Promises A+

The Promise Abstraction

Callbacks are low level

- They are the simplest abstraction that works.
- But they don’t
  - Encapsulate their contract
  - Enforce their contract
  - Provide any control-flow
  - Provide any error handling or protection from stack-tearing

Promises are encapsulate their contract

Return a promise instead of calling a passed callback:

```
readFile("file.txt", function (err, result) {
    // continue here...
});

// becomes

var promiseForResult = readFile("file.txt");
```

Promise Contract

```
promiseForResult.then(onFulfilled, onRejected);
```

- Either **onFulfilled** will be called OR **onRejected** (never BOTH)
- **onFulfilled** will recieve a single **fulfillment value** (⇔ return value).
- **onRejected** will be called with a single **error value** (⇔ thrown exception).
- Attach handlers at any time (before or after fulfillment)
- Handlers will always be called asynchronously.

Promises can be chained

```
var transformedPromise = originalPromise.then(onFulfilled, onRejected);
```

- **onFulfilled**'s return value will be used to resolve **transformedPromise**
  - **transformedPromise** is fulfilled with **onFulfilled**'s return value
  - If **onFulfilled**'s value is a promise, **transformedPromise** is fulfilled with **onFulfilled**'s promise's fulfilled value
- If **onFulfilled** or **onRejected** throws an exception, **transformedPromise** will be rejected with that exception.

The Sync ⇔ Async Parallel

```
try {
    var result = doSomethingSync();
    process(result);
} catch (ex) {
    handle(ex);
}
```

```
doSomethingAsync().then(
    result => process(result),
    handle
);
```

Case 1: Simple Functional Transform

```
var user = getUser();
var userName = user.name;

// becomes

var userNamePromise = getUser().then(function (user) {
    return user.name;
});
```

Case 2: Reacting with an Exception

```
var user = getUser();
if (user === null)
    throw new Error("null user!");

// becomes

var userPromise = getUser().then(function (user) {
    if (user === null)
        throw new Error("null user!");
    return user;
});
```

Case 3: Handling an Exception

```
try {
    updateUser(data);
} catch (ex) {
    console.log("There was an error:", ex.stack);
}

// becomes

var updatePromise = updateUser(data).then(undefined, ex => {
    console.log("There was an error:", ex.stack);
});
```

Case 4: Rethrowing an Exception

```
try {
    updateUser(data);
} catch (ex) {
    throw new Error("Updating user failed. Details: " + ex.stack);
}

// becomes

var updatePromise = updateUser(data).then(undefined, function (ex) {
    throw new Error("Updating user failed. Details: " + ex.stack);
});
```

Case 5: Async Chaining

```
var name = promptForNewUserName();
updateUser({ name: name });
refreshUI();

// becomes

promptForNewUserName()
    .then(name => {
        return updateUser({ name: name });
    })
    .then(refreshUI);
```

Promises Give You Back Exception Propagation

```
getUser("User", function (outerErr, user) {
    getBestFriend(user, function (innerErr, friend) {
        ui.showBestFriend(friend);
    });
});
```

Promises Give You Back Exception Propagation

```
getUser("User", function (err, user) {
    if (err) return handleError(err);

    getBestFriend(user, function (err, friend) {
        if (err) return handleError(err);

        showBestFriend(friend);
    });
});
```

Promises Give You Back Exception Propagation

```
getUser("User")
    .then(getBestFriend)
    .then(showBestFriend, handleError);
```

Promises as First-Class Objects

- Because promises are first-class objects, you can build simple operations on them instead of tying callbacks together:

```
// Fulfills with an array of results, or rejects if any reject
all([getUserData(), getCompanyData()]);

// Fulfills as soon as either completes, or rejects if both reject
any([storeDataOnServer1(), storeDataOnServer2()]);

// If writeFile accepts promises as arguments, and readFile returns one:
writeFile("dest.txt", readFile("source.txt"));
```

Promises in Your Code

Some practical guidance

First, Choose a Library

- My top picks:
  - Q, by Kris Kowal and myself: https://github.com/kriskowal/q
  - When.js, by Brian Cavalier: https://github.com/cujojs/when

If you ever see a jQuery promise, kill it with fire:

```
var realPromise = Q(jQueryPromise);
var realPromise = when(jQueryPromise);
```

Keep The Sync ⇔ Async Parallel In Mind

- Use promises for single operations that can result in fulfillment (⇔ returning a value) or rejection (⇔ throwing an exception).

- If you're ever stuck, ask "how would I structure this code if it were synchronous?"
  - Keep in mind, multiple parallel operations have no sync counterpart.

Promises Are *Not*

- A replacement for events
- A replacement for streams
- A way of doing functional reactive programming

They work together:

- An event can trigger from one part of your code, used to fulfill a promise, and cause the promise handlers to *trigger*
- A HTTP request function can return *a promise for a stream*

The Unhandled Rejection Pitfall

This hits the top of the stack:

```
throw new Error("boo!");
```

This stays inert:

```
var promise = doSomething().then(function () {
    throw new Error("boo!");
});
```

Avoiding the Unhandled Rejection Pitfall

- *Always* either:
  - **return** the promise to your caller;
  - or call **.done()** on it to signal that any unhandled rejections should explode

```
function getUserName() {
    return getUser().then(function (user) {
      return user.name;
    });
}

getUserName().then(function (userName) {
    console.log("User name: ", userName);
}).done();
```

Promise Patterns: try/catch/finally

```
ui.startSpinner();
getUser("Domenic")
    .then(getBestFriend)
    .then(ui.showBestFriend)
    .catch(ui.error)
    .finally(ui.stopSpinner)
    .done();
```

Promise Patterns: all + spread

```
Q.all([getUser(), getCompany()]).then(function (results) {
    console.log("user = ", results[0]);
    console.log("company = ", results[1]);
}).done();

Q.all([getUser(), getCompany()]).spread(function (user, company) {
    console.log("user = ", user);
    console.log("company = ", company);
}).done();
```

Promise Patterns: map + all

```
var userIds = ["123", "456", "789"];

Q.all(userIds.map(getUserById))
    .then(function (users) {
        console.log("all the users: ", users);
    })
    .done();
```

Promise Patterns: message sending

```
var userData = getUserData();

userData
    .then(createUserViewModel)
    .invoke("setStatus", "loaded")
    .done();

userData
    .get("friends")
    .get("0")
    .get("name")
    .then(setBestFriendsNameInUI)
    .done();
```

Promise Patterns: Denodeify

```
var readFile = Q.denodeify(fs.readFile);
var readDir = Q.denodeify(fs.readdir);

readDir("/tmp")
    .get("0")
    .then(readFile)
    .then(function (data) {
        console.log("The first temporary file contains: ", data);
    })
    .catch(function (error) {
        console.log("One of the steps failed: ", error);
    })
    .done();
```

Advanced Promise Magic

(Bonus round!)

Coroutines

"Coroutines are computer program components that generalize subroutines to allow multiple entry points for suspending and resuming execution at certain locations."

Generators = Shallow Coroutines

```
function* fibonacci() {
    var [prev, curr] = [0, 1];
    while (true) {
        [prev, curr] = [curr, prev + curr];
        yield curr;
    }
}

for (n of fibonnaci()) {
    console.log(n);
}
```

q.spawn: Generators + Promises

```
q.spawn(function* () {
    var data = yield $.ajax(url);
    $("#result").html(data);
    var status = $("#status").html("Download complete.");
    yield status.fadeIn().promise();
    yield sleep(2000);
    status.fadeOut();
});
```

q.spawn: Even Works on Exceptions

```
q.spawn(function* () {
    var user;
    try {
        user = yield getUser();
    } catch (err) {
        ui.showError(err);
        return;
    }

    ui.updateUser(user);
});
```