8

BUGS AND ERRORS

Flaws in computer programs are usually called *bugs*. It makes programmers feel good to imagine them as little things that just happen to crawl into our work. In reality, of course, we put them there ourselves.

If a program is crystallized thought, we can roughly categorize bugs into those caused by the thoughts being confused and those caused by mistakes introduced while converting a thought to code. The former type is generally harder to diagnose and fix than the latter.

Language

Many mistakes could be pointed out to us automatically by the computer if it knew enough about what we're trying to do. But here, JavaScript's looseness is a hindrance. Its concept of bindings and properties is vague enough that it will rarely catch typos before actually running the program. Even then, it allows you to do some clearly nonsensical things without complaint, such as computing `true * "monkey"`.

There are some things that JavaScript does complain about. Writing a program that does not follow the language's grammar will immediately make the computer complain. Other things, such as calling something that's not a function or looking up a property on an undefined value, will cause an error to be reported when the program tries to perform the action.

Often, however, your nonsense computation will merely produce `NaN` (not a number) or an undefined value, while the program happily continues, convinced that it's doing something meaningful. The mistake will manifest itself only later, after the bogus value has traveled through several functions. It might not trigger an error at all, but silently cause the program's output to be wrong. Finding the source of such problems can be difficult.

The process of finding mistakes—bugs—in programs is called *debugging*.

Strict Mode

JavaScript can be made a *little* stricter by enabling *strict mode*. This can be done by putting the string "`use strict`" at the top of a file or a function body. Here's an example:

```
function canYouSpotTheProblem() {
  "use strict";
  for (counter = 0; counter < 10; counter++) {
    console.log("Happy happy");
  }
}

canYouSpotTheProblem();
// → ReferenceError: counter is not defined
```

Code inside classes and modules (which we will discuss in Chapter 10) is automatically strict. The old nonstrict behavior still exists only because some old code might depend on it, and the language designers work hard to avoid breaking any existing programs.

Normally, when you forget to put `let` in front of your binding, as with `counter` in the example, JavaScript quietly creates a global binding and uses that. In strict mode, an error is reported instead. This is very helpful. It should be noted, though, that this doesn't work when the binding in question already exists somewhere in scope. In that case, the loop will still quietly overwrite the value of the binding.

Another change in strict mode is that the `this` binding holds the value `undefined` in functions that are not called as methods. When making such a call outside of strict mode, `this` refers to the global scope object, which is an object whose properties are the global bindings. So if you accidentally call a method or constructor incorrectly in strict mode, JavaScript will produce an error as soon as it tries to read something from `this`, rather than happily writing to the global scope.

For example, consider the following code, which calls a constructor function without the `new` keyword so that its `this` will *not* refer to a newly constructed

object:

```
function Person(name) { this.name = name; }
let ferdinand = Person("Ferdinand"); // Oops
console.log(name);
// → Ferdinand
```

The bogus call to `Person` succeeded, but returned an undefined value and created the global binding `name`. In strict mode, the result is different.

```
"use strict";
function Person(name) { this.name = name; }
let ferdinand = Person("Ferdinand"); // Forgot new
// → TypeError: Cannot set property 'name' of undefinec
```

We are immediately told that something is wrong. This is helpful. Fortunately, constructors created with the `class` notation will always complain if they are called without `new`, making this less of a problem even in nonstrict mode.

Strict mode does a few more things. It disallows giving a function multiple parameters with the same name and removes certain problematic language features entirely (such as the `with` statement, which is so wrong it is not further discussed in this book).

In short, putting "`use strict`" at the top of your program rarely hurts and might help you spot a problem.

Types

Some languages want to know the types of all your bindings and expressions before even running a program. They will tell you right away when a type is used in an inconsistent way. JavaScript considers types only when actually running the program, and even there often tries to implicitly convert values to the type it expects, so it's not much help.

Still, types provide a useful framework for talking about programs. A lot of mistakes come from being confused about the kind of value that goes into or

comes out of a function. If you have that information written down, you're less likely to get confused.

You could add a comment like the following before the `findRoute` function from the previous chapter to describe its type:

```
// (graph: Object, from: string, to: string) => string[
function findRoute(graph, from, to) {
  // ...
}
```

There are a number of different conventions for annotating JavaScript programs with types.

A thing about types is that they need to introduce their own complexity to be able to describe enough code to be useful. What do you think would be the type of the `randomPick` function that returns a random element from an array? You'd need to introduce a *type variable*, *T*, which can stand in for any type, so that you can give `randomPick` a type like `(T[])` → `T` (function from an array of *T*s to a *T*).

When the types of a program are known, it is possible for the computer to *check* them for you, pointing out mistakes before the program is run. There are several JavaScript dialects that add types to the language and check them. The most popular one is called TypeScript. If you are interested in adding more rigor to your programs, I recommend you give it a try.

In this book, we will continue using raw, dangerous, untyped Java-Script code.

Testing

If the language is not going to do much to help us find mistakes, we'll have to find them the hard way: by running the program and seeing whether it does the right thing.

Doing this by hand, again and again, is a really bad idea. Not only is it annoying but it also tends to be ineffective, since it takes too much time to exhaustively test everything every time you make a change.

Computers are good at repetitive tasks, and testing is the ideal repetitive task. Automated testing is the process of writing a program that tests another program. Writing tests is a bit more work than testing manually, but once you've done it, you gain a kind of superpower: it takes you only a few seconds to verify that your program still behaves properly in all the situations you wrote tests for. When you break something, you'll immediately notice rather than randomly running into it at some later time.

Tests usually take the form of little labeled programs that verify some aspect of your code. For example, a set of tests for the (standard, probably already tested by someone else) `toUpperCase` method might look like this:

```
function test(label, body) {
  if (!body()) console.log(`Failed: ${label}`);
}

test("convert Latin text to uppercase", () => {
  return "hello".toUpperCase() == "HELLO";
});
test("convert Greek text to uppercase", () => {
  return "Χαίρετε".toUpperCase() == "ΧΑΙΡΕΤΕ";
});
test("don't convert case-less characters", () => {
  return "مرحبا".toUpperCase() == "مرحبا";
});
```

Writing tests like this tends to produce rather repetitive, awkward code. Fortunately, there exist pieces of software that help you build and run collections of tests (*test suites*) by providing a language (in the form of functions and methods) suited to expressing tests and by outputting informative information when a test fails. These are usually called *test runners*.

Some code is easier to test than other code. Generally, the more external objects that the code interacts with, the harder it is to set up the context in which to test it. The style of programming shown in the previous chapter, which uses self-contained persistent values rather than changing objects, tends to be easy to test.

Debugging

Once you notice there is something wrong with your program because it misbehaves or produces errors, the next step is to figure out *what* the problem is.

Sometimes it is obvious. The error message will point at a specific line of your program, and if you look at the error description and that line of code, you can often see the problem.

But not always. Sometimes the line that triggered the problem is simply the first place where a flaky value produced elsewhere gets used in an invalid way. If you have been solving the exercises in earlier chapters, you will probably have already experienced such situations.

The following example program tries to convert a whole number to a string in a given base (decimal, binary, and so on) by repeatedly picking out the last digit and then dividing the number to get rid of this digit. But the strange output that it currently produces suggests that it has a bug.

```
function numberToString(n, base = 10) {
  let result = "", sign = "";
  if (n < 0) {
    sign = "-";
    n = -n;
  }
  do {
    result = String(n % base) + result;
    n /= base;
  } while (n > 0);
  return sign + result;
}
console.log(numberToString(13, 10));
// → 1.5e-3231.3e-3221.3e-3211.3e-3201.3e-3191.3e-3181.
```

Even if you see the problem already, pretend for a moment that you don't. We know that our program is malfunctioning, and we want to find out why.

This is where you must resist the urge to start making random changes to the code to see whether that makes it better. Instead, *think*. Analyze what is

happening and come up with a theory of why it might be happening. Then make additional observations to test this theory—or, if you don't yet have a theory, make additional observations to help you come up with one.

Putting a few strategic `console.log` calls into the program is a good way to get additional information about what the program is doing. In this case, we want `n` to take the values `13`, `1`, and then `0`. Let's write out its value at the start of the loop.

```
13
1.3
0.13
0.013
...
1.5e-323
```

Right. Dividing 13 by 10 does not produce a whole number. Instead of `n /= base`, what we actually want is `n = Math.floor(n / base)` so that the number is properly "shifted" to the right.

An alternative to using `console.log` to peek into the program's behavior is to use the *debugger* capabilities of your browser. Browsers come with the ability to set a *breakpoint* on a specific line of your code. When the execution of the program reaches a line with a breakpoint, it is paused, and you can inspect the values of bindings at that point. I won't go into details, as debuggers differ from browser to browser, but look in your browser's developer tools or search the web for instructions.

Another way to set a breakpoint is to include a `debugger` statement (consisting simply of that keyword) in your program. If the developer tools of your browser are active, the program will pause whenever it reaches such a statement.

Error Propagation

Not all problems can be prevented by the programmer, unfortunately. If your program communicates with the outside world in any way, it is possible to get malformed input, to become overloaded with work, or to have the network fail.

If you're programming only for yourself, you can afford to just ignore such problems until they occur. But if you build something that is going to be used by anybody else, you usually want the program to do better than just crash. Sometimes the right thing to do is take the bad input in stride and continue running. In other cases, it is better to report to the user what went wrong and then give up. In either situation the program has to actively do something in response to the problem.

Say you have a function `promptNumber` that asks the user for a number and returns it. What should it return if the user inputs "orange"?

One option is to make it return a special value. Common choices for such values are `null`, `undefined`, or `-1`.

```
function promptNumber(question) {
  let result = Number(prompt(question));
  if (Number.isNaN(result)) return null;
  else return result;
}

console.log(promptNumber("How many trees do you see?"))
```

Now any code that calls `promptNumber` must check whether an actual number was read and, failing that, must somehow recover—maybe by asking again or by filling in a default value. Or it could again return a special value to *its* caller to indicate that it failed to do what it was asked.

In many situations, mostly when errors are common and the caller should be explicitly taking them into account, returning a special value is a good way to indicate an error. It does, however, have its downsides. First, what if the function can already return every possible kind of value? In such a function, you'll have to do something like wrap the result in an object to be able to distinguish success from failure, the way the `next` method on the iterator interface does.

```
function lastElement(array) {
  if (array.length == 0) {
    return {failed: true};
  } else {
```

```
    return {value: array[array.length - 1]};
  }
}
```

The second issue with returning special values is that it can lead to awkward code. If a piece of code calls `promptNumber` 10 times, it has to check 10 times whether `null` was returned. If its response to finding `null` is to simply return `null` itself, callers of the function will in turn have to check for it, and so on.

Exceptions

When a function cannot proceed normally, what we would often *like* to do is just stop what we are doing and immediately jump to a place that knows how to handle the problem. This is what *exception handling* does.

Exceptions are a mechanism that makes it possible for code that runs into a problem to *raise* (or *throw*) an exception. An exception can be any value. Raising one somewhat resembles a supercharged return from a function: it jumps out of not just the current function but also its callers, all the way down to the first call that started the current execution. This is called *unwinding the stack*. You may remember the stack of function calls mentioned in Chapter 3. An exception zooms down this stack, throwing away all the call contexts it encounters.

If exceptions always zoomed right down to the bottom of the stack, they would not be of much use. They'd just provide a novel way to blow up your program. Their power lies in the fact that you can set "obstacles" along the stack to *catch* the exception as it is zooming down. Once you've caught an exception, you can do something with it to address the problem and then continue to run the program.

Here's an example:

```
function promptDirection(question) {
  let result = prompt(question);
  if (result.toLowerCase() == "left") return "L";
  if (result.toLowerCase() == "right") return "R";
  throw new Error("Invalid direction: " + result);
}

function look() {
```

```
  if (promptDirection("Which way?") == "L") {
    return "a house";
  } else {
    return "two angry bears";
  }
}

try {
  console.log("You see", look());
} catch (error) {
  console.log("Something went wrong: " + error);
}
```

The `throw` keyword is used to raise an exception. Catching one is done by wrapping a piece of code in a `try` block, followed by the keyword `catch`. When the code in the `try` block causes an exception to be raised, the `catch` block is evaluated, with the name in parentheses bound to the exception value. After the `catch` block finishes—or if the `try` block finishes without problems—the program proceeds beneath the entire `try`/`catch` statement.

In this case, we used the `Error` constructor to create our exception value. This is a standard JavaScript constructor that creates an object with a `message` property. Instances of `Error` also gather information about the call stack that existed when the exception was created, a so-called *stack trace*. This information is stored in the `stack` property and can be helpful when trying to debug a problem: it tells us the function where the problem occurred and which functions made the failing call.

Note that the `look` function completely ignores the possibility that `promptDirection` might go wrong. This is the big advantage of exceptions: error-handling code is necessary only at the point where the error occurs and at the point where it is handled. The functions in between can forget all about it.

Well, almost . . .

Cleaning Up After Exceptions

The effect of an exception is another kind of control flow. Every action that might cause an exception, which is pretty much every function call and property access, might cause control to suddenly leave your code.

This means when code has several side effects, even if its “regular” control flow looks like they’ll always all happen, an exception might prevent some of them from taking place.

Here is some really bad banking code:

```
const accounts = {
  a: 100,
  b: 0,
  c: 20
};

function getAccount() {
  let accountName = prompt("Enter an account name");
  if (!Object.hasOwn(accounts, accountName)) {
    throw new Error(`No such account: ${accountName}`);
  }
  return accountName;
}

function transfer(from, amount) {
  if (accounts[from] < amount) return;
  accounts[from] -= amount;
  accounts[getAccount()] += amount;
}
```

The `transfer` function transfers a sum of money from a given account to another, asking for the name of the other account in the process. If given an invalid account name, `getAccount` throws an exception.

But `transfer` *first* removes the money from the account and *then* calls `getAccount` before it adds it to another account. If it is broken off by an exception at that point, it’ll just make the money disappear.

That code could have been written a little more intelligently, for example by calling `getAccount` before it starts moving money around. But often problems like this occur in more subtle ways. Even functions that don’t look like they will throw an exception might do so in exceptional circumstances or when they contain a programmer mistake.

One way to address this is to use fewer side effects. Again, a programming style that computes new values instead of changing existing data helps. If a piece of code stops running in the middle of creating a new value, no existing data structures were damaged, making it easier to recover.

Since that isn't always practical, `try` statements have another feature: they may be followed by a `finally` block either instead of or in addition to a `catch` block. A `finally` block says "no matter *what* happens, run this code after trying to run the code in the `try` block."

```
function transfer(from, amount) {
  if (accounts[from] < amount) return;
  let progress = 0;
  try {
    accounts[from] -= amount;
    progress = 1;
    accounts[getAccount()] += amount;
    progress = 2;
  } finally {
    if (progress == 1) {
      accounts[from] += amount;
    }
  }
}
```

This version of the function tracks its progress, and if, when leaving, it notices that it was aborted at a point where it had created an inconsistent program state, it repairs the damage it did.

Note that even though the `finally` code is run when an exception is thrown in the `try` block, it does not interfere with the exception. After the `finally` block runs, the stack continues unwinding.

Writing programs that operate reliably even when exceptions pop up in unexpected places is hard. Many people simply don't bother, and because exceptions are typically reserved for exceptional circumstances, the problem may occur so rarely that it is never even noticed. Whether that is a good thing or a really bad thing depends on how much damage the software will do when it fails.

Selective Catching

When an exception makes it all the way to the bottom of the stack without being caught, it gets handled by the environment. What this means differs between environments. In browsers, a description of the error typically gets written to the JavaScript console (reachable through the browser's Tools or Developer menu). Node.js, the browserless JavaScript environment we will discuss in Chapter 20, is more careful about data corruption. It aborts the whole process when an unhandled exception occurs.

For programmer mistakes, just letting the error go through is often the best you can do. An unhandled exception is a reasonable way to signal a broken program, and the JavaScript console will, on modern browsers, provide you with some information about which function calls were on the stack when the problem occurred.

For problems that are *expected* to happen during routine use, crashing with an unhandled exception is a terrible strategy.

Invalid uses of the language, such as referencing a nonexistent binding, looking up a property on `null`, or calling something that's not a function, will also result in exceptions being raised. Such exceptions can also be caught.

When a `catch` body is entered, all we know is that *something* in our `try` body caused an exception. But we don't know *what* did or *which* exception it caused.

JavaScript (in a rather glaring omission) doesn't provide direct support for selectively catching exceptions: either you catch them all or you don't catch any. This makes it tempting to *assume* that the exception you get is the one you were thinking about when you wrote the `catch` block.

But it might not be. Some other assumption might be violated, or you might have introduced a bug that is causing an exception. Here is an example that *attempts* to keep on calling `promptDirection` until it gets a valid answer:

```
for (;;) {
  try {
    let dir = promtDirection("Where?"); // ← Typo!
    console.log("You chose ", dir);
```

```
    break;
  } catch (e) {
    console.log("Not a valid direction. Try again.");
  }
}
```

The `for (;;)` construct is a way to intentionally create a loop that doesn't terminate on its own. We break out of the loop only when a valid direction is given. Unfortunately, we misspelled `promptDirection`, which will result in an "undefined variable" error. Because the `catch` block completely ignores its exception value (`e`), assuming it knows what the problem is, it wrongly treats the binding error as indicating bad input. Not only does this cause an infinite loop but it also "buries" the useful error message about the misspelled binding.

As a general rule, don't blanket-catch exceptions unless it is for the purpose of "routing" them somewhere—for example, over the network to tell another system that our program crashed. And even then, think carefully about how you might be hiding information.

We want to catch a *specific* kind of exception. We can do this by checking in the `catch` block whether the exception we got is the one we are interested in, and if not, rethrow it. But how do we recognize an exception?

We could compare its `message` property against the error message we happen to expect. But that's a shaky way to write code—we'd be using information that's intended for human consumption (the message) to make a programmatic decision. As soon as someone changes (or translates) the message, the code will stop working.

Rather, let's define a new type of error and use `instanceof` to identify it.

```
class InputError extends Error {}

function promptDirection(question) {
  let result = prompt(question);
  if (result.toLowerCase() == "left") return "L";
  if (result.toLowerCase() == "right") return "R";
  throw new InputError("Invalid direction: " + result);
}
```

The new error class extends `Error`. It doesn't define its own constructor, which means that it inherits the `Error` constructor, which expects a string message as its argument. In fact, it doesn't define anything at all—the class is empty. `InputError` objects behave like `Error` objects, except that they have a different class by which we can recognize them.

Now the loop can catch these more carefully.

```
for (;;) {
  try {
    let dir = promptDirection("Where?");
    console.log("You chose ", dir);
    break;
  } catch (e) {
    if (e instanceof InputError) {
      console.log("Not a valid direction. Try again.");
    } else {
      throw e;
    }
  }
}
```

This will catch only instances of `InputError` and let unrelated exceptions through. If you reintroduce the typo, the undefined binding error will be properly reported.

Assertions

Assertions are checks inside a program that verify that something is the way it is supposed to be. They are used not to handle situations that can come up in normal operation but to find programmer mistakes.

If, for example, `firstElement` is described as a function that should never be called on empty arrays, we might write it like this:

```
function firstElement(array) {
  if (array.length == 0) {
    throw new Error("firstElement called with []");
  }
```

```
  return array[0];
}
```

Now, instead of silently returning `undefined` (which you get when reading an array property that does not exist), this will loudly blow up your program as soon as you misuse it. This makes it less likely for such mistakes to go unnoticed and easier to find their cause when they occur.

I do not recommend trying to write assertions for every possible kind of bad input. That'd be a lot of work and would lead to very noisy code. You'll want to reserve them for mistakes that are easy to make (or that you find yourself making).

Summary

An important part of programming is finding, diagnosing, and fixing bugs. Problems can become easier to notice if you have an automated test suite or add assertions to your programs.

Problems caused by factors outside the program's control should usually be actively planned for. Sometimes, when the problem can be handled locally, special return values are a good way to track them. Otherwise, exceptions may be preferable.

Throwing an exception causes the call stack to be unwound until the next enclosing `try`/`catch` block or until the bottom of the stack. The exception value will be given to the `catch` block that catches it, which should verify that it is actually the expected kind of exception and then do something with it. To help address the unpredictable control flow caused by exceptions, `finally` blocks can be used to ensure that a piece of code *always* runs when a block finishes.

Exercises

Retry

Say you have a function `primitiveMultiply` that in 20 percent of cases multiplies two numbers and in the other 80 percent of cases raises an exception of type `MultiplicatorUnitFailure`. Write a function that wraps this clunky function and just keeps trying until a call succeeds, after which it returns the result.

Make sure you handle only the exceptions you are trying to handle.

The Locked Box

Consider the following (rather contrived) object:

```
const box = new class {
  locked = true;
  #content = [];

  unlock() { this.locked = false; }
  lock() { this.locked = true; }
  get content() {
    if (this.locked) throw new Error("Locked!");
    return this.#content;
  }
};
```

It is a box with a lock. There is an array in the box, but you can get at it only when the box is unlocked.

Write a function called `withBoxUnlocked` that takes a function value as its argument, unlocks the box, runs the function, and then ensures that the box is locked again before returning, regardless of whether the argument function returned normally or threw an exception.

For extra points, make sure that if you call `withBoxUnlocked` when the box is already unlocked, the box stays unlocked.