## **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](https://github.com/enhorse/java-interview/blob/master/jcf.md" \l "java-collections-framework)

## **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. Элементы такой коллекции пронумерованы, начиная от нуля, к ним можно обратиться по индексу. Реализации:
  + ArrayList - инкапсулирует в себе обычный массив, длина которого автоматически увеличивается при добавлении новых элементов.
  + 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).
  + Почитать 2 файл про коллеции

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).

## **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

## **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».

## **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.

## **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](https://github.com/enhorse/java-interview/blob/master/jcf.md" \l "java-collections-framework)

## **Как между собой связаны Iterable и Iterator?**

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

public interface Iterable<T>{

Iterator<T> iterator();}

public interface Iterator<E>{

boolean hasNext();

E next();

void remove();}

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

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

[к оглавлению](https://github.com/enhorse/java-interview/blob/master/jcf.md" \l "java-collections-framework)

## **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.

## **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](https://github.com/enhorse/java-interview/blob/master/jcf.md" \l "java-collections-framework)

## **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](https://github.com/enhorse/java-interview/blob/master/jcf.md" \l "java-collections-framework)

## **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](https://github.com/enhorse/java-interview/blob/master/jcf.md" \l "java-collections-framework)

## **Какая коллекция реализует дисциплину обслуживания FIFO?**

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

[к оглавлению](https://github.com/enhorse/java-interview/blob/master/jcf.md" \l "java-collections-framework)

## **Какая коллекция реализует дисциплину обслуживания FILO?**

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

[к оглавлению](https://github.com/enhorse/java-interview/blob/master/jcf.md" \l "java-collections-framework)

## **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](https://github.com/enhorse/java-interview/blob/master/jcf.md" \l "java-collections-framework)

## **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);
* на добавление и удаление в начало или конец списка потребуется константное O(1);
* вставка или удаление в/из произвольного место константное O(1);
* 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](https://github.com/enhorse/java-interview/blob/master/jcf.md" \l "java-collections-framework)

## **Что работает быстрее ArrayList или LinkedList?**

Смотря какие действия будут выполняться над структурой.

## **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](https://github.com/enhorse/java-interview/blob/master/jcf.md" \l "java-collections-framework)

## **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](https://github.com/enhorse/java-interview/blob/master/jcf.md" \l "java-collections-framework)

## **Какое худшее время работы метода add() для LinkedList?**

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

[к оглавлению](https://github.com/enhorse/java-interview/blob/master/jcf.md" \l "java-collections-framework)

## **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](https://github.com/enhorse/java-interview/blob/master/jcf.md" \l "java-collections-framework)

## **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](https://github.com/enhorse/java-interview/blob/master/jcf.md" \l "%D0%A7%D0%B5%D0%BC-%D0%BE%D1%82%D0%BB%D0%B8%D1%87%D0%B0%D0%B5%D1%82%D1%81%D1%8F-arraylist-%D0%BE%D1%82-linkedlist-%D0%92-%D0%BA%D0%B0%D0%BA%D0%B8%D1%85-%D1%81%D0%BB%D1%83%D1%87%D0%B0%D1%8F%D1%85-%D0%BB%D1%83%D1%87%D1%88%D0%B5-%D0%B8%D1%81%D0%BF%D0%BE%D0%BB%D1%8C%D0%B7%D0%BE%D0%B2%D0%B0%D1%82%D1%8C-%D0%BF%D0%B5%D1%80%D0%B2%D1%8B%D0%B9-%D0%B0-%D0%B2-%D0%BA%D0%B0%D0%BA%D0%B8%D1%85-%D0%B2%D1%82%D0%BE%D1%80%D0%BE%D0%B9)

[к оглавлению](https://github.com/enhorse/java-interview/blob/master/jcf.md" \l "java-collections-framework)

## **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](https://github.com/enhorse/java-interview/blob/master/jcf.md" \l "java-collections-framework)

## **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)

## **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](https://github.com/enhorse/java-interview/blob/master/jcf.md" \l "java-collections-framework)

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

Создается один новый экземпляр вложенного класса Node.

## **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](https://github.com/enhorse/java-interview/blob/master/jcf.md" \l "java-collections-framework)

## **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](https://github.com/enhorse/java-interview/blob/master/jcf.md" \l "java-collections-framework)

## **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](https://github.com/enhorse/java-interview/blob/master/jcf.md" \l "java-collections-framework)

## **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](https://github.com/enhorse/java-interview/blob/master/jcf.md" \l "java-collections-framework)

## **LinkedList — это односвязный, двусвязный или четырехсвязный список?**

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

[к оглавлению](https://github.com/enhorse/java-interview/blob/master/jcf.md" \l "java-collections-framework)

## **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](https://github.com/enhorse/java-interview/blob/master/jcf.md" \l "java-collections-framework)

## **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](https://github.com/enhorse/java-interview/blob/master/jcf.md" \l "java-collections-framework)

## **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](https://github.com/enhorse/java-interview/blob/master/jcf.md" \l "java-collections-framework)

## **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](https://github.com/enhorse/java-interview/blob/master/jcf.md" \l "java-collections-framework)

## **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.

## **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](https://github.com/enhorse/java-interview/blob/master/jcf.md" \l "java-collections-framework)

## **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](https://github.com/enhorse/java-interview/blob/master/jcf.md" \l "java-collections-framework)

## **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](https://github.com/enhorse/java-interview/blob/master/jcf.md" \l "java-collections-framework)

## **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](https://github.com/enhorse/java-interview/blob/master/jcf.md" \l "java-collections-framework)

## **Какое начальное количество корзин в HashMap?**

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

[к оглавлению](https://github.com/enhorse/java-interview/blob/master/jcf.md" \l "java-collections-framework)

## **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](https://ru.wikipedia.org/wiki/%D0%92%D1%80%D0%B5%D0%BC%D0%B5%D0%BD%D0%BD%D0%B0%D1%8F_%D1%81%D0%BB%D0%BE%D0%B6%D0%BD%D0%BE%D1%81%D1%82%D1%8C_%D0%B0%D0%BB%D0%B3%D0%BE%D1%80%D0%B8%D1%82%D0%BC%D0%B0" \l "%D0%9B%D0%BE%D0%B3%D0%B0%D1%80%D0%B8%D1%84%D0%BC%D0%B8%D1%87%D0%B5%D1%81%D0%BA%D0%BE%D0%B5_%D0%B2%D1%80%D0%B5%D0%BC%D1%8F)O(log(N)), а в случае, когда хэш-функция постоянно возвращает одно и то же значение, HashMap превратится в связный список со сложностью О(n).

## **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](https://github.com/enhorse/java-interview/blob/master/jcf.md" \l "java-collections-framework)

## **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](https://github.com/enhorse/java-interview/blob/master/jcf.md" \l "java-collections-framework)

## **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](https://github.com/enhorse/java-interview/blob/master/jcf.md" \l "java-collections-framework)

## **Каково максимальное число значений hashCode()?**

Число значений следует из сигнатуры int hashCode() и равно диапазону типа int — ****232****.

[к оглавлению](https://github.com/enhorse/java-interview/blob/master/jcf.md" \l "java-collections-framework)

## **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.

[к оглавлению](https://github.com/enhorse/java-interview/blob/master/jcf.md" \l "java-collections-framework)

## **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](https://github.com/enhorse/java-interview/blob/master/jcf.md" \l "java-collections-framework)

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

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

[к оглавлению](https://github.com/enhorse/java-interview/blob/master/jcf.md" \l "java-collections-framework)

## **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](https://github.com/enhorse/java-interview/blob/master/jcf.md" \l "java-collections-framework)

## **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](https://github.com/enhorse/java-interview/blob/master/jcf.md" \l "java-collections-framework)

## **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](https://github.com/enhorse/java-interview/blob/master/jcf.md" \l "java-collections-framework)

## **Как перебрать все значения Map?**

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

[к оглавлению](https://github.com/enhorse/java-interview/blob/master/jcf.md" \l "java-collections-framework)

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

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

[к оглавлению](https://github.com/enhorse/java-interview/blob/master/jcf.md" \l "java-collections-framework)

## **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](https://github.com/enhorse/java-interview/blob/master/jcf.md" \l "java-collections-framework)

## **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](https://github.com/enhorse/java-interview/blob/master/jcf.md" \l "java-collections-framework)

## **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](https://github.com/enhorse/java-interview/blob/master/jcf.md" \l "java-collections-framework)

## **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](https://github.com/enhorse/java-interview/blob/master/jcf.md" \l "java-collections-framework)

## **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](https://github.com/enhorse/java-interview/blob/master/jcf.md" \l "java-collections-framework)

## **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](https://github.com/enhorse/java-interview/blob/master/jcf.md" \l "java-collections-framework)

## **Приведите пример, когда какая-либо коллекция выбрасывает UnsupportedOperationException.**

public static void main(String[] args) {

List<Integer> list = Collections.emptyList();

list.add(0);

}

[к оглавлению](https://github.com/enhorse/java-interview/blob/master/jcf.md" \l "java-collections-framework)

## **Как одной строчкой скопировать элементы любой collection в массив?**

Object[] array = collection.toArray();

[к оглавлению](https://github.com/enhorse/java-interview/blob/master/jcf.md" \l "java-collections-framework)

## **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](https://github.com/enhorse/java-interview/blob/master/jcf.md" \l "java-collections-framework)

## **Как одной строчкой преобразовать HashSet в ArrayList?**

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

[к оглавлению](https://github.com/enhorse/java-interview/blob/master/jcf.md" \l "java-collections-framework)

## **Как одной строчкой преобразовать ArrayList в HashSet?**

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

[к оглавлению](https://github.com/enhorse/java-interview/blob/master/jcf.md" \l "java-collections-framework)

## **Сделайте HashSet из ключей HashMap.**

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