Key Collections: Ein Baukasten für Java Collections

Collections aus dem Baukasten

Jede Applikation benötigt Daten. Diese werden mit dem (Java) Collections Framework verwaltet, das deshalb zu den wichtigsten Teilen des JDK gehört. Es definiert Interfaces und stellt Klassen mit Standardimplementierungen bereit. Allerdings kommt es häufig vor, dass man keine Klasse findet, die genau den Anforderungen entspricht. Entweder akzeptiert man diese Lücke, schließt sie mit Programmieraufwand oder zieht Collections aus zusätzlichen Bibliotheken hinzu. Viel besser wäre es doch, wenn man die benötigten Collections selbst wie aus einem Baukasten zusammenstellen könnte. Genau diese Funktionalität stellt die Brownies Collections Library mit den Key Collections zur Verfügung.

von Thomas Mauch

Der Artikel „High-Performance Lists für Java“ in der letzten Ausgabe des Java Magazins hat mit *GapList* und *BigList* Alternativen zu den List-Implementierungen des JDK vorgestellt, die in allen Fällen schnell sind und gut skalieren. In diesem Artikel stellen wir mit den Key Collections den zweiten Hauptteil der Brownies Collections Library [1] vor. Sie erlauben es, Collections mit konfigurierbarer Funktionalität zur Laufzeit zu erzeugen.

Bereits im ersten Artikel haben wir das spartanische API des List-Interface bemängelt. Es gibt aber noch weitere Kritikpunkte am Java-Collections-Framework: So decken die zur Verfügung gestellten Interfaces und Klassen nur die einfachsten Anwendungsfälle ab. Neben dem *List*-Interface gibt es nur noch *Set* und *Queue*/*Deque*, die ebenfalls von *Collection* erben, und getrennt von dieser Hierarchie noch *Map* – komplexere Datenstrukturen bleiben außen vor. Wenn man solche verwenden will, muss man sie entweder selbst aus den bestehenden zusammenbauen oder zusätzliche Libraries wie z. B. Googles Guava [2] einsetzen. Guava fügt dann beispielsweise die Interfaces *Multiset*, *Multimap* und *BiMap* mit entsprechenden Implementierungen hinzu, die weitere häufige Anwendungsfälle abdecken.

Allerdings wird man so mit einer Vielzahl von Interfaces und Klassen konfrontiert, die neu, unterschiedlich und zu erlernen sind, bevor die volle Funktionalität genutzt werden kann. Und da sich die Klassen typischerweise nicht konfigurieren lassen, findet man dann vielleicht doch wieder bloß eine Klasse, die den gewünschten Anwendungsfall nur fast, aber eben nicht ganz abdeckt. Viel besser wäre es doch, wenn man die benötigten Collections selbst wie aus einem Baukasten zusammenstellen könnte. Genau diese Funktionalität stellt die Brownies Collections Library mit den Key Collections zur Verfügung. Die Vielfältigkeit wird durch die Integration der Konzepte von Keys und Constraints ermöglicht, die durch ihre Mächtigkeit die Produktivität beim Entwickeln um ein Vielfaches erhöhen kann.

Artikelserie

Teil 1: Brownies Collections: GapList und BigList
Teil 2: Key Collections

Erstes Beispiel

Am einfachsten wird die Nützlichkeit der Funktionalität mit einem Beispiel erklärt: Wir wollen in unserem Programm die Metainformationen zu den Spalten einer Datenbanktabelle repräsentieren, ähnlich wie sie beispielsweise *java.sql.ResultSetMetaData* bereitstellt, aber es sollen ebenfalls Änderungen unterstützt werden. Damit ergeben sich folgende Anforderungen:

- Spalten haben eine explizite Ordnung, wie sie z. B. für eine Abfrage mit *select ** verwendet wird.
- Die Spaltennamen einer Tabelle müssen einzigartig sein, doppelte Namen müssen also verhindert werden.
- Effizienter Zugriff auf die Spalteninformationen soll sowohl über die Position der Spalte in der Tabelle als auch über den Spaltennamen möglich sein, analog wie *java.sql.ResultSet* den Zugriff auf Daten mit *getObject(int)* und *getObject(String)* erlaubt.

Wenn wir überlegen, welche *Collection* wir für diese Aufgabe wählen sollen, stellen wir fest, dass uns das JDK keine optimale Lösung bietet. Eine *List* erlaubt nur effizienten Zugriff über die Position, nicht aber über den

Namen, während es bei einer *Map* gerade umgekehrt ist. Nur wenn man davon ausgehen kann, dass die Liste immer wenig Elemente enthält, können wir den Zugriff über den Namen durch Iteration realisieren. Eine Tabelle wird zwar nie eine wirklich riesige Menge von Spalten haben, aber das Iterieren wird bereits bei tausend Einträgen nicht mehr wirklich effizient sein. Um eine wirklich skalierbare Lösung zu erhalten, müssen wir die *List* deshalb mit einer *Map* synchronisieren. Dies ist keine unmögliche Aufgabe, aber doch jede Menge Arbeit für eine eigentlich alltägliche Anforderung. Mithilfe der Key Collections hingegen können wir eine Datenstruktur mit den gewünschten Eigenschaften mit einem einzigen Aufruf erzeugen (Listing 1).

Um die gewünschte Funktionalität zu erreichen, nutzt das Beispiel sowohl Keys als auch Constraints. Diese beiden Konzepte wollen wir nun vorstellen.

Constraints

Ein Constraint auf einer Collection definiert, welche Bedingungen Elemente erfüllen müssen, damit sie in der Collection enthalten sein können. Beispiele für Constraints sind also, dass Elemente nicht null sein oder dass gespeicherte Strings nur Großbuchstaben enthalten dürfen. Zwar haben auch gewisse JDK-Klassen implizite Constraints, z. B., dass ein *Set* keine Duplikate enthalten kann oder dass *ArrayDeque* keine Nullwerte erlaubt, aber diese Einschränkungen sind fix und können nicht nach Bedarf geändert werden.

Will man konfigurierbare Constraints mit den JDK-Klassen realisieren, stellt man schnell fest, dass die Klassen nicht für solche Erweiterbarkeit entwickelt wurden: Um die erlaubten Elemente von *ArrayDeque* zu beschränken, müsste man z. B. alle Methoden, die das Hinzufügen von Elementen erlauben (*add/addLast/addFirst, offer/offerFirst/offerLast, addAll*), einzeln überschreiben, und kann dies nicht an einem zentralen Ort tun.

Weshalb sind aber Constraints so wichtig? Constraints sind nicht nur zur Datenhaltung selbst wichtig, sondern vor allem ein zentraler Baustein, um ein mächtiges API zur Verfügung stellen zu können. Gehen wir wieder zurück zu unserem Beispiel mit den Datenbankspalten und betrachten die Beispielklasse *Table*:

```
class Table { List<Column> columns; }
```

Welches API stellen wir den Nutzern der Klasse zur Verfügung, um die Spaltendefinition anlegen oder ändern zu können? Klar ist, dass die einfache Methode

```
List<Column> getColumns { return columns; }
```

gefährlich ist, da wir damit nicht garantieren können, dass die auf der zurückgegebenen Liste durchgeführten Änderungen die definierten Constraints erfüllen. Mit einer leicht angepassten Methode können wir zumindest den Lesezugriff realisieren:

```
List<Column> getColumns { return Collection.unmodifiableList(columns); }
```

Das API für Änderungen fehlt uns aber noch immer. Dazu bleiben zwei Möglichkeiten: eine ineffiziente für den faulen Programmierer, die nur eine einzige Methode zur Verfügung stellt:

```
void setColumns(List<Column> columns) {
  check(columns); this.columns = new ArrayList(columns);
}
```

Das Problem mit diesem Ansatz ist, dass wir bei jedem Aufruf die Constraints für alle Elemente wieder prüfen müssen, auch wenn vielleicht nur eine einzige Änderung gemacht wurde. Dieser Ansatz skaliert damit nicht wirklich gut.

Damit bleibt nur noch die andere, aufwändigere Möglichkeit: Wir stellen für jede Art von Änderung, die durchgeführt werden können muss, eine separate Methode zur Verfügung, wie z. B.:

```
void addColumn(Column column) {
  check(column); columns.add(columns);
}
```

Für ein vollständiges API brauchen wir dann aber auch noch die Methoden *setColumn*, *removeColumn*, vielleicht noch *addColumn* an einer angegebenen Position, und auch das Löschen von allen Spalteninformationen sollte möglich sein. Wie man sieht, artet das einfache API schnell in Fleißarbeit aus – oder aber das API bleibt unvollständig und ist damit unhandlich zu bedienen. Aber auch wenn diese Arbeit machbar ist, sollte man sich doch wieder die Frage stellen, ob es dafür keine einfachere Lösung gibt.

Tatsächlich können wir alle Anforderungen durch Einsatz der Key Collections mit der einfachstmöglichen Implementierung realisieren, da dann die Collection selbst sicherstellt, dass alle gespeicherten Elemente die Constraints jederzeit erfüllen:

Listing 1

```
// Definition of type Column
class Column {
  String name;
  String type;
  // Add constructor / getters / setters
}

// Create list with columns
Key1List<Column,String> cols = new Key1List.Builder<Column,String>.
  withKey1Map(Column::getName).withKey1Null(false).withKey1Duplicates
                                                    (false).build();

// Populate and query list
cols.add(new Column("name", "varchar"));
Column col = cols1.getByKey1("name");
```

```
class Table {
  Key1List<Column,String> columns = ...;
  List<Column> getColumns { return columns; }
}
```

Keys

Ein Key ist ein Wert, der von einer Funktion aus einem in einer Collection gespeicherten Element bestimmt wird. Wie man in Listing 1 sehen kann, speichern die Key Collections deshalb immer Elemente – dies als Gegensatz zum JDK-*Map*-Interface, das Elemente mit externen Keys assoziiert. Der Nachteil der JDK-Entscheidung ist, dass *Map* deshalb komplett getrennt von allen anderen Interfaces ist und nicht einmal das Basisinterface *Collection* erweitert. Schaut man sich dann aber Code an, der mit Maps arbeitet, stammt der Key in der Mehrzahl der Fälle aus dem Element selbst, d. h., die Map wird mit Aufrufen wie

```
map.put(elem.getName(), elem);
```

befüllt. So gesehen kann man sagen, dass sich die Key Collections auf den Normalfall konzentrieren, wenn sie grundsätzlich nur die Elemente selbst speichern und die Keys bei Bedarf mit Funktionen aus den gespeicherten Elementen bestimmen. Diese Entscheidung stellt auch keine wirkliche Einschränkung dar, denn sollte der Key einmal nicht in der *Element*-Klasse selbst vorhanden sein, kann dies durch die Definition einer Hilfsklasse einfach aufgefangen werden:

```
class Entry { String key; Column column; }
```

Wir bezeichnen die von der Funktion definierten Werte als *Key Map*. Wie auch Einträge in einer *Map* oder einem *Set* sollten die als Keys genutzten Werte grundsätzlich immutable sein, d. h., nach dem Hinzufügen des Elements zur Collection dürfen sich diese nicht mehr ändern.

Während Listing 1 eine Collection mit einem definierten Key zeigt, kann es pro Collection auch keine oder zwei Key Maps geben. Auch das Element selbst kann als Key genutzt werden; in diesem Fall sprechen wir von einem *Element Set*.

Das Beispiel zeigt auch, dass die Keys einer Collection nicht nur für den effizienten Zugriff auf die Elemente, sondern auch wieder für die Definition von Constraints verwendet werden können. Für jede Key Map oder das Element Set können folgende Eigenschaften festgelegt werden:

- **Nullwerte:** Erlaubt oder verboten.
- **Duplikate:** Erlaubt oder verboten. Ebenfalls kann spezifiziert werden, dass zwar Nullwerte mehrmals vorkommen können, andere Werte aber einzigartig sein müssen.
- **Sortierung:** Sortiert oder nicht. Beim Sortieren kann entweder die natürliche Ordnung der Klasse verwendet oder explizit ein eigener *Comparator* angegeben werden. Wenn eine Key Map sortiert ist, kann ebenfalls festgelegt werden, dass diese Reihenfolge auch für die Elemente selbst verwendet werden soll, wodurch wir eine sortierte Collection erhalten.

Mit Kenntnissen relationaler Datenbanken werden einem diese Konzepte bekannt vorkommen. So haben wir in unserem Beispiel eigentlich den Spaltennamen als Primary Key der Collection definiert. Und analog zum Primary Key in einer Datenbanktabelle garantiert der definierte Constraint, dass die zu speichernden Elemente korrekt sind und erlaubt einen effizienten Zugriff über den Constraint Key. Auch andere Konzepte aus der Datenbankwelt, wie Check Constraints oder Triggers, die für das manuelle Prüfen von Elementen nützlich sein können, haben ihre Entsprechung in den Key Collections.

Beispiele

Doch nun genug der Theorie. Anhand von Beispielen wollen wir nun die Möglichkeiten aufzeigen, die sich aus den besprochenen Konzepten ergeben.

Die Key Collections implementieren das *Collection*- oder das *List*-Interface. Durch die Kombination mit Anzahl der genutzten Keys erhalten wir sechs Klassen, die Tabelle 1 im Überblick zeigt.

Um die Vielzahl der Optionen vernünftig unterstützen zu können, geschieht das Anlegen von Key Collections immer über ein Builder-Pattern. Das zeigt unser erstes Beispiel, das eine einfache Integer-Liste ohne Constraints und Keys erzeugt:

```
KeyList<Integer> list = new KeyList.Builder<Integer>().build()
```

Die erzeugte *KeyList* verhält sich im Wesentlichen wie eine *GapList* und bietet über das *IList*-Interface auch alle bekannten Methoden an. Im Gegensatz zu einer *GapList* kann die *KeyList* nun aber beim Erstellen nach Belieben konfiguriert werden, indem wir z. B. die erlaubten Elemente mit einem Constraint einschränken:

Tabelle 1: Übersicht über die Key-Collections-Klassen

Klasse	Beschreibung
KeyCollection<E> Key1Collection<E,K1> Key2Collection<E,K1,K2>	Die Klassen *KeyCollection*, *Key1Collection* und *Key2Collection* implementieren das *Collection*-Interface. Die Reihenfolge der Elemente ist per Default nicht definiert, sie kann aber von einer der Key Maps bestimmt werden.
KeyList<E> Key1List<E,K1> Key2List<E,K1,K2>	Die Klassen *KeyList*, *Key1List* und *Key2List* implementieren das *List*-Interface, d. h., die Reihenfolge der Elemente wird immer durch die Liste bestimmt. Es kann aber festgelegt werden, dass die Reihenfolge der Liste der Reihenfolge einer Key Map entspricht, wodurch eine sortierte Liste entsteht.

- Eine Integer-Liste, die keine Nullwerte zulässt:
```
new KeyList.Builder<Integer>().withElemNull(false).build()
```
- Eine Integer-Liste, die keine Nullwerte und nur positive Zahlen zulässt:
```
new KeyList.Builder<Integer>().withElemNull(false).withConstraint(i -> i>=0).
                                                                    build()
```

Wenn nun versucht wird, ein Element hinzuzufügen, das die definierten Bedingungen nicht erfüllt, wird eine Exception geworfen. Neben den Elementen selbst kann auch die Anzahl der erlaubten Elemente eingeschränkt werden:

- Eine Liste mit einer fixen maximalen Anzahl von Elementen, d. h. das Hinzufügen eines Elements wird fehlschlagen, wenn die Liste bereits voll ist:
```
new KeyList.Builder<Integer>().withMaxSize(10).build()
```
- Eine Liste mit einer maximalen Anzahl von Elementen, die als rollendes Fenster organisiert sind, d. h. wenn die Liste voll ist, wird das Hinzufügen eines neuen Elements automatisch das erste Element aus der Liste entfernen, damit die festgelegte Größe nicht überschritten wird:
```
new KeyList.Builder<Integer>().withWindowSize(10).build()
```

Wenn komplexere Bedingungen geprüft werden müssen, die nicht nur vom Element selbst oder von den sich bereits in der Collection befindlichen Elementen abhängt, kann dies analog der Datenbankwelt mit Triggern gelöst werden.

- Eine Liste, die vor dem Einfügen eines Elements eine Trigger-Funktion aufruft. Diese Funktion wird als *Consumer*-Interface definiert und so dem Trigger übergeben:
```
new KeyList.Builder<Integer>().withBeforeInsert(...).build()
```

Die Funktionen *withBeforeInsert* und *withBeforeDelete* erlauben das explizite Prüfen von Elementen, bevor eine Änderung durchgeführt wird. Wird in der angegebenen Funktion eine Exception geworfen, wird die Operation abgebrochen. Die Funktionen *withAfterInsert* und *withAfterDelete* werden entsprechend nach dem Durchführen der Operation aufgerufen.

Ebenfalls kann die zur Speicherung der Elemente verwendete Datenstruktur angepasst werden, sodass Speicherverbrauch und Performance für jeden Fall optimiert werden können:

- Integer-Werte werden in einer *GapList* gespeichert:
```
new KeyList.Builder<Integer>().build()
```
- Integer-Werte werden in einer *BigList* gespeichert:
```
new KeyList.Builder<Integer>().withElemBig(true).build()
```
- Integer-Werte werden in einer *IntObjGapList* gespeichert:
```
new KeyList.Builder<Integer>().withElemClass(int.class).build()
```
- Integer-Werte werden in einer *IntObjBigList* gespeichert:
```
new KeyList.Builder<Integer>().withElemBig(true).withElemClass(int.class).
                                                                    build()
```

Alle offerierte Funktionalität ist bisher alleine auf der zugrunde liegenden Listenstruktur aufgebaut. So kann zwar die Standardfunktion *contains* verwendet werden, die Implementierung ist aber langsam, da zur Ausführung über alle Elemente der Liste iteriert werden muss. Diese Operation kann beschleunigt werden, indem die *KeyList* angewiesen wird, neben der eigentlichen Liste auch noch ein Element Set zu führen – eine Integer-Liste mit einem Element Set für schnelle Zugriffe:

```
new KeyList.Builder<Integer>().withElemSet().build()
```

Damit sind nun auch Operationen wie *contains* oder *remove* sehr schnell, da diese über eine intern angelegte *Map* ausgeführt werden. Als einzige Einschränkung ist das Entfernen von Elementen über den Key bei Listen langsam, da das Element in der Liste durch Iterieren gesucht werden muss. In größeren Listen sollten deshalb Elemente immer über den Index gelöscht werden – oder man verwendet sortierte Listen. Die Map erlaubt auch eine effiziente Prüfung auf Duplikate – eine Integer-Liste, die keine Duplikate zulässt:

```
new KeyList.Builder<Integer>().withElemDuplicates(false).build()
```

Während die Key Maps normalerweise nicht sortiert sind, kann auch dies angepasst werden – eine Integer-Liste mit einer sortierten Key Map:

```
new KeyList.Builder<Integer>().withElemSort(true).build()
```

Durch das Sortieren des Element Set wird die Reihenfolge der Liste als solche noch nicht beeinflusst. Man kann die Collection aber so konfigurieren, dass eine sortierte Liste entsteht – eine sortierte Integer-Liste:

```
new KeyList.Builder<Integer>().withElemSort(true).withElemOrderBy(true).build()
```

Zu sortierten Lists gibt es zahlreiche Diskussionen [3], weshalb Java diese nicht kennt. Es gibt zwei Hauptargumente, die dagegen sprechen: Das erste, puristische Argument ist, dass die *add*-Funktion den Contract des *List*-Interface verletzt, da das Element nicht am Ende der Liste hinzugefügt wird. Das zweite, praktische Argument ist, dass es nicht möglich ist, eine sortierte Liste zu implementieren, in der Elemente in zufälliger Reihenfolge effizient hinzugefügt werden können (da in diesem Fall immer Elemente verschoben werden müssen, wodurch die Performance leidet). Während beide Argumente eine gewisse Berechtigung haben, kann eine sortierte Liste dennoch nützlich und – wenn sie richtig genutzt wird – auch effizient sein.

Nun erweitern wir das Beispiel um die Definition eines Keys, wie wir ihn bereits in Listing 1 gesehen haben – eine Liste mit Spaltenelementen und Zugriff über den Spaltennamen:

```
new Key1List.Builder<Column,String>.withKey1Map(Column::getName).build()
```

Für die definierten Keys stehen im Wesentlichen dieselben Möglichkeiten zur Verfügung wie für das Element Set. So definiert z. B. die Methode *withKey1Null* einen Constraint auf dem definierten Key, wie *withElemNull* einen Constraint auf dem Element selbst definiert hat – eine Liste mit Spaltenelementen mit einzigartigen Namen und Zugriff über den Spaltennamen:

```
new Key1List.Builder<Column,String>.withKey1Map(Column::getName).
                    withKey1Null(false).withKey1Duplicates(false).build()
```

Da das Definieren eines solchen Primary Keys eine häufige Anforderung ist, gibt es als Shortcut die Methoden *withPrimaryKey1* und *withUniqueKey1*, die Nullwerte und Duplikate in einem Aufruf definieren. Damit vereinfacht sich auch die Definition der Liste – eine Liste mit Spaltenelementen mit einzigartigen Namen und Zugriff über den Spaltennamen:

```
new Key1List.Builder<Column,String>().withKey1Map(Column::getName).
                                        withPrimaryKey1().build()
```

Während wir bisher nur die Funktionen zum Definieren der Key Collections untersucht haben, stellen wir nun noch die Funktionen zum Arbeiten mit den erstellten Datenstrukturen vor.

Für den Zugriff auf das Element Set können neben der Standardfunktion *contains* die neuen Methoden *getAll*, *getCount*, *getDistinct* und *removeAll* genutzt werden.

Für das Arbeiten mit Key Maps gibt es die Methoden *containsKey*, *getByKey*, *getAllByKey*, *getCountByKey*, *getDistinctKeys*, *removeByKey*, *removeAllByKey* und *indexOfKey* (nur für Listen).

Ebenfalls kann mithilfe der Methoden *asSet* und *asMap* über die JDK-Standardinterfaces auf Element Set und Key Maps zugegriffen werden.

Dadurch, dass sich die aufgezeigten Möglichkeiten beliebig kombinieren lassen, kann man mit den Key Collections maßgeschneiderte Collections für alle Anwendungsfälle erzeugen. Listing 2 enthält weitere Beispiele. Unter anderem wird gezeigt, wie man mit wenigen Codezeilen eine *BiMap* erstellen kann, also eine *Map*, wo man sowohl effizient über den Key auf den Wert als auch umgekehrt über den Wert auf den Key zugreifen kann. Weitere Beispiele finden sich auch in der Dokumentation der Library [4].

Es gibt einen Spezialfall, der nicht orthogonal auf alle Key Collections angewendet werden kann, sondern nur von *KeyCollection* unterstützt wird. Es handelt sich dabei um die Möglichkeit, ein *Multiset* zu realisieren, in dem identische Elemente nur einmal zusammen mit einem Zähler für ihr Vorkommen gespeichert werden

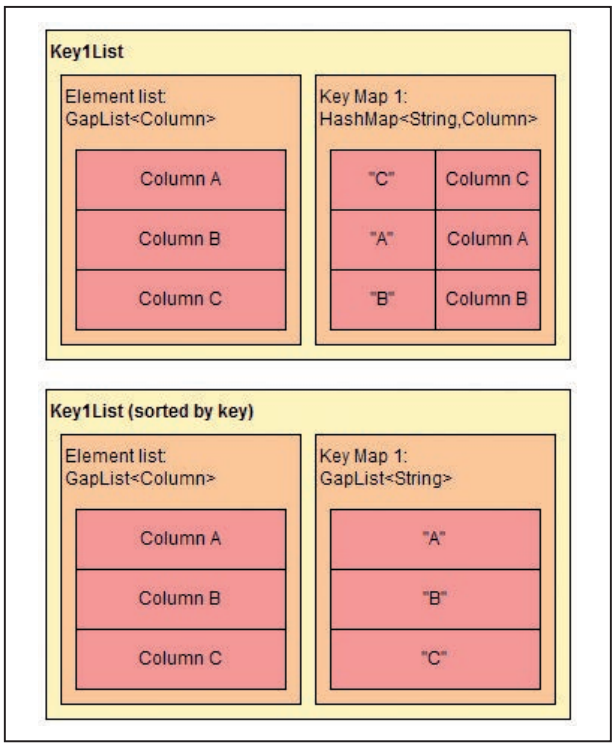

Abb. 1: Datenspeicherung in „Key1List“ (unsortiert und sortiert)

– ein *Multiset* von Strings, in dem für jeden String die Häufigkeit gespeichert wird:

```
new KeyCollection.Builder<String>.withElemCount().build()
```

Implementierung

Wir wollen auch noch einen Blick hinter die Kulissen werfen und sehen, wie die Key Collections die vorgestellte Funktionalität implementieren. Vielleicht erinnern wir uns noch daran, wie wir bei der Vorstellung des ersten Beispiels erwähnt haben, dass man für eine skalierende Lösung eine *List* mit einer *Map* synchronisieren muss – genau das tun die Key Collections bei Bedarf für uns.

Abbildung 1 zeigt, wie eine *Key1List* je nach Anforderung, ob die Liste sortiert sein soll oder nicht, aus unterschiedlichen *Collection*-Klassen zusammengestellt wird, die die gewünschte Funktionalität effizient implementieren. Key Maps und Element Set werden typischerweise mit *HashMap/TreeMap* realisiert, bei sortierten Listen wird auch die Key Map, die die Sortierreihenfolge vorgibt, als *List* gespeichert. Die Elemente der Liste selbst werden per Default in einer *GapList* gespeichert, bei Bedarf kann aber, wie gesehen, auch eine *BigList* oder eine primitive Wrapper-Klasse wie *IntObjGapList* oder *IntObjBigList* verwendet werden.

Fazit

Die Key Collections erweitern die Möglichkeiten der Java Collections auf eine faszinierende Art und Weise. Die einfache Definition von mächtigen Datenstrukturen, denen die gewünschte Funktionalität deklarativ zur Laufzeit zugewiesen werden kann, erlaubt ein präzises Abbilden des Datenmodells.

Die Brownies Collections Library erhöht damit die Effizienz beim Entwickeln und ermöglicht dem Entwickler mit geringem Programmieraufwand, Applikationen zu erstellen, die in jeder Situation effizient bezüglich Performance und Speicherplatzverbrauch sind und dadurch gut skalieren.

Thomas Mauch arbeitet als Softwarearchitekt bei Swisslog in Buchs, Schweiz. Seine Schwerpunkte liegen im Bereich Java und Datenbanken. Er interessiert sich seit den Zeiten des C64 für alle Aspekte der Softwareentwicklung.

Links & Literatur

[1] http://www.magicwerk.org/collections

[2] https://code.google.com/p/guava-libraries/

[3] http://stackoverflow.com/questions/8725387/why-there-is-no-sortedlist-in-java

[4] http://www.magicwerk.org/page-collections-examples.html

Listing 2

```
// Eine nach Namen sortierte File-List mit Namen als
// Primary Key
Key1List<File,String> coll1 =
  new Key1List.Builder<File,String>().
  withKey1Map(File::getName).
  withPrimaryKey1().withKey1OrderBy(true).build();

// Eine Collection mit 2 Keys, wobei 1 Key optional
// ist
Key2Collection<Ticket,String,String> coll2 =
  new Key2Collection.Builder<Ticket,String,String>().
  withKey1Map(Ticket::getId).withPrimaryKey1().
  withKey2Map(Ticket::getExtId).withUniqueKey2().
                                        build();
```

```
// Eine BiMap, welche Zip-Codes verwaltet, wobei
// alle Key Maps sortiert sind
class Zip {
  int code;
  String city;

  Zip(int code, String city) {
    this.code = code;
    this.city = city;
  }
  int getCode() {
    return code;
  }
  String getCity() {
    return city;
  }
}

void useBiMap() {
  Key2Collection<Zip, Integer, String> zips =
    new Key2Collection.Builder<Zip, Integer,
                                        String>().
    withKey1Map(Zip::getCode).withKey1Sort(true).
    withKey2Map(Zip::getCity).withKey2Sort(true).
                                        build();

  zips.add(new Zip(1000, “city1000”));
  String city = zips.getByKey1(1000).getCity();
  int code = zips.getByKey2(“city1000”).getCode();
}
```