Пара 3: Популярные библиотеки

1. Введение

Java имеет довольно большую стандартную библиотеку, но вместе с тем она имеет хороший пакетный менеджер и не имеет проблем с совместимостью бинарников (как, например, C++), поэтому появилась куча библиотек и фреймворков сторонних производителей, делающих всё, что угодно. Поскольку воспользоваться ими из любого Java-проекта практически ничего не стоит, все ими активно пользуются. Библиотеки бывают специализированными (типа библиотеки для работы с документами Microsoft Office) и общего назначения, которые, наверное, должны были бы быть частью стандартной библиотеки или даже языка. Многие из библиотек имеют открытый исходный код, многие распространяются по свободным лицензиям — при выборе библиотеки очень важно, чтобы её лицензия удовлетворяла потребностям вашего проекта, обращайте на это внимание.

Искать библиотеки проще всего гуглением, но есть известные места, где можно смотреть новые интересные штуки:

- The Apache Software Foundation[1] — организация, управляющая разработкой кучи интересных библиотек и технологий;
- Awesome Java[2] — список (модерируемый) известных библиотек и программ на Java.

Поскольку всем лень читать про библиотеки самим, кое про что будет у нас на парах по практике. Сегодня речь пойдёт про наиболее популярные библиотеки (самую популярную[3], кстати, вы и так знаете — это jUnit). Следующие по популярности библиотеки — это slf4j и log4j, нужные для логгирования, и guava, библиотека общего назначения, приятное дополнение к стандартной библиотеке. С неё и начнём.

2. Google Guava

Guava[4] — это библиотека, реализующая новые коллекции (например, мультимножество и мультихеш), немутабельные коллекции, функциональные типы, утилиты для ввода-вывода и многопоточности, хеширования, работы со строками и т.д. и т.п., делает программирование на Java более приятным в разных мелочах (так что guava для Java — что-то вроде boost для C++). Правда, она местами идеологически отличается от JDK, например, не

[1] https://httpd.apache.org/
[2] https://github.com/akullpp/awesome-java
[3] http://blog.takipi.com/the-top-100-java-libraries-in-2016-after-analyzing-47251-dependencies/
[4] Первоисточник: https://github.com/google/guava/wiki

гарантируется обратная совместимость (любой класс или метод может быть удалён из библиотеки через два года после предупреждения) и, что важно, Guava не любит null (многие коллекции бросают исключение при попытке добавить null).

2.1. Утилиты

Здесь будут описаны классы и методы общего назначения, полезные во всём коде. Начнём с **Preconditions** (класса `com.google.common.base.Preconditions`) — что-то вроде продвинутого `assert`. Его статические методы рекомендуется импортировать как `import static`. Пример:

```
checkArgument(i >= 0, "Argument was %s but expected nonnegative", i);
checkArgument(i < j, "Expected i < j, but %s > %s", i, j);
```

Это пишется вместо `if (i >= 0) { throw new IllegalArgimentException("Argument was " + i +` что гораздо симпатичнее. Ещё бывают:

Сигнатура	Описание	Бросаемое исключение
checkArgument(boolean)	Проверить условие на аргумент	IllegalArgumentException
checkNotNull(T)	Проверить на не $null$	NullPointerException
checkState(boolean)	Проверить состояние объекта безотносительно параметров (например, состояние итератора)	IllegalStateException
checkElementIndex(int index, int size)	Проверить, что $index$ от 0 до $size - 1$	IndexOutOfBoundsException
checkPositionIndex(int index, int size)	Проверить, что $index$ от 0 до $size$	IndexOutOfBoundsException
checkPositionIndexes(int start, int end, int size)	Проверить, что переданный полуинтервал является валидным подынтервалом для списка заданного размера	IndexOutOfBoundsException

Как видно, методы избыточны и частично перекрывают функциональность друг друга, и вообще, без них легко можно было бы обойтись или написать свои такие же, но в этом весь смысл Guava (по крайней мере, её утилит) — она делает код более простым и понятным, а программирование более удобным.

Следующая полезная штука — **MoreObjects**. Есть ещё класс **Objects**, в котором есть удобные методы для реализации методов `Object`, но его функциональность начиная с Java 7 уже есть в JDK, а вот `MoreObjects` ещё может быть полезен. Пример:

```
// Returns "ClassName{x=1}"
MoreObjects.toStringHelper(this)
    .add("x", 1)
    .toString();

// Returns "MyObject{x=1}"
MoreObjects.toStringHelper("MyObject")
    .add("x", 1)
    .toString();
```

Ещё есть класс **ComparisonChain**, позволяющий удобно реализовать `compareTo()` из стандартного интерфейса `Comparable`:

```
public int compareTo(Foo that) {
  return ComparisonChain.start()
      .compare(this.aString, that.aString)
      .compare(this.anInt, that.anInt)
      .compare(this.anEnum, that.anEnum, Ordering.natural().nullsLast())
      .result();
}
```

Тут ещё полезно хотя бы упомянуть про класс **Ordering**, который используется в этом примере, он нужен для быстрой реализации `Comparator`. В Java 8 появились стримы, так что этот класс теперь не очень полезен.

Класс **Throwables** делает работу с исключениями несколько более удобной. Например, работу с цепочкой вложенных исключений: `List<Throwable> getCausalChain(Throwable throwable)`, `Throwable getRootCause(Throwable throwable)`. И методы для удобного перебрасывания исключений: `void propagateIfPossible(Throwable throwable, Class<X> declaredType)`, `void throwIfInstanceOf(Throwable throwable, Class<X> declaredType)`. Пример:

```
try {
  someMethodThatCouldThrowAnything();
} catch (IKnowWhatToDoWithThisException e) {
  handle(e);
} catch (Throwable t) {
  Throwables.propagateIfPossible(t, OtherException.class);
  throw new RuntimeException("unexpected", t);
}
```

2.2. Коллекции

Для каждой коллекции из стандартной библиотеки и каждой коллекции из Guava есть Immutable-вариант, например, **ImmutableMap**, **ImmutableList** и т.д. Зачем они нужны:

- многопоточность — не может быть гонок;
- эффективность — нет оверхеда на хранение мутабельного состояния и поддержание возможности менять коллекцию;
- хороший стиль — дополнительные ограничения на код, «настоящая» константность.

Обратите внимание, что в JDK есть `Collections.unmodifiable...`, которые очень похожи на немодифицируемые коллекции из Guava, но их проблема в том, что они не более чем вид на исходную коллекцию, так что если исходная коллекция поменялась, то поменялось и содержимое «немутабельной» коллекции. К тому же, это не даёт никаких плюсов в плане

эффективности по времени и по памяти, внутри ведь всё те же модифицируемые коллекции. Guava пошла другим путём — немутабельная коллекция является копией мутабельной, из которой она создана, то есть никогда не меняет своих элементов. Для создания коллекции может применяться метод `copyOf`, например, `ImmutableSet.copyOf(set)`. Можно создать немутабельную коллекцию и напрямую, методом `of`: `ImmutableSet.of("a", "b", "c")` или `ImmutableMap.of("a", 1, "b", 2)`. При этом `copyOf` на самом деле выполняет копирование только тогда, когда это нужно, например, если вы хотите получить представление в виде немутабельного списка немутабельного же множества, то это выполняется за константное время просто созданием вида на множество — оно ведь всё равно немутабельно. Обычно `copyOf` не копирует данные, когда:

- можно создать вид на копируемую коллекцию за константное время (как в примере с немутабельным списком из немутабельного множества);
- это не приведёт к утечкам памяти, например, немутабельный поддиапазон огромного списка не будет держать ссылку на весь огромный список и все элементы, что в нём лежат;
- это не меняет семантику коллекции, например, немутабельное множество из немутабельного сортированного множества получается копированием, потому что в одном случае для элементов используется `equals` и `hashCode`, в другом — компаратор.

Это не то чтобы железные правила, реализация `copyOf` не специфицирована, но важно понимать, что она может быть умнее, чем кажется.

Ещё у всех немутабельных коллекций есть метод `asList()`, который возвращает `ImmutableList`.

Список всех немутабельных коллекций можно посмотреть в документации[5], поэтому вдаваться в подробности здесь не будем, а перейдём к мутабельным.

Multiset — мутабельное мультимножество. Семантика довольно очевидна — это неупорядоченная коллекция возможно повторяющихся элементов. Мы что-то такое писали на одной из контрольных, кстати. Идеологически его можно понимать по-разному: как неупорядоченный `ArrayList<E>` или как `Map<E, Integer>` с элементами и их количеством. Оно работает и так и так — оно реализует Collection (то есть, например, имеет метод add, добавляющий один экземпляр элемента) и реализует операции, ожидаемые от Map (с поправкой на множественность элементов, поэтому интерфейс Map Multiset не реализует), причём с ожидаемой от Map трудоёмкостью. Интересные методы у него такие:

[5] https://github.com/google/guava/wiki/ImmutableCollectionsExplained

Метод	Описание
count(E)	Возвращает количество вхождений элемента
elementSet()	Возвращает множество (нормальное) различных элементов (на самом деле, view)
entrySet()	Возвращает множество объектов Multiset.Entry<E>, у которых можно узнать элемент и число вхождений
add(E, int)	Добавляет указанное количество вхождений указанного элемента
remove(E, int)	Удаляет указанное количество вхождений указанного элемента
setCount(E, int)	Устанавливает количество вхождений заданного элемента в заданное число
size()	Возвращает общее количество элементов в мультимножестве

При этом, естественно, выполняются разумные ограничения на все методы:

- количество вхождений элемента не может быть отрицательным;
- установление количества вхождений элемента в 0 равносильно удалению всех его вхождений;
- `multiset.count(elem)` для элемента, не принадлежащего мультимножеству, вернёт 0.

Реализации (примерно соответствуют стандартным реализациям Map): **HashMultiset**, **TreeMultiset**, **LinkedHashMultiset**, **ConcurrentHashMultiset**, **ImmutableMultiset**. Думаю, примерно понятно, чем они отличаются, так что не будем на этом останавливаться.

Есть и похожий на мультимножество класс — **Multimap**. Его концептуально можно понимать как коллекцию отображений из ключа в значение, где ключи не уникальны (`a -> 1 a -> 2 a -> 4 b -> 3 c -> 5`) или как отображение из ключа в список значений (`a -> [1, 2, 4] b -> [3] c -> [5]`). Он умеет и так и так, второе представление можно получить с помощью метода asMap(), который возвращает Map<K, Collection<V». Однако ключ всегда должен отображаться хотя бы в одно значение, иначе он считается удалённым из multimap-а. Полезны также интерфейсы-наследники ListMultimap и SetMultimap, которые отображают ключи в List и Set соответственно. Методы наподобие get(key) или asMap() возвращают вид на коллекцию, так что модификации в том, что эти методы вернули, отразятся и на исходной коллекции. Например:

```
Set<Person> aliceChildren = childrenMultimap.get(alice);
aliceChildren.clear();
aliceChildren.add(bob);
aliceChildren.add(carol);
```

childrenMultimap в этом случае будет тоже модифицирована. Обратите внимание, что get(key) всегда возвращает коллекцию, возможно, пустую.

Реализации: ArrayListMultimap, HashMultimap, LinkedListMultimap, LinkedHashMultimap, TreeMultimap, ImmutableListMultimap, ImmutableSetMultimap.

Следующая интересная коллекция — это BiMap, представляющая собой отображение «туда и обратно», позволяющее эффективно искать ключи по значениям и значения по ключам. Тут и ключи, и значения должны быть уникальны, иначе оно упадёт с

IllegalArgumentException при попытке добавления. Затирать старое значение можно методом BiMap.forcePut(key, value). Сам по себе BiMap отображает ключ в значение, но у него есть метод inverse(), который возвращает обратный BiMap, из значений в ключи. Реализации: HashBiMap, ImmutableBiMap, EnumBiMap, EnumHashBiMap.

Следующий полезный класс — это Table, то, что с помощью стандартной библиотеки делается как, например, Map<FirstName, Map<LastName, Person». Table отображает пару ключей в значение, делая удобной работу с чем-то вроде двумерных таблиц:

```
Table<DateOfBirth, LastName, PersonalRecord> records = HashBasedTable.create();
records.put(someBirthday, "Schmo", recordA);
records.put(someBirthday, "Doe", recordB);
records.put(otherBirthday, "Doe", recordC);

records.row(someBirthday); // returns a Map mapping "Schmo" to recordA, "Doe" to recordB
records.column("Doe"); // returns a Map mapping someBirthday to recordB, otherBirthday to recor
```

По ключу, идентифицирующему строку (первому из пары ключей), можно получить столбец целиком, и наоборот, по идентификатору столбца можно получить целиком строку. Получение строки, однако же, обычно эффективнее, чем столбца. Есть метод cellSet(), возвращающий просто коллекцию ячеек в таблице. Реализации: HashBasedTable, TreeBasedTable, ImmutableTable, ArrayTable.

Следующий, более специализированный класс — это ClassToInstanceMap, это отображение из класса в объект этого класса (реализует интерфейс Map<Class<? extends B>, B>). Пример использования:

```
ClassToInstanceMap<Number> numberDefaults = MutableClassToInstanceMap.create();
numberDefaults.putInstance(Integer.class, Integer.valueOf(0));
```

Зачем — такие штуки часто встречаются, если кто-то хочет использовать паттерны «приспособленец» или «прототип» и, естественно, легко делаются вручную. Но, как обычно, то, что можно легко сделать вручную, с помощью Guava можно не делать вообще. Реализаций всего две: MutableClassToInstanceMap и ImmutableClassToInstanceMap.

Следующий класс — RangeSet, это набор промежутков: отрезков, интервалов или полуинтервалов. Пример использования:

```
   RangeSet<Integer> rangeSet = TreeRangeSet.create();
   rangeSet.add(Range.closed(1, 10)); // {[1, 10]}
   rangeSet.add(Range.closedOpen(11, 15)); // disconnected range: {[1, 10], [11, 15)}
   rangeSet.add(Range.closedOpen(15, 20)); // connected range; {[1, 10], [11, 20)}
   rangeSet.add(Range.openClosed(0, 0)); // empty range; {[1, 10], [11, 20)}
   rangeSet.remove(Range.open(5, 10)); // splits [1, 10]; {[1, 5], [10, 10], [11, 20)}
\begin{minted}{java}

Он умеет автоматически объединять перекрывающие друг друга интервалы, умеет генерить кучу разны

\begin{minted}{java}
RangeMap<Integer, String> rangeMap = TreeRangeMap.create();
```

```
rangeMap.put(Range.closed(1, 10), "foo"); // {[1, 10] => "foo"}
rangeMap.put(Range.open(3, 6), "bar"); // {[1, 3] => "foo", (3, 6) => "bar", [6, 10] => "foo"}
```

Для всех новых коллекций из Guava и многих стандатных коллекций есть ещё классы-утилиты со стаитческими методами, делающими работу с коллекциями более удобной: например, для Multiset есть класс Multisets, для Table — Tables, для List — Lists (идея, я думаю, понятна). Необходимость в них несколько уменьшилась с выходом Java 8, но они до сих пор делают жизнь Java-программиста приятнее. Туда имеет смысл посмотреть, потому что многие операции Guava реализует в чём-то лучше, чем JDK, например, хоть немножко, но короче и симпатичнее. Что важно, Guava, в отличие от JDK, имеет тенденцию делать операции лениво и не модифицировать коллекцию, а просто делать на неё вид. Но в силу несколько меньшей полезности оных классов ограничимся здесь лишь несколькими примерами, которые, я надеюсь, покажут общий «дух» и стиль классов-утилит Guava.

```
Set<String> wordsWithPrimeLength = ImmutableSet.of("one", "two", "three", "six", "seven", "eigh
Set<String> primes = ImmutableSet.of("two", "three", "five", "seven");

SetView<String> intersection = Sets.intersection(primes, wordsWithPrimeLength); // contains "tw
// I can use intersection as a Set directly, but copying it can be more efficient if I use it a
return intersection.immutableCopy();

Set<String> animals = ImmutableSet.of("gerbil", "hamster");
Set<String> fruits = ImmutableSet.of("apple", "orange", "banana");

Set<List<String>> product = Sets.cartesianProduct(animals, fruits);
// {{"gerbil", "apple"}, {"gerbil", "orange"}, {"gerbil", "banana"},
//  {"hamster", "apple"}, {"hamster", "orange"}, {"hamster", "banana"}}

Set<Set<String>> animalSets = Sets.powerSet(animals);
// {{}, {"gerbil"}, {"hamster"}, {"gerbil", "hamster"}}

Map<String, Integer> left = ImmutableMap.of("a", 1, "b", 2, "c", 3);
Map<String, Integer> right = ImmutableMap.of("b", 2, "c", 4, "d", 5);
MapDifference<String, Integer> diff = Maps.difference(left, right);

diff.entriesInCommon(); // {"b" => 2}
diff.entriesDiffering(); // {"c" => (3, 4)}
diff.entriesOnlyOnLeft(); // {"a" => 1}
diff.entriesOnlyOnRight(); // {"d" => 5}
```

Есть несколько полезных утилит для быстрого создания своих коллекций и итераторов (которыми пользуются сами авторы Guava). Для создания колекций используются так называемые Forwarding Decorators. Идея такая же, как классы Abstract* из JDK, но идеология немного другая: вместо наследования испольуется композиция. Коллекция, которую мы хотим взять за основу, помещается внутрь нашего класса-декоратора, наследуемого от декоратора из Guava, мы реализуем метод delegate(), возвращающий декорируемый объект, дальше библиотечный декоратор делает всё за нас, просто перенаправляя все вызовы

методов декорируемой коллекции (паттерн «декоратор» в действии). Профит в том, что любой метод мы можем переопределить и заставить его делать то, что хотим мы, быть может, вызвав потом и метод декорируемого объекта. Чем это лучше подхода, принятого в JDK — тем же, чем и обычно композиция и делегирование лучше наследования: декораторы и декорируемые объекты можно менять во время выполнения. Берёте список, хотите, чтобы все операции добавления добавлялись в лог — делаете для него декоратор и оборачиваете в него исходный список. Хотите, чтобы и операции удаления добавлялись в лог — делаете ещё один декоратор и оборачиваете в него предыдущий декоратор (ну а что, он же тоже реализует интерфейс списка). Подробности лучше показать на примере:

```
class AddLoggingList<E> extends ForwardingList<E> {
  final List<E> delegate; // backing list
  @Override protected List<E> delegate() {
    return delegate;
  }
  @Override public void add(int index, E elem) {
    log(index, elem);
    super.add(index, elem);
  }
  @Override public boolean add(E elem) {
    return standardAdd(elem); // implements in terms of add(int, E)
  }
  @Override public boolean addAll(Collection<? extends E> c) {
    return standardAddAll(c); // implements in terms of add
  }
}
```

Это список, который логирует все операции добавления. final List<E> delegate — это декорируемый список, delegate() — единственный обязательный метод, возвращающий объект, которому нужно перенаправлять запросы. add и addAll добавляют полезную нам функциональность, вызывая и базовую. Обратите внимание, standardAdd, standardAddAll и вообще, standard* — это методы, говорящие декоратору вызвать «исходную» реализацию метода, и тут они нужны, потому что реализации по умолчанию add и addAll просто перенаправят запрос delegate-у, так что наш перегруженный add с логированием вызван не будет. Forwarding*-классы есть для всех интерфейсов коллекций из JDK.

Для создания итераторов есть классы PeekingIterator и AbstractIterator. Первый оборачивает итератор и добавляет ему метод peek(), позволяющий заглянуть вперёд и получить элемент, который вернёт следующий next() (правда, после этого нельзя будет удалить элемент методом remove()). Семантика PeekingIterator, кстати, очень похожа на итераторы (точнее, энумераторы) в .NET. AbstractIterator позволяет реализовать итератор, определив только один метод — computeNext(), который должен возвращать следующий элемент последовательности, либо endOfData(), если процесс итерирования должен быть закончен. Есть ещё AbstractSequentialIterator, у которого computeNext() принимает предыдущее значение итератора, и для обозначения конца итерирования возвращает null. Оба этих класса делают итераторы, которые не реализуют remove(). Нужны эти классы, как обычно, чтобы сэкономить несколько строк кода, которые потребовалось бы написать, реализуй мы итераторы вручную. Небольшой пример:

```
Iterator<Integer> powersOfTwo = new AbstractSequentialIterator<Integer>(1) { // note the ini
  protected Integer computeNext(Integer previous) {
    return (previous == 1 << 30) ? null : previous * 2;
  }
};
```

2.3. Графы

Guava имеет целую подсистему для работы с графами. Казалось бы, граф — это довольно специфичная штука, применяемая не очень часто, но авторы библиотеки считают, что это связано с тем, что существующими инструментами работать с графами сложно и если сделать годную библиотеку, применения у графов сами собой найдутся. Например, вы хотите multimap, где некоторые ключи могут не иметь ассоциированных с ними значений — Multimap не подойдёт, он удалит такие ключи — конечно же используйте графы. Кроме того, иногда абстракция графа оказывается очень в тему, а писать код работы с графами руками не только неприятно, но и чревато ошибками — поэтому в Guava они есть и довольно неплохо поддержаны. Но, поскольку графы — это всё-таки не очень важно, мы в том обзоре ограничимся общей архитектурой подсистемы и некоторыми неожиданными вещами, детали, как всегда, смотрите в документации[6].

Графы в Guava бывают трёх сортов.

- **Graph** — просто множество вершин, связанных непомеченными рёбрами кратности 1.
- **ValueGraph** — наследник Graph, добавляющий метки на рёбрах (метки имеют семантику значений, в том смысле, что ребро не имеет собственной идентичности и одна и та же метка может отмечать несколько рёбер одновременно). Рёбра тоже имеют кратность 1, хотя с помощью меток можно проэмулировать произвольную кратность (впрочем, стандартные методы про это не знают и будут ваши метки игнорировать).
- **Network** — граф, в котором и вершины, и рёбра имеют собственную идентичность, поэтому он умеет выполнять запросы наподобие outEdges(node), incidentNodes(edge) и edgesConnecting(nodeU, nodeV). Тут уже рёбра могут быть произвольной кратности. У Network есть также метод asGraph(), который возвращает представление в виде графа (выкидывая лишнюю информацию), так что все алгоритмы, написанные для Graph, могут работать и для Network, хотя он от Graph не наследует.

Любой граф имеет помеченные вершины (в том смысле, что вершина — это произвольный объект, который сам может хранить какую угодно информацию), вершины можно понимать как ключи для внутренних структур данных, хранящих топологию графа. Графы могут быть направленными и ненаправленными, иметь или не иметь петли, разрешать или не разрешать параллельные рёбра (для Network). Это всё выбирается при создании графа, которое выполняется классами GraphBuilder, ValueGraphBuilder или NetworkBuilder, реализующие паттерн «Строитель». Обратите внимание, что Graph, ValueGraph и Network — это интерфейсы, а их конкретные реализации (за исключением Immutable*) библиотека

6 https://github.com/google/guava/wiki/GraphsExplained

вообще не публикует, их можно создать только Buiilder-ом. Это для того, чтобы пользователь не заморачивался выбором правильной реализации графа, а просто указал, что ему нужно и библиотека сделала бы всё за него. Пример создания графа:

```
MutableGraph<Integer> graph = GraphBuilder.undirected().build();

MutableValueGraph<City, Distance> roads = ValueGraphBuilder.directed().build();

MutableNetwork<Webpage, Link> webSnapshot = NetworkBuilder.directed()
    .allowsParallelEdges(true)
    .nodeOrder(ElementOrder.natural())
    .expectedNodeCount(100000)
    .expectedEdgeCount(1000000)
    .build();
```

Насчтё Immutable*-вариантов графов в Guava сделано не совсем канонично. В JDK принято немутабельные коллекции делать реализующими мутабельные интерфейсы и бросать UnsupportedOperationException в изменяющих коллекцию методах. Поскольку это ужасно, тут интерфейсы Graph, ValueGraph и Network вообще не предоставляют изменяющих граф методов, но от этих интерфейсов наследуются интерфейсы MutableGraph, MutableValueGraph и MutableNetwork, в которых оные методы и определены. Так что если метод не намеревается менять граф, он может принимать Graph и спокойной с ним работать с меньшим риском что-то сломать. Но — то, что метод принимает Graph, не значит, что он его не меняет. Ничто не мешает внутри метода откастать Graph к MutableGraph и делать с ним что угодно. Чтобы так было сделать нельзя, у Graph, ValueGraph и Network есть ещё наследники ImmutableGraph, ImmutableValueGraph и ImmutableNetwork.

Immutable* версии могут быть получены из обычного графа методом copyOf(), который выполняет мелкую копию (в противоположность глубокой, то есть топология копируется, но метки — нет). Поэтому может случиться печаль, если в Immutable-графе кто-то начнёт менять состояние вершин (особенно ту его часть, что используется в equals()). Вообще, стандартные коллекции тоже можно так сломать, поэтому в Guava просто вежливо просят так не делать. В остальном Immutable-версии ведут себя ожидаемо: элементы графа нельзя добавлять/удалять, обращения к ним потокобезопасны.

Полезные алгоритмы для работы с графами находятся в классе Graphs — такие как hasCycle(Graph<?> graph), inducedSubgraph(Graph<N> graph, Iterable<? extends N> nodes), reachableNodes(Graph<N> graph, Object node), transitiveClosure(Graph<N> graph). При желании можно встроить в библиотеку свои реализации графов, для этого есть классы AbstractGraph<N>, AbstractValueGraph<N,V>, AbstractNetwork<N,E>. Стандартные реализации, говорят, работают неплохо на графах порядка нескольких миллионов узлов, но для по-настоящему больших графов Guava не подходит.

Ну и немного примеров из документации:

```
MutableValueGraph<Integer, Double> weightedGraph = ValueGraphBuilder.directed().build();
weightedGraph.addNode(1);
weightedGraph.putEdgeValue(2, 3, 1.5);  // also adds nodes 2 and 3 if not already present
weightedGraph.putEdgeValue(3, 5, 1.5);  // edge values (like Map values) need not be unique
```

```
...
weightedGraph.putEdgeValue(2, 3, 2.0);  // updates the value for (2,3) to 2.0

MutableNetwork<Integer, String> network = NetworkBuilder.directed().build();
network.addNode(1);
network.addEdge("2->3", 2, 3);  // also adds nodes 2 and 3 if not already present

Set<Integer> successorsOfTwo = network.successors(2);  // returns {3}
Set<String> outEdgesOfTwo = network.outEdges(2);   // returns {"2->3"}

network.addEdge("2->3 too", 2, 3);  // throws; Network disallows parallel edges
                                    // by default
network.addEdge("2->3", 2, 3);  // no effect; this edge is already present
                                // and connecting these nodes in this order

Set<String> inEdgesOfFour = network.inEdges(4); // throws; node not in graph
```

2.4. Кеши

Первая домашка в этом курсе была про Lazy, класс, который выполняет вычисление при первом обращении к значению, после чего запоминает его и больше не вычисляет. Эта штука обобщается до понятия «кеш» — хешмапы таких значений. Ещё неплохо бы, раз хранимых значений теперь может быть много, уметь выкидывать их из памяти, когда памяти становится мало, либо по истечении времени (особенно, если значения в кеше могут протухать по естественным причинам, например, прогноз погоды). А ещё всё это надо уметь делать потокобезопасно. В общем-то, это часто возникающая задача, поэтому такая структура данных реализована в Guava.

Начнём с примера:

```
LoadingCache<Key, Graph> graphs = CacheBuilder.newBuilder()
       .maximumSize(1000)
       .build(
           new CacheLoader<Key, Graph>() {
             public Graph load(Key key) throws AnyException {
               return createExpensiveGraph(key);
             }
           });


...
try {
  return graphs.get(key);
} catch (ExecutionException e) {
  throw new OtherException(e.getCause());
}
```

Здесь мы видим кеш, который умеет грузить значения с помощью реализации интерфейса CacheLoader, который предоставляем мы, и который не может содержать больше

1000 элементов (на самом деле, кеш может начать выкидывать значения заранее). Метод get(key) делает то, что делал get у Lazy — если значение есть в кеше, возвращает его, если нет, вызывает метод load у переданного CacheLoader, получает от него значение и возвращает, сохраняя в кеше. Есть ещё метод getUnchecked(), который не декларирует исключений и полезен, если ваш CacheLoader исключений не кидает. Ещё есть метод getAll(Iterable<? extends K>), который позволяет загрузить из кеша сразу набор значений, и, если CacheLoader перегружает метод loadAll(), получить какой-то профит от того, что грузится сразу несколько значений (например, сделать один запрос к серверу, а не N разных). Можно использовать и более легковесный вариант, без CacheLoader-а:

```
Cache<Key, Value> cache = CacheBuilder.newBuilder()
    .maximumSize(1000)
    .build(); // look Ma, no CacheLoader
...
try {
  cache.get(key, () -> doThingsTheHardWay(key));
} catch (ExecutionException e) {
  throw new OtherException(e.getCause());
}
```

А можно вообще воспользоваться cache.put(key, value) и вставить значение в кеш вручную.

Выкидывание значений из кеша устроено менее очевидно. Во-первых, чистка кеша происходит только при записи и (иногда) чтении, даже если мы просили выкидывать значения по таймеру. Если элементы в кеш могут иметь сильно разный размер, имеет смысл пользоваться не maximumSize(), а методом CacheBuilder.maximumWeight(long), которому ещё требуется вызов CacheBuilder.weigher(Weigher). Weigher — это такой интерфейс, который должен по ключу и значению выдать число, определяющее «вес» значения, на основе которого кеш принимает решение, какие объекты удалять. Вот пример:

```
LoadingCache<Key, Graph> graphs = CacheBuilder.newBuilder()
       .maximumWeight(100000)
       .weigher(new Weigher<Key, Graph>() {
          public int weigh(Key k, Graph g) {
            return g.vertices().size();
          }
        })
       .build(
           new CacheLoader<Key, Graph>() {
             public Graph load(Key key) { // no checked exception
               return createExpensiveGraph(key);
             }
           });
```

Ещё можно выкидывать значения из кеша по времени (методами CacheBuilder.expireAfterAccess(long, TimeUnit) и CacheBuilder.expireAfterWrite(long, TimeUnit)), при этом они даже позаботились о модульном тестировании — можно

воспользоваться методом CacheBuilder.ticker(Ticker) для того, чтобы передать кешу тестовые часы и не ждать по N секунд инвалидации кеша в юнит-тестах. Можно выкидывать значения и вручную, методами Cache.invalidate(key) или Cache.invalidateAll(). Интересно, что кеш может приводить к неоправданным расходам памяти, если будет держать ссылки на значения, которые больше никому не нужны. Кеш в Guava можно попросить хранить weak references на значения или ключи, так что когда сборщик мусора решит собрать ключи и значения, они уберутся и из кеша. Если такое поведение пугает, есть метод CacheBuilder.removalListener(RemovalListener), позволяющий подписываться на удаление значения из кеша. Ещё годная штука, про которую надо сказать, это метод CacheBuilder.recordStats(), включающий запись статистики, которую можно потом получить методом Cache.stats() — количество попаданий в кеш, время, проведённое за загрузкой значений, число выталкиваний из кеша и т.д., это может быть очень полезно при насройке кеша в критических по функциональности участках кода.