Symbols

Symbols have existed prior to ES6, but now we have a public interface to using them directly. Symbols are immutable and unique and can be used as keys in any hash.

Symbol( )

Calling `Symbol()` or `Symbol(description)` will create a unique symbol that cannot be looked up globally. A Use case for `Symbol()` is to patch objects or namespaces from third parties with your own logic, but be confident that you won't collide with updates to that library. For example, if you wanted to add a method `refreshComponent` to the `React.Component` class, and be certain that you didn't trample a method they add in a later update:

```
const refreshComponent = Symbol();

React.Component.prototype[refreshComponent] = () => {
    // do something
}
```

Symbol.for(key)

`Symbol.for(key)` will create a Symbol that is still immutable and unique, but can be looked up globally. Two identical calls to `Symbol.for(key)` will return the same Symbol instance. NOTE: This is not true for `Symbol(description)`:

```
Symbol('foo') === Symbol('foo') // false
Symbol.for('foo') === Symbol('foo') // false
Symbol.for('foo') === Symbol.for('foo') // true
```

A common use case for Symbols, and in particular with `Symbol.for(key)` is for interoperability. This can be achieved by having your code look for a Symbol member on object arguments from third parties that contain some known interface. For example:

```
function reader(obj) {
    const specialRead = Symbol.for('specialRead');
    if (obj[specialRead]) {
        const reader = obj[specialRead]();
        // do something with reader
    } else {
        throw new TypeError('object cannot be read');
    }
}
```

And then in another library:

```
const specialRead = Symbol.for('specialRead');

class SomeReadableType {
    [specialRead]() {
        const reader = createSomeReaderFrom(this);
        return reader;
    }
}
```

> A notable example of Symbol use for interoperability is `Symbol.iterator` which exists on all iterable types in ES6: Arrays, strings, generators, etc. When called as a method it returns an object with an Iterator interface.

Async Await

While this is actually an upcoming ES2016 feature, `async await` allows us to perform the same thing we accomplished using Generators and Promises with less effort:

```
var request = require('request');

function getJSON(url) {
  return new Promise(function(resolve, reject) {
    request(url, function(error, response, body) {
      resolve(body);
    });
  });
}

async function main() {
  var data = await getJSON();
  console.log(data); // NOT undefined!
}

main();
```

Under the hood, it performs similarly to Generators. I highly recommend using them over Generators + Promises. A great resource for getting up and running with ES7 and Babel can be found here.