Вопросы для собеседования

Java Collections Framework

- Что такое *«коллекция»*?
- Назовите основные интерфейсы JCF и их реализации.
- Расположите в виде иерархии следующие интерфейсы: `List`, `Set`, `Map`, `SortedSet`, `SortedMap`, `Collection`, `Iterable`, `Iterator`, `NavigableSet`, `NavigableMap`.
- Почему `Map` — это не `Collection`, в то время как `List` и `Set` являются `Collection`?
- В чем разница между классами `java.util.Collection` и `java.util.Collections`?
- Что такое «fail-fast поведение»?
- Какая разница между fail-fast и fail-safe?
- Приведите примеры итераторов, реализующих поведение fail-safe
- Чем различаются `Enumeration` и `Iterator`.
- Как между собой связаны `Iterable` и `Iterator`?
- Как между собой связаны `Iterable`, `Iterator` и «for-each»?
- Сравните `Iterator` и `ListIterator`.

- Что произойдет при вызове `Iterator.next()` без предварительного вызова `Iterator.hasNext()` ?
- Сколько элементов будет пропущено, если `Iterator.next()` будет вызван после 10-ти вызовов `Iterator.hasNext()` ?
- Как поведёт себя коллекция, если вызвать `iterator.remove()` ?
- Как поведёт себя уже инстанциированный итератор для `collection` , если вызвать `collection.remove()` ?
- Как избежать `ConcurrentModificationException` во время перебора коллекции?
- Какая коллекция реализует дисциплину обслуживания FIFO?
- Какая коллекция реализует дисциплину обслуживания FILO?
- Чем отличается `ArrayList` от `Vector` ?
- Зачем добавили `ArrayList` , если уже был `Vector` ?
- Чем отличается `ArrayList` от `LinkedList` ? В каких случаях лучше использовать первый, а в каких второй?
- Что работает быстрее `ArrayList` или `LinkedList` ?
- Какое худшее время работы метода `contains()` для элемента, который есть в `LinkedList` ?
- Какое худшее время работы метода `contains()` для элемента, который есть в `ArrayList` ?
- Какое худшее время работы метода `add()` для `LinkedList` ?
- Какое худшее время работы метода `add()` для `ArrayList` ?
- Необходимо добавить 1 млн. элементов, какую структуру вы используете?
- Как происходит удаление элементов из `ArrayList` ? Как меняется в этом случае размер `ArrayList` ?
- Предложите эффективный алгоритм удаления нескольких рядом стоящих элементов из середины списка, реализуемого `ArrayList` .
- Сколько необходимо дополнительной памяти при вызове `ArrayList.add()` ?
- Сколько выделяется дополнительно памяти при вызове `LinkedList.add()` ?
- Оцените количество памяти на хранение одного примитива типа `byte` в `LinkedList` ?
- Оцените количество памяти на хранение одного примитива типа `byte` в `ArrayList` ?
- Для `ArrayList` или для `LinkedList` операция добавления элемента в середину (`list.add(list.size()/2, newElement)`) медленнее?
- В реализации класса `ArrayList` есть следующие поля: `Object[] elementData` ,

`int size`. Объясните, зачем хранить отдельно `size`, если всегда можно взять `elementData.length`?

- Сравните интерфейсы `Queue` и `Deque`.
- Кто кого расширяет: `Queue` расширяет `Deque`, или `Deque` расширяет `Queue`?
- Почему `LinkedList` реализует и `List`, и `Deque`?
- `LinkedList` — это односвязный, двусвязный или четырехсвязный список?
- Как перебрать элементы `LinkedList` в обратном порядке, не используя медленный `get(index)`?
- Что позволяет сделать `PriorityQueue`?
- `Stack` считается «устаревшим». Чем его рекомендуют заменять? Почему?
- Зачем нужен `HashMap`, если есть `Hashtable`?
- В чем разница между `HashMap` и `IdentityHashMap`? Для чего нужна `IdentityHashMap`?
- В чем разница между `HashMap` и `WeakHashMap`? Для чего используется `WeakHashMap`?
- В `WeakHashMap` используются WeakReferences. А почему бы не создать `SoftHashMap` на SoftReferences?
- В `WeakHashMap` используются WeakReferences. А почему бы не создать `PhantomHashMap` на PhantomReferences?
- `LinkedHashMap` - что в нем от `LinkedList`, а что от `HashMap`?
- В чем проявляется «сортированность» `SortedMap`, кроме того, что `toString()` выводит все элементы по порядку?
- Как устроен `HashMap`?
- Согласно Кнуту и Кормену существует две основных реализации хэш-таблицы: на основе открытой адресации и на основе метода цепочек. Как реализована `HashMap`? Почему, по вашему мнению, была выбрана именно эта реализация? В чем плюсы и минусы каждого подхода?
- Как работает `HashMap` при попытке сохранить в него два элемента по ключам с одинаковым `hashCode()`, но для которых `equals() == false`?
- Какое начальное количество корзин в `HashMap`?
- Какова оценка временной сложности операций над элементами из `HashMap`? Гарантирует ли `HashMap` указанную сложность выборки элемента?
- Возможна ли ситуация, когда `HashMap` выродится в список даже с ключами имеющими разные `hashCode()`?
- В каком случае может быть потерян элемент в `HashMap`?

- Почему нельзя использовать `byte[]` в качестве ключа в `HashMap`?
- Какова роль `equals()` и `hashCode()` в `HashMap`?
- Каково максимальное число значений `hashCode()`?
- Какое худшее время работы метода get(key) для ключа, которого нет в `HashMap`?
- Какое худшее время работы метода get(key) для ключа, который есть в `HashMap`?
- Сколько переходов происходит в момент вызова `HashMap.get(key)` по ключу, который есть в таблице?
- Сколько создается новых объектов, когда вы добавляете новый элемент в `HashMap`?
- Как и когда происходит увеличение количества корзин в `HashMap`?
- Объясните смысл параметров в конструкторе `HashMap(int initialCapacity, float loadFactor)`.
- Будет ли работать `HashMap`, если все добавляемые ключи будут иметь одинаковый `hashCode()`?
- Как перебрать все ключи `Map`?
- Как перебрать все значения `Map`?
- Как перебрать все пары «ключ-значение» в `Map`?
- В чем отличия `TreeSet` и `HashSet`?
- Что будет, если добавлять элементы в `TreeSet` по возрастанию?
- Чем `LinkedHashSet` отличается от `HashSet`?
- Для `Enum` есть специальный класс `java.util.EnumSet`. Зачем? Чем авторов не устраивал `HashSet` или `TreeSet`?
- Какие существуют способы перебирать элементы списка?
- Каким образом можно получить синхронизированные объекты стандартных коллекций?
- Как получить коллекцию только для чтения?
- Напишите однопоточную программу, которая заставляет коллекцию выбросить `ConcurrentModificationException`.
- Приведите пример, когда какая-либо коллекция выбрасывает `UnsupportedOperationException`.
- Реализуйте симметрическую разность двух коллекций используя методы `Collection` (`addAll(...)`, `removeAll(...)`, `retainAll(...)`).
- Как, используя LinkedHashMap, сделать кэш с «invalidation policy»?

- Как одной строчкой скопировать элементы любой `collection` в массив?
- Как одним вызовом из `List` получить `List` со всеми элементами, кроме первых и последних 3-х?
- Как одной строчкой преобразовать `HashSet` в `ArrayList`?
- Как одной строчкой преобразовать `ArrayList` в `HashSet`?
- Сделайте `HashSet` из ключей `HashMap`.
- Сделайте `HashMap` из `HashSet<Map.Entry<K, V>>`.

Что такое *«коллекция»*?

«Коллекция» - это структура данных, набор каких-либо объектов. Данными (объектами в наборе) могут быть числа, строки, объекты пользовательских классов и т.п.

к оглавлению

Назовите основные интерфейсы JCF и их реализации.

На вершине иерархии в Java Collection Framework располагаются 2 интерфейса: `Collection` и `Map`. Эти интерфейсы разделяют все коллекции, входящие во фреймворк на две части по типу хранения данных: простые последовательные наборы элементов и наборы пар «ключ — значение» соответственно.

Интерфейс `Collection` расширяют интерфейсы:

- `List` (список) представляет собой коллекцию, в которой допустимы дублирующие значения. Реализации:
  - `ArrayList` - инкапсулирует в себе обычный массив, длина которого автоматически увеличивается при добавлении новых элементов. Элементы такой коллекции пронумерованы, начиная от нуля, к ним можно обратиться по индексу.
  - `LinkedList` (двунаправленный связный список) - состоит из узлов, каждый из которых содержит как собственно данные, так и две ссылки на следующий и предыдущий узел.
  - `Vector` — реализация динамического массива объектов, методы которой синхронизированы.
  - `Stack` — реализация стека LIFO (last-in-first-out).
- `Set` (сет) описывает неупорядоченную коллекцию, не содержащую

повторяющихся элементов. Реализации:

- `HashSet` - использует HashMap для хранения данных. В качестве ключа и значения используется добавляемый элемент. Из-за особенностей реализации порядок элементов не гарантируется при добавлении.
- `LinkedHashSet` — гарантирует, что порядок элементов при обходе коллекции будет идентичен порядку добавления элементов.
- `TreeSet` — предоставляет возможность управлять порядком элементов в коллекции при помощи объекта `Comparator`, либо сохраняет элементы с использованием «natural ordering».

- `Queue` (очередь) предназначена для хранения элементов с предопределённым способом вставки и извлечения FIFO (first-in-first-out):
  - `PriorityQueue` — предоставляет возможность управлять порядком элементов в коллекции при помощи объекта `Comparator`, либо сохраняет элементы с использованием «natural ordering».
  - `ArrayDeque` — реализация интерфейса `Deque`, который расширяет интерфейс `Queue` методами, позволяющими реализовать конструкцию вида LIFO (last-in-first-out).

Интерфейс `Map` реализован классами:

- `Hashtable` — хэш-таблица, методы которой синхронизированы. Не позволяет использовать `null` в качестве значения или ключа и не является упорядоченной.
- `HashMap` — хэш-таблица. Позволяет использовать `null` в качестве значения или ключа и не является упорядоченной.
- `LinkedHashMap` — упорядоченная реализация хэш-таблицы.
- `TreeMap` — реализация, основанная на красно-чёрных деревьях. Является упорядоченной и предоставляет возможность управлять порядком элементов в коллекции при помощи объекта `Comparator`, либо сохраняет элементы с использованием «natural ordering».
- `WeakHashMap` — реализация хэш-таблицы, которая организована с использованием *weak references* для ключей (сборщик мусора автоматически удалит элемент из коллекции при следующей сборке мусора, если на ключ этого элемента нет жёстких ссылок).

к оглавлению

Расположите в виде иерархии следующие

интерфейсы: List, Set, Map, SortedSet, SortedMap, Collection, Iterable, Iterator, NavigableSet, NavigableMap.

- Iterable
  - Collection
    - List
    - Set
      - SortedSet
        - NavigableSet
- Map
  - SortedMap
    - NavigableMap
- Iterator

к оглавлению

Почему Map — это не Collection, в то время как List и Set являются Collection?

`Collection` представляет собой совокупность некоторых элементов. `Map` - это совокупность пар «ключ-значение».

к оглавлению

В чем разница между классами java.util.Collection и java.util.Collections?

`java.util.Collections` - набор статических методов для работы с коллекциями.

`java.util.Collection` - один из основных интерфейсов Java Collections Framework.

к оглавлению

Что такое «fail-fast поведение»?

fail-fast поведение означает, что при возникновении ошибки или состояния, которое может привести к ошибке, система немедленно прекращает дальнейшую работу и уведомляет об этом. Использование fail-fast подхода позволяет избежать недетерминированного поведения программы в течение времени.

В Java Collections API некоторые итераторы ведут себя как fail-fast и выбрасывают `ConcurrentModificationException`, если после его создания была произведена модификация коллекции, т.е. добавлен или удален элемент напрямую из коллекции, а не используя методы итератора.

Реализация такого поведения осуществляется за счет подсчета количества модификаций коллекции (modification count):

- при изменении коллекции счетчик модификаций так же изменяется;
- при создании итератора ему передается текущее значение счетчика;
- при каждом обращении к итератору сохраненное значение счетчика сравнивается с текущим, и, если они не совпадают, возникает исключение.

к оглавлению

Какая разница между fail-fast и fail-safe?

В противоположность fail-fast, итераторы fail-safe не вызывают никаких исключений при изменении структуры, потому что они работают с клоном коллекции вместо оригинала.

к оглавлению

Приведите примеры итераторов, реализующих поведение fail-safe

Итератор коллекции `CopyOnWriteArrayList` и итератор представления `keySet` коллекции `ConcurrentHashMap` являются примерами итераторов fail-safe.

к оглавлению

Чем различаются `Enumeration` и `Iterator`.

Хотя оба интерфейса и предназначены для обхода коллекций между ними имеются существенные различия:

- с помощью `Enumeration` нельзя добавлять/удалять элементы;
- в `Iterator` исправлены имена методов для повышения читаемости кода (`Enumeration.hasMoreElements()` соответствует `Iterator.hasNext()`, `Enumeration.nextElement()` соответствует `Iterator.next()` и т.д);
- `Enumeration` присутствуют в устаревших классах, таких как `Vector` / `Stack`, тогда как `Iterator` есть во всех современных классах-коллекциях.

к оглавлению

Как между собой связаны `Iterable` и `Iterator`?

Интерфейс `Iterable` имеет только один метод - `iterator()`, который возвращает `Iterator`.

к оглавлению

Как между собой связаны `Iterable`, `Iterator` и «for-each»?

Классы, реализующие интерфейс `Iterable`, могут применяться в конструкции `for-each`, которая использует `Iterator`.

к оглавлению

Сравните `Iterator` и `ListIterator`.

- `ListIterator` расширяет интерфейс `Iterator`
- `ListIterator` может быть использован только для перебора элементов коллекции `List`;
- `Iterator` позволяет перебирать элементы только в одном направлении, при помощи метода `next()`. Тогда как `ListIterator` позволяет перебирать список в обоих направлениях, при помощи методов `next()` и `previous()`;
- `ListIterator` не указывает на конкретный элемент: его текущая позиция располагается между элементами, которые возвращают методы `previous()` и `next()`.
- При помощи `ListIterator` вы можете модифицировать список, добавляя/удаляя элементы с помощью методов `add()` и `remove()`. `Iterator` не поддерживает данного функционала.

к оглавлению

Что произойдет при вызове `Iterator.next()` без предварительного вызова `Iterator.hasNext()`?

Если итератор указывает на последний элемент коллекции, то возникнет исключение `NoSuchElementException`, иначе будет возвращен следующий элемент.

к оглавлению

Сколько элементов будет пропущено, если `Iterator.next()` будет вызван после 10-ти вызовов `Iterator.hasNext()`?

Нисколько - `hasNext()` осуществляет только проверку наличия следующего элемента.

к оглавлению

Как поведёт себя коллекция, если вызвать `iterator.remove()`?

Если вызову `iterator.remove()` предшествовал вызов `iterator.next()`, то `iterator.remove()` удалит элемент коллекции, на который указывает итератор, в противном случае будет выброшено `IllegalStateException()`.

к оглавлению

Как поведёт себя уже инстанциированный итератор для `collection`, если вызвать `collection.remove()`?

При следующем вызове методов итератора будет выброшено `ConcurrentModificationException`.

к оглавлению

Как избежать `ConcurrentModificationException` во время перебора коллекции?

- Попробовать подобрать или реализовать самостоятельно другой итератор, работающий по принципу fail-safe.
- Использовать `ConcurrentHashMap` и `CopyOnWriteArrayList`.
- Преобразовать список в массив и перебирать массив.
- Блокировать изменения списка на время перебора с помощью блока `synchronized`.

Отрицательная сторона последних двух вариантов - ухудшение производительности.

к оглавлению

Какая коллекция реализует дисциплину обслуживания FIFO?

FIFO, First-In-First-Out («первым пришел-первым ушел») - по этому принципу построена коллекция `Queue`.

к оглавлению

Какая коллекция реализует дисциплину обслуживания FILO?

FILO, First-In-Last-Out («первым пришел, последним ушел») - по этому принципу построена коллекция `Stack`.

к оглавлению

Чем отличается `ArrayList` от `Vector`?

Зачем добавили `ArrayList`, если уже был `Vector`?

- Методы класса `Vector` синхронизированы, а `ArrayList` - нет;
- По умолчанию, `Vector` удваивает свой размер, когда заканчивается

выделенная под элементы память. `ArrayList` же увеличивает свой размер только на половину.

`Vector` это устаревший класс и его использование не рекомендовано.

к оглавлению

Чем отличается `ArrayList` от `LinkedList`? В каких случаях лучше использовать первый, а в каких второй?

`ArrayList` это список, реализованный на основе массива, а `LinkedList` — это классический двусвязный список, основанный на объектах с ссылками между ними.

`ArrayList`:

- доступ к произвольному элементу по индексу за *константное* время *O(1)*;
- доступ к элементам по значению за *линейное* время *O(N)*;
- вставка в конец в среднем производится за *константное* время *O(1)*;
- удаление произвольного элемента из списка занимает значительное время т.к. при этом все элементы, находящиеся «правее» смещаются на одну ячейку влево (реальный размер массива (capacity) не изменяется);
- вставка элемента в произвольное место списка занимает значительное время т.к. при этом все элементы, находящиеся «правее» смещаются на одну ячейку вправо;
- минимум накладных расходов при хранении.

`LinkedList`:

- на получение элемента по индексу или значению потребуется *линейное* время *O(N)*;
- но доступ к первому и последнему элементу списка всегда осуществляется за *константное* время *O(1)* — ссылки постоянно хранятся на первый и последний элемент;
- на добавление и удаление в начало или конец списка потребуется *константное O(1)*;
- вставка или удаление в/из произвольного место *константное O(1)*;
- но поиск позиции вставки и удаления за *линейное* время *O(N)*;

- требует больше памяти для хранения такого же количества элементов, потому что кроме самого элемента хранятся еще указатели на следующий и предыдущий элементы списка.

В целом, `LinkedList` в абсолютных величинах проигрывает `ArrayList` и по потребляемой памяти, и по скорости выполнения операций. `LinkedList` предпочтительно применять, когда нужны частые операции вставки/удаления или в случаях, когда необходимо гарантированное время добавления элемента в список.

к оглавлению

Что работает быстрее `ArrayList` или `LinkedList`?

Смотря какие действия будут выполняться над структурой.

см. Чем отличается `ArrayList` от `LinkedList`

к оглавлению

Какое худшее время работы метода `contains()` для элемента, который есть в `LinkedList`?

O(N). Время поиска элемента линейно пропорционально количеству элементов в списке.

к оглавлению

Какое худшее время работы метода `contains()` для элемента, который есть в `ArrayList`?

O(N). Время поиска элемента линейно пропорционально количеству элементов с списке.

к оглавлению

Какое худшее время работы метода `add()` для `LinkedList`?

O(N). Добавление в начало/конец списка осуществляется за время *O(1)*.

к оглавлению

Какое худшее время работы метода `add()` для `ArrayList`?

O(N). Вставка элемента в конец списка осуществляется за время *O(1)*, но если вместимость массива недостаточна, то происходит создание нового массива с увеличенным размером и копирование всех элементов из старого массива в новый.

к оглавлению

Необходимо добавить 1 млн. элементов, какую структуру вы используете?

Однозначный ответ можно дать только исходя из информации о том в какую часть списка происходит добавление элементов, что потом будет происходить с элементами списка, существуют ли какие-то ограничения по памяти или скорости выполнения.

см. Чем отличается `ArrayList` от `LinkedList`

к оглавлению

Как происходит удаление элементов из `ArrayList`? Как меняется в этом случае размер `ArrayList`?

При удалении произвольного элемента из списка, все элементы, находящиеся «правее» смещаются на одну ячейку влево и реальный размер массива (его емкость, capacity) не изменяется никак. Механизм автоматического «расширения» массива существует, а вот автоматического «сжатия» нет, можно только явно выполнить «сжатие» командой `trimToSize()`.

к оглавлению

Предложите эффективный алгоритм удаления нескольких рядом стоящих элементов из середины

списка, реализуемого `ArrayList`.

Допустим нужно удалить `n` элементов с позиции `m` в списке. Вместо выполнения удаления одного элемента `n` раз (каждый раз смещая на 1 позицию элементы, стоящие «правее» в списке), нужно выполнить смещение всех элементов, стоящих «правее» `n + m` позиции на `n` элементов «левее» к началу списка. Таким образом, вместо выполнения `n` итераций перемещения элементов списка, все выполняется за 1 проход. Но если говорить об общей эффективности - то самый быстрый способ будет с использованием `System.arraycopy()`, и получить к нему доступ можно через метод - `subList(int fromIndex, int toIndex)`

Пример:

```
import java.io.*;
import java.util.ArrayList;

public class Main {
    //позиция, с которой удаляем
    private static int m = 0;
    //количество удаляемых элементов
    private static int n = 0;
    //количество элементов в списке
    private static final int size = 1000000;
    //основной список (для удаления вызовом remove() и его копия для удаления пут
    private static ArrayList<Integer> initList, copyList;

    public static void main(String[] args){

        initList = new ArrayList<>(size);
        for (int i = 0; i < size; i++) {
            initList.add(i);
        }
        System.out.println("Список из 1.000.000 элементов заполнен");

        copyList = new ArrayList<>(initList);
        System.out.println("Создана копия списка\n");

        BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
        try{
            System.out.print("С какой позиции удаляем? > ");
            m = Integer.parseInt(br.readLine());
            System.out.print("Сколько удаляем? > ");
            n = Integer.parseInt(br.readLine());
        } catch(IOException e){
            System.err.println(e.toString());
```

```
        }
        System.out.println("\nВыполняем удаление вызовом remove()...");
        long start = System.currentTimeMillis();

        for (int i = m - 1; i < m + n - 1; i++) {
            initList.remove(i);
        }

        long finish = System.currentTimeMillis() - start;
        System.out.println("Время удаления с помощью вызова remove(): " + finish)
        System.out.println("Размер исходного списка после удаления: " + initList.

        System.out.println("\nВыполняем удаление путем перезаписи...\n");
        start = System.currentTimeMillis();

        removeEfficiently();

        finish = System.currentTimeMillis() - start;
        System.out.println("Время удаления путём смещения: " + finish);
        System.out.println("Размер копии списка:" + copyList.size());

        System.out.println("\nВыполняем удаление через SubList...\n");
        start = System.currentTimeMillis();

        initList.subList(m - 1, m + n).clear();

        finish = System.currentTimeMillis() - start;
        System.out.println("Время удаления через саблист: " + finish);
        System.out.println("Размер копии списка:" + copyList.size());
    }

    private static void removeEfficiently(){
        /* если необходимо удалить все элементы, начиная с указанного,
         * то удаляем элементы с конца до m
         */
        if (m + n >= size){
            int i = size - 1;
            while (i != m - 1){
                copyList.remove(i);
                i--;
            }
        } else{
            //переменная k необходима для отсчёта сдвига начиная от места вставка
            for (int i  = m + n, k = 0; i < size; i++, k++) {
               copyList.set(m + k, copyList.get(i));
            }

            /* удаляем ненужные элементы в конце списка
```

```
                 * удаляется всегда последний элемент, так как время этого действия
                 * фиксировано и не зависит от размера списка
                 */
                int i = size - 1;
                while (i != size - n - 1){
                    copyList.remove(i);
                    i--;
                }
                //сокращаем длину списка путём удаления пустых ячеек
                copyList.trimToSize();
            }
        }
    }
```

Результат выполнения:

```
run:
Список из 1.000.000 элементов заполнен
Создана копия списка

С какой позиции удаляем? > 600000
Сколько удаляем? > 20000

Выполняем удаление вызовом remove()...
Время удаления с помощью вызова remove(): 928
Размер исходного списка после удаления: 980000

Выполняем удаление путем перезаписи...

Время удаления путём смещения: 17
Размер копии списка:980000

Выполняем удаление через SubList...

Время удаления через саблист: 1
Размер копии списка:980000
СБОРКА УСПЕШНО ЗАВЕРШЕНА (общее время: 33 секунды)
```

к оглавлению

Сколько необходимо дополнительной памяти при вызове `ArrayList.add()`?

Если в массиве достаточно места для размещения нового элемента, то дополнительной памяти не требуется. Иначе происходит создание нового массива размером в 1,5 раза превышающим существующий (это верно для JDK выше 1.7, в более ранних версиях размер увеличения иной).

к оглавлению

Сколько выделяется дополнительно памяти при вызове `LinkedList.add()`?

Создается один новый экземпляр вложенного класса `Node`.

к оглавлению

Оцените количество памяти на хранение одного примитива типа `byte` в `LinkedList`?

Каждый элемент `LinkedList` хранит ссылку на предыдущий элемент, следующий элемент и ссылку на данные.

```
private static class Node<E> {
        E item;
        Node<E> next;
        Node<E> prev;
//...
}
```

Для 32-битных систем каждая ссылка занимает 32 бита (4 байта). Сам объект (заголовок) вложенного класса `Node` занимает 8 байт. 4 + 4 + 4 + 8 = 20 байт, а т.к. размер каждого объекта в Java кратен 8, соответственно получаем 24 байта. Примитив типа `byte` занимает 1 байт памяти, но в JCF примитивы упаковываются: объект типа `Byte` занимает в памяти 16 байт (8 байт на заголовок объекта, 1 байт на поле типа `byte` и 7 байт для кратности 8). Также напомню, что значения от -128 до 127 кэшируются и для них новые объекты каждый раз не создаются. Таким образом, в x32 JVM 24 байта тратятся на хранение одного элемента в списке и 16 байт - на хранение упакованного объекта типа `Byte`. Итого 40 байт.

Для 64-битной JVM каждая ссылка занимает 64 бита (8 байт), размер заголовка каждого объекта составляет 16 байт (два машинных слова). Вычисления аналогичны: 8 + 8 + 8 + 16 = 40байт и 24 байта. Итого 64 байта.

к оглавлению

Оцените количество памяти на хранение одного примитива типа `byte` в `ArrayList`?

`ArrayList` основан на массиве, для примитивных типов данных осуществляется автоматическая упаковка значения, поэтому 16 байт тратится на хранение упакованного объекта и 4 байта (8 для x64) - на хранение ссылки на этот объект в самой структуре данных. Таким образом, в x32 JVM 4 байта используются на хранение одного элемента и 16 байт - на хранение упакованного объекта типа `Byte`. Для x64 - 8 байт и 24 байта соответственно.

к оглавлению

Для `ArrayList` или для `LinkedList` операция добавления элемента в середину (`list.add(list.size()/2, newElement)`) медленнее?

Для `ArrayList`:

- проверка массива на вместимость. Если вместимости недостаточно, то увеличение размера массива и копирование всех элементов в новый массив (*O(N)*);
- копирование всех элементов, расположенных правее от позиции вставки, на одну позицию вправо (*O(N)*);
- вставка элемента (*O(1)*).

Для `LinkedList`:

- поиск позиции вставки (*O(N)*);
- вставка элемента (*O(1)*).

В худшем случае вставка в середину списка эффективнее для `LinkedList`. В остальных - скорее всего, для `ArrayList`, поскольку копирование элементов осуществляется за счет вызова быстрого системного метода `System.arraycopy()`.

к оглавлению

В реализации класса `ArrayList` есть следующие поля: `Object[] elementData`, `int size`. Объясните, зачем хранить отдельно `size`, если всегда можно взять `elementData.length`?

Размер массива `elementData` представляет собой вместимость (capacity) `ArrayList`, которая всегда больше переменной `size` - реального количества хранимых элементов. При необходимости вместимость автоматически возрастает.

к оглавлению

Сравните интерфейсы `Queue` и `Deque`.

Кто кого расширяет: `Queue` расширяет `Deque`, или `Deque` расширяет `Queue`?

`Queue` - это очередь, которая обычно (но необязательно) строится по принципу FIFO (First-In-First-Out) - соответственно извлечение элемента осуществляется с начала очереди, вставка элемента - в конец очереди. Хотя этот принцип нарушает, к примеру, `PriorityQueue`, использующая «natural ordering» или переданный `Comparator` при вставке нового элемента.

`Deque` (Double Ended Queue) расширяет `Queue` и согласно документации, это линейная коллекция, поддерживающая вставку/извлечение элементов с обоих концов. Помимо этого, реализации интерфейса `Deque` могут строится по принципу FIFO, либо LIFO.

Реализации и `Deque`, и `Queue` обычно не переопределяют методы `equals()` и `hashCode()`, вместо этого используются унаследованные методы класса Object, основанные на сравнении ссылок.

к оглавлению

Почему `LinkedList` реализует и `List`, и `Deque`?

`LinkedList` позволяет добавлять элементы в начало и конец списка за константное время, что хорошо согласуется с поведением интерфейса `Deque`.

к оглавлению

`LinkedList` — это односвязный, двусвязный или четырехсвязный список?

`Двусвязный` : каждый элемент `LinkedList` хранит ссылку на предыдущий и следующий элементы.

к оглавлению

Как перебрать элементы `LinkedList` в обратном порядке, не используя медленный `get(index)` ?

Для этого в `LinkedList` есть обратный итератор, который можно получить вызва метод `descendingIterator()` .

к оглавлению

Что позволяет сделать `PriorityQueue` ?

Особенностью `PriorityQueue` является возможность управления порядком элементов. По-умолчанию, элементы сортируются с использованием «natural ordering», но это поведение может быть переопределено при помощи объекта `Comparator` , который задаётся при создании очереди. Данная коллекция не поддерживает null в качестве элементов.

Используя `PriorityQueue` , можно, например, реализовать алгоритм Дейкстры для поиска кратчайшего пути от одной вершины графа к другой. Либо для хранения объектов согласно определённого свойства.

к оглавлению

`Stack` считается «устаревшим». Чем его рекомендуют заменять? Почему?

`Stack` был добавлен в Java 1.0 как реализация стека LIFO (last-in-first-out) и является расширением коллекции `Vector`, хотя это несколько нарушает понятие стека (например, класс `Vector` предоставляет возможность обращаться к любому элементу по индексу). Является частично синхронизированной коллекцией (кроме метода добавления `push()`) с вытекающими отсюда последствиями в виде негативного воздействия на производительность. После добавления в Java 1.6 интерфейса `Deque`, рекомендуется использовать реализации именно этого интерфейса, например, `ArrayDeque`.

к оглавлению

Зачем нужен `HashMap`, если есть `Hashtable`?

- Методы класса `Hashtable` синхронизированы, что приводит к снижению производительности, а `HashMap` - нет;
- `HashTable` не может содержать элементы `null`, тогда как `HashMap` может содержать один ключ `null` и любое количество значений `null`;
- Iterator у `HashMap`, в отличие от Enumeration у `HashTable`, работает по принципу «fail-fast» (выдает исключение при любой несогласованности данных).

`Hashtable` это устаревший класс и его использование не рекомендовано.

к оглавлению

В чем разница между `HashMap` и `IdentityHashMap`? Для чего нужна `IdentityHashMap`?

`IdentityHashMap` - это структура данных, так же реализующая интерфейс `Map` и использующая при сравнении ключей (значений) сравнение ссылок, а не вызов метода `equals()`. Другими словами, в `IdentityHashMap` два ключа `k1` и `k2` будут считаться равными, если они указывают на один объект, т.е. выполняется условие `k1 == k2`.

`IdentityHashMap` не использует метод `hashCode()`, вместо которого применяется метод `System.identityHashCode()`, по этой причине `IdentityHashMap` по сравнению с `HashMap` имеет более высокую производительность, особенно если последний хранит объекты с дорогостоящими методами `equals()` и `hashCode()`.

Одним из основных требований к использованию `HashMap` является неизменяемость ключа, а, т.к. `IdentityHashMap` не использует методы `equals()` и `hashCode()`, то это правило на него не распространяется.

`IdentityHashMap` может применяться для реализации сериализации/клонирования. При выполнении подобных алгоритмов программе необходимо обслуживать хэш-таблицу со всеми ссылками на объекты, которые уже были обработаны. Такая структура не должна рассматривать уникальные объекты как равные, даже если метод `equals()` возвращает `true`.

Пример кода:

```
import java.util.HashMap;
import java.util.IdentityHashMap;
import java.util.Map;

public class Q2 {

    public static void main(String[] args) {
        Q2 q = new Q2();
        q.testHashMapAndIdentityHashMap();
    }

    private void testHashMapAndIdentityHashMap() {
        CreditCard visa = new CreditCard("VISA", "04/12/2019");

        Map<CreditCard, String> cardToExpiry = new HashMap<>();
        Map<CreditCard, String> cardToExpiryIdenity = new IdentityHashMap<>();

        System.out.println("adding to HM");
        // inserting objects to HashMap
        cardToExpiry.put(visa, visa.getExpiryDate());

        // inserting objects to IdentityHashMap
        cardToExpiryIdenity.put(visa, visa.getExpiryDate());
        System.out.println("adding to IHM");

        System.out.println("before modifying keys");
        String result = cardToExpiry.get(visa) != null ? "Yes" : "No";
        System.out.println("Does VISA card exists in HashMap? " + result);

        result = cardToExpiryIdenity.get(visa) != null ? "Yes" : "No";
        System.out.println("Does VISA card exists in IdenityHashMap? " + result);

        // modifying value object
        visa.setExpiryDate("02/11/2030");
```

```
        System.out.println("after modifying keys");
        result = cardToExpiry.get(visa) != null ? "Yes" : "No";
        System.out.println("Does VISA card exists in HashMap? " + result);

        result = cardToExpiryIdenity.get(visa) != null ? "Yes" : "No";
        System.out.println("Does VISA card exists in IdenityHashMap? " + result);

        System.out.println("cardToExpiry.containsKey");
        System.out.println(cardToExpiry.containsKey(visa));
        System.out.println("cardToExpiryIdenity.containsKey");
        System.out.println(cardToExpiryIdenity.containsKey(visa));
    }

}

class CreditCard {
    private String issuer;
    private String expiryDate;

    public CreditCard(String issuer, String expiryDate) {
        this.issuer = issuer;
        this.expiryDate = expiryDate;
    }

    public String getIssuer() {
        return issuer;
    }

    public String getExpiryDate() {
        return expiryDate;
    }

    public void setExpiryDate(String expiry) {
        this.expiryDate = expiry;
    }

    @Override
    public int hashCode() {
        final int prime = 31;
        int result = 1;
        result = prime * result + ((expiryDate == null) ? 0 : expiryDate.hashCode
        result = prime * result + ((issuer == null) ? 0 : issuer.hashCode());
        System.out.println("hashCode = " + result);
        return result;
    }

    @Override
```

```
        public boolean equals(Object obj) {
            System.out.println("equals !!! ");
            if (this == obj)
                return true;
            if (obj == null)
                return false;
            if (getClass() != obj.getClass())
                return false;
            CreditCard other = (CreditCard) obj;
            if (expiryDate == null) {
                if (other.expiryDate != null)
                    return false;
            } else if (!expiryDate.equals(other.expiryDate))
                return false;
            if (issuer == null) {
                if (other.issuer != null)
                    return false;
            } else if (!issuer.equals(other.issuer))
                return false;
            return true;
        }

    }
```

Результат выполнения кода:

```
adding to HM
hashCode = 1285631513
adding to IHM
before modifying keys
hashCode = 1285631513
Does VISA card exists in HashMap? Yes
Does VISA card exists in IdenityHashMap? Yes
after modifying keys
hashCode = 791156485
Does VISA card exists in HashMap? No
Does VISA card exists in IdenityHashMap? Yes
cardToExpiry.containsKey
hashCode = 791156485
false
cardToExpiryIdenity.containsKey
true
```

к оглавлению

В чем разница между `HashMap` и `WeakHashMap`? Для чего используется `WeakHashMap`?

В Java существует 4 типа ссылок: *сильные (strong reference)*, *мягкие (SoftReference)*, *слабые (WeakReference)* и *фантомные (PhantomReference)*. Особенности каждого типа ссылок связаны с работой Garbage Collector. Если объект можно достичь только с помощью цепочки WeakReference (то есть на него отсутствуют сильные и мягкие ссылки), то данный объект будет помечен на удаление.

`WeakHashMap` - это структура данных, реализующая интерфейс `Map` и основанная на использовании WeakReference для хранения ключей. Таким образом, пара «ключ-значение» будет удалена из `WeakHashMap`, если на объект-ключ более не имеется сильных ссылок.

В качестве примера использования такой структуры данных можно привести следующую ситуацию: допустим имеются объекты, которые необходимо расширить дополнительной информацией, при этом изменение класса этих объектов нежелательно либо невозможно. В этом случае добавляем каждый объект в `WeakHashMap` в качестве ключа, а в качестве значения - нужную информацию. Таким образом, пока на объект имеется сильная ссылка (либо мягкая), можно проверять хэш-таблицу и извлекать информацию. Как только объект будет удален, то WeakReference для этого ключа будет помещен в ReferenceQueue и затем соответствующая запись для этой слабой ссылки будет удалена из `WeakHashMap`.

к оглавлению

В `WeakHashMap` используются WeakReferences. А почему бы не создать `SoftHashMap` на SoftReferences?

`SoftHashMap` представлена в сторонних библиотеках, например, в `Apache Commons`.

к оглавлению

В `WeakHashMap` используются WeakReferences. А почему бы не создать `PhantomHashMap` на PhantomReferences?

PhantomReference при вызове метода `get()` возвращает всегда `null`, поэтому тяжело представить назначение такой структуры данных.

к оглавлению

`LinkedHashMap` - что в нем от `LinkedList`, а что от `HashMap`?

Реализация `LinkedHashMap` отличается от `HashMap` поддержкой двухсвязанного списка, определяющего порядок итерации по элементам структуры данных. По умолчанию элементы списка упорядочены согласно их порядку добавления в `LinkedHashMap` (insertion-order). Однако порядок итерации можно изменить, установив параметр конструктора `accessOrder` в значение `true`. В этом случае доступ осуществляется по порядку последнего обращения к элементу (access-order). Это означает, что при вызове методов `get()` или `put()` элемент, к которому обращаемся, перемещается в конец списка.

При добавлении элемента, который уже присутствует в `LinkedHashMap` (т.е. с одинаковым ключом), порядок итерации по элементам не изменяется.

к оглавлению

В чем проявляется «сортированность» `SortedMap`, кроме того, что `toString()` выводит все элементы по порядку?

Так же оно проявляется при итерации по коллекции.

к оглавлению

Как устроен `HashMap`?

`HashMap` состоит из «корзин» (bucket). С технической точки зрения «корзины» — это элементы массива, которые хранят ссылки на списки элементов. При добавлении новой пары «ключ-значение», вычисляет хэш-код ключа, на основании которого вычисляется номер корзины (номер ячейки массива), в которую попадет новый элемент. Если корзина пустая, то в нее сохраняется ссылка на вновь добавляемый элемент, если же там уже есть элемент, то происходит последовательный переход по ссылкам между элементами в цепочке, в поисках последнего элемента, от которого и ставится ссылка на вновь добавленный элемент. Если в списке был найден элемент с таким же ключом, то он заменяется.

к оглавлению

Согласно Кнуту и Кормену существует две основных реализации хэш-таблицы: на основе открытой адресации и на основе метода цепочек. Как реализована `HashMap`? Почему, по вашему мнению, была выбрана именно эта реализация? В чем плюсы и минусы каждого подхода?

`HashMap` реализован с использованием метода цепочек, т.е. каждой ячейке массива (корзине) соответствует свой связный список и при возникновении коллизии осуществляется добавление нового элемента в этот список.

Для метода цепочек коэффициент заполнения может быть больше 1 и с увеличением числа элементов производительность убывает линейно. Такие таблицы удобно использовать, если заранее неизвестно количество хранимых элементов, либо их может быть достаточно много, что приводит к большим значениям коэффициента заполнения.

Среди методов открытой реализации различают:

- линейное пробирование;
- квадратичное пробирование;
- двойное хэширование.

Недостатки структур с методом открытой адресации:

- Количество элементов в хэш-таблице не может превышать размера массива. По мере увеличения числа элементов и повышения коэффициента заполнения

производительность структуры резко падает, поэтому необходимо проводить перехэширование.
- Сложно организовать удаление элемента.
- Первые два метода открытой адресации приводят к проблеме первичной и вторичной группировок.

Преимущества хэш-таблицы с открытой адресацией:

- отсутствие затрат на создание и хранение объектов списка;
- простота организации сериализации/десериализации объекта.

к оглавлению

Как работает `HashMap` при попытке сохранить в него два элемента по ключам с одинаковым `hashCode()`, но для которых `equals() == false`?

По значению `hashCode()` вычисляется индекс ячейки массива, в список которой этот элемент будет добавлен. Перед добавлением осуществляется проверка на наличие элементов в этой ячейке. Если элементы с таким `hashCode()` уже присутствует, но их `equals()` методы не равны, то элемент будет добавлен в конец списка.

к оглавлению

Какое начальное количество корзин в `HashMap`?

В конструкторе по умолчанию - 16, используя конструкторы с параметрами можно задавать произвольное начальное количество корзин.

к оглавлению

Какова оценка временной сложности операций над элементами из `HashMap`? Гарантирует ли `HashMap` указанную сложность выборки элемента?

В общем случае операции добавления, поиска и удаления элементов занимают константное время.

Данная сложность не гарантируется, т.к. если хэш-функция распределяет элементы по корзинам равномерно, временная сложность станет не хуже *Логарифмического времени* O(log(N)), а в случае, когда хэш-функция постоянно возвращает одно и то же значение, `HashMap` превратится в связный список со сложностью O(n).

Пример кода двоичного поиска:

```
public class Q {
    public static void main(String[] args) {
        Q q = new Q();
        q.binSearch();
    }

    private void binSearch() {
        int[] inpArr = {1, 2, 3, 4, 5, 6, 7, 8, 9};
        Integer result = binSearchF(inpArr, 1, 0, inpArr.length - 1);
        System.out.println("-----------------------");
        result = binSearchF(inpArr, 2, 0, inpArr.length - 1);
        System.out.println("Found at position " + result);
    }

    private Integer binSearchF(int[] inpArr, int searchValue, int low, int high)
        Integer index = null;
        while (low <= high) {
            System.out.println("New iteration, low = " + low + ", high = " + high
            int mid = (low + high) / 2;
            System.out.println("trying mid = " + mid + " inpArr[mid] = " + inpArr
            if (inpArr[mid] < searchValue) {
                low = mid + 1;
                System.out.println("inpArr[mid] (" + inpArr[mid] + ") < searchVal
                        + ", setting low = " + low);
            } else if (inpArr[mid] > searchValue) {
                high = mid - 1;
                System.out.println("inpArr[mid] (" + inpArr[mid] + ") > searchVal
                        + ", setting high = " + high);
            } else if (inpArr[mid] == searchValue) {
                index = mid;
                System.out.println("found at index " + mid);
                break;
            }
        }
        return index;
    }
}
```

к оглавлению

Возможна ли ситуация, когда HashMap выродится в список даже с ключами имеющими разные hashCode()?

Это возможно в случае, если метод, определяющий номер корзины будет возвращать одинаковые значения.

к оглавлению

В каком случае может быть потерян элемент в HashMap?

Допустим, в качестве ключа используется не примитив, а объект с несколькими полями. После добавления элемента в `HashMap` у объекта, который выступает в качестве ключа, изменяют одно поле, которое участвует в вычислении хэш-кода. В результате при попытке найти данный элемент по исходному ключу, будет происходить обращение к правильной корзине, а вот `equals` уже не найдет указанный ключ в списке элементов. Тем не менее, даже если `equals` реализован таким образом, что изменение данного поля объекта не влияет на результат, то после увеличения размера корзин и пересчета хэш-кодов элементов, указанный элемент, с измененным значением поля, с большой долей вероятности попадет в совершенно другую корзину и тогда уже потеряется совсем.

к оглавлению

Почему нельзя использовать byte[] в качестве ключа в HashMap?

Хэш-код массива не зависит от хранимых в нем элементов, а присваивается при создании массива (метод вычисления хэш-кода массива не переопределен и вычисляется по стандартному `Object.hashCode()` на основании адреса массива). Так же у массивов не переопределен `equals` и выполняется сравнение указателей. Это приводит к тому, что обратиться к сохраненному с ключом-массивом элементу не получится при использовании другого массива такого же размера и с такими же элементами, доступ можно осуществить лишь в одном случае — при использовании той же самой ссылки на массив, что использовалась для сохранения элемента.

к оглавлению

Какова роль `equals()` и `hashCode()` в `HashMap`?

`hashCode` позволяет определить корзину для поиска элемента, а `equals` используется для сравнения ключей элементов в списке корзины и искомого ключа.

к оглавлению

Каково максимальное число значений `hashCode()`?

Число значений следует из сигнатуры `int hashCode()` и равно диапазону типа `int` — 2^{32}.

к оглавлению

Какое худшее время работы метода get(key) для ключа, которого нет в `HashMap`?

Какое худшее время работы метода get(key) для ключа, который есть в `HashMap`?

O(N). Худший случай - это поиск ключа в `HashMap`, вырожденного в список по причине совпадения ключей по `hashCode()` и для выяснения хранится ли элемент с определённым ключом может потребоваться перебор всего списка.

Но начиная с Java 8, после определенного числа элементов в списке, связный список преобразовывается в красно-черное дерево и сложность выборки, даже в случае плохой хеш-функции, не хуже *логарифмической O(log(N))*

к оглавлению

Сколько переходов происходит в момент вызова `HashMap.get(key)` по ключу, который есть в таблице?

- ключ равен `null` : 1 - выполняется единственный метод `getForNullKey()`.

- любой ключ отличный от `null` : **4** - вычисление хэш-кода ключа; определение номера корзины; поиск значения; возврат значения.

к оглавлению

Сколько создается новых объектов, когда вы добавляете новый элемент в `HashMap`?

Один новый объект статического вложенного класса `Entry<K,V>`.

к оглавлению

Как и когда происходит увеличение количества корзин в `HashMap`?

Помимо `capacity` у `HashMap` есть еще поле `loadFactor`, на основании которого, вычисляется предельное количество занятых корзин `capacity * loadFactor`. По умолчанию `loadFactor = 0.75`. По достижению предельного значения, число корзин увеличивается в 2 раза и для всех хранимых элементов вычисляется новое «местоположение» с учетом нового числа корзин.

к оглавлению

Объясните смысл параметров в конструкторе `HashMap(int initialCapacity, float loadFactor)`.

- `initialCapacity` - исходный размер `HashMap`, количество корзин в хэш-таблице в момент её создания.
- `loadFactor` - коэффициент заполнения `HashMap`, при превышении которого происходит увеличение количества корзин и автоматическое перехэширование. Равен отношению числа уже хранимых элементов в таблице к её размеру.

к оглавлению

Будет ли работать `HashMap`, если все добавляемые ключи будут иметь одинаковый `hashCode()`?

Да, будет, но в этом случае `HashMap` вырождается в связный список и теряет свои преимущества.

Как перебрать все ключи `Map`?

Использовать метод `keySet()`, который возвращает множество `Set<K>` ключей.

к оглавлению

Как перебрать все значения `Map`?

Использовать метод `values()`, который возвращает коллекцию `Collection<V>` значений.

к оглавлению

Как перебрать все пары «ключ-значение» в `Map`?

Использовать метод `entrySet()`, который возвращает множество `Set<Map.Entry<K, V>` пар «ключ-значение».

к оглавлению

В чем отличия `TreeSet` и `HashSet`?

`TreeSet` обеспечивает упорядоченно хранение элементов в виде красно-черного дерева. Сложность выполнения основных операций не хуже *O(log(N))* (*Логарифмическое время*).

`HashSet` использует для хранения элементов такой же подход, что и `HashMap`, за тем отличием, что в `HashSet` в качестве ключа и значения выступает сам `элемент`, кроме того, `HashSet` не поддерживает упорядоченное хранение элементов и обеспечивает временную сложность выполнения операций аналогично `HashMap`.

к оглавлению

Что будет, если добавлять элементы в `TreeSet` по возрастанию?

В основе `TreeSet` лежит красно-черное дерево, которое умеет само себя балансировать. В итоге, `TreeSet` все равно в каком порядке вы добавляете в него элементы, преимущества этой структуры данных будут сохраняться.

к оглавлению

Чем `LinkedHashSet` отличается от `HashSet`?

`LinkedHashSet` отличается от `HashSet` только тем, что в его основе лежит `LinkedHashMap` вместо `HashMap`. Благодаря этому порядок элементов при обходе коллекции является идентичным порядку добавления элементов (insertion-order). При добавлении элемента, который уже присутствует в `LinkedHashSet` (т.е. с одинаковым ключом), порядок обхода элементов не изменяется.

к оглавлению

Для `Enum` есть специальный класс `java.util.EnumSet`. Зачем? Чем авторов не устраивал `HashSet` или `TreeSet`?

`EnumSet` - это реализация интерфейса `Set` для использования с перечислениями (`Enum`). В структуре данных хранятся объекты только одного типа `Enum`, указываемого при создании. Для хранения значений `EnumSet` использует массив битов (*bit vector*), - это позволяет получить высокую компактность и эффективность. Проход по `EnumSet` осуществляется согласно порядку объявления элементов перечисления.

Все основные операции выполняются за *O(1)* и обычно (но негарантированно) быстрей аналогов из `HashSet`, а пакетные операции (*bulk operations*), такие как `containsAll()` и `retainAll()` выполняются даже гораздо быстрей.

Помимо всего `EnumSet` предоставляет множество статических методов инициализации для упрощенного и удобного создания экземпляров.

к оглавлению

Какие существуют способы перебирать элементы списка?

- Цикл с итератором

```
Iterator<String> iterator = list.iterator();
while (iterator.hasNext()) {
    //iterator.next();
}
```

- Цикл `for`

```
for (int i = 0; i < list.size(); i++) {
    //list.get(i);
}
```

- Цикл `while`

```
int i = 0;
while (i < list.size()) {
    //list.get(i);
    i++;
}
```

- «for-each»

```
for (String element : list) {
    //element;
}
```

к оглавлению

Каким образом можно получить синхронизированные объекты стандартных коллекций?

С помощью статических методов `synchronizedMap()` и `synchronizedList()` класса `Collections`. Данные методы возвращают синхронизированный декоратор переданной коллекции. При этом все равно в случае обхода по коллекции требуется ручная синхронизация.

```
Map m = Collections.synchronizedMap(new HashMap());
List l = Collections.synchronizedList(new ArrayList());
```

Начиная с Java 6 JCF был расширен специальными коллекциями, поддерживающими многопоточный доступ, такими как `CopyOnWriteArrayList` и `ConcurrentHashMap`.

к оглавлению

Как получить коллекцию только для чтения?

При помощи:

- `Collections.unmodifiableList(list);`
- `Collections.unmodifiableSet(set);`
- `Collections.unmodifiableMap(map).`

Эти методы принимают коллекцию в качестве параметра, и возвращают коллекцию только для чтения с теми же элементами внутри.

к оглавлению

Напишите однопоточную программу, которая заставляет коллекцию выбросить `ConcurrentModificationException`.

```
public static void main(String[] args) {
    List<Integer> list = new ArrayList<>();
    list.add(1);
    list.add(2);
    list.add(3);

    for (Integer integer : list) {
        list.remove(1);
    }
}
```

к оглавлению

Приведите пример, когда какая-либо коллекция выбрасывает `UnsupportedOperationException`.

```
public static void main(String[] args) {
    List<Integer> list = Collections.emptyList();
    list.add(0);
}
```

к оглавлению

Реализуйте симметрическую разность двух коллекций используя методы `Collection` (`addAll(...)`, `removeAll(...)`, `retainAll(...)`).

Симметрическая разность двух коллекций - это множество элементов, одновременно не принадлежащих обоим исходным коллекциям.

```
<T> Collection<T> symmetricDifference(Collection<T> a, Collection<T> b) {
    // Объединяем коллекции.
    Collection<T> result = new ArrayList<>(a);
    result.addAll(b);

    // Получаем пересечение коллекций.
    Collection<T> intersection = new ArrayList<>(a);
    intersection.retainAll(b);

    // Удаляем элементы, расположенные в обоих коллекциях.
    result.removeAll(intersection);

    return result;
}
```

к оглавлению

Как, используя LinkedHashMap, сделать кэш с «invalidation policy»?

Необходимо использовать *LRU-алгоритм* (*Least Recently Used algorithm*) и `LinkedHashMap` с access-order. В этом случае при обращении к элементу он будет перемещаться в конец списка, а наименее используемые элементы будут постепенно группироваться в начале списка. Так же в стандартной реализации `LinkedHashMap` есть метод `removeEldestEntries()`, который возвращает `true`, если текущий объект `LinkedHashMap` должен удалить наименее используемый элемент из коллекции при использовании методов `put()` и `putAll()`.

```
public class LRUCache<K, V> extends LinkedHashMap<K, V> {
    private static final int MAX_ENTRIES = 10;

    public LRUCache(int initialCapacity) {
        super(initialCapacity, 0.85f, true);
    }

    @Override
    protected boolean removeEldestEntry(Map.Entry<K, V> eldest) {
        return size() > MAX_ENTRIES;
    }
}
```

Стоит заметить, что `LinkedHashMap` не позволяет полностью реализовать LRU-алгоритм, поскольку при вставке уже имеющегося в коллекции элемента порядок итерации по элементам не меняется.

к оглавлению

Как одной строчкой скопировать элементы любой `collection` в массив?

```
Object[] array = collection.toArray();
```

к оглавлению

Как одним вызовом из `List` получить `List` со всеми элементами, кроме первых и последних 3-х?

```
List<Integer> subList = list.subList(3, list.size() - 3);
```

к оглавлению

Как одной строчкой преобразовать HashSet в ArrayList?

```
ArrayList<Integer> list = new ArrayList<>(new HashSet<>());
```

к оглавлению

Как одной строчкой преобразовать ArrayList в HashSet?

```
HashSet<Integer> set = new HashSet<>(new ArrayList<>());
```

к оглавлению

Сделайте HashSet из ключей HashMap.

```
HashSet<Object> set = new HashSet<>(map.keySet());
```

к оглавлению

Сделайте HashMap из HashSet<Map.Entry<K, V>>.

```
HashMap<K, V> map = new HashMap<>(set.size());
for (Map.Entry<K, V> entry : set) {
    map.put(entry.getKey(), entry.getValue());
}
```

к оглавлению

Источник

- parshinpn.pro

- Хабрахабр
- Quizful
- JavaRush
- Хабрахабр:Справочник по Java Collections Framework

Вопросы для собеседования