3 jQuery-Plug-ins

Bisher haben Sie einen Einblick in die allgemeinen Fähigkeiten von jQuery gewinnen können. Diese Einführung war nötig, um ein Verständnis für die Bibliothek und die mit ihr einhergehenden Arbeitsweisen zu bekommen. Im folgenden Kapitel wird zum ersten Mal über jQuery-Plug-ins gesprochen. Es wird beleuchtet, was Plug-ins ausmacht, welche verschiedenen Typen es gibt und welche Ziele mit ihrer Entwicklung verfolgt werden. Im Allgemeinen zu beachtende Regeln werden genauso aufgezeigt wie ein Grundgerüst zur Entwicklung vielseitiger und effizienter Plug-ins.

3.1 Was sind jQuery-Plug-ins?

Einige der ersten Fragen, die beantwortet werden müssen, wenn es um Plug-ins geht, sind die nach ihrer Bedeutung. Was macht ein jQuery-Plug-in aus? Warum will ich ein solches entwickeln? Gibt es verschiedene Arten von Plug-ins? All diese Fragen sollen im Folgenden beantwortet werden.

jQuery-Plug-ins sind von außen betrachtet nichts weiter als eine in sich abgeschlossene Funktionserweiterung der Bibliothek. Anstelle des Wortes Plug-in könnte man auch Modul oder Feature sagen. In der jQuery-Welt hat sich die Bezeichnung Plug-in jedoch etabliert und wird daher auch hier benutzt, auch wenn es sich nicht um die beste Wortwahl handelt.

Da ein Plug-in aufgrund seiner Integration in jQuery eine gewisse Struktur besitzen muss – wie sich im weiteren Verlauf dieses Kapitels noch zeigen wird – handelt es sich dabei immer um ein in sich abgeschlossenes Modul. Plug-ins können Funktionen und Methoden besitzen, die nur innerhalb des Plug-ins verfügbar sind. Nach außen wird lediglich die jQuery hinzugefügte Funktionalität sichtbar. Diese erzwungene Struktur besitzt den erfreulichen Nebeneffekt, dass Plug-ins nur selten voneinander abhängig sind. Auf jeden Fall aber wird eine Kapselung interner Funktionen erzwungen, die die Wartung und die Refaktorierung erleichtert und darüber hinaus für größere Wiederverwendbarkeit sorgt.

HINWEIS: Um all diese Eigenschaften von Plug-ins zu gewährleisten, müssen verschiedene Regeln eingehalten werden, die im weiteren Verlauf vorgestellt werden. Es existieren leider auch viele jQuery-Plug-ins, die sich nicht an diese Regeln halten und somit häufig nur von eingeschränktem Nutzen sind. So existieren Plug-ins, die ihre privaten und lediglich intern genutzten Funktionen dennoch in die globale Umgebung des Browsers exportieren. Sie verschmutzen somit nicht nur diesen Namensraum, sondern machen auch die Grundidee hinter der Kapselung von Plug-ins zunichte.

Ein Plug-in ist eine Funktion

Ähnlich wie es sich für die Werkzeuge auf der Unix-Kommandozeile etabliert hat, gilt auch in jQuery die Devise: Ein Plug-in sollte nur eine Funktion erfüllen, diese jedoch besonders gut und effizient. Durchdachtes Applikationsdesign führt mit jQuery also häufig zu dem Ergebnis, das gewünschte Ziel durch die Entwicklung mehrerer Plug-ins und deren anschließende Kombination zu erreichen.

PROFITIPP: Auch wenn die Umsetzung dieser Idee zunächst mit erhöhtem Aufwand verbunden ist, sind ihre Vorteile äußert reizvoll. Die zusätzlich investierte Entwicklungszeit amortisiert sich schnell durch den deutlichen Zugewinn an Wiederverwendbarkeit.

HINWEIS: Wird die Kapselung von Plug-ins konsequent eingehalten, so ist es unter anderem möglich, benötigte Applikationsteile erst bei Bedarf dynamisch nachzuladen anstatt sie initial auszuliefern. Bei Webapplikationen mit vielfältigem Funktionsumfang kann der Einsatz dieser Technik die Ladezeiten enorm verkürzen und somit das Benutzererlebnis deutlich verbessern.

3.2 Unterschiedliche Plug-in-Typen

Betrachtet man verschiedene jQuery-Plug-ins genauer, stellt man fest, dass es nicht nur eine Gattung von ihnen gibt. Die verschiedenen Ausprägungen, die sich unter anderem in der Art und Weise der Problemlösung manifestieren, charakterisieren unterschiedliche Plug-in-Typen. Während meiner täglichen Arbeit mit jQuery sind mir bereits viele dieser Typen untergekommen. Im folgenden Abschnitt werden diese unterschiedlichen Klassen identifiziert und kurz vorgestellt. Im weiteren Verlauf des Buches erhalten Sie detaillierte Informationen und Beispiele zu jeder der Klassen. Das wird Sie in die Lage versetzen, unterschiedliche Typen zu erkennen und für die Lösung beliebiger Probleme den korrekten auszuwählen.

3.2.1 Neue Methoden

Der erste vorgestellte Plug-in-Typ ist zugleich der am weitesten verbreitete: Die Erweiterung von jQuery-Sets um eine neue Methode. Diese neue Methode kann beliebige Operationen auf den Elementen des Sets ausführen. Hierbei kann es sich um vom DOM weitgehend unabhängige Aktionen wie die Berechnung der Durchschnittsfarbe handeln, oder um dem DOM nahe Funktionen wie die Veränderung von CSS-Eigenschaften und Positionierung. Auch das vollständige Entfernen oder Hinzufügen neuer Elemente ist Bestandteil dieser Plug-in-Klasse.

Die meisten Plug-ins in dieser Kategorie führen Veränderungen an den Elementen des Dokuments durch. In diesem Zusammenhang ist eine weitere Unterteilung erforderlich: neue Methoden mit und ohne Zustand.

Methoden ohne Zustand

Methoden, die bei jedem Aufruf eine Kette von Operationen auf den Zielelementen zur Ausführung bringen und nach deren Abschluss beendet sind, können als zustandslos bezeichnet werden. Methoden ohne einen Zustand erfüllen demnach zwei maßgebliche Kriterien:

1. Bei mehrmaligen Aufrufen ändert sich das Verhalten der Methode nicht.
2. Nach dem Aufruf wartet die Methode nicht auf ein weiteres Ereignis, um mit ihrer Arbeit fortzufahren.

> **HINWEIS:** Die zweite Anforderung ist nicht immer zu 100 % richtig. Falls es sich bei der stattfindenden Ereignisregistrierung um die Hauptaufgabe des Plug-ins handelt, kann es dennoch als zustandslos gelten, da das Ereignis nach der Registrierung nicht mehr von ihm abhängig ist und seine Tätigkeit somit abgeschlossen ist. Wird das registrierte Ereignis jedoch vom Plug-in zur weiteren Verarbeitung benötigt und ist lediglich ein Mittel zum Zweck, besitzt es aller Erfahrung nach einen Zustand.

Methoden mit Zustand

Methoden mit einem Zustand sind in den allermeisten Fällen sog. Pseudo-Widgets, also vollständig gekapselte Bedienelemente einer Webapplikation. Hierbei kann es sich z. B. um Dinge wie dynamische Baumstrukturen oder Lightboxen handeln. Den Zusatz „Pseudo“ erhalten diese Widgets, weil es seit der Veröffentlichung von jQuery UI (Kapitel 11) die Möglichkeit gibt, echte Widgets mit einem fest definierten API zu entwickeln. Grundsätzlich ist dieser Weg der Entwicklung einem Pseudo-Widget vorzuziehen, besitzt jedoch den Nachteil, dass damit auch eine Abhängigkeit auf die jQuery-UI-Bibliothek entsteht, die gegebenenfalls nicht wünschenswert ist.

HINWEIS: Als Lightbox wird im allgemeinen eine Technik bezeichnet, in der über die komplette Seite der Webapplikation ein semitransparenter Schleier gelegt und anschließend oberhalb dessen ein modaler Dialog angezeigt wird, der die unterschiedlichsten Inhalte präsentieren kann. In den meisten Fällen wird diese Technik jedoch zur Darstellung großformatiger Bilder eingesetzt, die das normale Seitenlayout durch ihre Dimensionen sprengen würden. Ein schöner Vertreter dieser Gattung ist das jQuery-Fancybox-Plug-in[1].

HINWEIS: Bei jQuery UI handelt es sich um eine weitreichende Erweiterung von jQuerys Fähigkeiten zur Entwicklung von Widgets und User-Interface-spezifischer Konzepte. Diese Erweiterung ist als Bibliothek, die auf jQuery basiert, umgesetzt und muss daher separat ausgeliefert werden. Detaillierte Informationen zum Einsatz und der Entwicklung von jQuery-UI-bezogenen Plug-ins erhalten Sie in Kapitel 11.

Weshalb Pseudo-Widgets einen gewissen Zustand speichern müssen, ist schnell ersichtlich. Häufig reagieren Widgets auf externe Ereignisse wie Klicken mit der Maus oder Drücken einer Tastenkombination. Daher müssen Event Handler registriert werden, die bei derartigen Ereignissen ausgelöst werden. Bei einem Aufruf müssen sie jedoch weitere Informationen über das aktuelle Widget, wie dessen Konfiguration oder aktuellen Status, abfragen können. Hierfür wird ein persistenter Zustand benötigt.

PROFITIPP: Plug-ins mit internem Zustand lassen sich oft an der Verwendung von jQuerys `data`-Methode erkennen. Diese Methode wird gerne eingesetzt, um ganze Instanzen des Plug-ins an ein DOM-Element zu binden oder einzelne Zustandswerte für die spätere Verwendung zu speichern. Allerdings ist auch hier Vorsicht geboten, da diese Schlussfolgerung nicht immer gilt. Sie ist jedoch ein deutliches Anzeichen.

Unterscheidungsgrundsatz

Wie die vielen Ausnahmen innerhalb der Beschreibungen zeigen, ist die Unterscheidung zwischen Methoden mit und ohne Zustand oft nicht einfach. Häufig ist die Grenze zwischen beiden Typen unscharf. Aus diesem Grund ist meist Bauchgefühl anstelle von konkreten Regeln erforderlich, um eine Differenzierung vorzunehmen.

In Kapitel 6 lernen Sie ein Pseudo-Widget am Beispiel einer Livesearch-Implementierung im Detail kennen.

1 Weitere Informationen zum Fancybox-Plug-in und alle benötigten Dateien zum Download finden Sie unter *http://fancybox.net*.

3.2.2 Neue Funktionen

Im Unterschied zu neuen Methoden, die auf jQuery-Sets aufgerufen werden, existiert ebenfalls die Möglichkeit, jQuery neue Funktionen hinzuzufügen, die direkt auf dem `jQuery`- oder `$`-Objekt zur Verfügung stehen. Sie haben somit keinen direkten Bezug zu einem Set.

Bei dieser Art von Erweiterung handelt es sich meist um Funktionalitäten mit sehr generischem Charakter. Oft erfüllen sie kleinere Aufgaben, die während der übrigen Programmierung häufig auftreten. Ein gutes Beispiel ist die Verarbeitung von Konfigurationsoptionen oder der Abruf von Daten eines Webservice mittels spezieller API.

> **HINWEIS:** Viele Funktionen, die durch Plug-ins bereitgestellt werden, könnten auch als normale JavaScript-Funktionen im globalen Namensraum des Browsers implementiert werden, da jQuery dafür gar nicht erforderlich ist. Innerhalb von Applikationen, die diese Bibliothek einsetzen, ist es dennoch üblich, eine Registrierung innerhalb des `jQuery`-Namensraums vorzunehmen, um zu verdeutlichen, dass die jeweilige Funktion nach den in jQuery üblichen Gepflogenheiten anwendbar ist. Im Besonderen bietet sich dieses Vorgehen natürlich an, falls die entwickelte Funktion in irgendeiner Form von jQuery abhängig ist.

Mit der Umsetzung einiger Shortcut-Funktionen zur einfachen Kommunikation mit einem RESTful-Webservice lernen Sie diesen Plug-in-Typ detailliert in Kapitel 7 kennen.

3.2.3 Special Events

Special Events stellen in jQuery eine einzigartige Möglichkeit dar, eigene Ereignisse für den Gebrauch innerhalb von Applikationen zu erzeugen. Durch die Art und Weise der Umsetzung können so manuell erzeugte Ereignisse innerhalb der bereits vorhandenen Infrastruktur zur Ereignisbehandlung genutzt werden. Während der Anwendung eines Special Events ist demnach keinerlei Unterschied zur Behandlung eines nativen Browserereignisses zu erkennen. Diese vollständige Transparenz macht dessen Einsatz sehr attraktiv, wenngleich auch nur wenige Entwickler um die Existenz dieses Plug-in-Typs wissen.

Das von jQuery bereitgestellte Interface zur Entwicklung erlaubt außerdem den einfachen Aufsatz auf bereits existierenden Ereignissen, was deren einfache Erweiterung ermöglicht.

Anhand der Implementierung der Ereignisse `dragstart`, `dragmove` und `dragend` zur Behandlung von Drag-and-Drop-Operationen wird Ihnen dieser Plug-in-Typ in Kapitel 8 vorgestellt.

3.2.4 Animations Easing-Funktionen

Easing-Funktionen bestimmen innerhalb aller Animationen in jQuery deren genauen Ablauf. Durch das Hinzufügen neuer Funktionen dieser Art kann das Animationsverhalten und der daraus resultierende optische Eindruck maßgeblich beeinflusst werden. Durch

den Einsatz entsprechender Easing-Funktionen ist es ein Leichtes, den Realismus diverser Animationen zu erhöhen und zugleich ein wiederverwendbares Plug-in zu schaffen.

Kapitel 10 beschäftigt sich eingehend mit dem Konzept der Easing-Funktionen sowie dessen genauer Umsetzung in jQuery. Neben dem nötigen Wissen zur Entwicklung eigener Easing-Effekte werden außerdem eine Vielzahl bereits existierender beleuchtet und anschaulich dargestellt.

3.2.5 CSS-Selektoren

In jQuery ist es möglich, der CSS-Selektor-Engine eigene Selektorregeln und Selektorfunktionen hinzuzufügen. Nur in den seltensten Fällen wird diese Fähigkeit in einem eigenständigen Plug-in eingesetzt. Üblich ist es hingegen, einem anderen Plug-in den letzten Feinschliff zu geben, indem eine Selektorfunktion geschaffen wird, mit dessen Hilfe alle beeinflussten oder erstellten Elemente ausgewählt werden können. Ein Plug-in, das Formulardaten validiert, könnte demnach eine CSS-Selektorfunktion mit dem Namen `:validfield` hinzufügen, die nur valide Felder des Formulars selektiert.

Informationen zur in jQuery eingesetzten CSS-Selektor-Engine und den Möglichkeiten zu deren Erweiterung finden Sie in Kapitel 9.

3.2.6 Widgets

Echte Widgets können nur in Kombination mit jQuery-UI entwickelt werden. Hierbei handelt es sich um eine Erweiterung von jQuery für häufig eingesetzte Widgets sowie Programmierschnittstellen zur Entwicklung verschiedener neuartiger Plug-ins. Da jQuery-UI mittlerweile seinen Kinderschuhen entwachsen und für den produktiven Einsatz bereit ist, behandelt dieses Buch auch diejenigen Plug-in-Klassen, die nur in Kooperation mit dieser Erweiterung möglich sind.

Widgets sind einer dieser Typen. Es handelt sich hierbei um die Bauelemente einer Webapplikation: Buttons, Listen, Progressbars und Dialoge. Im Allgemeinen sind Widgets all diejenigen Plug-ins, die einen Zustand besitzen und mithilfe der entsprechenden jQuery-UI-Interfaces konzipiert und entwickelt wurden.

Die Schritt für Schritt vollzogene Entwicklung eines vollwertigen jQuery-UI-Widgets finden Sie in Kapitel 11.6 dieses Buches.

3.2.7 Effects

Wie bei Widgets handelt es sich bei Effects um ein Konzept, das erst mit UI Einzug in jQuerys Sphären gehalten hat. Effects sind benutzerdefinierte Effekte. Auch wenn sie meist eng mit Animationen verbunden sind, stellen Effekte dennoch ein vollkommen anderes Konzept dar. Anstatt visuelle Veränderungen der Webapplikation durch das Ausführen

verschiedener Animationen zu erreichen, die nacheinander aufgerufen oder anderweitig kombiniert werden, bieten Effekte vorgefertigte, teils komplexe Animations- und Manipulationsabfolgen. So lassen sich in jQuery-UI z. B. beliebige Elemente mit nur einem Befehl in mehrere Teile sprengen und auf diese Weise ausblenden oder auf sonstige optisch ansprechende Art und Weise verändern.

Wie Sie unter Einsatz des jQuery-UI-API eigene spektakuläre Effekte entwickeln und einsetzen, erfahren Sie in Kapitel 11.8.

3.2.8 Behaviours

Behaviours stellen eine weitere Art der Plug-ins dar, die an jQuery-UI gebunden ist. Es handelt sich hierbei um eine Spezialform des Widgets, die der Anreicherung beliebiger Elemente im Dokument durch neue Fähigkeiten dient.

So ist es mit dessen Hilfe möglich, durch einen simplen Methodenaufruf beliebige Elemente beweglich oder größenveränderlich zu machen.

In Kapitel 11.7 wird präzise auf die Entwicklung von Behaviours eingegangen und an einem praxisnahen Beispiel erläutert.

3.3 Grundregeln der Plug-in-Entwicklung

Zur Entwicklung jeglicher Art von jQuery-Plug-ins ist das Befolgen der in diesem Abschnitt vorgestellten Grundregeln ein guter Ausgangspunkt, um schnell und erfolgreich das gewünschte Ziel zu erreichen. Einige der Regeln sind zwingend erforderlich, um ein lauffähiges Plug-in zu produzieren, andere sind lediglich guter Stil. Trotzdem wird ausdrücklich empfohlen, alle diese Regeln zu befolgen, um eine möglichst unkomplizierte Entwicklungsphase zu gewährleisten.

3.3.1 Konventionen zur Namensgebung

Bevor mit der Entwicklung begonnen werden kann, muss zunächst ein Dateiname für den späteren Plug-in-Quelltext ausgewählt werden. Bevor das geschehen kann, muss das Plug-in selbst zunächst einen Namen erhalten. Ein Name, der dessen Aufgabe möglichst präzise beschreibt, ist wünschenswert. Im Folgenden wird angenommen, das neue Plug-in trägt den äußerst kreativen Namen `MyPlugin`. Ist diese Entscheidung gefallen, kann ein Dateiname gebildet werden. jQuerys Regeln diesbezüglich sind relativ strikt. Nicht nur, wenn eine spätere Aufnahme in das jQuery-Plug-in-Verzeichnis angestrebt wird, bietet es sich an, die folgende Konvention zu befolgen:

> Der JavaScript-Quelltext jedes Plug-ins sollte sich in einer Datei mit dem Namen `jquery.[Pluginname].js` befinden

In dem hier skizzierten Szenario wäre das also `jquery.myplugin.js`. Grundsätzlich sind alle Buchstaben im Dateinamen klein zu schreiben, um Probleme auf Systemen zu vermeiden, die zwischen Groß- und Kleinbuchstaben im Dateisystem unterscheiden.

CSS eines Plug-ins

Oft ist es erforderlich, zusammen mit einem Plug-in CSS-Regeln auszuliefern. Natürlich ist es möglich, sie innerhalb des JavaScript-Quelltexts unterzubringen und dynamisch auf die jeweiligen Elemente anzuwenden. Es ist jedoch meist sinnvoller, diese Regeln innerhalb einer getrennten Datei zu speichern. Das erhöht nicht nur die Lesbarkeit des Quelltexts, sondern erleichtert auch spätere Anpassungen durch den Anwender des Plug-ins.

CSS-Dateien, die zu einem speziellen Plug-in gehören, sollten den Dateinamen `jquery.[Pluginname].css` tragen. In diesem Beispiel ist das `jquery.myplugin.css`. Auch hier gilt der Grundsatz, lediglich Kleinbuchstaben zu verwenden.

Weitere Dateien eines Plug-ins

Gelegentlich benötigt ein Plug-in neben JavaScript-Quelltext und CSS-Regeln noch weitere Dateien. Das können z. B. Bilder, Videodateien, oder jegliche andere Art von Daten sein. Für sie existiert kein konkretes Namensschema, da dieser Fall nur selten auftritt. Hier bleibt die Auswahl geeigneter Dateinamen Ihnen als Entwickler überlassen. Für den Autor haben sich zwei unterschiedliche Ansätze als praktikabel erwiesen:

Die erste Variante ist die Ablage der entsprechenden Dateien in Ordnern, die ihren Inhalt beschreiben. Bilder würden sich somit im Verzeichnis `images` wiederfinden, wohingegen Videos im Ordner `videos` residierten. Weil diese Ordner natürlich in größeren Applikationen nicht nur von einem Plug-in, sondern mehreren verwendet würden, sollte der Plug-in-Name als Präfix innerhalb des Dateinamens eingesetzt werden. Im konkreten Beispiel wäre z. B. die Grafik einer „Schließen"-Schaltfläche des hier skizzierten MyPlugins in der Datei `images/myplugin-close.png` abgelegt.

Die zweite Variante ist die Unterbringung aller weiteren Dateien in einem Ordner mit dem Namen `jquery.[Pluginname]`. Im Fall dieses konkreten Beispiels: `jquery.myplugin`. Auch hier bietet es sich an, die einzelnen Dateien nach Inhalt zu gruppieren. Also `jquery.myplugin/images/close.png` oder `jquery.myplugin/videos/warning.mp4`. Auf ein weiteres Präfix im Dateinamen kann bei dieser Umsetzung natürlich verzichtet werden.

Egal, welche der beiden Methoden Sie präferieren, es empfiehlt sich immer, dem Anwender zu ermöglichen, den Speicherort und Dateinamen mittels einer Option zu verändern. Somit kann beim Einsatz des Plug-ins der Ablageort der „Schließen"-Schaltfläche einfach verändert oder sogar die Grafik vollständig ausgetauscht werden. Wie Sie diese Optionen umsetzen und behandeln können, lernen Sie in Kapitel 4.3.

3.3.2 Zugriff auf die jQuery-Bibliothek

Innerhalb eines Plug-ins wird für gewöhnlich Zugriff auf Funktionen der jQuery-Bibliothek benötigt. Wäre das nicht der Fall, käme die Frage auf, warum ein jQuery-Plug-in als Lösungsweg gewählt wurde.

> Innerhalb von Plug-in-Quelltexten sollten Bibliotheksfunktionen nur unter Verwendung des `jQuery`-Objekts aufgerufen werden. Die Verwendung von Shortcuts wie dem `$`-Objekt ist gänzlich zu unterlassen.

Die Begründung für diese Einschränkung liegt in der Fähigkeit von jQuery, parallel zu weiteren JavaScript-Bibliotheken innerhalb einer Webseite zu koexistieren. Um das zu erreichen, kann auf dem `jQuery`-Objekt die Funktion `noConflict` aufgerufen werden, die sicherstellt, dass eventuelle Belegungen des `$`-Shortcuts durch andere Bibliotheken weiterhin Gültigkeit besitzen. Nach dem Aufruf dieser Methode ist ein Zugriff auf jQuery nur noch über das `jQuery`-Objekt direkt oder einen vom Benutzer festgelegten anderen Bezeichner möglich. Ein Plug-in kann sich also nicht auf die Existenz des `$`-Objekts verlassen.

HINWEIS: Viele verschiedene JavaScript-Bibliotheken haben das Dollarzeichen (`$`) als Bezeichner ausgewählt. Diese Entscheidung liegt nahe, da das `$`-Zeichen innerhalb von JavaScript keine besondere Bedeutung besitzt, optisch gut zu erkennen ist und sich leicht eingeben lässt. Außerdem ist bei häufig wiederkehrenden Aufrufen, wie z. B. der Selektion von DOM-Elementen, eine möglichst kurze Zeichenkette wünschenswert. Auch wenn die Verwendung dieses Symbols vollkommen plausibel ist, so haben einige Bibliotheken den gravierenden Nachteil, dass Sie dieses nicht als zusätzlichen Shortcut anbieten, sondern als alleinige Zugriffsmöglichkeit auf deren Funktionalität. Um dieses Manko anderer Bibliotheken zu kompensieren, existiert in jQuery die `noConflict`-Funktion.

$-Shortcut dennoch einsetzen

Sind die Gründe für den Einsatz des `jQuery`-Objekts anstelle des `$`-Shortcuts bekannt, lässt sich mit diesem Wissen ein Ausweg aus dem Dilemma finden, der die Verwendung des `$`-Symbols auch innerhalb eines jQuery-Plug-ins ermöglicht, ohne dabei dessen Integrität zu gefährden.

Um die Wirkungsweise dieses Tricks zu verstehen, muss zunächst eine Eigenschaft von JavaScript selbst erläutert werden: Lambda-Funktionen.

JavaScript erlaubt die Definition sog. Lambda- oder anonymer Funktionen. Hierbei handelt es sich um Funktionen, die nicht mit einem konkreten Namen versehen werden. Stattdessen können sie in einer Variablen gespeichert oder direkt zur Ausführung gebracht werden. Die Deklaration entspricht der einer üblichen Funktion, jedoch ohne Angabe

eines Namens. Listing 3.1 zeigt die Erzeugung einer solchen Lambda-Funktion und deren anschließende Speicherung in der Variablen `fn`.

```
var fn = function() {
  // Hier befindet sich der übliche Funktionsrumpf
};
```

Listing 3.1: Erzeugung einer Lambda-Funktion mit anschließender Zuweisung in die Variable `fn`

Wie zuvor erwähnt, ist nicht nur die Speicherung in einer Variablen möglich, sondern außerdem die direkte Ausführung (Listing 3.2).

```
(function() {
  // Hier befindet sich der übliche Funktionsrumpf
})();
```

Listing 3.2: Deklaration und direkte Ausführung einer Lambda-Funktion.

Um eine Lambda-Funktion direkt nach ihrer Deklaration zur Ausführung zu bringen, wird sie in runde Klammern eingeschlossen und dieses Konstrukt genau wie jede andere Funktion mittels Argumentenübergabe aufgerufen. Existieren wie im Beispiel aus Listing 3.2 keinerlei Funktionsargumente, erfolgt der Aufruf in gewohnter Weise durch ein leeres Klammernpaar.

HINWEIS: Lambda-Funktionen sind Ihnen bereits innerhalb der jQuery-Einführungskapitel über den Weg gelaufen. Anonyme Funktionen eigenen sich für die verschiedensten Anwendungsgebiete und werden daher in JavaScript sehr häufig zum Einsatz gebracht. Sie werden diesem Konstrukt also noch häufiger begegnen.

Mit diesem Wissen um Lambda-Funktionen ist es nun möglich, den $-Shortcut gefahrlos während der Plug-in-Entwicklung einzusetzen. Der gesamte Plug-in-Quelltext wird hierzu in einer Lambda-Funktion eingeschlossen, die ein Argument mit dem Bezeichner $ entgegennimmt. Bei dem unmittelbar darauf folgenden Aufruf der Funktion wird als Inhalt für eben dieses Argument das `jQuery`-Objekt übergeben. Da der Zugriff auf Variablen immer zunächst im aktuellen Scope erfolgt und jede Funktion einen eigenen Scope besitzt, führt ein Aufruf des $-Shortcuts innerhalb des Plug-in-Quelltexts nun unweigerlich zum `jQuery`-Objekt, unabhängig von seiner Belegung außerhalb der Funktion (Listing 3.3).

```
(function($) {
  // Hier befindet sich der komplette Plug-in-Quelltext.
  // Ein Zugriff auf $ innerhalb dieser Funktion führt immer
  // zum jQuery-Objekt.
})(jQuery);
```

Listing 3.3: Anwendung einer Lambda-Funktion zur Verwendung des $-Shortcuts innerhalb von jQuery-Plug-ins

PROFITIPP: Der Zugriff auf jegliche Art von Bezeichner in JavaScript (Variablennamen, Funktionen, …) erfolgt immer zunächst im lokalen Scope. Wird der jeweilige Bezeichner in diesem Scope nicht gefunden, wird die Suche im nächsthöheren fortgesetzt. Erst wenn der äußerste Scope erreicht und dort der Bezeichner nicht gefunden wurde, gilt er als nicht verfügbar. Für jede Funktion, egal ob anonym oder nicht, erstellen JavaScript Engines einen neuen Ausführungskontext und somit einen neuen Scope. Generell gilt die Regel: Der Zugriff auf Bezeichner aus dem lokalen Scope ist schnell, wohingegen jede weitere Ebene nach außen den Zugriff verlangsamt. Aus diesem Grund sollte es vermieden werden, Bezeichner anzufordern, die nicht im lokalen Scope liegen. Ist ein mehrfacher Zugriff auf diese Bezeichner erforderlich, bietet es sich an, sie in einer Variablen innerhalb des lokalen Scopes abzulegen und von dort aus zu benutzen. Auch wenn aktuelle JavaScript Engines durch diverse Optimierungen versuchen, dieses Problem zu minimieren, ist es dennoch sinnvoll, sich dieser Tatsache bewusst zu sein. Bezogen auf Listing 3.3 bedeutet das also, dass ein Zugriff auf `$` innerhalb der Funktion schneller ist als ein Zugriff auf `jQuery` selbst, da dieser Bezeichner nicht im lokalen Scope der Funktion liegt.

Lambda-Funktionen nicht nur für den $-Shortcut

Es ist nützlich, ein jQuery-Plug-in innerhalb einer Lambda-Funktion einzuschließen, egal ob der $-Shortcut eingesetzt werden soll oder nicht. Der Hauptgrund ist der zuvor bereits angesprochene neue Scope, der für diese Funktion erzeugt wird. Durch diese Gegebenheit ist garantiert, dass sämtliche innerhalb des Plug-ins deklarierten Bezeichner, also Variablen und Funktionen, innerhalb des Plug-in-Kontexts verbleiben und nicht den globalen Namensraum verschmutzen. Außerdem ist es so möglich, innerhalb des Plug-ins private Funktionen zu definieren, die nicht von außen zugreifbar sind.

3.3.3 Anlegen neuer Set-Methoden

Da das Erweitern von jQuery-Sets mit neuen Methoden, wie bereits in der Vorstellung der einzelnen Plug-in-Typen erwähnt, eines der wichtigsten und am häufigsten anzutreffenden Einsatzgebiete von Plug-ins ist, fällt dies in den Bereich der Grundregeln der Plug-in-Entwicklung.

Neue Methoden eines jQuery-Sets müssen innerhalb des `jQuery.fn`-Namensraums definiert werden. Innerhalb dieses Objekts tragen sie den gleichen Namen, mit dem sie fortan auf beliebigen Sets aufgerufen werden können. Es ist üblich, den Namen des Plug-ins und der Methode aufeinander abzustimmen.

Zur Erstellung einer solchen Methode wird erneut zur Lambda-Funktion gegriffen. Für das hier beispielhaft benannte Plug-in `MyPlugin` kann die Deklaration der gleichnamigen Set-Methode in Listing 3.4 nachvollzogen werden.

```
jQuery.fn.myPlugin = function() {
  // Inhalt der neuen Methode
};
```

Listing 3.4: Deklaration einer neuen Set-Methode mit dem Namen myPlugin

Wie bei allen auf jQuery-Sets verfügbaren Methoden ist es üblich, den Namen mit einem Kleinbuchstaben zu beginnen und einzelne Worte durch CamelCase-Schreibweise abzugrenzen.

Während diese neue Methode auf einem Set aufgerufen wird, ist es möglich, durch die Variable `this` auf das Set des Aufrufs Zugriff zu nehmen. In Kapitel 4 wird durch die praktische Anwendung dieser und aller bisherigen Regeln dieses Konzept verdeutlicht.

3.3.4 Anlegen neuer Funktionen

Neben dem Erzeugen neuer Methoden für Sets ist die Definition von neuen Funktionen ein weiterer Kernbereich für den Einsatz von Plug-ins.

> Neue Funktionen, die unabhängig von Sets agieren, werden direkt auf dem `jQuery`-Objekt registriert.

Auch hier gilt für die Namensgebung wie bei Set-Methoden die unausgesprochene Regel, sie dem Plug-in-Namen anzupassen und in CamelCase-Schreibweise mit beginnendem Kleinbuchstaben auszuführen (Listing 3.5).

```
jQuery.myPlugin = function() {
  // Hier befindet sich der Rumpf der Set unabhängigen Funktion.
};
```

Listing 3.5: Deklaration einer Set-unabhängigen Funktion innerhalb von jQuery

3.3.5 Beachtung des Fluent-Interface

Wie Sie sich aus dem Einführungskapitel sicherlich noch erinnern, unterstützt jQuery für die meisten seiner Methoden das Konzept des sog. Fluent-Interfaces. Kurz gesagt bedeutet das: Jede Methode liefert als Rückgabewert ihr Eingabeset. Durch diesen Trick ist es möglich, den Aufruf mehrerer Methoden direkt aneinanderzureihen. Natürlich soll diese Eigenschaft auch für Methoden aus der Eigenentwicklung gelten, um dem Bedienungsgrundsatz der übrigen Bibliothek zu genügen.

> Eine Methode auf einem jQuery-Set sollte immer das aktuelle Set als Rückgabewert liefern, es sei denn, besondere Umstände erzwingen einen anderen Wert.

Wie bereits angesprochen, ist das aktuelle Set beim Aufruf einer entsprechenden Methode über den Bezeichner `this` erreichbar. Es genügt zur Umsetzung dieser Regel, den Inhalt eben dieser Variablen als Rückgabewert einzusetzen (Listing 3.6).

HINWEIS: In Bezug auf die Regel bedeuten „besondere Umstände", dass es die vornehmliche Funktion der neuen Methode ist, einen anderen Wert als das Set zurückzugeben. Als Beispiel eignen sich hier z. B. die Methoden `height` und `width`, die ohne übergebenes Argument die Größe eines Elements bestimmen und sie zurückgeben. In solch speziellen Fällen darf die Kette des Fluent-Interface natürlich durchbrochen werden. Wird jedoch ein anderer Wert als Rückgabe eingesetzt, muss das in jedem Fall deutlich dokumentiert werden.

```
jQuery.fn.myPlugin = function() {
  // Der Plug-in-Quelltext steht hier
  return this;
};
```

Listing 3.6: Rückgabe des aktuellen Sets zur Gewährleistung des Fluent-Interface-Paradigmas

3.3.6 Multiple Elemente eines Sets korrekt behandeln

Während der Entwicklung von Methoden muss immer bedacht werden, dass diese auf Sets agieren, die möglicherweise mehr als ein Element beinhalten. jQuery besitzt mit der `each`-Methode eine einfache Möglichkeit, dieses Faktum zu berücksichtigen und korrekt zu behandeln. Auch wenn hierzu Ausnahmen existieren, ist es meistens sinnvoll, die Aufgabe des Plug-ins separat auf jedes Element eines Sets anzuwenden. In diesen Fällen ist der Einsatz von `each` die richtige Wahl.

Die `each`-Methode sollte eingesetzt werden, um alle Elemente des Ausgangssets korrekt zu behandeln.

Die each-Methode und das Fluent-Interface

Natürlich darf im Kontext von `each` nicht das Fluent-Interface vergessen werden. Da sich diese Methode selbst an das Paradigma hält, kann dessen Rückgabewert bequem durchgereicht werden (Listing 3.7).

```
jQuery.fn.myPlugin = function() {
  return this.each( function() {
    // this ist in diesem Kontext nur noch ein einzelnes Element
  } );
};
```

Listing 3.7: Einsatz der each-Methode zur Behandlung aller Elemente eines Sets

3.4 Grundgerüst

Aus allen im vorherigen Abschnitt beschriebenen Regeln lässt sich nun ein einfaches Grundgerüst zusammenstellen, das genügt, um die meisten Anforderungen mittels eines Plug-ins umzusetzen. Natürlich kann dieser Aufbau nur für einen Plug-in-Typ erfolgen. Da neue Set-Methoden der häufigste Anwendungsfall sind, fällt die Wahl hier relativ leicht. Außerdem können viele Ideen, wie die Verwendung einer Lambda-Funktion als Wrapper und die Beachtung des Fluent-Interfaces mit minimalen Anpassungen auch für andere Plug-in-Typen übernommen werden. Listing 3.8 zeigt das Grundgerüst und verdeutlicht durch Kommentare, an welchen Stellen welche Implementierungsdetails erwartet werden.

```
(function($, jQuery) {

  // Die Deklaration privater Methoden kann hier erfolgen.
  // …

  jQuery.fn.myPlugin = function() {
    // Alle Set-abhängigen Operationen erfolgen hier.
    // …

    // Beachtung des Fluent-Interface und Behandlung aller
    // Set-Elemente.
    return this.each( function() {
      // Hier werden alle einzelnen Elemente behandelt.
      // this ist in diesem Kontext ein einzelnes DOM-Element.
      // Ein spezifischer Rückgabewert ist nicht nötig.
    } );
  };
})(jQuery, jQuery);
```

Listing 3.8: Grundgerüst zur Entwicklung eines Plug-ins mit einer neuen Set-Methode

HINWEIS: In Listing 3.8 erfolgt eine doppelte Übergabe des `jQuery`-Objekts an die umschließende Lambda-Funktion. Der Grund ist der in Kapitel 3.3.2 erwähnte Geschwindigkeitsvorteil beim Zugriff auf Bezeichner im lokalen Scope. Würde der Bezeichner `jQuery` nicht als Funktionsargument erneut gesetzt, so wären Zugriffe auf das `$`-Objekt schneller als auf das `jQuery`-Objekt, weil `jQuery` sich einen Kontext weiter außen befindet. Der Autor hat in seinem Quelltext, zur Erhöhung der Lesbarkeit gerne die Möglichkeit, sowohl `$` als auch `jQuery` für den Zugriff zu nutzen. Um sich keinerlei Gedanken um die Performance dieses Wechsels zu machen, sollte man beide Bezeichner in den lokalen Scope ziehen.

3.5 Zusammenfassung

In diesem Kapitel haben Sie gelernt:

- Plug-ins sind der in jQuery gebräuchliche Name für abgeschlossene Funktionseinheiten (Module).
- Es existiert eine Vielzahl unterschiedlicher Plug-in-Typen: Methoden, Funktionen, Special Events, Easing-Funktionen, CSS-Selektoren, …
- Neue Methoden können in zwei Segmente unterteilt werden: Methoden mit und Methoden ohne persistenten Zustand.
- Zur Benennung von Dateinamen innerhalb von Plug-ins gibt es konkrete Regeln.
- Um der möglichen Veränderung des $-Objekts durch den Einsatz von noConflict in einem Plug-in vorzubeugen, ist eine anonyme Wrapper-Funktion nötig.
- Neue Methoden werden im jQuery.fn-Namespace registriert.
- Neue Funktionen werden direkt auf dem jQuery-Objekt registriert.
- Das Fluent-Interface-Paradigma sollte auch bei eigenen Plug-ins nicht durchbrochen werden.
- Die each-Methode hilft bei der korrekten Behandlung von Sets als Eingabe.

Vorschläge für die weitere Recherche:

- Abschnitt der jQuery-Dokumentation über die Erstellung von Plug-ins: http://docs.jquery.com/Plug-ins/Authoring