You Don't Know JS: ES6 & Beyond

Chapter 7: Meta Programming

Meta programming is programming where the operation targets the behavior of the program itself. In other words, it's programming the programming of your program. Yeah, a mouthful, huh?

For example, if you probe the relationship between one object `a` and another `b` -- are they `[[Prototype]]` linked? -- using `a.isPrototypeOf(b)`, this is commonly referred to as introspection, a form of meta programming. Macros (which don't exist in JS, yet) -- where the code modifies itself at compile time -- are another obvious example of meta programming. Enumerating the keys of an object with a `for..in` loop, or checking if an object is an *instance of* a "class constructor", are other common meta programming tasks.
Meta programming focuses on one or more of the following: code inspecting itself, code modifying itself, or code modifying default language behavior so other code is affected.

The goal of meta programming is to leverage the language's own intrinsic capabilities to make the rest of your code more descriptive, expressive, and/or flexible. Because of the *meta* nature of meta programming, it's somewhat difficult to put a more precise definition on it than that. The best way to understand meta programming is to see it through examples.

ES6 adds several new forms/features for meta programming on top of what JS already had.

Function Names

There are cases where your code may want to introspect on itself and ask what the name of some function is. If you ask what a function's name is, the answer is surprisingly somewhat ambiguous. Consider:

```
function daz() {
        // ..
}

var obj = {
        foo: function() {
                // ..
```

```
        },
        bar: function baz() {
                // ..
        },
        bam: daz,
        zim() {
                // ..
        }
};
```

In this previous snippet, "what is the name of `obj.foo()`" is slightly nuanced. Is it `"foo"`, `""`, or `undefined`? And what about `obj.bar()` -- is it named `"bar"` or `"baz"`? Is `obj.bam()` named `"bam"` or `"daz"`? What about `obj.zim()`?
Moreover, what about functions which are passed as callbacks, like:

```
function foo(cb) {
        // what is the name of `cb()` here?
}

foo( function(){
        // I'm anonymous!
} );
```

There are quite a few ways that functions can be expressed in programs, and it's not always clear and unambiguous what the "name" of that function should be.

More importantly, we need to distinguish whether the "name" of a function refers to its `name` property -- yes, functions have a property called `name` -- or whether it refers to the lexical binding name, such as `bar` in `function bar() { .. }`.
The lexical binding name is what you use for things like recursion:

```
function foo(i) {
        if (i < 10) return foo( i * 2 );
        return i;
}
```

The `name` property is what you'd use for meta programming purposes, so that's what we'll focus on in this discussion.
The confusion comes because by default, the lexical name a function has (if any) is also set as its `name` property. Actually there was no official requirement for that behavior by the ES5 (and prior) specifications. The setting of the `name` property was nonstandard but still fairly reliable. As of ES6, it has been standardized.
Tip: If a function has a `name` value assigned, that's typically the name used in stack traces in developer tools.

Inferences

But what happens to the `name` property if a function has no lexical name?

As of ES6, there are now inference rules which can determine a sensible `name` property value to assign a function even if that function doesn't have a lexical name to use. Consider:

```
var abc = function() {
        // ..
};

abc.name;                               // "abc"
```

Had we given the function a lexical name like `abc = function def() { .. }`, the `name` property would of course be `"def"`. But in the absence of the lexical name, intuitively the `"abc"` name seems appropriate.
Here are other forms that will infer a name (or not) in ES6:

```
(function(){ .. });                                     // name:
(function*(){ .. });                            // name:
window.foo = function(){ .. };          // name:

class Awesome {
        constructor() { .. }                    // name: Awesome
        funny() { .. }                                  // name: funny
}

var c = class Awesome { .. };           // name: Awesome

var o = {
        foo() { .. },                                   // name: foo
        *bar() { .. },                                  // name: bar
        baz: () => { .. },                              // name: baz
        bam: function(){ .. },                  // name: bam
        get qux() { .. },                               // name: get qux
        set fuz() { .. },                               // name: set fuz
        ["b" + "iz"]:
                function(){ .. },                       // name: biz
        [Symbol( "buz" )]:
                function(){ .. }                // name: [buz]
};

var x = o.foo.bind( o );                // name: bound foo
(function(){ .. }).bind( o );           // name: bound

export default function() { .. } // name: default

var y = new Function();                         // name: anonymous
var GeneratorFunction =
        function*(){}.__proto__.constructor;
var z = new GeneratorFunction(); // name: anonymous
```

The `name` property is not writable by default, but it is configurable, meaning you can use `Object.defineProperty(..)` to manually change it if so desired.

Meta Properties

In the "`new.target`" section of Chapter 3, we introduced a concept new to JS in ES6: the meta property. As the name suggests, meta properties are intended to provide special meta information in the form of a property access that would otherwise not have been possible.

In the case of `new.target`, the keyword `new` serves as the context for a property access. Clearly `new` is itself not an object, which makes this capability special. However, when `new.target` is used inside a constructor call (a function/method invoked with `new`), `new` becomes a virtual context, so that `new.target` can refer to the target constructor that `new` invoked.

This is a clear example of a meta programming operation, as the intent is to determine from inside a constructor call what the original `new` target was, generally for the purposes of introspection (examining typing/structure) or static property access.

For example, you may want to have different behavior in a constructor depending on if it's directly invoked or invoked via a child class:

```
class Parent {
	constructor() {
		if (new.target === Parent) {
			console.log( "Parent instantiated" );
		}
		else {
			console.log( "A child instantiated" );
		}
	}
}

class Child extends Parent {}

var a = new Parent();
// Parent instantiated

var b = new Child();
// A child instantiated
```

There's a slight nuance here, which is that the `constructor()` inside the `Parent` class definition is actually given the lexical name of the class (`Parent`), even though the syntax implies that the class is a separate entity from the constructor.

Warning: As with all meta programming techniques, be careful of creating code that's too clever for your future self or others maintaining your code to understand. Use these tricks with caution.

Well Known Symbols

In the "Symbols" section of Chapter 2, we covered the new ES6 primitive type `symbol`. In addition to symbols you can define in your own program, JS predefines a number of built-in symbols, referred to as *Well Known Symbols* (WKS).
These symbol values are defined primarily to expose special meta properties that are being exposed to your JS programs to give you more control over JS's behavior.

We'll briefly introduce each and discuss their purpose.

`Symbol.iterator`

In Chapters 2 and 3, we introduced and used the `@@iterator` symbol, automatically used by `...` spreads and `for..of` loops. We also saw `@@iterator` as defined on the new ES6 collections as defined in Chapter 5.
`Symbol.iterator` represents the special location (property) on any object where the language mechanisms automatically look to find a method that will construct an iterator instance for consuming that object's values. Many objects come with a default one defined.
However, we can define our own iterator logic for any object value by setting the `Symbol.iterator` property, even if that's overriding the default iterator. The meta programming aspect is that we are defining behavior which other parts of JS (namely, operators and looping constructs) use when processing an object value we define.
Consider:

```
var arr = [4,5,6,7,8,9];

for (var v of arr) {
        console.log( v );
}
// 4 5 6 7 8 9

// define iterator that only produces values
// from odd indexes
arr[Symbol.iterator] = function*() {
        var idx = 1;
        do {
                yield this[idx];
        } while ((idx += 2) < this.length);
};

for (var v of arr) {
        console.log( v );
}
// 5 7 9
```

`Symbol.toStringTag` and `Symbol.hasInstance`

One of the most common meta programming tasks is to introspect on a value to find out what *kind* it is, usually to decide what operations are appropriate to perform on it.

With objects, the two most common inspection techniques are `toString()` and `instanceof`.
Consider:

```
function Foo() {}

var a = new Foo();

a.toString();                           // [object Object]
a instanceof Foo;                       // true
```

As of ES6, you can control the behavior of these operations:

```
function Foo(greeting) {
        this.greeting = greeting;
}

Foo.prototype[Symbol.toStringTag] = "Foo";

Object.defineProperty( Foo, Symbol.hasInstance, {
        value: function(inst) {
                return inst.greeting == "hello";
        }
} );

var a = new Foo( "hello" ),
        b = new Foo( "world" );

b[Symbol.toStringTag] = "cool";

a.toString();                           // [object Foo]
String( b );                            // [object cool]

a instanceof Foo;                       // true
b instanceof Foo;                       // false
```

The `@@toStringTag` symbol on the prototype (or instance itself) specifies a string value to use in the `[object ___]` stringification.
The `@@hasInstance` symbol is a method on the constructor function which receives the instance object value and lets you decide by returning `true` or `false` if the value should be considered an instance or not.
Note: To set `@@hasInstance` on a function, you must use `Object.defineProperty(..)`, as the default one on `Function.prototype` is `writable: false`. See the *this & Object Prototypes* title of this series for more information.

`Symbol.species`

In "Classes" in Chapter 3, we introduced the `@@species` symbol, which controls which constructor is used by built-in methods of a class that needs to spawn new instances. The most common example is when subclassing `Array` and wanting to define which constructor (`Array(..)` or your subclass) inherited methods like `slice(..)` should use. By

default, `slice(..)` called on an instance of a subclass of `Array` would produce a new instance of that subclass, which is frankly what you'll likely often want.
However, you can meta program by overriding a class's default `@@species` definition:

```
class Cool {
        // defer `@@species` to derived constructor
        static get [Symbol.species]() { return this; }

        again() {
                return new this.constructor[Symbol.species]();
        }
}

class Fun extends Cool {}

class Awesome extends Cool {
        // force `@@species` to be parent constructor
        static get [Symbol.species]() { return Cool; }
}

var a = new Fun(),
        b = new Awesome(),
        c = a.again(),
        d = b.again();

c instanceof Fun;                       // true
d instanceof Awesome;           // false
d instanceof Cool;                      // true
```

The `Symbol.species` setting defaults on the built-in native constructors to the `return this` behavior as illustrated in the previous snippet in the `Cool` definition. It has no default on user classes, but as shown that behavior is easy to emulate.
If you need to define methods that generate new instances, use the meta programming of the `new this.constructor[Symbol.species](..)` pattern instead of the hard-wiring of `new this.constructor(..)` or `new XYZ(..)`. Derived classes will then be able to customize `Symbol.species` to control which constructor vends those instances.

`Symbol.toPrimitive`

In the *Types & Grammar* title of this series, we discussed the `ToPrimitive` abstract coercion operation, which is used when an object must be coerced to a primitive value for some operation (such as `==` comparison or `+` addition). Prior to ES6, there was no way to control this behavior.
As of ES6, the `@@toPrimitive` symbol as a property on any object value can customize that `ToPrimitive` coercion by specifying a method.
Consider:

```
var arr = [1,2,3,4,5];

arr + 10;                               // 1,2,3,4,510

arr[Symbol.toPrimitive] = function(hint) {
```

```
        if (hint == "default" || hint == "number") {
                // sum all numbers
                return this.reduce( function(acc,curr){
                        return acc + curr;
                }, 0 );
        }
};

arr + 10;                                   // 25
```

The `Symbol.toPrimitive` method will be provided with a *hint* of `"string"`, `"number"`, or `"default"` (which should be interpreted as `"number"`), depending on what type the operation invoking `ToPrimitive` is expecting. In the previous snippet, the additive `+` operation has no hint (`"default"` is passed). A multiplicative `*` operation would hint `"number"` and a `String(arr)` would hint `"string"`.

Warning: The `==` operator will invoke the `ToPrimitive` operation with no hint -- the `@@toPrimitive` method, if any is called with hint `"default"` -- on an object if the other value being compared is not an object. However, if both comparison values are objects, the behavior of `==` is identical to `===`, which is that the references themselves are directly compared. In this case, `@@toPrimitive` is not invoked at all. See the *Types & Grammar* title of this series for more information about coercion and the abstract operations.

Regular Expression Symbols

There are four well known symbols that can be overridden for regular expression objects, which control how those regular expressions are used by the four corresponding `String.prototype` functions of the same name:

- `@@match`: The `Symbol.match` value of a regular expression is the method used to match all or part of a string value with the given regular expression. It's used by `String.prototype.match(..)` if you pass it a regular expression for the pattern matching.

  The default algorithm for matching is laid out in section 21.2.5.6 of the ES6 specification (http://www.ecma-international.org/ecma-262/6.0/#sec-regexp.prototype-@@match). You could override this default algorithm and provide extra regex features, such as look-behind assertions.

  `Symbol.match` is also used by the `isRegExp` abstract operation (see the note in "String Inspection Functions" in Chapter 6) to determine if an object is intended to be used as a regular expression. To force this check to fail for an object so it's not treated as a regular expression, set the `Symbol.match` value to `false` (or something falsy).

- `@@replace`: The `Symbol.replace` value of a regular expression is the method used by `String.prototype.replace(..)` to replace within a string one or all occurrences of character sequences that match the given regular expression pattern.

  The default algorithm for replacing is laid out in section 21.2.5.8 of the ES6 specification (http://www.ecma-international.org/ecma-262/6.0/#sec-regexp.prototype-@@replace).

  One cool use for overriding the default algorithm is to provide additional `replacer` argument options, such as supporting `"abaca".replace(/a/g,[1,2,3])` producing `"1b2c3"` by consuming the iterable for successive replacement values.
- `@@search`: The `Symbol.search` value of a regular expression is the method used by `String.prototype.search(..)` to search for a sub-string within another string as matched by the given regular expression.

  The default algorithm for searching is laid out in section 21.2.5.9 of the ES6 specification (http://www.ecma-international.org/ecma-262/6.0/#sec-regexp.prototype-@@search).

- `@@split`: The `Symbol.split` value of a regular expression is the method used by `String.prototype.split(..)` to split a string into sub-strings at the location(s) of the delimiter as matched by the given regular expression.

  The default algorithm for splitting is laid out in section 21.2.5.11 of the ES6 specification (http://www.ecma-international.org/ecma-262/6.0/#sec-regexp.prototype-@@split).

Overriding the built-in regular expression algorithms is not for the faint of heart! JS ships with a highly optimized regular expression engine, so your own user code will likely be a lot slower. This kind of meta programming is neat and powerful, but it should only be used in cases where it's really necessary or beneficial.

`Symbol.isConcatSpreadable`

The `@@isConcatSpreadable` symbol can be defined as a boolean property (`Symbol.isConcatSpreadable`) on any object (like an array or other iterable) to indicate if it should be *spread out* if passed to an array `concat(..)`.
Consider:

```
var a = [1,2,3],
        b = [4,5,6];

b[Symbol.isConcatSpreadable] = false;
```

```
[].concat( a, b );                // [1,2,3,[4,5,6]]
```

`Symbol.unscopables`

The `@@unscopables` symbol can be defined as an object property (`Symbol.unscopables`) on any object to indicate which properties can and cannot be exposed as lexical variables in a `with` statement.

Consider:

```
var o = { a:1, b:2, c:3 },
        a = 10, b = 20, c = 30;

o[Symbol.unscopables] = {
        a: false,
        b: true,
        c: false
};

with (o) {
        console.log( a, b, c );          // 1 20 3
}
```

A `true` in the `@@unscopables` object indicates the property should be *unscopable*, and thus filtered out from the lexical scope variables. `false` means it's OK to be included in the lexical scope variables.

Warning: The `with` statement is disallowed entirely in `strict` mode, and as such should be considered deprecated from the language. Don't use it. See the *Scope & Closures* title of this series for more information. Because `with` should be avoided, the `@@unscopables` symbol is also moot.

Proxies

One of the most obviously meta programming features added to ES6 is the `Proxy` feature.

A proxy is a special kind of object you create that "wraps" -- or sits in front of -- another normal object. You can register special handlers (aka *traps*) on the proxy object which are called when various operations are performed against the proxy. These handlers have the opportunity to perform extra logic in addition to *forwarding* the operations on to the original target/wrapped object.

One example of the kind of *trap* handler you can define on a proxy is `get` that intercepts the `[[Get]]` operation -- performed when you try to access a property on an object.

Consider:

```
var obj = { a: 1 },
        handlers = {
                get(target,key,context) {
                        // note: target === obj,
```

```
                        // context === pobj
                        console.log( "accessing: ", key );
                        return Reflect.get(
                                target, key, context
                        );
                }
        },
        pobj = new Proxy( obj, handlers );

obj.a;
// 1

pobj.a;
// accessing: a
// 1
```

We declare a `get(..)` handler as a named method on the *handler* object (second argument to `Proxy(..)`), which receives a reference to the *target* object (`obj`), the *key* property name (`"a"`), and the `self`/receiver/proxy (`pobj`).
After the `console.log(..)` tracing statement, we "forward" the operation onto `obj` via `Reflect.get(..)`. We will cover the `Reflect` API in the next section, but note that each available proxy trap has a corresponding `Reflect` function of the same name. These mappings are symmetric on purpose. The proxy handlers each intercept when a respective meta programming task is performed, and the `Reflect` utilities each perform the respective meta programming task on an object. Each proxy handler has a default definition that automatically calls the corresponding `Reflect` utility. You will almost certainly use both `Proxy` and `Reflect` in tandem.
Here's a list of handlers you can define on a proxy for a *target* object/function, and how/when they are triggered:

- `get(..)`: via `[[Get]]`, a property is accessed on the proxy (`Reflect.get(..)`, `.` property operator, or `[ .. ]` property operator)
- `set(..)`: via `[[Set]]`, a property value is set on the proxy (`Reflect.set(..)`, the `=` assignment operator, or destructuring assignment if it targets an object property)
- `deleteProperty(..)`: via `[[Delete]]`, a property is deleted from the proxy (`Reflect.deleteProperty(..)` or `delete`)
- `apply(..)` (if *target* is a function): via `[[Call]]`, the proxy is invoked as a normal function/method (`Reflect.apply(..)`, `call(..)`, `apply(..)`, or the `(..)` call operator)
- `construct(..)` (if *target* is a constructor function): via `[[Construct]]`, the proxy is invoked as a constructor function (`Reflect.construct(..)` or `new`)
- `getOwnPropertyDescriptor(..)`: via `[[GetOwnProperty]]`, a property descriptor is retrieved from the proxy (`Object.getOwnPropertyDescriptor(..)` or `Reflect.getOwnPropertyDescriptor(..)`)

- `defineProperty(..)`: via `[[DefineOwnProperty]]`, a property descriptor is set on the proxy (`Object.defineProperty(..)` or `Reflect.defineProperty(..)`)
- `getPrototypeOf(..)`: via `[[GetPrototypeOf]]`, the `[[Prototype]]` of the proxy is retrieved (`Object.getPrototypeOf(..)`, `Reflect.getPrototypeOf(..)`, `__proto__`, `Object#isPrototypeOf(..)`, or `instanceof`)
- `setPrototypeOf(..)`: via `[[SetPrototypeOf]]`, the `[[Prototype]]` of the proxy is set (`Object.setPrototypeOf(..)`, `Reflect.setPrototypeOf(..)`, or `__proto__`)
- `preventExtensions(..)`: via `[[PreventExtensions]]`, the proxy is made non-extensible (`Object.preventExtensions(..)` or `Reflect.preventExtensions(..)`)
- `isExtensible(..)`: via `[[IsExtensible]]`, the extensibility of the proxy is probed (`Object.isExtensible(..)` or `Reflect.isExtensible(..)`)
- `ownKeys(..)`: via `[[OwnPropertyKeys]]`, the set of owned properties and/or owned symbol properties of the proxy is retrieved (`Object.keys(..)`, `Object.getOwnPropertyNames(..)`, `Object.getOwnSymbolProperties(..)`, `Reflect.ownKeys(..)`, or `JSON.stringify(..)`)
- `enumerate(..)`: via `[[Enumerate]]`, an iterator is requested for the proxy's enumerable owned and "inherited" properties (`Reflect.enumerate(..)` or `for..in`)
- `has(..)`: via `[[HasProperty]]`, the proxy is probed to see if it has an owned or "inherited" property (`Reflect.has(..)`, `Object#hasOwnProperty(..)`, or `"prop" in obj`)

Tip: For more information about each of these meta programming tasks, see the "`Reflect` API" section later in this chapter.

In addition to the notations in the preceding list about actions that will trigger the various traps, some traps are triggered indirectly by the default actions of another trap. For example:

```
var handlers = {
                getOwnPropertyDescriptor(target,prop) {
                        console.log(
                                "getOwnPropertyDescriptor"
                        );
                        return Object.getOwnPropertyDescriptor(
                                target, prop
                        );
                },
                defineProperty(target,prop,desc){
                        console.log( "defineProperty" );
                        return Object.defineProperty(
                                target, prop, desc
                        );
                }
        },
        proxy = new Proxy( {}, handlers );
```

```
proxy.a = 2;
// getOwnPropertyDescriptor
// defineProperty
```

The `getOwnPropertyDescriptor(..)` and `defineProperty(..)` handlers are triggered by the default `set(..)` handler's steps when setting a property value (whether newly adding or updating). If you also define your own `set(..)` handler, you may or may not make the corresponding calls against `context` (not `target`!) which would trigger these proxy traps.

Proxy Limitations

These meta programming handlers trap a wide array of fundamental operations you can perform against an object. However, there are some operations which are not (yet, at least) available to intercept.

For example, none of these operations are trapped and forwarded from `pobj` proxy to `obj` target:

```
var obj = { a:1, b:2 },
        handlers = { .. },
        pobj = new Proxy( obj, handlers );

typeof obj;
String( obj );
obj + "";
obj == pobj;
obj === pobj
```

Perhaps in the future, more of these underlying fundamental operations in the language will be interceptable, giving us even more power to extend JavaScript from within itself.

Warning: There are certain *invariants* -- behaviors which cannot be overridden -- that apply to the use of proxy handlers. For example, the result from the `isExtensible(..)` handler is always coerced to a `boolean`. These invariants restrict some of your ability to customize behaviors with proxies, but they do so only to prevent you from creating strange and unusual (or inconsistent) behavior. The conditions for these invariants are complicated so we won't fully go into them here, but this post (http://www.2ality.com/2014/12/es6-proxies.html#invariants) does a great job of covering them.

Revocable Proxies

A regular proxy always traps for the target object, and cannot be modified after creation -- as long as a reference is kept to the proxy, proxying remains possible. However, there

may be cases where you want to create a proxy that can be disabled when you want to stop allowing it to proxy. The solution is to create a *revocable proxy*:

```
var obj = { a: 1 },
        handlers = {
                get(target,key,context) {
                        // note: target === obj,
                        // context === pobj
                        console.log( "accessing: ", key );
                        return target[key];
                }
        },
        { proxy: pobj, revoke: prevoke } =
                Proxy.revocable( obj, handlers );

pobj.a;
// accessing: a
// 1

// later:
prevoke();

pobj.a;
// TypeError
```

A revocable proxy is created with `Proxy.revocable(..)`, which is a regular function, not a constructor like `Proxy(..)`. Otherwise, it takes the same two arguments: *target* and *handlers*.

The return value of `Proxy.revocable(..)` is not the proxy itself as with `new Proxy(..)`. Instead, it's an object with two properties: *proxy* and *revoke* -- we used object destructuring (see "Destructuring" in Chapter 2) to assign these properties to `pobj` and `prevoke()` variables, respectively.

Once a revocable proxy is revoked, any attempts to access it (trigger any of its traps) will throw a `TypeError`.

An example of using a revocable proxy might be handing out a proxy to another party in your application that manages data in your model, instead of giving them a reference to the real model object itself. If your model object changes or is replaced, you want to invalidate the proxy you handed out so the other party knows (via the errors!) to request an updated reference to the model.

Using Proxies

The meta programming benefits of these Proxy handlers should be obvious. We can almost fully intercept (and thus override) the behavior of objects, meaning we can extend object behavior beyond core JS in some very powerful ways. We'll look at a few example patterns to explore the possibilities.

Proxy First, Proxy Last

As we mentioned earlier, you typically think of a proxy as "wrapping" the target object. In that sense, the proxy becomes the primary object that the code interfaces with, and the actual target object remains hidden/protected.

You might do this because you want to pass the object somewhere that can't be fully "trusted," and so you need to enforce special rules around its access rather than passing the object itself.

Consider:

```
var messages = [],
        handlers = {
                get(target,key) {
                        // string value?
                        if (typeof target[key] == "string") {
                                // filter out punctuation
                                return target[key]
                                        .replace( /[^\w]/g, "" );
                        }

                        // pass everything else through
                        return target[key];
                },
                set(target,key,val) {
                        // only set unique strings, lowercased
                        if (typeof val == "string") {
                                val = val.toLowerCase();
                                if (target.indexOf( val ) == -1) {
                                        target.push(val);
                                }
                        }
                        return true;
                }
        },
        messages_proxy =
                new Proxy( messages, handlers );

// elsewhere:
messages_proxy.push(
        "heLLo...", 42, "wOrlD!!", "WoRld!!"
);

messages_proxy.forEach( function(val){
        console.log(val);
} );
// hello world

messages.forEach( function(val){
        console.log(val);
} );
```

```
// hello... world!!
```

I call this *proxy first* design, as we interact first (primarily, entirely) with the proxy.

We enforce some special rules on interacting with `messages_proxy` that aren't enforced for `messages` itself. We only add elements if the value is a string and is also unique; we also lowercase the value. When retrieving values from `messages_proxy`, we filter out any punctuation in the strings.
Alternatively, we can completely invert this pattern, where the target interacts with the proxy instead of the proxy interacting with the target. Thus, code really only interacts with the main object. The easiest way to accomplish this fallback is to have the proxy object in the `[[Prototype]]` chain of the main object.
Consider:

```
var handlers = {
                get(target,key,context) {
                        return function() {
                                context.speak(key + "!");
                        };
                }
        },
        catchall = new Proxy( {}, handlers ),
        greeter = {
                speak(who = "someone") {
                        console.log( "hello", who );
                }
        };

// setup `greeter` to fall back to `catchall`
Object.setPrototypeOf( greeter, catchall );

greeter.speak();                        // hello someone
greeter.speak( "world" );               // hello world

greeter.everyone();                             // hello everyone!
```

We interact directly with `greeter` instead of `catchall`. When we call `speak(..)`, it's found on `greeter` and used directly. But when we try to access a method like `everyone()`, that function doesn't exist on `greeter`.
The default object property behavior is to check up the `[[Prototype]]` chain (see the *this & Object Prototypes* title of this series), so `catchall` is consulted for an `everyone` property. The proxy `get()` handler then kicks in and returns a function that calls `speak(..)` with the name of the property being accessed (`"everyone"`).
I call this pattern *proxy last*, as the proxy is used only as a last resort.

"No Such Property/Method"

A common complaint about JS is that objects aren't by default very defensive in the situation where you try to access or set a property that doesn't already exist. You may wish to predefine all the properties/methods for an object, and have an error thrown if a nonexistent property name is subsequently used.

We can accomplish this with a proxy, either in *proxy first* or *proxy last* design. Let's consider both.

```
var obj = {
                a: 1,
                foo() {
                        console.log( "a:", this.a );
                }
        },
        handlers = {
                get(target,key,context) {
                        if (Reflect.has( target, key )) {
                                return Reflect.get(
                                        target, key, context
                                );
                        }
                        else {
                                throw "No such property/method!";
                        }
                },
                set(target,key,val,context) {
                        if (Reflect.has( target, key )) {
                                return Reflect.set(
                                        target, key, val, context
                                );
                        }
                        else {
                                throw "No such property/method!";
                        }
                }
        },
        pobj = new Proxy( obj, handlers );

pobj.a = 3;
pobj.foo();                     // a: 3

pobj.b = 4;                     // Error: No such property/method!
pobj.bar();                     // Error: No such property/method!
```

For both `get(..)` and `set(..)`, we only forward the operation if the target object's property already exists; error thrown otherwise. The proxy object (`pobj`) is the main object code should interact with, as it intercepts these actions to provide the protections.
Now, let's consider inverting with *proxy last* design:

```
var handlers = {
```

```
                get() {
                        throw "No such property/method!";
                },
                set() {
                        throw "No such property/method!";
                }
        },
        pobj = new Proxy( {}, handlers ),
        obj = {
                a: 1,
                foo() {
                        console.log( "a:", this.a );
                }
        };

// setup `obj` to fall back to `pobj`
Object.setPrototypeOf( obj, pobj );

obj.a = 3;
obj.foo();                      // a: 3

obj.b = 4;                      // Error: No such property/method!
obj.bar();                      // Error: No such property/method!
```

The *proxy last* design here is a fair bit simpler with respect to how the handlers are defined. Instead of needing to intercept the `[[Get]]` and `[[Set]]` operations and only forward them if the target property exists, we instead rely on the fact that if either `[[Get]]` or `[[Set]]` get to our `pobj` fallback, the action has already traversed the whole `[[Prototype]]` chain and not found a matching property. We are free at that point to unconditionally throw the error. Cool, huh?

Proxy Hacking the `[[Prototype]]` Chain

The `[[Get]]` operation is the primary channel by which the `[[Prototype]]` mechanism is invoked. When a property is not found on the immediate object, `[[Get]]` automatically hands off the operation to the `[[Prototype]]` object.

That means you can use the `get(..)` trap of a proxy to emulate or extend the notion of this `[[Prototype]]` mechanism.

The first hack we'll consider is creating two objects which are circularly linked via `[[Prototype]]` (or, at least it appears that way!). You cannot actually create a real circular `[[Prototype]]` chain, as the engine will throw an error. But a proxy can fake it! Consider:

```
var handlers = {
                get(target,key,context) {
                        if (Reflect.has( target, key )) {
                                return Reflect.get(
                                        target, key, context
                                );
                        }
                        // fake circular `[[Prototype]]`
```

```
                        else {
                                return Reflect.get(
                                        target[
                                                Symbol.for( "[[Prototype]]" )
                                        ],
                                        key,
                                        context
                                );
                        }
                }
        },
        obj1 = new Proxy(
                {
                        name: "obj-1",
                        foo() {
                                console.log( "foo:", this.name );
                        }
                },
                handlers
        ),
        obj2 = Object.assign(
                Object.create( obj1 ),
                {
                        name: "obj-2",
                        bar() {
                                console.log( "bar:", this.name );
                                this.foo();
                        }
                }
        );

// fake circular `[[Prototype]]` link
obj1[ Symbol.for( "[[Prototype]]" ) ] = obj2;

obj1.bar();
// bar: obj-1 <-- through proxy faking [[Prototype]]
// foo: obj-1 <-- `this` context still preserved

obj2.foo();
// foo: obj-2 <-- through [[Prototype]]
```

Note: We didn't need to proxy/forward `[[Set]]` in this example, so we kept things simpler. To be fully `[[Prototype]]` emulation compliant, you'd want to implement a `set(..)` handler that searches the `[[Prototype]]` chain for a matching property and respects its descriptor behavior (e.g., set, writable). See the *this & Object Prototypes* title of this series.

In the previous snippet, `obj2` is `[[Prototype]]` linked to `obj1` by virtue of the `Object.create(..)` statement. But to create the reverse (circular) linkage, we create property on `obj1` at the symbol location `Symbol.for("[[Prototype]]")` (see "Symbols" in Chapter 2). This symbol may look sort of special/magical, but it isn't. It just allows me a conveniently named hook that semantically appears related to the task I'm performing.

Then, the proxy's `get(..)` handler looks first to see if a requested `key` is on the proxy. If not, the operation is manually handed off to the object reference stored in the `Symbol.for("[[Prototype]]")` location of `target`.

One important advantage of this pattern is that the definitions of `obj1` and `obj2` are mostly not intruded by the setting up of this circular relationship between them.

Although the previous snippet has all the steps intertwined for brevity's sake, if you look closely, the proxy handler logic is entirely generic (doesn't know about `obj1` or `obj2` specifically). So, that logic could be pulled out into a simple helper that wires them up, like a `setCircularPrototypeOf(..)` for example. We'll leave that as an exercise for the reader.

Now that we've seen how we can use `get(..)` to emulate a `[[Prototype]]` link, let's push the hackery a bit further. Instead of a circular `[[Prototype]]`, what about multiple `[[Prototype]]` linkages (aka "multiple inheritance")? This turns out to be fairly straightforward:

```
var obj1 = {
                name: "obj-1",
                foo() {
                        console.log( "obj1.foo:", this.name );
                },
        },
        obj2 = {
                name: "obj-2",
                foo() {
                        console.log( "obj2.foo:", this.name );
                },
                bar() {
                        console.log( "obj2.bar:", this.name );
                }
        },
        handlers = {
                get(target,key,context) {
                        if (Reflect.has( target, key )) {
                                return Reflect.get(
                                        target, key, context
                                );
                        }
                        // fake multiple `[[Prototype]]`
                        else {
                                for (var P of target[
                                        Symbol.for( "[[Prototype]]" )
                                ]) {
                                        if (Reflect.has( P, key )) {
                                                return Reflect.get(
                                                        P, key, context
                                                );
                                        }
                                }
                        }
                }
        },
```

```
        obj3 = new Proxy(
                {
                        name: "obj-3",
                        baz() {
                                this.foo();
                                this.bar();
                        }
                },
                handlers
        );

// fake multiple `[[Prototype]]` links
obj3[ Symbol.for( "[[Prototype]]" ) ] = [
        obj1, obj2
];

obj3.baz();
// obj1.foo: obj-3
// obj2.bar: obj-3
```

Note: As mentioned in the note after the earlier circular `[[Prototype]]` example, we didn't implement the `set(..)` handler, but it would be necessary for a complete solution that emulates `[[Set]]` actions as normal `[[Prototype]]`s behave.
`obj3` is set up to multiple-delegate to both `obj1` and `obj2`. In `obj3.baz()`, the `this.foo()` call ends up pulling `foo()` from `obj1` (first-come, first-served, even though there's also a `foo()` on `obj2`). If we reordered the linkage as `obj2, obj1`, the `obj2.foo()` would have been found and used.
But as is, the `this.bar()` call doesn't find a `bar()` on `obj1`, so it falls over to check `obj2`, where it finds a match.
`obj1` and `obj2` represent two parallel `[[Prototype]]` chains of `obj3`. `obj1` and/or `obj2` could themselves have normal `[[Prototype]]` delegation to other objects, or either could themself be a proxy (like `obj3` is) that can multiple-delegate.
Just as with the circular `[[Prototype]]` example earlier, the definitions of `obj1`, `obj2`, and `obj3` are almost entirely separate from the generic proxy logic that handles the multiple-delegation. It would be trivial to define a utility like `setPrototypesOf(..)` (notice the "s"!) that takes a main object and a list of objects to fake the multiple `[[Prototype]]` linkage to. Again, we'll leave that as an exercise for the reader.
Hopefully the power of proxies is now becoming clearer after these various examples. There are many other powerful meta programming tasks that proxies enable.

`Reflect` API

The `Reflect` object is a plain object (like `Math`), not a function/constructor like the other built-in natives.

It holds static functions which correspond to various meta programming tasks that you can control. These functions correspond one-to-one with the handler methods (*traps*) that Proxies can define.

Some of the functions will look familiar as functions of the same names on `Object`:

- `Reflect.getOwnPropertyDescriptor(..)`
- `Reflect.defineProperty(..)`
- `Reflect.getPrototypeOf(..)`
- `Reflect.setPrototypeOf(..)`
- `Reflect.preventExtensions(..)`
- `Reflect.isExtensible(..)`

These utilities in general behave the same as their `Object.*` counterparts. However, one difference is that the `Object.*` counterparts attempt to coerce their first argument (the target object) to an object if it's not already one. The `Reflect.*` methods simply throw an error in that case.
An object's keys can be accessed/inspected using these utilities:

- `Reflect.ownKeys(..)`: Returns the list of all owned keys (not "inherited"), as returned by
  both `Object.getOwnPropertyNames(..)` and `Object.getOwnPropertySymbols(..)`. See the "Property Enumeration Order" section for information about the order of keys.
- `Reflect.enumerate(..)`: Returns an iterator that produces the set of all non-symbol keys (owned and "inherited") that are *enumerable* (see the *this & Object Prototypes* title of this series). Essentially, this set of keys is the same as those processed by a `for..in` loop. See the "Property Enumeration Order" section for information about the order of keys.
- `Reflect.has(..)`: Essentially the same as the `in` operator for checking if a property is on an object or its `[[Prototype]]` chain. For
  example, `Reflect.has(o,"foo")` essentially performs `"foo" in o`.

Function calls and constructor invocations can be performed manually, separate of the normal syntax (e.g., `(..)` and `new`) using these utilities:

- `Reflect.apply(..)`: For example, `Reflect.apply(foo,thisObj,[42,"bar"])` calls the `foo(..)` function with `thisObj` as its `this`, and passes in
  the `42` and `"bar"` arguments.
- `Reflect.construct(..)`: For example, `Reflect.construct(foo,[42,"bar"])` essentially calls `new foo(42,"bar")`.

Object property access, setting, and deletion can be performed manually using these utilities:

- `Reflect.get(..)`: For example, `Reflect.get(o,"foo")` retrieves `o.foo`.
- `Reflect.set(..)`: For example, `Reflect.set(o,"foo",42)` essentially performs `o.foo = 42`.
- `Reflect.deleteProperty(..)`: For example, `Reflect.deleteProperty(o,"foo")` essentially performs `delete o.foo`.

The meta programming capabilities of `Reflect` give you programmatic equivalents to emulate various syntactic features, exposing previously hidden-only abstract operations. For example, you can use these capabilities to extend features and APIs for *domain specific languages* (DSLs).

Property Ordering

Prior to ES6, the order used to list an object's keys/properties was implementation dependent and undefined by the specification. Generally, most engines have enumerated them in creation order, though developers have been strongly encouraged not to ever rely on this ordering.

As of ES6, the order for listing owned properties is now defined (ES6 specification, section 9.1.12) by the `[[OwnPropertyKeys]]` algorithm, which produces all owned properties (strings or symbols), regardless of enumerability. This ordering is only guaranteed for `Reflect.ownKeys(..)` (and by extension, `Object.getOwnPropertyNames(..)` and `Object.getOwnPropertySymbols(..)`). The ordering is:

1. First, enumerate any owned properties that are integer indexes, in ascending numeric order.
2. Next, enumerate the rest of the owned string property names in creation order.
3. Finally, enumerate owned symbol properties in creation order.

Consider:

```
var o = {};

o[Symbol("c")] = "yay";
o[2] = true;
o[1] = true;
o.b = "awesome";
o.a = "cool";
```

```
Reflect.ownKeys( o );                              // [1,2,"b","a",Symbol(c)]
Object.getOwnPropertyNames( o ); // [1,2,"b","a"]
Object.getOwnPropertySymbols( o );        // [Symbol(c)]
```

On the other hand, the `[[Enumerate]]` algorithm (ES6 specification, section 9.1.11) produces only enumerable properties, from the target object as well as its `[[Prototype]]` chain. It is used by both `Reflect.enumerate(..)` and `for..in`. The observable ordering is implementation dependent and not controlled by the specification.

By contrast, `Object.keys(..)` invokes the `[[OwnPropertyKeys]]` algorithm to get a list of all owned keys. However, it filters out non-enumerable properties and then reorders the list to match legacy implementation-dependent behavior, specifically with `JSON.stringify(..)` and `for..in`. So, by extension the ordering *also* matches that of `Reflect.enumerate(..)`.

In other words, all four mechanisms (`Reflect.enumerate(..)`, `Object.keys(..)`, `for..in`, and `JSON.stringify(..)`) will match with the same implementation-dependent ordering, though they technically get there in different ways.

Implementations are allowed to match these four to the ordering of `[[OwnPropertyKeys]]`, but are not required to. Nevertheless, you will likely observe the following ordering behavior from them:

```
var o = { a: 1, b: 2 };
var p = Object.create( o );
p.c = 3;
p.d = 4;

for (var prop of Reflect.enumerate( p )) {
        console.log( prop );
}
// c d a b

for (var prop in p) {
        console.log( prop );
}
// c d a b

JSON.stringify( p );
// {"c":3,"d":4}

Object.keys( p );
// ["c","d"]
```

Boiling this all down: as of ES6, `Reflect.ownKeys(..)`, `Object.getOwnPropertyNames(..)`, and `Object.getOwnPropertySymbols(..)` all have predictable and reliable ordering guaranteed by the specification. So it's safe to build code that relies on this ordering. `Reflect.enumerate(..)`, `Object.keys(..)`, and `for..in` (as well as `JSON.stringify(..)` by extension) continue to share an observable ordering with each other, as they always

have. But that ordering will not necessarily be the same as that of `Reflect.ownKeys(..)`. Care should still be taken in relying on their implementation-dependent ordering.

Feature Testing

What is a feature test? It's a test that you run to determine if a feature is available or not. Sometimes, the test is not just for existence, but for conformance to specified behavior -- features can exist but be buggy.

This is a meta programming technique, to test the environment your program runs in to then determine how your program should behave.

The most common use of feature tests in JS is checking for the existence of an API and if it's not present, defining a polyfill (see Chapter 1). For example:

```
if (!Number.isNaN) {
        Number.isNaN = function(x) {
                return x !== x;
        };
}
```

The `if` statement in this snippet is meta programming: we're probing our program and its runtime environment to determine if and how we should proceed.
But what about testing for features that involve new syntax?

You might try something like:

```
try {
        a = () => {};
        ARROW_FUNCS_ENABLED = true;
}
catch (err) {
        ARROW_FUNCS_ENABLED = false;
}
```

Unfortunately, this doesn't work, because our JS programs are compiled. Thus, the engine will choke on the `() => {}` syntax if it is not already supporting ES6 arrow functions. Having a syntax error in your program prevents it from running, which prevents your program from subsequently responding differently if the feature is supported or not.
To meta program with feature tests around syntax-related features, we need a way to insulate the test from the initial compile step our program runs through. For instance, if we could store the code for the test in a string, then the JS engine wouldn't by default try to compile the contents of that string, until we asked it to.

Did your mind just jump to using `eval(..)`?
Not so fast. See the *Scope & Closures* title of this series for why `eval(..)` is a bad idea.
But there's another option with less downsides: the `Function(..)` constructor.
Consider:

```
try {
        new Function( "( () => {} )" );
        ARROW_FUNCS_ENABLED = true;
}
catch (err) {
        ARROW_FUNCS_ENABLED = false;
}
```

OK, so now we're meta programming by determining if a feature like arrow functions *can* compile in the current engine or not. You might then wonder, what would we do with this information?

With existence checks for APIs, and defining fallback API polyfills, there's a clear path for what to do with either test success or failure. But what can we do with the information that we get from `ARROW_FUNCS_ENABLED` being `true` or `false`?
Because the syntax can't appear in a file if the engine doesn't support that feature, you can't just have different functions defined in the file with and without the syntax in question.

What you can do is use the test to determine which of a set of JS files you should load. For example, if you had a set of these feature tests in a bootstrapper for your JS application, it could then test the environment to determine if your ES6 code can be loaded and run directly, or if you need to load a transpiled version of your code (see Chapter 1).

This technique is called *split delivery*.

It recognizes the reality that your ES6 authored JS programs will sometimes be able to entirely run "natively" in ES6+ browsers, but other times need transpilation to run in pre-ES6 browsers. If you always load and use the transpiled code, even in the new ES6-compliant environments, you're running suboptimal code at least some of the time. This is not ideal.

Split delivery is more complicated and sophisticated, but it represents a more mature and robust approach to bridging the gap between the code you write and the feature support in browsers your programs must run in.

FeatureTests.io

Defining feature tests for all of the ES6+ syntax, as well as the semantic behaviors, is a daunting task you probably don't want to tackle yourself. Because these tests require dynamic compilation (`new Function(..)`), there's some unfortunate performance cost. Moreover, running these tests every single time your app runs is probably wasteful, as on average a user's browser only updates once in a several week period at most, and even then, new features aren't necessarily showing up with every update.

Finally, managing the list of feature tests that apply to your specific code base -- rarely will your programs use the entirety of ES6 -- is unruly and error-prone.

The "https://featuretests.io" feature-tests-as-a-service offers solutions to these frustrations.

You can load the service's library into your page, and it loads the latest test definitions and runs all the feature tests. It does so using background processing with Web Workers, if possible, to reduce the performance overhead. It also uses LocalStorage persistence to cache the results in a way that can be shared across all sites you visit which use the service, which drastically reduces how often the tests need to run on each browser instance.

You get runtime feature tests in each of your users' browsers, and you can use those tests results dynamically to serve users the most appropriate code (no more, no less) for their environments.

Moreover, the service provides tools and APIs to scan your files to determine what features you need, so you can fully automate your split delivery build processes.

FeatureTests.io makes it practical to use feature tests for all parts of ES6 and beyond to make sure that only the best code is ever loaded and run for any given environment.

Tail Call Optimization (TCO)

Normally, when a function call is made from inside another function, a second *stack frame* is allocated to separately manage the variables/state of that other function invocation. Not only does this allocation cost some processing time, but it also takes up some extra memory.

A call stack chain typically has at most 10-15 jumps from one function to another and another. In those scenarios, the memory usage is not likely any kind of practical problem.

However, when you consider recursive programming (a function calling itself repeatedly) -- or mutual recursion with two or more functions calling each other -- the call stack could easily be hundreds, thousands, or more levels deep. You can probably see the problems that could cause, if memory usage grows unbounded.

JavaScript engines have to set an arbitrary limit to prevent such programming techniques from crashing by running the browser and device out of memory. That's why we get the frustrating "RangeError: Maximum call stack size exceeded" thrown if the limit is hit.

Warning: The limit of call stack depth is not controlled by the specification. It's implementation dependent, and will vary between browsers and devices. You should never code with strong assumptions of exact observable limits, as they may very well change from release to release.

Certain patterns of function calls, called *tail calls*, can be optimized in a way to avoid the extra allocation of stack frames. If the extra allocation can be avoided, there's no reason to arbitrarily limit the call stack depth, so the engines can let them run unbounded.

A tail call is a `return` statement with a function call, where nothing has to happen after the call except returning its value.
This optimization can only be applied in `strict` mode. Yet another reason to always be writing all your code as `strict`!
Here's a function call that is *not* in tail position:

```
"use strict";

function foo(x) {
        return x * 2;
}

function bar(x) {
        // not a tail call
        return 1 + foo( x );
}

bar( 10 );                              // 21
```

`1 + ..` has to be performed after the `foo(x)` call completes, so the state of that `bar(..)` invocation needs to be preserved.
But the following snippet demonstrates calls to `foo(..)` and `bar(..)` where both *are* in tail position, as they're the last thing to happen in their code path (other than the `return`):

```
"use strict";

function foo(x) {
```

```
        return x * 2;
}

function bar(x) {
        x = x + 1;
        if (x > 10) {
                return foo( x );
        }
        else {
                return bar( x + 1 );
        }
}

bar( 5 );                               // 24
bar( 15 );                              // 32
```

In this program, `bar(..)` is clearly recursive, but `foo(..)` is just a regular function call. In both cases, the function calls are in *proper tail position*. The `x + 1` is evaluated before the `bar(..)` call, and whenever that call finishes, all that happens is the `return`.
Proper Tail Calls (PTC) of these forms can be optimized -- called tail call optimization (TCO) -- so that the extra stack frame allocation is unnecessary. Instead of creating a new stack frame for the next function call, the engine just reuses the existing stack frame. That works because a function doesn't need to preserve any of the current state, as nothing happens with that state after the PTC.

TCO means there's practically no limit to how deep the call stack can be. That trick slightly improves regular function calls in normal programs, but more importantly opens the door to using recursion for program expression even if the call stack could be tens of thousands of calls deep.

We're no longer relegated to simply theorizing about recursion for problem solving, but can actually use it in real JavaScript programs!

As of ES6, all PTC should be optimizable in this way, recursion or not.

Tail Call Rewrite

The hitch, however, is that only PTC can be optimized; non-PTC will still work of course, but will cause stack frame allocation as they always did. You'll have to be careful about structuring your functions with PTC if you expect the optimizations to kick in.

If you have a function that's not written with PTC, you may find the need to manually rearrange your code to be eligible for TCO.

Consider:

```
"use strict";

function foo(x) {
        if (x <= 1) return 1;
        return (x / 2) + foo( x - 1 );
}

foo( 123456 );                  // RangeError
```

The call to `foo(x-1)` isn't a PTC because its result has to be added to `(x / 2)` before `return`ing.
However, to make this code eligible for TCO in an ES6 engine, we can rewrite it as follows:

```
"use strict";

var foo = (function(){
        function _foo(acc,x) {
                if (x <= 1) return acc;
                return _foo( (x / 2) + acc, x - 1 );
        }

        return function(x) {
                return _foo( 1, x );
        };
})();

foo( 123456 );                  // 3810376848.5
```

If you run the previous snippet in an ES6 engine that implements TCO, you'll get the `3810376848.5` answer as shown. However, it'll still fail with a `RangeError` in non-TCO engines.

Non-TCO Optimizations

There are other techniques to rewrite the code so that the call stack isn't growing with each call.

One such technique is called *trampolining*, which amounts to having each partial result represented as a function that either returns another partial result function or the final result. Then you can simply loop until you stop getting a function, and you'll have the result. Consider:

```
"use strict";

function trampoline( res ) {
        while (typeof res == "function") {
                res = res();
        }
```

```
        return res;
}

var foo = (function(){
        function _foo(acc,x) {
                if (x <= 1) return acc;
                return function partial(){
                        return _foo( (x / 2) + acc, x - 1 );
                };
        }

        return function(x) {
                return trampoline( _foo( 1, x ) );
        };
})();

foo( 123456 );                  // 3810376848.5
```

This reworking required minimal changes to factor out the recursion into the loop in `trampoline(..)`:

1. First, we wrapped the `return _foo ..` line in the `return partial() { ..` function expression.
2. Then we wrapped the `_foo(1,x)` call in the `trampoline(..)` call.

The reason this technique doesn't suffer the call stack limitation is that each of those inner `partial(..)` functions is just returned back to the `while` loop in `trampoline(..)`, which runs it and then loop iterates again. In other words, `partial(..)` doesn't recursively call itself, it just returns another function. The stack depth remains constant, so it can run as long as it needs to.

Trampolining expressed in this way uses the closure that the inner `partial()` function has over the `x` and `acc` variables to keep the state from iteration to iteration. The advantage is that the looping logic is pulled out into a reusable `trampoline(..)` utility function, which many libraries provide versions of. You can reuse `trampoline(..)` multiple times in your program with different trampolined algorithms.

Of course, if you really wanted to deeply optimize (and the reusability wasn't a concern), you could discard the closure state and inline the state tracking of `acc` into just one function's scope along with a loop. This technique is generally called *recursion unrolling*:

```
"use strict";

function foo(x) {
        var acc = 1;
        while (x > 1) {
                acc = (x / 2) + acc;
                x = x - 1;
        }
        return acc;
```

```
}

foo( 123456 );                  // 3810376848.5
```

This expression of the algorithm is simpler to read, and will likely perform the best (strictly speaking) of the various forms we've explored. That may seem like a clear winner, and you may wonder why you would ever try the other approaches.

There are some reasons why you might not want to always manually unroll your recursions:

- Instead of factoring out the trampolining (loop) logic for reusability, we've inlined it. This works great when there's only one example to consider, but as soon as you have a half dozen or more of these in your program, there's a good chance you'll want some reusability to keep things shorter and more manageable.
- The example here is deliberately simple enough to illustrate the different forms. In practice, there are many more complications in recursion algorithms, such as mutual recursion (more than just one function calling itself).

  The farther you go down this rabbit hole, the more manual and intricate the *unrolling* optimizations are. You'll quickly lose all the perceived value of readability. The primary advantage of recursion, even in the PTC form, is that it preserves the algorithm readability, and offloads the performance optimization to the engine.

If you write your algorithms with PTC, the ES6 engine will apply TCO to let your code run in constant stack depth (by reusing stack frames). You get the readability of recursion with most of the performance benefits and no limitations of run length.

Meta?

What does TCO have to do with meta programming?

As we covered in the "Feature Testing" section earlier, you can determine at runtime what features an engine supports. This includes TCO, though determining it is quite brute force. Consider:

```
"use strict";

try {
        (function foo(x){
                if (x < 5E5) return foo( x + 1 );
        })( 1 );
```

```
        TCO_ENABLED = true;
}
catch (err) {
        TCO_ENABLED = false;
}
```

In a non-TCO engine, the recursive loop will fail out eventually, throwing an exception caught by the `try..catch`. Otherwise, the loop completes easily thanks to TCO.
Yuck, right?

But how could meta programming around the TCO feature (or rather, the lack thereof) benefit our code? The simple answer is that you could use such a feature test to decide to load a version of your application's code that uses recursion, or an alternative one that's been converted/transpiled to not need recursion.

Self-Adjusting Code

But here's another way of looking at the problem:

```
"use strict";

function foo(x) {
        function _foo() {
                if (x > 1) {
                        acc = acc + (x / 2);
                        x = x - 1;
                        return _foo();
                }
        }

        var acc = 1;

        while (x > 1) {
                try {
                        _foo();
                }
                catch (err) { }
        }

        return acc;
}

foo( 123456 );                  // 3810376848.5
```

This algorithm works by attempting to do as much of the work with recursion as possible, but keeping track of the progress via scoped variables `x` and `acc`. If the entire problem can be solved with recursion without an error, great. If the engine kills the recursion at some point, we simply catch that with the `try..catch` and then try again, picking up where we left off.

I consider this a form of meta programming in that you are probing during runtime the ability of the engine to fully (recursively) finish the task, and working around any (non-TCO) engine limitations that may restrict you.

At first (or even second!) glance, my bet is this code seems much uglier to you compared to some of the earlier versions. It also runs a fair bit slower (on larger runs in a non-TCO environment).

The primary advantage, other than it being able to complete any size task even in non-TCO engines, is that this "solution" to the recursion stack limitation is much more flexible than the trampolining or manual unrolling techniques shown previously.

Essentially, `_foo()` in this case is a sort of stand-in for practically any recursive task, even mutual recursion. The rest is the boilerplate that should work for just about any algorithm.
The only "catch" is that to be able to resume in the event of a recursion limit being hit, the state of the recursion must be in scoped variables that exist outside the recursive function(s). We did that by leaving `x` and `acc` outside of the `_foo()` function, instead of passing them as arguments to `_foo()` as earlier.
Almost any recursive algorithm can be adapted to work this way. That means it's the most widely applicable way of leveraging TCO with recursion in your programs, with minimal rewriting.

This approach still uses a PTC, meaning that this code will *progressively enhance* from running using the loop many times (recursion batches) in an older browser to fully leveraging TCO'd recursion in an ES6+ environment. I think that's pretty cool!

Review

Meta programming is when you turn the logic of your program to focus on itself (or its runtime environment), either to inspect its own structure or to modify it. The primary value of meta programming is to extend the normal mechanisms of the language to provide additional capabilities.

Prior to ES6, JavaScript already had quite a bit of meta programming capability, but ES6 significantly ramps that up with several new features.

From function name inferences for anonymous functions to meta properties that give you information about things like how a constructor was invoked, you can inspect the program structure while it runs more than ever before. Well Known Symbols let you override intrinsic behaviors, such as coercion of an object to a primitive value. Proxies

can intercept and customize various low-level operations on objects,
and `Reflect` provides utilities to emulate them.
Feature testing, even for subtle semantic behaviors like Tail Call Optimization, shifts the meta programming focus from your program to the JS engine capabilities itself. By knowing more about what the environment can do, your programs can adjust themselves to the best fit as they run.

Should you meta program? My advice is: first focus on learning how the core mechanics of the language really work. But once you fully know what JS itself can do, it's time to start leveraging these powerful meta programming capabilities to push the language further!