satz zu herkömmlichen Threads werden solche Worker-Threads vollständig isoliert und ohne Zugriff auf den globalen Gültigkeitsbereich oder die Webseiteninhalte im Hauptthread der Anwendung ausgeführt. Dadurch können sie die Ausführung von Code in der Hauptereigniswarteschlange nicht stören. In einem Worker-Thread ist es daher weniger problematisch, die synchrone Variante von `XMLHttpRequest` zu verwenden. Bei einer Blockierung durch einen Download kann der Worker nicht weitermachen, was aber nicht verhindert, dass die Seite dargestellt wird oder die Event Queue auf Ereignisse reagiert. In einem Server sind blockierende APIs in der Startphase harmlos, also bevor der Server beginnt, auf eingehende Anforderungen zu reagieren. Sobald der Server aber Anforderungen bearbeitet, sind blockierende APIs genauso katastrophal wie in der Event Queue des Browsers.

Was Sie sich merken sollten

- Asynchrone APIs verwenden Callbacks, um die Verarbeitung aufwendiger Operationen zu verzögern und eine Blockierung der Hauptanwendung zu verhindern.
- JavaScript nimmt Ereignisse gleichzeitig entgegen, verarbeitet die Ereignishandler aber nacheinander in einer Event Queue.
- Verwenden Sie in der Event Queue einer Anwendung niemals blockierende I/O-Vorgänge.

Verwenden Sie verschachtelte oder benannte Callbacks für die asynchrone Abarbeitung — Thema 62

In Thema 61 haben Sie gesehen, wie asynchrone APIs potenziell aufwendige I/O-Operationen ausführen, ohne die Anwendung daran zu hindern, andere Arbeit zu erledigen und weitere Eingaben zu verarbeiten. Die Reihenfolge der Operationen in asynchronen Programmen erscheint zu Anfang jedoch etwas verwirrend. Beispielsweise gibt das folgende Programm erst `"starting"` aus und dann `"finished"`, obwohl diese beiden Aktionen im Quellcode in umgekehrter Reihenfolge stehen:

```
downloadAsync("file.txt", function(file) {
   console.log("finished");
});
console.log("starting");
```

Der Aufruf von `downloadAsync` gibt die Steuerung sofort zurück, ohne erst darauf zu warten, dass die Datei komplett heruntergeladen ist.

Inzwischen sorgt die Run-to-Completion-Garantie von JavaScript dafür, dass die nächste Zeile ausgeführt wird, bevor irgendwelche Ereignishandler laufen können. Daher wird `"starting"` auf jeden Fall vor `"finished"` ausgegeben.

Um diese Operationsreihenfolge zu verstehen, sollten Sie sich immer vorstellen, dass eine asynchrone API Operationen nicht *durchführt*, sondern *auslöst*. Der vorstehende Code löst erst den Download der Datei aus und gibt dann sofort `"starting"` aus. Wenn der Download abgeschlossen ist, gibt der registrierte Ereignishandler in einem anderen Durchlauf der Ereignisschleife `"finished"` aus.

Mehrere Anweisungen in einer Sequenz, also in einer Reihe nacheinander auszuführen funktioniert also nur dann, wenn Sie etwas unmittelbar nach dem Auslösen einer Operation tun müssen. Wie aber bauen Sie abgeschlossene asynchrone Operationen in solch eine Abfolge ein? Nehmen wir beispielsweise an, dass wir einen URL asynchron in einer Datenbank nachschlagen und danach den Inhalt dieses URLs herunterladen wollen. Es ist nicht möglich, diese beiden Anforderungen einfach hintereinander zu schreiben:

```
db.lookupAsync("url", function(url) {
   // ?
});
downloadAsync(url, function(text) {    // Fehler: URL ist
                                       // nicht gebunden.
   console.log("contents of " + url + ": " + text);
});
```

Das kann nicht funktionieren, da wir den URL, den die Datenbanksuche ergibt, als Argument für `downloadAsync` definieren, in deren Gültigkeitsbereich er aber nicht zu finden ist. Das liegt daran, dass wir die Suche in der Datenbank an dieser Stelle nur ausgelöst haben, das Ergebnis aber noch nicht vorliegt.

Callbacks verschachteln

Die offensichtlichste Lösung für dieses Problem besteht darin, die Aufrufe zu verschachteln. Mithilfe von Closures (siehe Thema 11) können wir die zweite Aktion in den Callback der ersten einbetten:

```
db.lookupAsync("url", function(url) {
   downloadAsync(url, function(text) {
      console.log("contents of " + url + ": " + text);
   });
});
```

Wir haben zwar immer noch zwei Callbacks, aber diesmal befindet sich der zweite innerhalb des ersten. Dadurch entsteht eine Closure, die Zugriff auf die Variablen des äußeren Callbacks hat. Beachten Sie, wie der zweite Callback auf `url` verweist.

Die Verschachtelung asynchroner Operationen ist einfach, wird aber schnell unhandlich, wenn noch mehrere Operationen hinzukommen:

```
db.lookupAsync("url", function(url) {
   downloadAsync(url, function(file) {
      downloadAsync("a.txt", function(a) {
         downloadAsync("b.txt", function(b) {
            downloadAsync("c.txt", function(c) {
               // ...
            });
         });
      });
   });
});
```

Callbacks benennen

Eine Möglichkeit, eine solche übermäßige Verschachtelung einzudämmen, besteht darin, die verschachtelten Callbacks als benannte Funktionen (*Named Functions*) herauszunehmen und ihnen die zusätzlichen Daten, die sie benötigen, als weitere Argumente zu übergeben. Das Beispiel mit zweistufiger Verschachtelung können wir damit wie folgt umschreiben:

```
db.lookupAsync("url", downloadURL);

function downloadURL(url) {
   downloadAsync(url, function(text) { // Immer noch verschachtelt
      showContents(url, text);
   });
}

function showContents(url, text) {
   console.log("contents of " + url + ": " + text);
}
```

Hierbei wird immer noch ein verschachtelter Callback in `downloadURL` verwendet, um die äußere Variable `url` und die innere Variable `text` als Argumente von `showContents` zu kombinieren. Diese letzte Verschachtelung können wir mit `bind` aufheben (siehe Thema 25):

```
db.lookupAsync("url", downloadURL);

function downloadURL(url) {
   downloadAsync(url, showContents.bind(null, url));
}

function showContents(url, text) {
   console.log("contents of " + url + ": " + text);
}
```

Dadurch sieht der Code zwar mehr nach sequenzieller Abarbeitung aus, erfordert aber auch etwas mehr Aufwand, da Sie die Zwischenschritte der Folge einzeln benennen und die Bindungen von einem Schritt zum nächsten kopieren müssen. In Fällen wie dem zuvor angeführten längeren Beispiel kann das ziemlich umständlich werden:

```
db.lookupAsync("url", downloadURLAndFiles);

function downloadURLAndFiles(url) {
   downloadAsync(url, downloadABC.bind(null, url));
}

// Umständlicher Name
function downloadABC(url, file) {
   downloadAsync("a.txt",
                 // Duplizierte Bindungen
                 downloadABC.bind(null, url, file));
}

// Umständlicher Name
function downloadBC(url, file, a) {
   downloadAsync("b.txt",
                 // Weitere duplizierte Bindungen
                 downloadBC.bind(null, url, file, a));
}

// Umständlicher Name
function downloadC(url, file, a, b) {
   downloadAsync("c.txt",
                 // Noch mehr duplizierte Bindungen
                 finish.bind(null, url, file, a, b));
}

function finish(url, file, a, b, c) {
   // ...
}
```

Ein Kompromiss

Manchmal kann eine Kombination der beiden Vorgehensweisen ausgeglichener sein, wenngleich Sie auch dabei immer noch eine gewisse Verschachtelung in Kauf nehmen müssen:

```
db.lookupAsync("url", function(url) {
   downloadURLAndFiles(url);
});

function downloadURLAndFiles(url) {
   downloadAsync(url, downloadFiles.bind(null, url));
}

function downloadFiles(url, file) {
   downloadAsync("a.txt", function(a) {
```

```
        downloadAsync("b.txt", function(b) {
           downloadAsync("c.txt", function(c) {
              // ...
           });
        });
     });
}
```

Diese letzte Version lässt sich aber noch weiter verbessern, indem Sie eine Abstraktion für das Herunterladen mehrerer Dateien und deren Speicherung in einem Array hinzufügen:

```
function downloadFiles(url, file) {
   downloadAllAsync(["a.txt", "b.txt", "c.txt"],
      function(all) {
      var a = all[0], b = all[1], c = all[2];
      // ...
   });
}
```

Mithilfe von `downloadAllAsync` können Sie auch mehrere Dateien gleichzeitig herunterladen. Sequenzielle Abarbeitung bedeutet, dass eine Operation erst dann ausgelöst werden kann, wenn die vorherige abgeschlossen ist. Manche Operationen sind naturgemäß sequenziell, wie der Download von einem URL, den wir zunächst mit einer Datenbanksuche ermitteln. Wenn Sie aber eine Liste mit den Namen der herunterzuladenden Dateien haben, gibt es meistens keinen Grund dafür, auf den Abschluss eines Downloads zu warten, bevor Sie den nächsten anfordern. In Thema 66 erfahren Sie, wie Sie nebenläufige Abstraktionen wie `downloadAllAsync` implementieren.

Abgesehen von der Verschachtelung und Benennung von Callbacks ist es auch möglich, Abstraktionen höherer Ebene zu erstellen, um den asynchronen Steuerungsfluss einfacher und kompakter zu machen. Thema 68 beschreibt eine besonders weitverbreitete Vorgehensweise dazu. Außerdem lohnt es sich, Bibliotheken für die asynchrone Verarbeitung zu studieren oder eigene Abstraktionen auszuprobieren.

Was Sie sich merken sollten

- Verwenden Sie verschachtelte oder benannte Callbacks, um mehrere asynchrone Operationen nacheinander auszuführen.
- Versuchen Sie einen Kompromiss zwischen übermäßiger Verschachtelung von Callbacks und der umständlichen Benennung nicht verschachtelter Callbacks zu finden.
- Vermeiden Sie eine sequenzielle Abarbeitung von Operationen, die gleichzeitig ausgeführt werden können.