Datenstrukturen Maps und Sets

Objekte und Arrays = vorherrschenden Datenstrukturen zur Speicherung von Datensammlungen

- Objekte zum Speichern von Schlüssel/Wert-Paaren
- Arrays zum Speichern indizierter Listen

für mehr Flexibilität mit ECMAScript 2015 (ES6) 2 neue Arten von iterierbaren Objekten eingeführt:

- Maps, d. h. geordnete Sammlungen von Schlüssel/Wert-Paaren
- Sets, d. h. Sammlungen von eindeutigen Werten

Maps

- = Datenstruktur, in der Schlüssel-Wert-Paare gespeichert werden
- = Sammlung von Schlüssel-Wert-Paaren
- kann jeden Datentyp als Schlüssel verwenden
- Reihenfolge der Einträge bleiben erhalten

Maps haben Elemente von Objekten (eindeutiges Schlüssel/Wert-Paar) und Arrays (geordnete Sammlung) - sind aber konzeptionell den Objekten ähnlicher

liegt daran, dass Einträge selbst Schlüssel/Wert-Paare wie Objekte sind, obwohl Größe und Reihenfolge der Einträge wie bei Array beibehalten werden

innerhalb einer Map können zu Schlüsseln einzelne Werte abgelegt werden, die dann über Schlüssel abgerufen werden können

Schlüssel innerhalb einer Map müssen eindeutig sein, Werte können mehrfach vorkommen

Instanzen können über Konstruktorfunktion erzeugt werden

Maps haben Elemente von Objekten und Arrays

- wie bei Array haben wir eine nullindizierte Sammlung
- können auch sehen, wie viele Elemente standardmäßig in Map sind
- Maps verwenden Syntax => um Schlüssel/Wert-Paare als key => value zu kennzeichnen:

Maps werden über Konstruktorfunktion new Map() erzeugt

als Parameter kann iterierbares Objekt - bsw. Array von Arrays mit Schlüssel-Wert-Paaren übergeben werden

iterierbares Objekt = Objekt, das eigenes Iterationsverhalten definiert (sich daher bsw. direkt in Schleife als Eingabe verwenden lässt)

einige Standardobjekte wie String, Array, Map und Set sind standardmäßig iterierbar - können auch eigene iterierbare Objekte erstellen

Umwandlung Map in Object: `const object = Object.fromEntries(map);`

Umwandlung Map in Array: `const array = Array.from(map);`

Map-Schlüssel

- Map akzeptiert jeden Datentyp als Schlüssel (in Objekten werden Schlüssel in Zeichenfolge geändert)
- lassen das Duplizieren von Schlüsselwerten nicht zu – müssen eindeutig sein
- Map verwendet zum Vergleich der Gleichheit Referenz zum Objekt, nicht Literalwert des Objekts – d.h.:
- Hinzufügen von 2 eindeutigen Objekten mit gleichem Wert erstellt Map mit zwei Einträgen
- mehrmalige Verwendung einer Objekt-Referenz erstellt Map mit einem Eintrag

Eigenschaften und Methoden von Map

- Nachteil von Objekten = ist schwierig, diese aufzuzählen oder mit allen Schlüsseln oder Werten zu arbeiten
- Map-Struktur hat Menge integrierter Eigenschaften, wodurch man direkter mit ihren Elementen arbeitet kann

Methode/Eigenschaft	**Beschreibung**
set(key, value)	setzt zu Schlüssel entsprechenden Wert ist Schlüssel bereits in Map vorhanden, wird damit assoziierter Wert überschrieben (Rückgabe: Map-Object)
get(key)	liefert zu Schlüssel den assoziierten Wert (gibt es Schlüssel in Map nicht, Rückgabe = undefined)
has(key)	prüft, ob es zu Schlüssel Wert in Map gibt (Rückgabe: true/false)
delete(key)	löscht zu Schlüssel den Wert aus Map (gibt bei erfolgreichem Löschen true zurück, sonst false)
clear()	löscht Schlüssel-Wert-Paare aus Map
size	enthält Anzahl an Schlüssel-Wert-Paaren in Map
keys()	gibt Iterator für Schlüssel der Map zurück
values()	gibt Iterator für Werte der Map zurück
entries()	gibt Iterator, um über Schlüssel-Wert- Paare der Map zu iterieren
foreach()	iteriert über Map in Reihenfolge der Einfügungen

besonderer Vorteil von Maps gegenüber Objekten: können jederzeit Größe einer Map finden, wie es auch bei Array möglich (size beinhaltet weniger Schritte als Umwandlung eines Objekts in Array, um length zu finden)

Schlüssel, Werte und Einträge für Maps

Objekte können Schlüssel, Werte und Einträge über Methoden des Objekt-Konstruktors abrufen Object.keys() etc.

Maps haben Prototyp-Methoden, die es ermöglichen, Schlüssel, Werte und Einträge der Map-Instanz direkt abzurufen

Methoden keys(), values() und entries() geben einen MapIterator zurück, der es wie bei Array ermöglicht, mit for...of Werte zu durchlaufen

for-of-Schleife seit ES2015

- iteriert über mit Eigenschaften assoziierte Werte
- (zur Erinnerung: ältere for-in-Schleife gibt Indizes - die bei Arrays die Eigenschaften darstellen)

Iterator, der von entries() zurückgegeben = Iterator, der standardmäßig dem (iterierbaren) Objekt Map hinterlegt - d.h. kann in diesem Fall auch direkt Map als Eingabe für for-of-Schleife verwenden,

Map verfügt – ähnlich wie Array – über integrierte forEach-Methode für integrierte Iteration

Map.prototype.forEach((value, key, map) = () => {})

ist großer Vorteil für Maps gegenüber Objekten, da Objekte mit keys(), values() oder entries() umgewandelt werden müssen – gibt keinen einfachen Weg, Eigenschaften eines Objekts ohne Umwandlung abzurufen

Vorteile Maps gegenüber Objekten:

- Size – Maps haben die Eigenschaft size, während Objekte keine integrierte Möglichkeit zum Abruf ihrer Größe haben
- Iteration – Maps sind direkt iterierbar, Objekte nicht
- Flexibilität – Maps können einen beliebigen Datentyp (primitiv oder Objekt) als Schlüssel für einen Wert haben, während Objekte nur Zeichenfolgen haben können
- Geordnet – Maps behalten die Reihenfolge ihrer Einfügungen bei, während Objekte keine garantierte Reihenfolge haben

daher sind Maps leistungsstarke Datenstruktur

Vorteile von Objekten:

- JSON – Objekte funktionieren einwandfrei mit JSON.parse() und JSON.stringify(), zwei wesentliche Funktionen zum Arbeiten mit JSON, das u.a. gängiges Datenformat, für Arbeit mit vielen REST-APIs
- können mit .key direkt auf einzelnen Wert zugreifen, ohne Methode wie Maps get() verwenden zu müssen
- auf dieser Grundlage lässt sich Entscheidung treffen, ob Map oder Objekt richtige Datenstruktur für Anwendungsfall ist

Set (dt. Menge)

Datenstruktur ähnlich einer Liste, in der doppelte Werte nicht erlaubt – d.h. Set = Sammlung von eindeutigen Werten

Set ist Array konzeptionell ähnlicher als Objekt, da Liste von Werten und nicht Schlüssel/Wert-Paare

Set kein Ersatz für Arrays, sondern Ergänzung für zusätzliche Unterstützung der Arbeit mit duplizierten Daten

Initialisierung über Konstruktorfunktion: const set = new Set();

Set hat viele gleiche Methoden / Eigenschaften wie Map, bsw. delete(), has(), clear() und size

Elemente in Set hinzugefügen mit add() (nicht mit set() für Map verwechseln, obwohl ähnlich)

Sets können nur eindeutige Werte enthalten - jeder Versuch, bereits vorhandenen Wert hinzuzufügen, wird ignoriert

Objekte, die gleichen Wert haben, aber nicht gleiche Referenz teilen, werden nicht als gleich angesehen

Set lässt sich mit mit Array von Werten initialisieren - doppelte Werte im Array werden vom Set entfernt

const setHeros = new Set(['Jessica', 'Luke', 'Odin', 'Odin']); // Set(3) ["Jessica", "Luke", "Odin"]

Umwandlung Set zu Array mit Spread-Syntax: const arr = [...set]

keine Möglichkeit, auf Wert über Schlüssel oder Index zuzugreifen, wie Map.get(key) oder arr[index]

Methoden keys(), values(), entries(), geben Iterator zurück

- Sets haben keine Schlüssel - daher Schlüssel = Alias für Werte
- daher keys() und values() haben gleichen Iterator
- entries() gibt Wert zweimal zurück

am sinnvollsten, nur values() mit Set zu verwenden, da andere Methoden der Einheitlichkeit und übergreifenden Kompatibilität mit Map dienen

hat auch integrierte forEach()-Methode

- da keine Schlüssel, geben 1. und 2. Parameter des forEach()-Callbacks gleichen Wert zurück, sodass es dafür keinen Anwendungsfall außerhalb der Kompatibilität mit Map gibt
- Set.prototype.forEach((value, key, set) = () => {})

Eigenschaften und Methoden von Set

Methode/Eigenschaft	Beschreibung
add()	fügt Set ein Element hinzu
has()	prüft, ob übergebenes Element im Set enthalten
delete()	löscht übergebenes Element aus Set
clear()	löscht alle Elemente aus Set
entries()	gibt Iterator mit wert-wert-pairs des Sets Reihenfolge der Paare entspricht Reihenfolge des Hinzufügens
keys()	gibt Iterator mit Werten des Sets Reihenfolge der Paare entspricht Reihenfolge des Hinzufügens
values()	gibt wie keys() Iterator mit Werten des Sets (Standart-Iterator) Reihenfolge der Paare entspricht Reihenfolge des Hinzufügens
size	Anzahl an Elementen im Set

Wann Set verwenden?

- Set = nützliche Ergänzung, insbesondere für Arbeit mit doppelten Werten in Daten
- in einzelner Zeile können wir neues Array ohne doppelte Werte eines Arrays, das doppelte Werte hat, erstellen const uniqueArray = [...new Set(myArray)]
- Set kann verwendet werden, um nach Gemeinsamkeiten (Union), Überschneidung (Intersection) und Unterschied (Difference) zwischen zwei Datensätzen zu suchen

Vorteil von Arrays gegenüber Sets:

- Methoden sort(), map(), filter() und reduce()
- direkte Kompatibilität mit JSON-Methoden

Iteratoren zum standardisierten Iterieren von Collections

Prinzip von Iteratoren

- Iteratoren abstrahieren Iteration über Datenstrukturen
- sind Objekte, die Art Zeiger auf unterliegende Datenstruktur enthalten, der über Aufrufe der Methode next() am Iterator verschoben werden kann
- jeweiliges Element, auf das Zeiger zeigt, wird von next() zurückgegeben

Methode next() wird aufgerufen, bis Ende des Iterators erreicht und alle Werte ausgegeben

next() liefert Objekt mit Eigenschaften done (Angabe, ob Ende des Iterators erreicht) und value (Wert der aktuellen Iteration) und rückt internen Zeiger des Iterators weiter

Generatoren

seit ES2015 (ES6) hinzugefügt

mit ihnen möglich, Funktionen an bestimmten Stellen anzuhalten / zu unterbrechen und später fortzusetzen

werden über Generatorfunktionen erzeugt

Generatorfunktion erstellen

Generatorfunktionen werden über function* definiert

bei Aufruf liefert Generatorfunktion einen Generator zurück, über den man Ausführung des in Generatorfunktion definierten Codes steuern kann

können mehrere Generatoren auf Basis der Funktion erzeugen, die alle gleiches Verhalten zeigen

unterbrechen eines Generators über neu eingeführten yield-Operator

yield liefert - wie return - Wert zurück und sorgt dafür, dass aus Generator herausgesprungen wird

wird Generator nächstes Mal aufgerufen, setzt Ausführung der Funktion nach yield fort

Aufruf des Generators über Methode next() am Generatorobjekt setzt Generator in Gang

Generatoren = spezielle Art von Iteratoren

- Aufrufe von next() liefern am Generator immer Objekt mit Eigenschaften done und value
- done gibt Auskunft, ob Ende des Generators erreicht
- value enthält jeweiligen Wert, der vom aktuellen yield zurückgegeben

durch wiederholte Aufrufe von next() am Generator wird erreicht, dass Kontrollfluss bei yield aus Generatorfunktion heraus- und dahinter wieder hereinspringt

in einigen Browsern kann es TypeError "generator has already finished" geben, wenn versucht wird, weitere Aufrufe von next() zu tätigen, wenn done: true

Über Generatoren iterieren

Generatoren = spezielle Form von Iteratoren

lassen sich auch direkt als Eingabe für for-of-Schleife verwenden

ist auch möglich, Generatoren für unendliche Sequenzen zu generieren

yield-Operator kann prinzipiell an beliebigen Stellen innerhalb der Funktion auftauchen - bsw. innerhalb einer unendlichen Schleife… würde dann endlos laufen...

Generatoren mit Parametern steuern

- kann hilfreich sein, Generator von außen beeinflussen zu können
- bsw. um zu steuern, wann er beendet werden soll
- dafür kann an next() Parameter übergeben werden, der innerhalb des Generators als Eingangswert der yield-Anweisung verwendet wird