g., getting the logged-in user's ID, then getting their list of friends, then getting each friend's name):

```
class API {
  getLoggedInUserID(): UserID
  getFriendIDs(userID: UserID): UserID[]
  getUserName(userID: UserID): string
}
```

1 If you haven't worked with Java before, a `throws` clause indicates which types of runtime exceptions a method might throw, so a consumer has to handle those exceptions.

2 Also called the `Maybe` type.

3 Google "try type" or "either type" for more information on those types.