Оглавление

[Что такое дженерики? Какую проблему они решают. 1](#_Toc59377346)

[Wildcard 1](#_Toc59377347)

[Принцип PECS 2](#_Toc59377348)

[Параметризация статических методов. Что можно параметризовать 3](#_Toc59377349)

[Что такое коллекция. Иерархия коллекций. 4](#_Toc59377350)

[В каких случаях нужно использовать iterator? И почему? 5](#_Toc59377351)

[Как работает HashSet? Внутреннее устройство HashSet? 6](#_Toc59377352)

[Как работает HashMap? Расскажите подробно, как работает метод put? Что происходит при коллизии? 7](#_Toc59377353)

[Как будет разрешаться коллизия? 8](#_Toc59377354)

[**Может ли null быть ключём в HashMap?** 9](#_Toc59377355)

[Как работает метод get в HashMap? 9](#_Toc59377356)

[При каких условиях в HashMap лист преобразуется в красно-черное дерево?" 9](#_Toc59377357)

[Что такое даймонд оператор? 9](#_Toc59377358)

[Что такое raw type? К чему приводит использование raw type? +++от других1. Чему эквивалентно <?> 9](#_Toc59377359)

[Что было до дженериков? 10](#_Toc59377360)

[Если поле типизировано дженериком как в байт коде будет представлен этот тип? 10](#_Toc59377361)

[Почему Map не входит в Collection? 11](#_Toc59377362)

[Зачем в итераторе метод remove? 11](#_Toc59377363)

[Что такое Map? Что должно быть уникальным? 11](#_Toc59377364)

[что кладётся на место значения в HashSet? 12](#_Toc59377365)

[В чём разница между Iterable и Iterator? 12](#_Toc59377366)

[как устроен LinkedList 14](#_Toc59377367)

[Отличие ArrayList от LinkedList? 14](#_Toc59377368)

[Когда лучше использовать ArrayList, а когда LinkedList? 14](#_Toc59377369)

[Расскажи отличие List от Set? 15](#_Toc59377370)

[В чём разница между Queue и Deque и Stack? 15](#_Toc59377371)

[какой хэш-код у null в HashMap 17](#_Toc59377372)

[Отличие двусвязного и односвязного списка? 17](#_Toc59377373)

[Почему в HashSet вместо value не null а new Object? 17](#_Toc59377374)

# Что такое дженерики? Какую проблему они решают.

Дженерики (обобщения) — это особые средства языка Java для реализации обобщённого программирования: особого подхода к описанию данных и алгоритмов, позволяющего работать с различными типами данных без изменения их описания.

одно из назначений — более сильная проверка типов во время компиляции и устранение необходимости явного приведения.

Код, использующий универсальные шаблоны, имеет много преимуществ по сравнению с обычным кодом:

* Более строгие проверки типов во время компиляции.  
  Компилятор Java применяет строгую проверку типов к универсальному коду и выдает ошибки, если код нарушает безопасность типов. Исправлять ошибки во время компиляции проще, чем исправлять ошибки во время выполнения (в Runtime), которые бывает сложно найти.
* Устранение слепков.  
  Следующий фрагмент кода без универсальных шаблонов требует приведения:
* List list = new ArrayList();
* list.add ("привет");
* String s = **(String)** list.get (0);

При переписывании для использования дженериков код не требует приведения:

List <String> list = new ArrayList<String>();

list.add ("привет");

String s = list.get(0); // без приведения

* Предоставление программистам возможности реализовать общие алгоритмы.  
  Используя универсальные шаблоны, программисты могут реализовать универсальные алгоритмы, которые работают с коллекциями различных типов, могут быть настроены, безопасны по типу и легче читаются.

# Wildcard

Wildcards – специальные вопросительные знаки, отмечающие место, где должно оказаться название параметра типа. Они являются средством выражения ограничений в условиях неизвестности какого-нибудь типа. Первоначально эти знаки не являлись частью структуры generic-ов, унаследованной от проекта Generic Java (GJ), а были добавлены в процессе разработки дизайна за те более чем пять лет, что прошли между созданием JSR 14 и его последним выпуском.

Wildcards играют важную роль в системе типов. Они позволяют задать границы для семейства типов, определенных каким-нибудь generic-классом. Для generic-класса ArrayList тип ArrayList<?> является супер-типом ArrayList<T> с параметром любого типа T (так же, как обычный тип (raw type) ArrayList и корневой тип (root type) Object, но эти супертипы менее удобны при выполнении вывода типа).

Тип List<?> отличается как от raw-типа List, так и от конкретного типа List<Object>. Сказать, что переменная x имеет тип List<?>, означает, что существует некоторый тип T, для которого x имеет тип List<T>, и что при этом x – однородная переменная, хотя неизвестно, какой конкретно тип имеют здесь ее элементы. Это не значит, что содержимое тут может быть любым – имеется в виду, что мы не знаем заранее, какие ограничения типа использованы, но знаем, что ограничение есть. С другой стороны, raw-тип List является гетерогенным; мы не можем наложить какие-либо ограничения на его элементы, а конкретный тип List<Object> предполагает, что мы ясно представляем себе, что в него могут быть переданы любые объекты. (Конечно, система generic-типов не содержит концепции «содержимое списка», но generic-и проще всего понять на примере типов, описывающих такие коллекции как List).

Wildcards удобно использовать в системе типов, в частности, потому, что generic-типы не являются ковариантными. Массивы же ковариантны. Поскольку Integer является подтипом Number, то массив типа Integer[] – подтип от Number[], и, следовательно, значение переменной типа Integer[]можно передать туда, где требуется Number[]. С другой стороны, generic-типы – не ковариантны; List<Integer> не является подтипом от List<Number>, попытка передать List<Integer> туда, где необходим List<Number>, вызовет ошибку несоответствия типов. Это не случайность, но и не обязательно ошибка, как кажется многим. На самом деле большая часть проблем вызвана разницей между принципами работы generic-типов и обычных массивов.

Если класс B является наследником класса А, то Collection<B> при этом — не наследник Collection<A>.

String является наследником Object, но List<String> не является наследником List<Object>.

Да, <? super Number> действительно подразумевает "любой тип, являющийся супер-классом для Number". Плюс сам Number.

В случае <? extends Number> подразумевается "любой тип, являющийся дочерним классом для Number". Плюс сам Number.

http://www.k-press.ru/cs/2008/3/generic/generic.asp

# Принцип PECS

"PECS" Producer Extends Consumer Super - это с точки зрения коллекции. Если вы только вытаскиваете элементы из общей коллекции, это производитель, и вы должны использовать extends ; если вы только набиваете элементы, это потребитель, и вы должны использовать super . Если вы делаете и то и другое с одной и той же коллекцией, вы не должны использовать ни extends , ни super .

Предположим, у вас есть метод, который принимает в качестве своего параметра набор вещей, но вы хотите, чтобы он был более гибким, чем просто принятие Collection<Thing> .

Случай 1: Вы хотите пройти через коллекцию и сделать что-то с каждым предметом.  
Тогда список является производителем, поэтому вы должны использовать Collection<? extends Thing> .

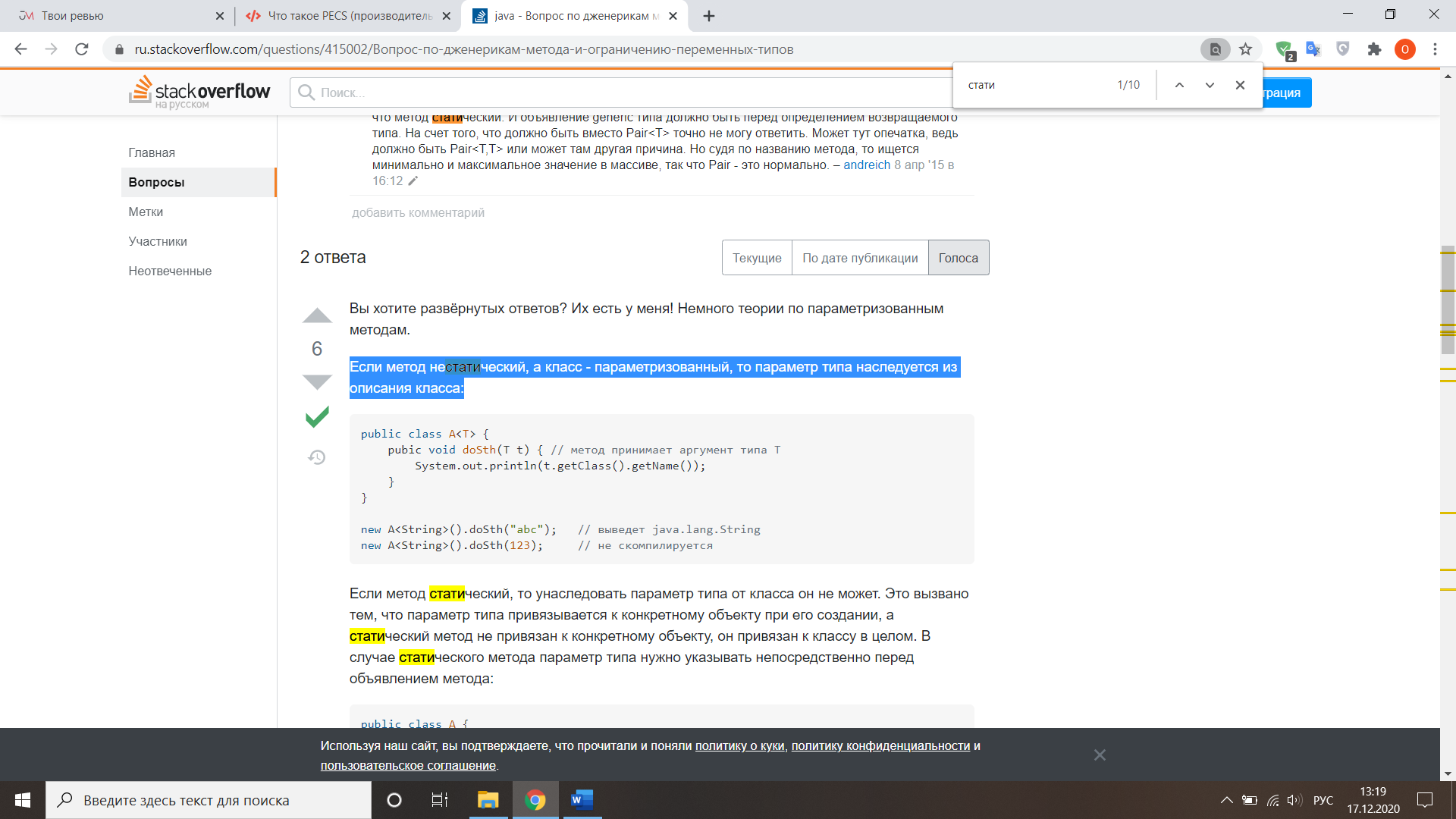
Рассуждение состоит в том , что Collection<? extends Thing> может содержать любой подтип Thing, и поэтому каждый элемент будет вести себя как Thing , когда вы выполняете свою операцию. (На самом деле вы не можете ничего добавить к Collection<? extends Thing> , потому что вы не можете знать во время выполнения, какой конкретный подтип Thing содержит коллекция.)

Случай 2: вы хотите добавить вещи в коллекцию.  
Тогда список является потребителем, поэтому вы должны использовать Collection<? super Thing> .

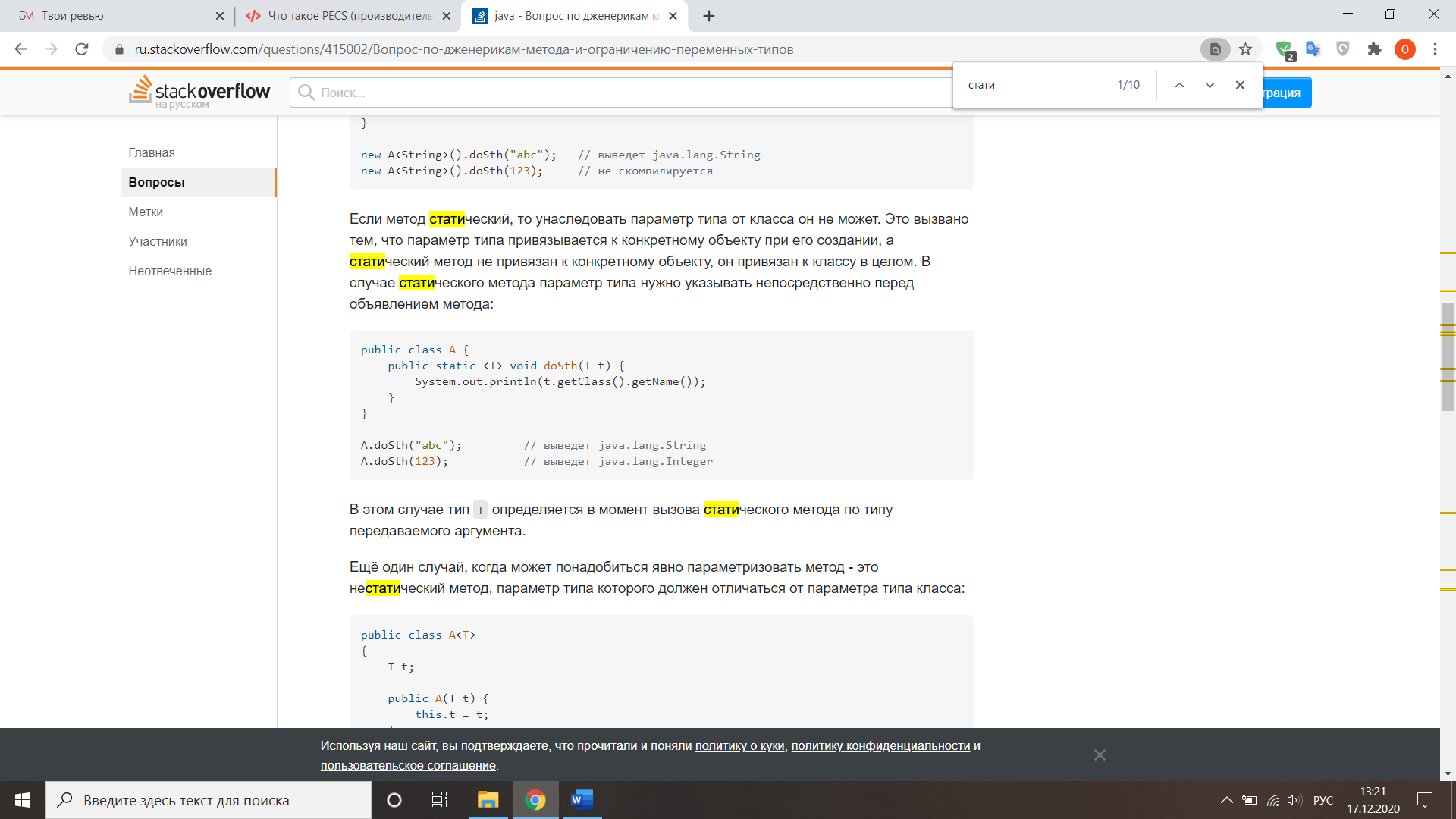
Рассуждение здесь состоит в том , что в отличие от Collection<? extends Thing>, Collection<? super Thing> всегда может содержать Thing , независимо от того, какой фактический параметризованный тип является. Здесь вам все равно, что уже есть в списке, если это позволит добавить Thing ; это то, что гарантирует ? super Thing .

# Параметризация статических методов. Что можно параметризовать

Если метод нестатический, а класс - параметризованный, то параметр типа наследуется из описания класса:

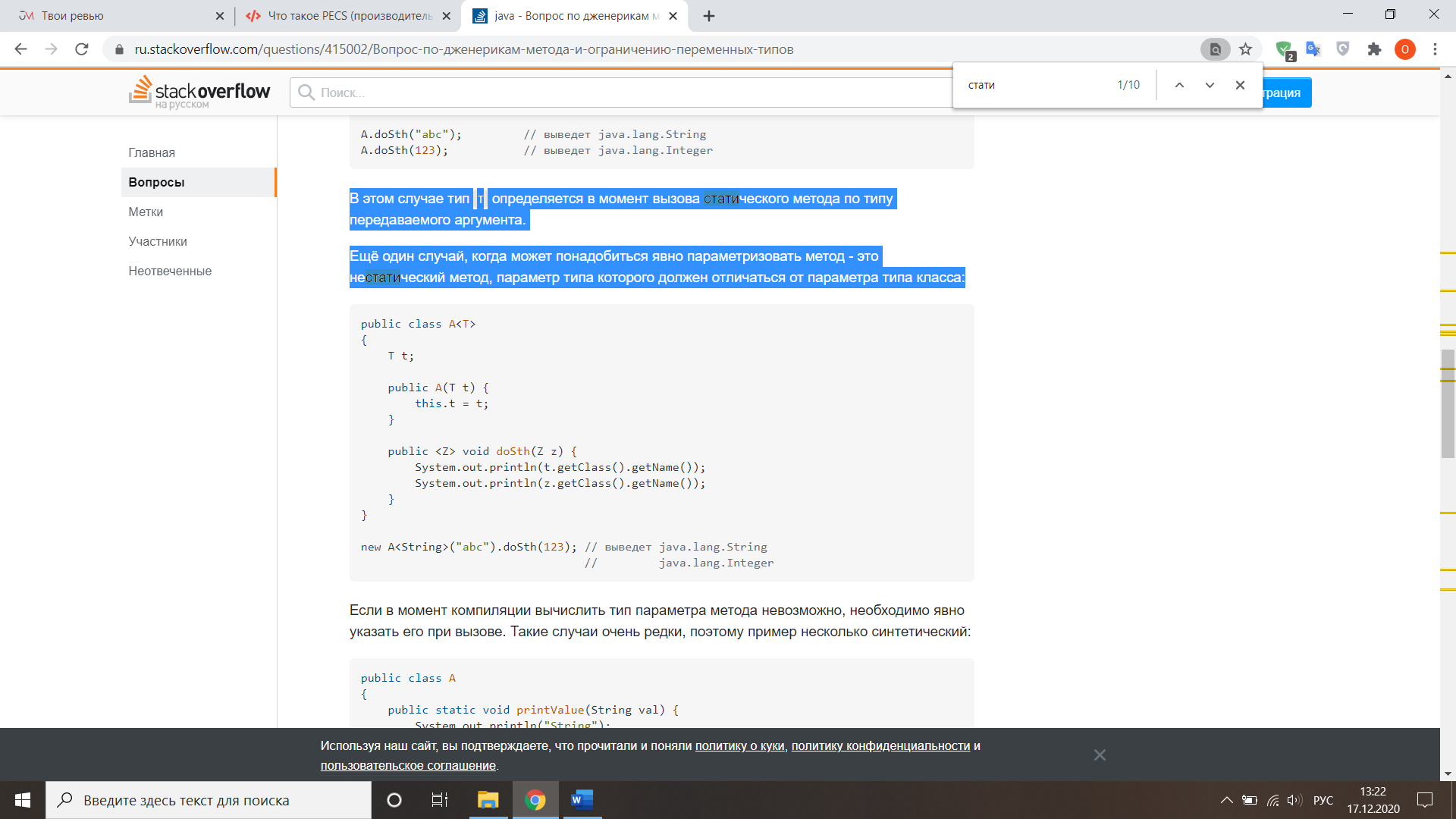


Если метод статический, то унаследовать параметр типа от класса он не может. Это вызвано тем, что параметр типа привязывается к конкретному объекту при его создании, а статический метод не привязан к конкретному объекту, он привязан к классу в целом. В случае статического метода параметр типа нужно указывать непосредственно перед объявлением метода:

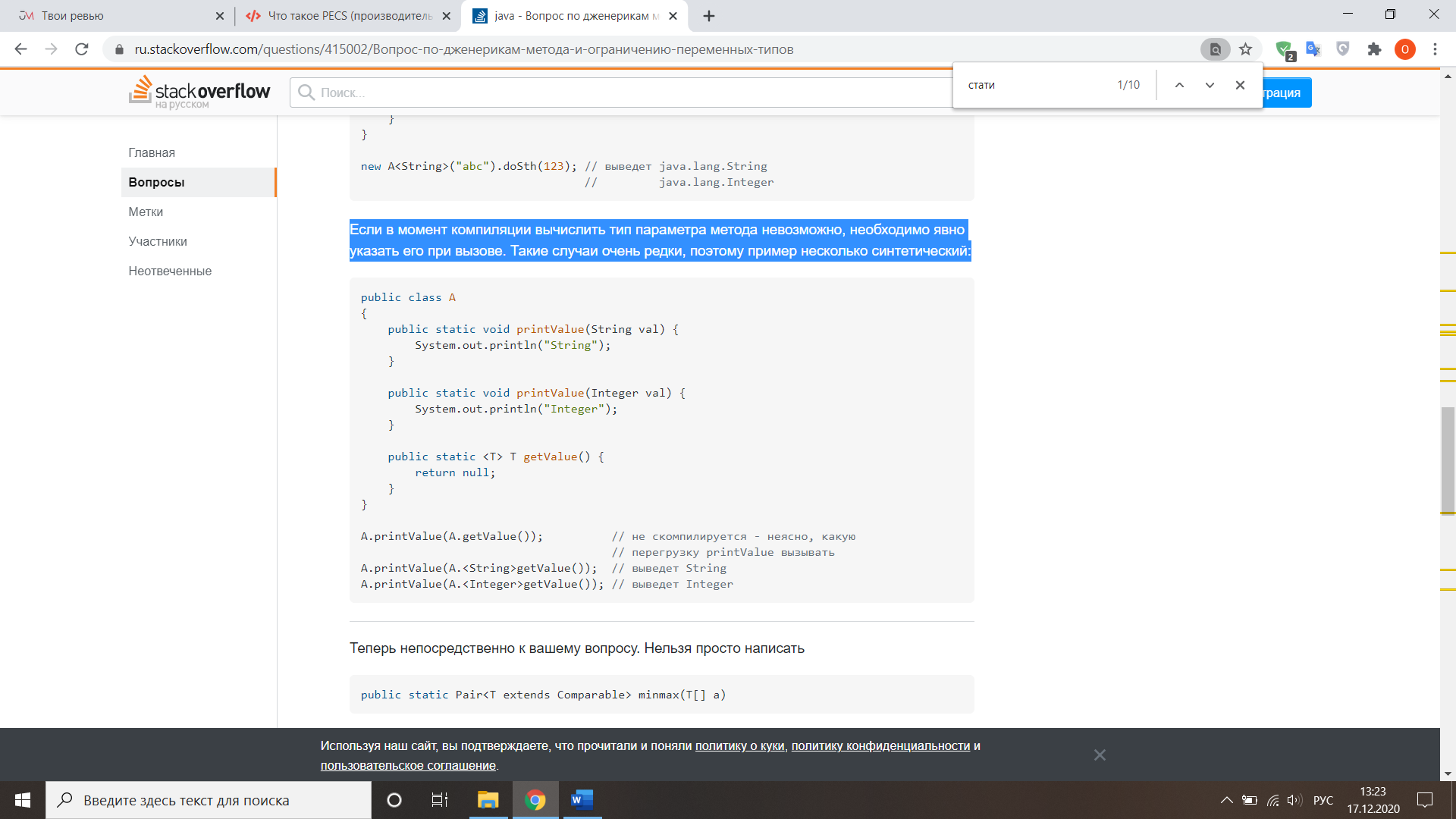


В этом случае тип T определяется в момент вызова статического метода по типу передаваемого аргумента.

Ещё один случай, когда может понадобиться явно параметризовать метод - это нестатический метод, параметр типа которого должен отличаться от параметра типа класса:



Если в момент компиляции вычислить тип параметра метода невозможно, необходимо явно указать его при вызове. Такие случаи очень редки, поэтому пример несколько синтетический:



Параметризация позволяет создавать классы, интерфейсы и методы, в которых тип обрабатываемых данных задается как параметр.

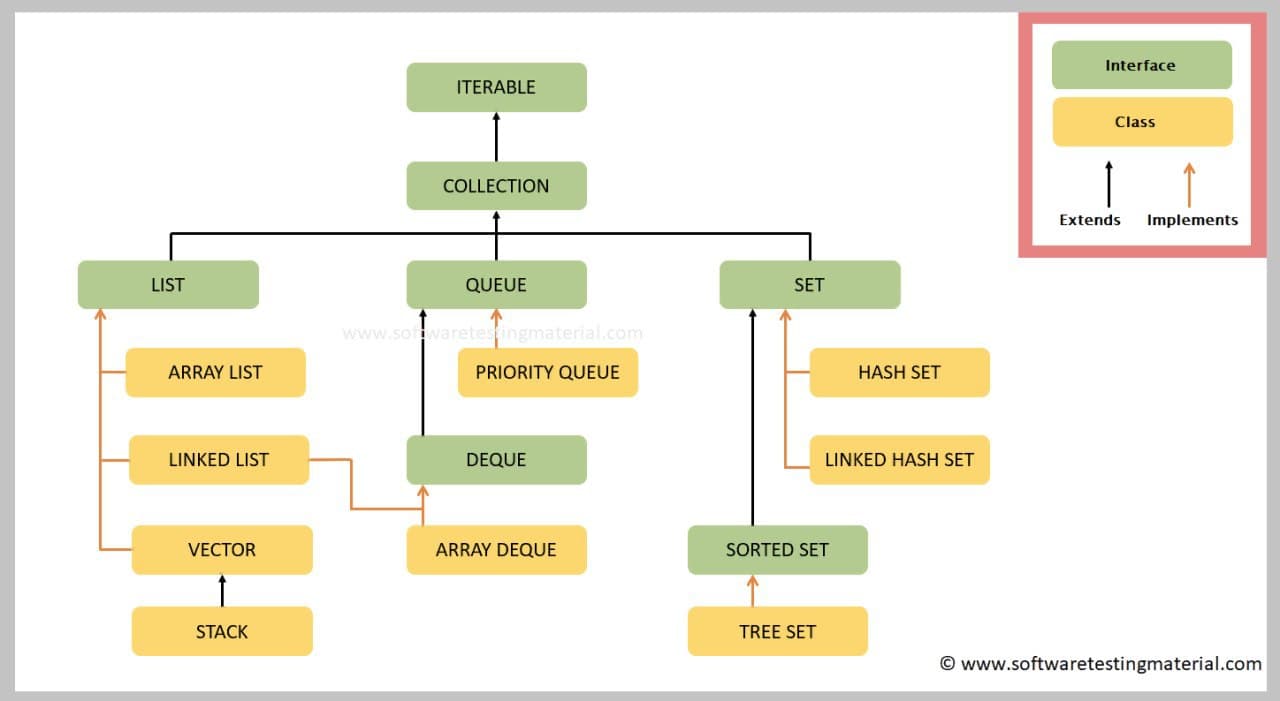
# Что такое коллекция. Иерархия коллекций.

Для хранения наборов данных в Java предназначены массивы. Однако их не всегда удобно использовать, прежде всего потому, что они имеют фиксированную длину. Эту проблему в Java решают коллекции. Однако суть не только в гибких по размеру наборах объектов, но в и том, что классы коллекций реализуют различные алгоритмы и структуры данных, например, такие как стек, очередь, дерево и ряд других.

Классы коллекций располагаются в пакете java.util, поэтому перед применением коллекций следует подключить данный пакет.

Хотя в Java существует множество коллекций, но все они образуют стройную и логичную систему. Во-первых, в основе всех коллекций лежит применение того или иного интерфейса, который определяет базовый функционал. Среди этих интерфейсов можно выделить следующие:

* **Collection:** базовый интерфейс для всех коллекций и других интерфейсов коллекций
* **Queue:** наследует интерфейс Collection и представляет функционал для структур данных в виде очереди
* **Deque:** наследует интерфейс Queue и представляет функционал для двунаправленных очередей
* **List:** наследует интерфейс Collection и представляет функциональность простых списков
* **Set:** также расширяет интерфейс Collection и используется для хранения множеств уникальных объектов
* **SortedSet:** расширяет интерфейс Set для создания сортированных коллекций
* **NavigableSet:** расширяет интерфейс SortedSet для создания коллекций, в которых можно осуществлять поиск по соответствию
* **Map:** предназначен для созданий структур данных в виде словаря, где каждый элемент имеет определенный ключ и значение. В отличие от других интерфейсов коллекций не наследуется от интерфейса Collection



https://metanit.com/java/tutorial/5.1.php

1. **Внутреннее устройство коллекций.**

# В каких случаях нужно использовать iterator? И почему?

В java.util есть класс Iterator, который может использоваться, когда итерация по списку или другой коллекции предполагает удаление элементов. Удалять элементы в цикле обхода последовательности for each нельзя.

Из списка с помощью метода iterator() создается объект-итератор, который присваивается переменной типа Iterator.

Далее из объекта-итератора в цикле извлекаются элементы с помощью метода next(). Проверка, есть ли следующий элемент, выполняется методом hasNext(). Удаляют элементы методом remove().

Итератор не лучше цикла. Итератор — способ перебрать все элементы какой-либо сущности. А цикл не обязательно что-то перебирает. Например, while (true){} не перебирает ничего. Цикл для простоты понимания лучше рассматривать, как хождение по кругу. Причем этот круг может вообще намертво замкнуться и цикл станет бесконечным.  
Сравним два применения цикла:

Когда вы делаете так

ArrayList <String> list = new Arraylist <>();

for (int i=0; i<list.size(); i++)System.out.printl(list [i]);

И так

ArrayList <String> list = new Arraylist <>();

for (String s : list) System.out.printl(s);

На вид происходит то же самое, а на самом деле — это принципиально разный код.  
Во втором случае вы используете итератор, просто неявно. Его использует сокращенный цикл for. В первом случае вы используете особенности реализации ArrayList, а именно то, что в ArrayList есть индексы, по которым можно получить доступ к элементам ArrayList.  
Здесь итератор, который имплементируется коллекцией, не вызывается, а цикл просто перебирает все натуральные числа от 0 до тех пор, пока это самое натуральное число не станет равным длине коллекции, тогда цикл прерывается. Посему перебрать, например, коллекцию типа Set первым способом вы уже не сможете, ведь индексов у сета нет, а вот вторым без проблем, потому как сет имплементирует итератор.

Исходя из сказанного, становится понятно, что сортировка элементов тут ни при чем. Назначение итератора, как уже было сказано — перебрать все элементы сущности.  
Если вам нужна сортировка, то существует другая замечательная штука, называемая компаратор, но сейчас не об этом. Итератор — не средство сортировки. Вы должны понимать, что мы говорим об интерфейсе, имплементировать который могут классы с абсолютно разной логикой. К примеру, коллекция HashSet не гарантирует порядок элементов и вы никак не можете их упорядочить в пределах этой коллекции. Соответственно итератор, как интерфейс, невозможно приспособить для таких задач. Сущность, его имплементирующая, может просто не поддерживать сортировку элементов в принципе в виду особенности своей внутренней логики, как HashSet.

И последнее…  
Вызывать итератор в явном виде зачастую нецелесообразно и делается это, как правило в одном случае — когда во время итерации вам нужно удалять элементы. У итератора есть методы hasNext, next и remove, которые проверяют наличие следующего элемента, получают следующий элемент и удаляют элемент соотвественно. Попытка удаления элемента при итерации с помощью цикла приведет к исключению. Причем исключение будет брошено даже, если вы используете сокращенный цикл типа for (String s : list). Происходит это потому что, хотя этот цикл неявно и использует итератор, у вас нет к нему доступа. Соответственно, единственный доступный вам способ удалить элементы — вызывать метод remove у коллекции list, что приведет к исключению, ведь для удаления элементов во время итерации по ней нужно вызывать метод remove не у коллекции list, а у самого итератора, к которому, как уже было сказано, у вас в любом из циклов доступа нет.

И еще. В предыдущем абзаце сказано что итератор, как правило, вызывается в одном случае. На самом деле не в одном. Просто зачастую он вызывается именно для указанного случая. А вот остальные случаи вам придется изучить самостоятельно, так как их назначение не столь тривиально, чтобы объяснить это в таком контексте. Речь идет, например, о том, что у одного объекта может быть вызвано несколько итераторов, которым задана разная логика работы. Но это придет со временем.

# Как работает HashSet? Внутреннее устройство HashSet?

Класс HashSet реализует интерфейс Set, основан на хэш-таблице, а также поддерживается с помощью экземпляра HashMap. В HashSet элементы не упорядочены, нет никаких гарантий, что элементы будут в том же порядке спустя какое-то время. Операции добавления, удаления и поиска будут выполняться за константное время при условии, что хэш-функция правильно распределяет элементы по «корзинам», о чем будет рассказано далее.

Несколько важных пунктов о HashSet:

* Т.к. класс реализует интерфейс Set, он может хранить только уникальные значения;
* Может хранить NULL – значения;
* Порядок добавления элементов вычисляется с помощью хэш-кода;
* HashSet также реализует интерфейсы Serializable и Cloneable.

Для поддержания постоянного времени выполнения операций время, затрачиваемое на действия с HashSet, должно быть прямо пропорционально количеству элементов в HashSet + «емкость» встроенного экземпляра HashMap (количество «корзин»). Поэтому для поддержания производительности очень важно не устанавливать слишком высокую начальную ёмкость (или слишком низкий коэффициент загрузки).

Начальная емкость – изначальное количество ячеек («корзин») в хэш-таблице. Если все ячейки будут заполнены, их количество увеличится автоматически.

Коэффициент загрузки – показатель того, насколько заполненным может быть HashSet до того момента, когда его емкость автоматически увеличится. Когда количество элементов в HashSet становится больше, чем произведение начальной емкости и коэффициента загрузки, хэш-таблица ре-хэшируется (заново вычисляются хэшкоды элементов, и таблица перестраивается согласно полученным значениям) и количество ячеек в ней увеличивается в 2 раза.

Коэффициент загрузки = Количество хранимых элементов в таблице / размер хэш-таблицы

Например, если изначальное количество ячеек в таблице равно 16, и коэффициент загрузки равен 0,75, то из этого следует, что когда количество заполненных ячеек достигнет 12, их количество автоматически увеличится.

Коэффициент загрузки и начальная емкость – два главных фактора, от которых зависит производительность операций с HashSet. Коэффициент загрузки, равный 0,75, в среднем обеспечивает хорошую производительность. Если этот параметр увеличить, тогда уменьшится нагрузка на память (так как это уменьшит количество операций ре-хэширования и перестраивания), но это повлияет на операции добавления и поиска. Чтобы минимизировать время, затрачиваемое на ре-хэширование, нужно правильно подобрать параметр начальной емкости. Если начальная емкость больше, чем максимальное количество элементов, поделенное на коэффициент загрузки, то никакой операции ре-хэширования не произойдет в принципе.

Важно: HashSet не является структурой данных с встроенной синхронизацией, поэтому если с ним работают одновременно несколько потоков, и как минимум один из них пытается внести изменения, необходимо обеспечить синхронизированный доступ извне. Часто это делается за счет другого синхронизируемого объекта, инкапсулирующего HashSet. Если такого объекта нет, то лучше всего подойдет метод Collections.synchronizedSet(). На данный момент это лучшее средство для предотвращения несинхронизированных операций с HashSet.

Все классы, реализующие интерфейс Set, внутренне поддерживаются реализациями Map. HashSet хранит элементы с помощью HashMap. Хоть и для добавления элемента в HashMap он должен быть представлен в виде пары «ключ-значение», в HashSet добавляется только значение.

На самом деле значение, которые мы передаем в HashSet, является ключом к объекту HashMap, а в качестве значения в HashMap используется константа. Таким образом, в каждой паре «ключ-значение» все ключи будут иметь одинаковые значения.

# Как работает HashMap? Расскажите подробно, как работает метод put? Что происходит при коллизии?

Внутри Map данные хранятся в формате “ключ”-”значение”, то есть по парам. И в качестве ключей, и в качестве значений могут выступать любые объекты — числа, строки или объекты других классов.

Реализация интерфейса Map на основе хэш-таблиц . Эта реализация предоставляет все необязательные операции map и допускает нулевые значения и нулевой ключ. (Класс HashMap примерно эквивалентен Hashtable , за исключением того, что он несинхронизирован и допускает значения NULL.) Этот класс не дает никаких гарантий относительно порядка отображения; в частности, это не гарантирует, что порядок останется постоянным с течением времени.

Эта реализация обеспечивает постоянную производительность для основных операций ( получение и размещение ), предполагая, что хеш-функция правильно распределяет элементы по сегментам. Итерация по представлениям коллекций требует времени, пропорционального «емкости» экземпляра HashMap (количеству сегментов) плюс его размеру (количеству сопоставлений «ключ-значение»). Таким образом, очень важно не устанавливать слишком высокую начальную емкость (или слишком низкий коэффициент загрузки), если важна производительность итераций.

Экземпляр HashMap имеет два параметра, которые влияют на его производительность: начальная емкость и коэффициент загрузки . Емкость является количество ковшей в хэш - таблице, а начальная емкость просто емкость в момент создания таблицы хеширования. Коэффициент загрузки - это мера того, насколько может быть заполнена хеш-таблица до того, как ее емкость автоматически увеличится. Когда количество записей в хеш-таблице превышает произведение коэффициента загрузки и текущей емкости, хеш-таблица перестраивается (то есть перестраиваются внутренние структуры данных), так что хеш-таблица имеет примерно в два раза больше сегментов.

Как правило, коэффициент загрузки по умолчанию (0,75) предлагает хороший компромисс между затратами времени и места. Более высокие значения уменьшают накладные расходы на пространство, но увеличивают стоимость поиска (отражается в большинстве операций класса HashMap , включая получение и размещение ). Ожидаемое количество записей на карте и ее коэффициент загрузки следует учитывать при установке ее начальной емкости, чтобы минимизировать количество операций повторного хеширования. Если начальная емкость больше, чем максимальное количество записей, разделенное на коэффициент загрузки, операции повторного хеширования никогда не будут выполняться.

Если в экземпляре HashMap должно храниться много сопоставлений , создание его с достаточно большой емкостью позволит хранить сопоставления более эффективно, чем выполнение автоматического повторного хеширования по мере необходимости для увеличения таблицы.

# Как будет разрешаться коллизия?

Когда вы вставляете пару (10, 17) и затем (10, 20), технически никакого столкновения не происходит. Вы просто заменяете старое значение новым значением для данного ключа 10 (так как в обоих случаях 10 равно 10, а код hash для 10 всегда равен 10).

Столкновение происходит, когда несколько ключей hash к одному и тому же ведру. В этом случае вам нужно убедиться, что вы можете различать эти ключи. Цепочка разрешения коллизий является одним из тех методов, которые используются для этого.

В качестве примера предположим, что две строки "abra ka dabra" и "wave my wand" дают hash кодов 100 и 200 соответственно. Предполагая, что общий размер массива равен 10, оба они оказываются в одном и том же ведре ( 100 % 10 и 200 % 10 ). Цепочка гарантирует, что всякий раз , когда вы делаете map.get( "abra ka dabra" );, вы получаете правильное значение, связанное с ключом. В случае hash map в Java это делается с помощью метода equals .

В A HashMap ключом является объект, содержащий методы hashCode() и equals(Object) .

Когда вы вставляете новую запись в карту, она проверяет, известна ли уже hashCode . Затем он будет перебирать все объекты с этим хэшкодом и проверять их равенство с .equals() . Если найден равный объект,то новое значение заменяет старое. Если нет, то он создаст новую запись на карте.

Обычно, говоря о картах, вы используете столкновение , когда два объекта имеют один и тот же hashCode , но они разные. Они внутренне хранятся в списке.

Про put и get

<https://habr.com/ru/post/421179/>

# **Может ли null быть ключём в HashMap?**

Поскольку HashCode допускает ключ типа null, хэш код null всегда будет равен 0.

# Как работает метод get в HashMap?

Про put и get

<https://habr.com/ru/post/421179/>

1. **Внутреннее устройство HashMap**

# При каких условиях в HashMap лист преобразуется в красно-черное дерево?"

Если в одном ведре есть по крайней мере 8 записей ( TREEIFY\_THRESHOLD ) и общее количество ведер больше 64 ( MIN\_TREEIFY\_CAPACITY), то это одно ведро будет преобразовано в идеально сбалансированный красно-черный узел дерева .

[**https://coderoad.ru/47921663/%D0%9A%D0%BE%D0%B3%D0%B4%D0%B0-%D0%B8-%D0%BA%D0%B0%D0%BA-HashMap-%D0%BF%D1%80%D0%B5%D0%BE%D0%B1%D1%80%D0%B0%D0%B7%D1%83%D0%B5%D1%82-%D0%B2%D0%B5%D0%B4%D1%80%D0%BE-%D0%B8%D0%B7-%D1%81%D0%B2%D1%8F%D0%B7%D0%B0%D0%BD%D0%BD%D0%BE%D0%B3%D0%BE-%D1%81%D0%BF%D0%B8%D1%81%D0%BA%D0%B0-%D0%B2-%D0%BA%D1%80%D0%B0%D1%81%D0%BD%D1%8B%D0%B5-%D1%87%D0%B5%D1%80%D0%BD%D1%8B%D0%B5**](https://coderoad.ru/47921663/%D0%9A%D0%BE%D0%B3%D0%B4%D0%B0-%D0%B8-%D0%BA%D0%B0%D0%BA-HashMap-%D0%BF%D1%80%D0%B5%D0%BE%D0%B1%D1%80%D0%B0%D0%B7%D1%83%D0%B5%D1%82-%D0%B2%D0%B5%D0%B4%D1%80%D0%BE-%D0%B8%D0%B7-%D1%81%D0%B2%D1%8F%D0%B7%D0%B0%D0%BD%D0%BD%D0%BE%D0%B3%D0%BE-%D1%81%D0%BF%D0%B8%D1%81%D0%BA%D0%B0-%D0%B2-%D0%BA%D1%80%D0%B0%D1%81%D0%BD%D1%8B%D0%B5-%D1%87%D0%B5%D1%80%D0%BD%D1%8B%D0%B5)

# Что такое даймонд оператор?

Раньше нужно было явно указывать тип справа. Теперь с даймонд оператором кавычки можно оставить пустыми. Главное указать тип слева.

# Что такое raw type? К чему приводит использование raw type? +++от других1. Чему эквивалентно <?>

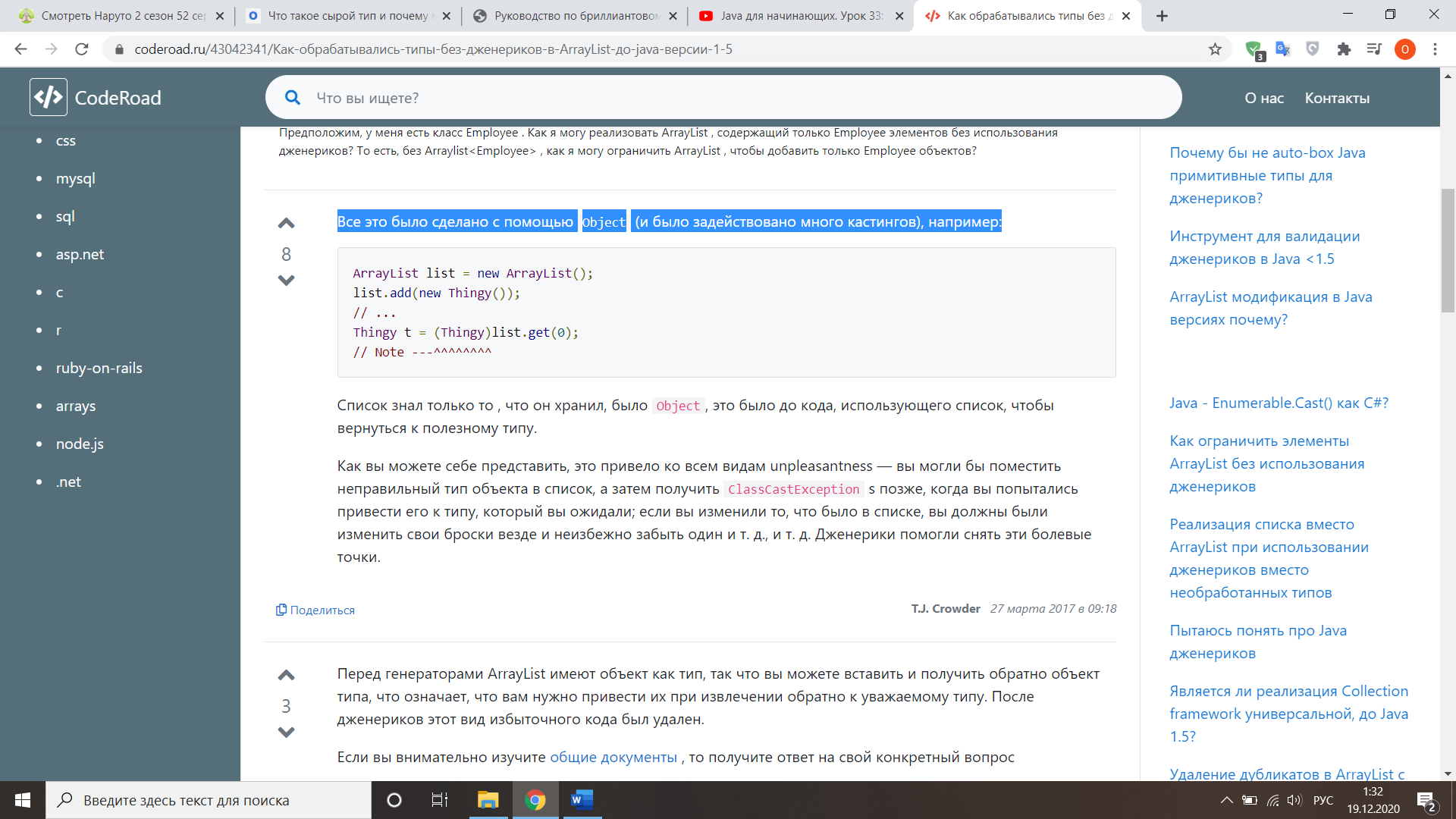
"raw type" - это использование универсального класса без указания аргумента(ОВ) типа для его параметризованного типа(ов), например, использование List вместо List<String> . Когда дженерики были введены в Java, несколько классов были обновлены для использования дженериков. Использование этого класса в качестве "raw type" (без указания аргумента типа) позволило унаследованному коду все еще компилироваться.

"Raw types" используются для обратной совместимости. Их использование в новом коде не рекомендуется, так как использование универсального класса с аргументом типа позволяет усилить типизацию, что в свою очередь может улучшить понятность кода и привести к более раннему обнаружению потенциальных проблем.

При использовании raw types нам нужно указывать явное приведение (даун кастинг) к нужному типу, т.к raw types хранит Object.

# Что было до дженериков?

Все это было сделано с помощью Object (и было задействовано много кастингов), например:



Список знал только то , что он хранил, было Object, это было до кода, использующего список, чтобы вернуться к полезному типу.

Как вы можете себе представить, это привело ко всем видам unpleasantness — вы могли бы поместить неправильный тип объекта в список, а затем получить ClassCastException s позже, когда вы попытались привести его к типу, который вы ожидали; если вы изменили то, что было в списке, вы должны были изменить свои броски везде и неизбежно забыть один и т. д., и т. д. Дженерики помогли снять эти болевые точки.

# Если поле типизировано дженериком как в байт коде будет представлен этот тип?

Дженерики появились только в версии языка Java 5. К моменту ее выхода программисты успели написать кучу кода с использованием Raw Types, и чтобы он не перестал работать, возможность создания и работы Raw Types в Java сохранилась.

Однако эта проблема оказалась гораздо обширнее.

Java-код, как ты знаешь, преобразуется в специальный байт-код, который потом выполняется виртуальной машиной Java.

И если бы в процессе перевода мы помещали в байт-код информацию о типах-параметрах, это сломало бы весь ранее написанный код, ведь до Java 5 никаких типов-параметров не существовало!

При работе с дженериками есть одна очень важная особенность, о которой необходимо помнить. Она называется “стирание типов” (type erasure).

Ее суть заключается в том, что внутри класса не хранится никакой информации о его типе-параметре.

Эта информация доступна только на этапе компиляции и стирается (становится недоступной) в runtime.

Если ты попытаешься положить объект не того типа в свой List<String>, компилятор выдаст ошибку. Этого как раз и добивались создатели языка, создавая дженерики — проверки на этапе компиляции.

Но когда весь написанный тобой Java-код превратится в байт-код, в нем не будет информации о типах-параметрах.

Внутри байт-кода твой список List<Cat> cats не будет отличаться от List<String> strings. В байт-коде ничто не будет говорить о том, что cats — это список объектов Cat. Информация об этом сотрется во время компиляции, и в байт код попадет только информация о том, что у тебя в программе есть некий список List<Object> cats.

# Почему Map не входит в Collection?

Реализации интерфейса Map также, как и реализации коллекций (List, Set), используются для формирования набора данных. Однако, если в коллекции элемент данных представляет определенный Object, то в реализации Map элемент является совокупностью пары "ключ-значение". Соответственно, некоторые методы Collection нельзя использовать в Map. Например, метод remove(Object o) в коллекции предназначен для удаления элемента, в то время, как такой же метод remove(Object key) в интерфейсе Map предназначен для удаления элемента по заданному ключу.

# Зачем в итераторе метод remove?

Как вы уже сказали, итератор используется, когда вы хотите удалить материал во время итерации по содержимому массива. Если вы не используете итератор, а просто имеете цикл for и внутри него используете метод remove, вы получите исключения, потому что содержимое массива изменяется во время итерации. например: вы можете подумать, что размер массива равен 10 в начале цикла for, но это не будет так, как только вы удалите материал.. поэтому, когда вы достигнете последних петель, вероятно, будет IndexOutofBoundsException и т. д.

# Что такое Map? Что должно быть уникальным?

Интерфейс Map<K, V> представляет отображение или иначе говоря словарь, где каждый элемент представляет пару "ключ-значение". При этом все ключи уникальные в рамках объекта Map. Такие коллекции облегчают поиск элемента, если нам известен ключ - уникальный идентификатор объекта.

Следует отметить, что в отличие от других интерфейсов, которые представляют коллекции, интерфейс Map НЕ расширяет интерфейс Collection.

Среди методов интерфейса Map можно выделить следующие:

* void clear(): очищает коллекцию
* boolean containsKey(Object k): возвращает true, если коллекция содержит ключ k
* boolean containsValue(Object v): возвращает true, если коллекция содержит значение v
* Set<Map.Entry<K, V>> entrySet(): возвращает набор элементов коллекции. Все элементы представляют объект Map.Entry
* boolean equals(Object obj): возвращает true, если коллекция идентична коллекции, передаваемой через параметр obj
* boolean isEmpty: возвращает true, если коллекция пуста
* V get(Object k): возвращает значение объекта, ключ которого равен k. Если такого элемента не окажется, то возвращается значение null
* V getOrDefault(Object k, V defaultValue): возвращает значение объекта, ключ которого равен k. Если такого элемента не окажется, то возвращается значение defaultVlue
* V put(K k, V v): помещает в коллекцию новый объект с ключом k и значением v. Если в коллекции уже есть объект с подобным ключом, то он перезаписывается. После добавления возвращает предыдущее значение для ключа k, если он уже был в коллекции. Если же ключа еще не было в коллекции, то возвращается значение null
* V putIfAbsent(K k, V v): помещает в коллекцию новый объект с ключом k и значением v, если в коллекции еще нет элемента с подобным ключом.
* Set<K> keySet(): возвращает набор всех ключей отображения
* Collection<V> values(): возвращает набор всех значений отображения
* void putAll(Map<? extends K, ? extends V> map): добавляет в коллекцию все объекты из отображения map
* V remove(Object k): удаляет объект с ключом k
* int size(): возвращает количество элементов коллекции

Чтобы положить объект в коллекцию, используется метод put, а чтобы получить по ключу - метод get. Реализация интерфейса Map также позволяет получить наборы как ключей, так и значений. А метод entrySet() возвращает набор всех элементов в виде объектов Map.Entry<K, V>.

Обобщенный интерфейс Map.Entry<K, V> представляет объект с ключом типа K и значением типа V и определяет следующие методы:

* boolean equals(Object obj): возвращает true, если объект obj, представляющий интерфейс Map.Entry, идентичен текущему
* K getKey(): возвращает ключ объекта отображения
* V getValue(): возвращает значение объекта отображения
* V setValue(V v): устанавливает для текущего объекта значение v
* int hashCode(): возвращает хеш-код данного объекта

При переборе объектов отображения мы будем оперировать этими методами для работы с ключами и значениями объектов.

# что кладётся на место значения в HashSet?

Все классы, реализующие интерфейс Set, внутренне поддерживаются реализациями Map. HashSet хранит элементы с помощью HashMap. Хоть и для добавления элемента в HashMap он должен быть представлен в виде пары «ключ-значение», в HashSet добавляется только значение.

На самом деле значение, которые мы передаем в HashSet, является ключом к объекту HashMap, а в качестве значения в HashMap используется константа. Таким образом, в каждой паре «ключ-значение» все ключи будут иметь одинаковые значения.

# В чём разница между Iterable и Iterator?

Iterable представляет собой простое представление серии элементов, которые могут быть повторены. У него нет состояния итерации, такого как «текущий элемент». Вместо этого у него есть один метод, который производит Iterator.

Iterator - объект с состоянием итерации. Он позволяет вам проверить, есть ли в нем больше элементов, hasNext()и перейти к следующему элементу (если есть), используя next().

Как правило, Iterable должно быть в состоянии произвести любое количество действительных Iterators.

Как работает метод contains в ArrayList, LinkedList, HashSet?

Метод коллекции contains() позволяет проверить наличие определенного элемента в наборе. Процесс проверки выполняется в цикле для всех элементов набора пока не будет найден соответствующий «эквивалент». При сравнении объектов используется метод equals() элемента набора. Поэтому, необходимо быть внимательным, если набор включает сложные объекты, для которых требуется метод [equals()](http://java-online.ru/java-interview-02.xhtml" \l "equals) переписать. Следующий пример демонстрирует использование метода contains() для простых строковых объектов набора; в консоль будет выведено логическое значение 'true'.

List<String> list = new ArrayList<String>();

list.add("Весна");

list.add("Лето");

list.add("Осень");

list.add("Зима");

System.out.println("" + list.contains("Лето"));

Незначительно усложняем пример и создаем коллекцию из объектов Saison, который включает два поля : id и name. Теперь, чтобы метод (проверки наличия в коллекции элемента) contains() работал корректно необходимо в классе Saison переписать метод equals(), в котором выполнить соответствующее сравнение двух объектов. В примере сравнение объектов выполняется для целочисленных идентификаторов id, поэтому метод equals() вернет true для любых строковых значений name и в консоль будет выведено 'true'.

class Saison {

public int id;

public String name;

public Saison(int id, String name) {

this.id = id;

this.name = name;

}

@Override

public boolean equals(Object obj) {

return (id == ((Saison)obj).id);

}

}

. . .

List<Saison> list = new ArrayList<Saison>();

list.add(new Saison(1, "Весна"));

list.add(new Saison(2, "Лето"));

list.add(new Saison(3, "Осень"));

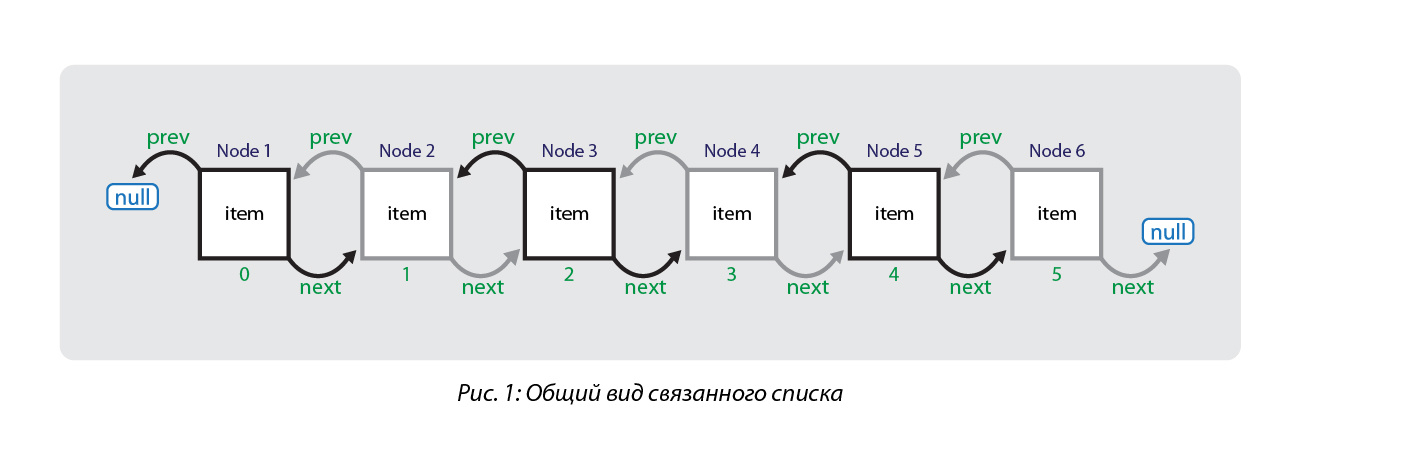
list.add(new Saison(4, "Зима"));

System.out.println("" + list.contains(new Saison(2, "Весна")));

Листинг метода contains() можно посмотреть [здесь](http://java-online.ru/java-collection.xhtml).

# как устроен LinkedList

LinkedList — класс, реализующий два интерфейса — List и Deque. Это обеспечивает возможность создания двунаправленной очереди из любых (в том числе и null) элементов. Каждый объект, помещенный в связанный список, является узлом (нодом). Каждый узел содержит элемент, ссылку на предыдущий и следующий узел. Фактически связанный список состоит из последовательности узлов, каждый из которых предназначен для хранения объекта определенного при создании типа.



# Отличие ArrayList от LinkedList?

ArrayList - это список на основе массива. LinkedList - связанный список на основе элементов и связи между ними. В качестве LinkedList лучше всего подходит представление вагонов поезда сцепленных последовательно.

ArrayList следует использовать, когда в приоритете доступ по индексу, так как эти операции выполняются за константное время. Добавление в конец списка в среднем тоже выполняется за константное время. Кроме того в ArrayList нет дополнительных расходов на хранение связки между элементами. Минусы в скорости вставки/удаления элементов находящихся не в конце списка, так как при этой операции все элементы правее добавляемого/удаляемого сдвигаются.

LinkedList удобен когда важнее быстродействие операций вставки/удаления, которые в LinkedList выполняются за константное время. Операции доступа по индексу производятся перебором с начала или конца (смотря что ближе) до нужного элемента. Дополнительные затраты на хранение связки между элементами.

Одним словом - если часто вставляете/удаляете - выбирайте в пользу LinkedList, в противном случае ArrayList

# Когда лучше использовать ArrayList, а когда LinkedList?

Ответ в 24 вопросе

# Расскажи отличие List от Set?

List - упорядоченная последовательность элементов, тогда как Set - это отдельный список элементов, которые неупорядочены

List - это тип упорядоченной коллекции, которая поддерживает элементы в порядке вставки, в то время как Set - это тип неупорядоченной коллекции, поэтому элементы не поддерживаются в каком-либо порядке.

List позволяет дублировать, а Set не позволяет дублировать элементы. Все элементы набора должны быть уникальными, если вы попытаетесь вставить дублирующий элемент в Set, это заменит существующее значение.

List разрешает любое количество нулевых значений в своей коллекции, в то время как Set разрешает только одно значение null в его коллекции.

Новые методы определены внутри интерфейса List. Но в интерфейсе Set нет новых методов, поэтому мы должны использовать методы интерфейса Collection только с помощью подклассов Set.

List можно вставлять как в прямом направлении, так и в обратном направлении с помощью Listiterator, в то время как Set можно перемещать только в прямом направлении с помощью итератора

**Когда использовать List и Set в Java**

List - упорядоченная последовательность элементов, однако set - это отличный список неупорядоченного элемента. Таким образом, используйте список для хранения неидеальных объектов в соответствии с порядком вставки и используйте набор для хранения уникальных объектов в случайном порядке

# В чём разница между Queue и Deque и Stack?

Современные программисты сейчас практически не используют Stack, который слишком прост и не очень гибок. Тем не менее, изучить его стоит, может именно он вам и пригодится.

Stack является подклассом класса Vector, который реализует простой механизм типа "последний вошёл - первый вышел" (LIFO). Можно представить процесс в виде детской пирамидки, когда вы по одному нанизываете диск на колышек. И снять диски вы сможете только по порядку, начиная с верхнего.

Интерфейс Queue

Обобщенный интерфейс Queue<E> расширяет базовый интерфейс Collection и определяет поведение класса в качестве однонаправленной очереди. Свою функциональность он раскрывает через следующие методы:

* E element(): возвращает, но не удаляет, элемент из начала очереди. Если очередь пуста, генерирует исключение NoSuchElementException
* boolean offer(E obj): добавляет элемент obj в конец очереди. Если элемент удачно добавлен, возвращает true, иначе - false
* E peek(): возвращает без удаления элемент из начала очереди. Если очередь пуста, возвращает значение null
* E poll(): возвращает с удалением элемент из начала очереди. Если очередь пуста, возвращает значение null
* E remove(): возвращает с удалением элемент из начала очереди. Если очередь пуста, генерирует исключение NoSuchElementException

Таким образом, у всех классов, которые реализуют данный интерфейс, будет метод offer для добавления в очередь, метод poll для извлечения элемента из головы очереди, и методы peek и element, позволяющие просто получить элемент из головы очереди.

Интерфейс Deque

Интерфейс Deque расширяет вышеописанный интерфейс Queue и определяет поведение двунаправленной очереди, которая работает как обычная однонаправленная очередь, либо как стек, действующий по принципу LIFO (последний вошел - первый вышел).

Интерфейс Deque определяет следующие методы:

* void addFirst(E obj): добавляет элемент в начало очереди
* void addLast(E obj): добавляет элемент obj в конец очереди
* E getFirst(): возвращает без удаления элемент из головы очереди. Если очередь пуста, генерирует исключение NoSuchElementException
* E getLast(): возвращает без удаления последний элемент очереди. Если очередь пуста, генерирует исключение NoSuchElementException
* boolean offerFirst(E obj): добавляет элемент obj в самое начало очереди. Если элемент удачно добавлен, возвращает true, иначе - false
* boolean offerLast(E obj): добавляет элемент obj в конец очереди. Если элемент удачно добавлен, возвращает true, иначе - false
* E peekFirst(): возвращает без удаления элемент из начала очереди. Если очередь пуста, возвращает значение null
* E peekLast(): возвращает без удаления последний элемент очереди. Если очередь пуста, возвращает значение null
* E pollFirst(): возвращает с удалением элемент из начала очереди. Если очередь пуста, возвращает значение null
* E pollLast(): возвращает с удалением последний элемент очереди. Если очередь пуста, возвращает значение null
* E pop(): возвращает с удалением элемент из начала очереди. Если очередь пуста, генерирует исключение NoSuchElementException
* void push(E element): добавляет элемент в самое начало очереди
* E removeFirst(): возвращает с удалением элемент из начала очереди. Если очередь пуста, генерирует исключение NoSuchElementException
* E removeLast(): возвращает с удалением элемент из конца очереди. Если очередь пуста, генерирует исключение NoSuchElementException
* boolean removeFirstOccurrence(Object obj): удаляет первый встреченный элемент obj из очереди. Если удаление произшло, то возвращает true, иначе возвращает false.
* boolean removeLastOccurrence(Object obj): удаляет последний встреченный элемент obj из очереди. Если удаление произшло, то возвращает true, иначе возвращает false.

Таким образом, наличие методов pop и push позволяет классам, реализующим этот элемент, действовать в качестве стека. В тоже время имеющийся функционал также позволяет создавать двунаправленные очереди, что делает классы, применяющие данный интерфейс, довольно гибкими.

1. **public class Ex1 { public static void main(String[] args) {**

**List<Integer> integerList = Arrays.asList(1, 2, 3);**

**Gen gen = new Gen();**

**gen.m(integerList);**

**} static class Gen<T> {**

**<T> void m(Collection<T> collection) {**

**for (T s : collection) {**

**System.out.println(s);**

**}**

**} <T> void m(List<String> list) {**

**for (String s : list) {**

**System.out.println(s);**

**}**

**}**

**}**

**}Что произойдет? Почему отработает 2 метод? Как это исправить?**

**java.lang.Integer cannot be cast to class java.lang.String**

# какой хэш-код у null в HashMap

0

# +

Односвязанный список хранит ссылку только на следующий элемент, тогда как двусвязаный список хранит в обе стороны

# Почему в HashSet вместо value не null а new Object?

Потому что контракт HashSet указывает, что remove() возвращает true , если указанный объект существовал и был удален. Для этого он использует обернутый HashMap#remove() , который возвращает удаленное значение.

Если бы вы хранили null вместо объекта , то вызов HashMap#remove() вернул бы null, что было бы неотличимо от результата попытки удалить несуществующий объект, и контракт HashSet.remove() не мог бы быть выполнен.

**Бинарное дерево** представляет собой иерархическую структуру данных, в которой каждый узел имеет не более двух дочерних узлов. Как правило, первый называется родительским узлом или корнем **дерева** (root), а дочерние узлы называются левым и правым наследниками.