Ruim tien jaar geleden schreef Joshua Bloch het Collections Framework. De drie belangrijke concepten waren toen: Interfaces die verzamelingen definiëren, implementaties van de interfaces en algoritmes om de collections te manipuleren. Bloch had het idee dat het framework zou worden uitgebreid met een veel rijkere set aan interfaces, algoritmes en implementaties. Nu hebben Kevin Bourrillion en Jared Levy van Google een uitbreiding op het framework ontwikkeld met de naam Guava.

Guava bibliotheek onder de loep

Google Core Libraries voor Java 1.5+

Voordat we naar Guava kunnen kijken, moeten we het eerst downloaden. Via de link http://guava-libraries.googlecode.com kom je op de website van het project. Daar kun je een bestand downloaden van rond de 3Mb die de binary en sources bevat. De eigenlijke bibliotheek bestaat uit het bestand guava-r07.jar (van rond de 1Mb); deze is overigens ook via de maven repository beschikbaar. Op de website staat ook nog een verwijzing naar de uitgebreide en zeer complete JavaDoc.

Guava is een samenvoeging van de eerdere bibliotheek: Google (concurrent) Collections en meer algemene utility classes voor primitive collections, network en I/O. De bibliotheek valt onder de Apache License 2.0 en bevat de volgende onderdelen:

- **com.google.common.collect**: bevat de interfaces/implementaties/utility functies die dienen als uitbreiding op de bestaande Java collections.
- **com.google.common.base**: bevat de basis utility libraries en interfaces zoals Predicates en Preconditions.
- **com.google.common.io**: bevat utility functies en classes voor gebruik in combinatie met java I/O, bijvoorbeeld input streams, output streams, readers, writers en files.
- **com.google.common.net**: bevat utility functies voor gebruik in combinatie met numerieke IP adressen en domeinnamen om bijvoorbeeld te controleren of een domeinnaam syntactisch correct is.
- **com.google.common.primitives**: bevat utility functies voor het gebruik van arrays bestaande uit primitieven.

Collect Package

De collect package is de grootste en veruit de meest interessante package. Bijna alle classes en interfaces die met collections te maken hebben zitten in deze package. In onderstaande afbeelding staan een aantal belangrijke interfaces uit deze package. Een aantal overige classes in deze package zijn:

- PeekingIterator: Een iterator waarmee je 1 element vooruit kunt kijken;
- Interner: Een Interface waarmee je immutable classes kunt "poolen" zodat elke unieke instantie maar één keer in het geheugen wordt geladen;
- Constraints en MapConstraints: Factories waarmee collections kunnen worden geforceerd om aan bepaalde voorwaarden te voldoen.

Hierna volgen een aantal voorbeelden van het gebruik van de classes en interfaces in de collect package. Omdat in dit artikel niet alle classes en interfaces kunnen worden besproken zijn de voorbeelden beperkt tot Multimap, MultiSet, BiMap en MapDifference.

MultiMap, een Map waabij de value een Collection van elementen is. Handig om bijvoorbeeld de verkochte items van een verkoper in op te slaan. Zonder com.google.common.collect.

Rino Kadijk
is Java-ontwikkelaar bij Qiy.

Multimap<K,V>:

```
public Sale getBiggestSale() {
  Sale biggestSale = null;
  for (List<Sale> sales : map.values()) {
    Sale biggestSaleForSalesman
     = Collections.max(sales, SALE_COST_COMPARATOR);
    if (biggestSale == null
      || biggestSaleForSalesman.getCharge() >
                        biggestSale().getCharge()) {
      biggestSale = biggestSaleForSalesman;
    }
  }
  return biggestSale;
}
```

Met com.google.common.collect.Multimap<K,V>:

```
public Sale getBiggestSale() {
  return Collections.max(multimap.values(), SALE_COST_
  COMPARATOR);
}
```

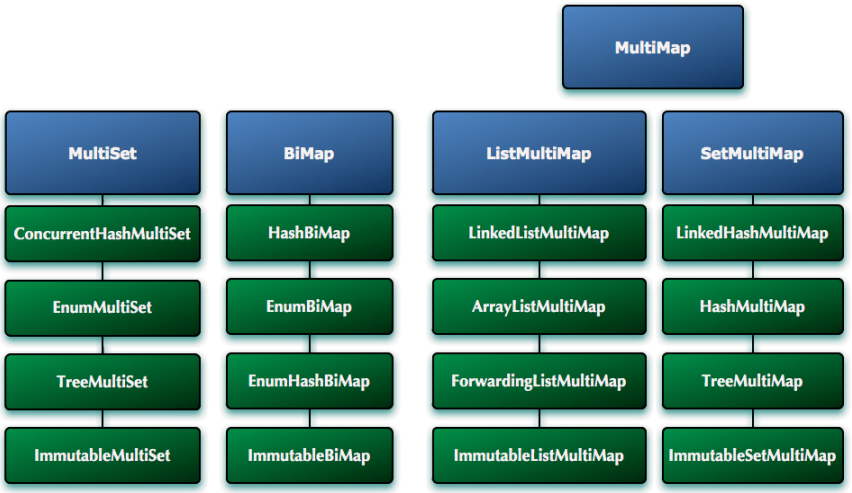
De twee voorgaande codevoorbeelden (zie referentie [1]) hebben dezelfde uitkomst, namelijk het product dat het meest heeft gekost. De values() methode van de multimap geeft een Collection<V> van alle values in de hele map waar vervolgens de hoogste waarde uit wordt gehaald. De Multimap abstractie uit de Google Collections brengt beter tot uiting wat de bedoeling is en zorgt dat de methode getBiggestSale een stuk eenvoudiger en leesbaarder is. Ook is de methode die de multimap gebruikt minder foutgevoelig, omdat deze logica niet zelf hoeft te worden geschreven.

MultiSet, een Set waarbij ieder uniek element meerdere keren voor mag komen en het aantal voorkomens van een uniek element wordt bijgehouden. Handig voor bijvoorbeeld het bijhouden van het aantal unieke bezoekers.

```
Multiset<InternetDomainName> visitedDomains =
 HashMultiset.create();
visitedDomains.add(InternetDomainName.from("www.qiy.
nl"));
visitedDomains.add(InternetDomainName.from("www.google.
com"));
visitedDomains.add(InternetDomainName.from("www.google.
com"));
System.out.println("Hit count for google.com: " +
 visitedDomains.count(InternetDomainName.from("www.
google.com")));
```

In bovenstaand voorbeeld wordt een HashMultiSet aangemaakt door de create methode om het aantal bezoeken aan een domein bij te houden. De create() functie heeft geen parameters nodig om een generieke Multiset aan te maken. De broncode blijft daardoor korter en beter leesbaar. Bij de tweede aanroep van de add methode met de domeinnaam www.google.com is de set van domeinnamen nog even groot. Het verschil met een java.util.Set is dat de count wordt opgehoogd op het moment dat een element al aanwezig is in de Set.

BiMap, een Map waarbij de mapping van sleutels naar waarden met één methode kan worden omgekeerd. Handig voor als je bijvoorbeeld van RGB waarde naar kleurnaam wilt, of omgekeerd.



Afbeelding 1: een subset van com.google.common.collect.

```
BiMap<Color, String> colorMap = HashBiMap.create();
colorMap.put(Color.RED, "Rood");
colorMap.put(Color.BLACK, "Zwart");
System.out.println(colorMap.inverse());

Resultaat:
{Zwart=java.awt.Color[r=0,g=0,b=0], Rood=java.awt.
Color[r=255,g=0,b=0]}
```

In de bovenstaande BiMap worden één-op-één relaties gelegd tussen kleuren en de bijbehorende Nederlandse vertaling. De BiMap kan eenvoudig worden omgekeerd door de inverse() methode aan te roepen. Dit zorgt er voor dat alle sleutels worden omgewisseld met de gerelateerde waarden. Dat impliceert dat twee achtereenvolgende aanroepen van de inverse() methode resulteert in de originele BiMap. Als de waarden in de BiMap niet uniek zijn, dan resulteert een aanroep van de inverse() methode in een IllegalArgumentException.

MapDifference, een representatie van een doorsnede van twee maps, vergelijkbaar met een left/right/inner-join in SQL. Handig bijvoorbeeld om configuratie settings van twee gebruikers te vergelijken.

```
ImmutableMap<String, Integer> map1 = ImmutableMap.of("1"
, 1);
ImmutableMap<String, Integer> map2 = ImmutableMap.of("1"
, 1, "2", 2);
MapDifference<String, Integer> difference = Maps.
difference(map1, map2);
System.out.println(difference.entriesInCommon());

Resultaat:
{1=1}
```

Het bovenstaande voorbeeld maakt twee immutable maps aan die beide het element met de sleutel '1' bevatten. Via de methode Maps.difference() wordt een object aangemaakt waarmee de verschillen en overeenkomsten kunnen worden opgevraagd. De laatste regel uit het voorbeeld print het eerste element, omdat het element in beide maps aanwezig is.

De BiMap kan simpel worden omgekeerd door de inverse() methode aan te roepen.

In 95% van de gevallen is een null waarde in een collection onwenselijk.

Primitive Collections

De primitives package bevat statische methoden om berekeningen of transformaties op verzamelingen van primitieven toe te passen. De methoden zijn ingedeeld op basis van de naam van de primitieve. Zo kan voor een byte[] de class Bytes worden gebruikt en voor een double[] de class Doubles. De meeste methoden in de classes bieden dezelfde functionaliteit zoals die aanwezig is in de collections van java.util. Op een List zou je bijvoorbeeld contains() kunnen aanroepen. Het equivalent van de contains methode voor een double[] is de methode Doubles.contains(double[] array, double target).

Er is voor elk type primitieve een aparte class. Een mooie toevoeging daarop is UnsignedBytes die het sign-bit negeert. Methoden die je in deze package onder andere kunt vinden zijn:

- checkedCast, die een IllegalArgumentException gooit als er een overflow plaatsvindt;
- saturatedCast, die een de maximale waarde teruggeeft als er een overflow plaatsvindt;
- max, om de maximale waarde in een array van het primitive type te vinden;
- join, om bijvoorbeeld een comma-seperated String van een array te terug te geven;
- lexicographicalComparator, die een Comparator teruggeeft waarmee eenvoudig twee arrays kunnen worden vergelijken. Daarbij geldt dat kortere arrays altijd eerder komen dan langere arrays in geval van gelijke getallen en volgorde.

```
UnsignedBytes.join(":", (byte) 192, (byte) 168, (byte) 1
, (byte) 1);
```

Bovenstaand voorbeeld geeft de String '192:168:1:1'. De getallen moeten expliciet worden gecast, omdat de methoden alleen werken voor het type van de class. Dit heeft als voordeel dat de compiler waarschuwt op het moment dat de methode met verkeerde parameters wordt aangeroepen.

Null Hostile

Uit onderzoek onder de ontwikkelaars bij Google blijkt dat in 95% van de gevallen een null waarde in een collection onwenselijk is en zorgt voor onnodig veel extra checks en/of nullpointer fouten. De google collections proberen zoveel mogelijk null waarden te vermijden.

```
ImmutableSet.of("1","2","3","4",null);
```

In bovenstaand voorbeeld wordt geprobeerd om een ImmutableSet van vijf elementen aan te maken. De ImmutableSet zal een exceptie gooien omdat bij het aanmaken van een collection geen null waarden worden toegestaan. Dit is overeenkomstig met de collections die toegevoegd zijn vanaf JDK 1.4. Mocht het toch absoluut een vereiste zijn om een collection met null waarden te maken, dan is het altijd mogelijk om op de Java Collection classes terug te vallen.

De ImmutableList garandeert aan de gebruiker van de List dat deze na creatie niet meer zal worden gewijzigd. Dit is sterker dan het doorgeven van een Collection in een unmodifiable wrapper. Want een unmodifiable wrapper van de List kan niet worden gewijzigd, maar het object dat de List aanbiedt kan dat nog wel! De samenstelling van een ImmutableList is met zekerheid na creatie niet meer te wijzigen en dus zijn er ook sterkere aannames over te doen. Zo kan bijvoorbeeld een reverse view worden teruggeven. Om een ImmutableList deeply immutable te maken wordt aangeraden om enkel Immutable objecten aan deze List toe te voegen.

Immutability

Immutability resulteert naast thread-safety en eenvoudigere redenering, ook in een efficiënter geheugen gebruik. Daarnaast kan in een aantal gevallen snelheid worden gewonnen. Een voorbeeld daarvan zijn de equals en de hashcode van een immutable collectie. Bij het uitvoeren van de equals methode kan de hashCode worden gebruikt om snel false terug te geven als de Sets niet gelijk zijn. Ook kan de hashCode al worden berekend bij het aanmaken van de collectie.

```
ImmutableList<String> list = ImmutableList.<String>
 builder()

.add("A")
.add("B")
.add("BB")
.build()
.reverse();

Iterable<String> filtered = Iterables.filter(list,
         new Predicate<String>() {
           public boolean apply(String input) {
                return input == null ||
                    input.startsWith("B");
                }
         });
```

De ImmutableList in bovenstaand voorbeeld wordt aangemaakt met behulp van een builder. Daarna wordt direct de omgekeerde versie van de lijst gemaakt. Vervolgens kunnen elementen worden gefilterd door een predicaat op te geven. De filter methode van de Iterables class zorgt er voor dat over een lijst kan worden gelopen waarvoor het predikaat geldt.

Een handige feature van de Guava collections is dat ze naast Builders ook factory-methods aanbieden om de collections te creëren. Dat betekent dat je tot vijf parameters type-safe tijdens het creëren aan je collection kunt toevoegen via de of() methode.

```
ImmutableSet.of(); // i.p.v. Collections.emptySet()
ImmutableSet.of("single") // i.p.v. Collections.
singleton("single")
ImmutableSet.of("single", "double")
```

Gebruikers van bibliotheken profiteren van verbeteringen in nieuwe versies. Ook voor het digitale tijdperk was dat al het geval. Tegenwoordig ziet een bibliotheek er al heel anders uit.

Als je later van twee parameters terug wilt naar één parameter dan hoef je niet een speciale methode aan te roepen, maar kun je nog steeds dezelfde factory-method met één parameter gebruiken. Op de achtergrond wordt dan een singletonSet voor je aangemaakt. Hetzelfde geldt voor een emptySet als je geen parameters gebruikt. Wil je meer parameters dan is er een factory-method die als zesde argument een vararg Object accepteert (of je kunt natuurlijk de Builder gebruiken).

Guava vs Apache Commons

De collections van Apache Commons bieden min of meer dezelfde functionaliteit als de Google Collections. De andere projecten Lang, IO, Net en Primitives van Apache Commons zijn waarschijnlijk het best te vergelijken met de packages base, io, net en primitives. De io en net packages van Guava zijn op een aantal plekken nog gemarkeerd als Beta en lijken nog niet zo ver ontwikkeld als de projecten van Apache Commons. Guava biedt speciale Concurrent Collections, maar Apache Commons is nogal vaag over de thread-safety van de collections.

Bijna alles wat met Collections te maken heeft is verzameld in de com.google.common.collect package. Dat heeft als voordeel dat de collections eenvoudiger te vinden zijn. Een van de belangrijkste verschillen is dat de Google Collections generified zijn en daardoor minder fout gevoelig. In het geval van de Google Collections zal de compiler je waarschuwen als er een kans is op fouten. Bij het gebruik van Apache Commons bestaat de kans op runtime fouten en dus is een statische analyse van de broncode noodzakelijker.

Een ander verschil is dat Guava niet afwijkt van de bestaande Collections Framework contracten. Apache Commons heeft bijvoorbeeld bij de Bag Interface staan "(Violation) Adds one copy the specified object to the Bag". De Java.util.Collection.add() methode zou eigenlijk altijd true moeten teruggeven, maar de Bag interface wijkt daar vanaf. Dit kan leiden tot onverwachte resultaten als een ontwikkelaar de Bag benadert als een Collection en niet naar de JavaDoc kijkt.

Conclusie

In dit artikel komt maar een zeer klein gedeelte van Guava aan bod. Als de bibliotheek je interesse heeft dan is de JavaDoc een goed startpunt. Release 7 van Guava biedt een aantal mooie toevoegingen op de bestaande Collections. De bibliotheek is overzichtelijk ingedeeld en de collections proberen zoveel mogelijk null values te vermijden en ondersteunen immutability waar mogelijk. Guava is generified en houdt zich strikt aan de bestaande interfaces van het Collections Framework. En, de abstracties zijn intuïtief en zorgen voor betere leesbaarheid van de broncode.

De bibliotheek is een goede verbetering ten opzichte van Apache Commons Collections. De io en net package van Guava zijn nog gemarkeerd als Beta en daarom misschien minder geschikt voor gebruik in productie. Over de tijd zal de implementatie van Guava waarschijnlijk efficiënter worden. Als gebruiker van de bibliotheek profiteer je mee van de verbeteringen in nieuwere versies. Denk hierbij aan de Collections.sort die zal worden gewijzigd in een TimSort (Zie Referenties [2]). «

Bibliotheek is een goede verbetering ten opzichte van Apache Commons Collections.

Referenties

[1] http://blog.publicobject.com/2007/09/series-recap-coding-in-small-with.html

[2] http://google-collections.googlecode.com/files/collections-connection.pdf

http://code.google.com/p/guava-libraries/

http://code.google.com/p/google-collections/

http://www.slideshare.net/GoogleTecTalks/using-the-google-collections-library-for-java

http://www.devx.com/Java/Article/36183/1954