Интерфейсы в Java

Сложно декларативно, в двух словах, описать что такое интерфейс. Ниже приведены описания с разных точек зрения, сумма которых формирует понятие интерфейса.

# Несколько определений из разных источников

* Назначение интерфейса - **описание** функциональности, которая **должна** быть реализована имплементирующими классами в дальнейшем [1, c. 150].
* Интерфейс определяет имена, список аргументов, возвращаемые значения методов, но не их реализацию. Интерфейс предоставляет только **форму**, но не **реализацию**. Интерфейс говорит: «все классы что реализуют этот конкретный интерфейс будут выглядеть так». То есть любой код, который использует данный интерфейс знает какие методы могут быть вызваны. Можно сказать, что интерфейс устанавливает «протокол» общения между классами [2, c. 222].
* Интерфейс - это регламент взаимодействия. Класс, который реализует интерфейс обязан реализовывать все его методы. В интерфейсе вы описываете лишь сигнатуры методов, то есть вы указываете что класс наследник **должен** уметь делать, но **как** он будет это делать, тот решает сам. Таким образом, вы уверенны что, если класс реализует тот или иной интерфейс, все объекты данного класса имеют определенный набор методов. Интерфейс - это еще одна абстракция (более высокого уровня чем наследование), позволяющая отделить **описание** от **реализации**.

# Примеры интерфейсов в реальном мире

* Интерфейс «способность летать», то есть способность к движению в воздухе. Эта способность присуща бабочкам, птицам, самолетам. На уровне объектов между ними родства нет - бабочка не является самолетом, а самолет не является птицей. То есть реализация у всех сущностей разная, а интерфейс общий.
* Интерфейс «управление наземным транспортным средством» предоставляет органы управления: руль, педали, средства подачи сигналов. При этом транспортное средство может быть реализовано в виде троллейбуса, легкового автомобиля, электропоезда.

# Отличия интерфейсов и абстрактных классов

Интерфейсы тесно связаны с абстрактными классами. Не все языки выделяют интерфейсы как отдельный элемент. Например, в С++ нет интерфейсов, а есть только абстрактные классы. То есть чисто абстрактный класс в С++ (где есть только абстрактные методы) можно рассматривать как интерфейс. Выяснение отличий подводит нас к тому, что такое интерфейс конкретно в языке Java [3], [4].

|  |  |
| --- | --- |
| **Классы** | **Интерфейсы** |
| Абстрактные классы предназначены для **наследования** (**inheritance**), а значит реализуют **«is a»** тип отношений. Например: класс «собака» унаследован от класса «животное». Значит мы можем сказать: «собака» есть (is a) животное. Но обратное неверно. При этом «собака» имеет доступ к свойствам и методам родительской сущности «животное». | Интерфейсы реализуют тип отношений **«has a»**,который называется **композиция** (**composition**). Например: есть класс «пчела» и класс «собака». Эти сущности никак не связаны друг с другом, но обе реализуют интерфейс «возможность укусить». То есть, мы можем сказать: «собака», «пчела» имеют (has a) «возможность укусить». |
| Множественное наследование в Java запрещено для классов. | В Java один класс может реализовывать множество интерфейсов. |
| Абстрактный класс может реализовывать методы, все или только некоторые. | Интерфейс не может реализовывать методы |
| К неабстрактным членам класса применимы любые модификаторы. Однако абстрактные методы (с модификатором abstract) могут иметь либо public, либо protected модификатор. Иными словами, абстрактные методы не могут быть приватными. | Все методы по умолчанию являются публичными (public) и абстрактными (abstract), а поля - public static final. Даже если они таковыми не объявлены. |
| При наследовании абстрактного класса **не все** методы могут быть переопределены. | При реализации интерфейса **все** методы должны быть реализованы. |
| В абстрактном классе можно описать конструктор (или несколько конструкторов). | Интерфейс не может содержать никаких конструкторов |

# Зачем нужны интерфейсы

Причины появления интерфейсов и устранения множественного наследования в Java заключаются в том, что Java стремиться более строго следовать паттернам в объектно-ориентированном проектировании (General Responsibility Assignment Software Patterns - общие шаблоны распределения обязанностей) [5].

Больше всего с классами и интерфейсами связаны такие принципы:

* **High сohesion** (высокое **сцепление**, часто переводится как связность). Понятие, определяющее насколько тесно связаны между собой элементы одного модуля. Можно считать "модулем" класс, а "элементами" - его методы. Тогда **сцепление** определяет, по сути, насколько узкий (специфичный) функционал реализует данный класс.   
    
  Высокое **сцепление** означает что класс выполняет однородные обязанности, то есть лишь свою определённую часть функционала системы. И наоборот, за каждый элемент функционала системы отвечает только один класс.   
    
  Например, если в системе понадобилось изменить политику безопасности, то изменения должны быть сделаны лишь в одном месте программы, отвечающем за проверку пароля на безопасность.

Высокое **сцепление** - это положительная характеристика и его нужно стремиться повысить.

* **Low сoupling** (низкая **связанность**). Это принцип, который позволяет распределить обязанности между объектами таким образом, чтобы они как можно меньше зависели друг от друга. При соблюдении это принципа мы получаем:
  + Малое число зависимостей между классами (подсистемами).
  + Слабая зависимость одного класса (подсистемы) от изменений в другом классе (подсистеме).
  + Высокая степень повторного использования подсистем.

Высокое сцепление и низкая связанность на практике означают простоту обслуживания. То есть, лёгкость модификации существующей программы для изменения существующих и добавления новых возможностей.

Использование интерфейсов понижает связанность в то время как наследование повышает ее. Поддержка интерфейсов на уровне языка обязывает программиста придерживаться паттернов ООП.   
  
Интерфейсы плюс одинарное наследование помогают добиться оптимального баланса между сцеплением и связанностью.

В других языках (например, в C++) есть возможность использовать правильные подходы, паттерны ООП, но это полностью ответственность программиста. Управление сцеплением и связанностью на уровне языка - это неоспоримое преимущество Java.

**Рекомендации по применению** **абстрактных классов и интерфейсов**

* Абстрактный класс уместно использовать, если вы собираетесь использовать наследование, которое будет обеспечивать общую структуру.
* Абстрактный класс также уместно использовать, если вы хотите объявить приватные экземпляры. В интерфейсах, все методы должны быть публичными.
* Если вы считаете, что в будущем вам понадобится прибегать к добавлению новых методов, тогда абстрактный класс – лучший вариант. В случае интерфейса вам необходимо будет объявлять новый метод в каждом классе, который реализует ваш интерфейс.
* Интерфейсы – хороший выбор, если вам известно, что API не будет меняться некоторое время.
* Также, для реализации «множественного наследования» кроме интерфейсов вы не найдете альтернативных решений.
* Если абстрактный класс не содержит никакой частичной реализации, то вместо него лучше использовать интерфейс [4].

**Синтаксис**

Ниже приведен пример описания интерфейса и его реализации.

public interface Love {  
 bolean initLove(int money);

bolean initLove(Feelings feelings);

Pleasure makeLove();  
}

class Man implements Love {

bolean initLove(int money) {

…

}

bolean initLove(Feelings feelings) {

…

}

Pleasure makeLove(){

…

}

}

Cтруктуры данных в Java

**Контейнеры** - это хранилища, поддерживающие различные способы хранения и упорядочения объектов с целью обеспечения возможностей эффективного доступа к ним. Они представляют собой реализацию абстрактных структур данных, поддерживающих три основные операции [1, c. 253]:

* Добавление нового элемента в коллекцию.
* Удаление элемента из коллекции.
* Изменение элемента в коллекции.

Применение контейнеров обусловливается возросшими объемами обрабатываемой информации. В этих условиях массивы не могут удовлетворить всем требованиям.

Контейнеры в Java можно разделить на:

* Коллекции (Collections);
* Отображение вида «ключ-значение» (Map);

Такое разделение обусловлено тем, что все контейнеры кроме Map принадлежат к иерархии классов Collections. Контейнеры Map реализованы отдельно, но они используют коллекции для итерирования.

# Итераторы

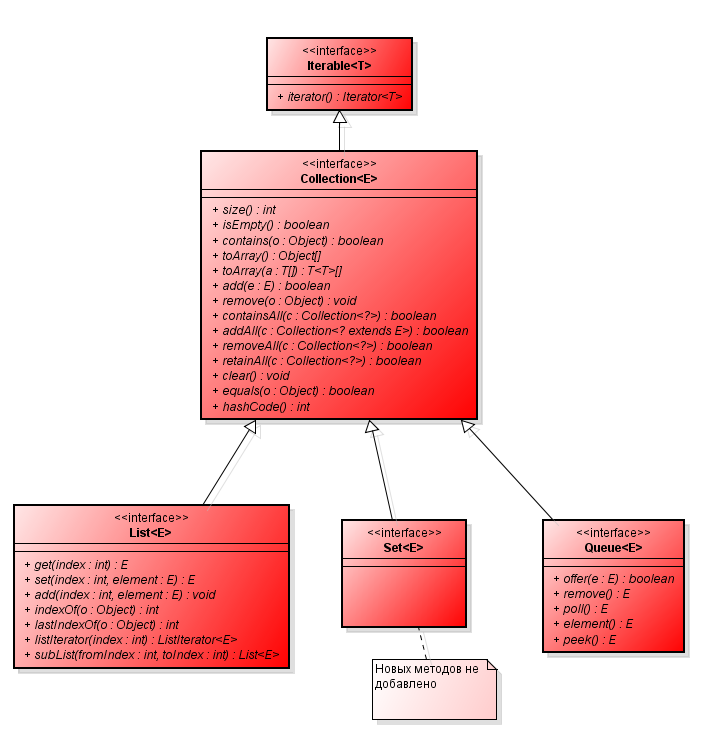
**О**бъекты, абстрагирующие за единым интерфейсом доступ к элементам коллекции. Итератор позволяет получить доступ к элементам любой коллекции без вникания в суть ее реализации.

Иерархия итераторов представлена ниже  


Интерфейс **Iterator** позволяет организовать цикл для перебора коллекции получая либо удаляя элементы.

Интерфейс **ListIterator** добавляет к интерфейсу **Iterator** двунаправленный проход по списку и возможность модификации элементов.

# Основные интерфейсы иерархии Collection



Как видно из диаграммы интерфейс **Collection** расширяет интерфейс **Iterable**, у которого есть только один метод iterator(). Это значит, что любая коллекция, которая является наследником **Iterable** должна возвращать итератор.

**List**

Представляет собой неупорядоченную коллекцию, в которой допустимы дублирующие значения. Иногда их называют последовательностями (sequence). Элементы такой коллекции пронумерованы, начиная от нуля, к ним можно обратиться по индексу [6].

**Set**

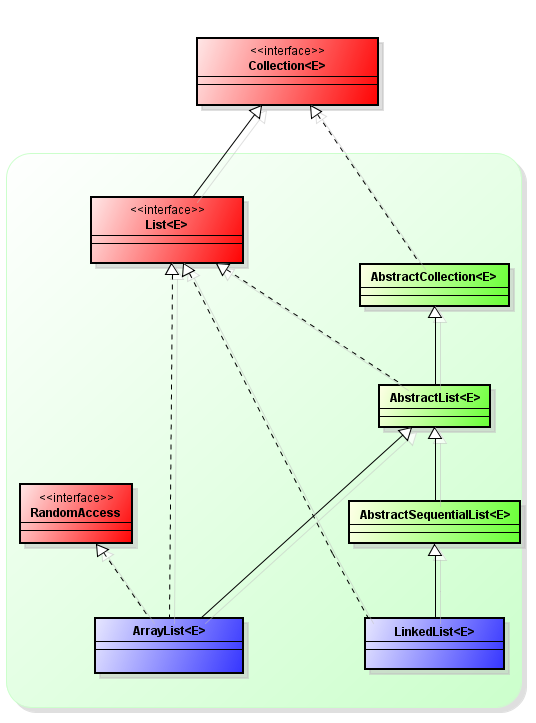
Описывает неупорядоченную коллекцию, не содержащую повторяющихся элементов. Это соответствует математическому понятию множества (set) [6].

**Queue**

Очередь. Это коллекция, предназначенная для хранения элементов в порядке, нужном для их обработки. В дополнение к базовым операциям интерфейса Collection, очередь предоставляет дополнительные операции вставки, получения и контроля [6].

# Реализации интерфейса List

Красным выделены интерфейсы, зеленым - абстрактные классы, а синим готовые реализации.

****

* **ArrayList** - Инкапсулирует в себе обычный массив, длина которого автоматически увеличивается при добавлении новых элементов. Так как ArrayList использует массив, то время доступа к элементу по индексу минимально. При удалении произвольного элемента из списка все элементы, находящиеся «правее» смещаются на одну ячейку влево, при этом реальный размер массива (его емкость, capacity) не изменяется. Если при добавлении элемента, оказывается, что массив полностью заполнен, будет создан новый массив размером (n \* 3) / 2 + 1, в него будут помещены все элементы из старого массива + новый, добавляемый элемент [6].
* **LinkedList** - Двусвязный список. Это структура данных, состоящая из узлов, каждый из которых содержит как собственно данные, так и две ссылки («связки») на следующий и предыдущий узел списка. Доступ к произвольному элементу осуществляется за линейное время, но доступ к первому и последнему элементу списка всегда осуществляется за константное время [6]. LinkedList обеспечивает оптимальный последовательный доступ с недорогой вставкой и удалением из середины списка. LinkedList относительно медленный для произвольного доступа, но предоставляет больше возможностей чем ArrayList [2].

**Сходства ArrayList и LinkedList**

* ArrayList и LinkedList являются основными реализациями интерфейса List
* Элементы в этих классах упорядочены и располагаются в порядке вставки (insertion order)
* Доступ по индексу. Мы можем вставлять и извлекать элементы по их индексу
* Допускается дублирование элементов
* Допускается использование null
* Обе имплементации не синхронизированы (not synchronized)
* Iterator и ListIterator в обоих классах fail-fast. Это означает, что изменение структуры коллекции во время итерирования приведет к выбросу ConcurrentModificationException

**Различия ArrayList и LinkedList**

|  |  |
| --- | --- |
| **ArrayList** | **LinkedList** |
| Для реализации ArrayList используется динамический массив. | Для реализации LinkedList используется двусвязный список (doubly linked list) |
|  | LinkedList дополнительно реализует интерфейс **Deque**, что позволяет вставлять и удалять элементы как в начало, так и в конец списка. Deque сокращение от double-ended queue (двусвязная очередь) |
| Так как в основе ArrayList - массив, то доступ к произвольному элементу по индексу осуществляется **за константное время O(1)**. Является важным преимуществом ArrayList. | В основе LinkedList мы имеем двусвязный список, поэтому для доступа к произвольному элементу по индексу нужно пройтись по списку пока мы не достигнем нужного индекса. Если индекс ближе к концу списка, то проход начинается с конца, однако это не спасает, в LinkedList - доступ к произвольному элементу по индексу осуществляется **за линейное время O(n)**. |
| При добавлении элементов в начало ArrayList все элементы сдвигаются вправо по средствам вызова нативного метода System.arraycopy(), таким образом, **сложность O(n)**. | В случае c LinkedList добавление элементов в начало происходит за константное время O(1). |
| Добавление элементов в конец списка происходит за константное время **O(1)**, однако может произойти так, что при добавлении элемента в конец ArrayList может потребоваться увеличение размера массива и тогда сложность станет линейной **O(n)**. | Добавление элементов в конец списка происходит за константное время **O(1)** |
| При вставке/удалении элемента в средине списка происходит перезапись всех элементов, размещенных «правее» в списке.  Это выполняется за **линейное время** с помощью метода System.arraycopy().  Кроме того, при удалении элементов размер массива не уменьшается, до явного вызова метода trimToSize(). | LinkedList производит вставку удаление в средину списка за **постоянное время** (плюс поиск позиции, который происходит за линейное время). |

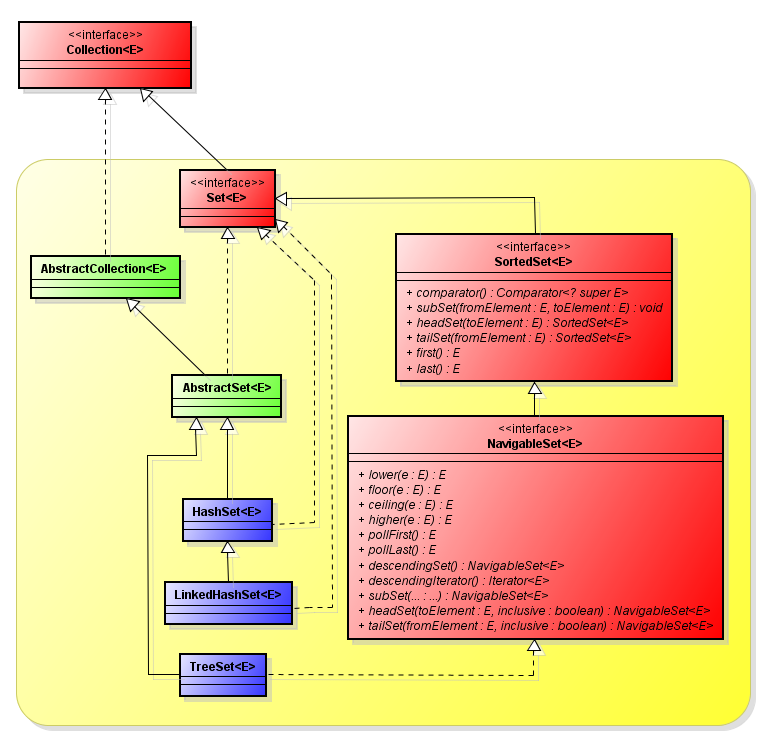
**Рекомендации по использованию ArrayList и LinkedList**

В большинстве случаев используется ArrayList, благодаря константному времени доступа к элементам и своей быстроте.

Если вы собираетесь часто вставлять элементы в начало списка или же, собираетесь часто удалять элементы из середины списка при помощи итератора, то можно рассмотреть возможность использования LinkedList.

Однако все-таки стоит предварительно сравнить, как меняется производительность приложения при использовании ArrayList и LinkedList.

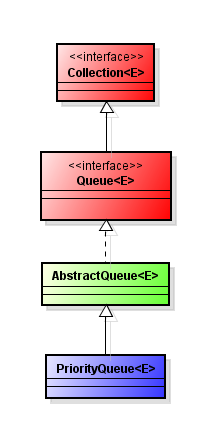
# Реализации интерфейса Set



* **HashSet** - коллекция, не позволяющая хранить одинаковые объекты. HashSet инкапсулирует в себе объект HashMap (то-есть использует для хранения хэш-таблицу). Выгода от хеширования состоит в том, что оно обеспечивает константное время выполнения методов add(), contains(), remove() и size() , даже для больших наборов.  
    
  Если использовать HashSet для хранения объектов пользовательский классов, то необходимо переопределить методы hashCode() и equals(), иначе два логически-одинаковых объекта будут считаться разными, так как при добавлении элемента в коллекцию будет вызываться метод hashCode() класса Object который скорее-всего вернет разный хэш-код для пользовательских объектов.  
    
  Важно отметить, что класс HashSet **не гарантирует упорядоченности** элементов, поскольку процесс хеширования сам по себе обычно не порождает сортированных наборов.
* **LinkedHashSet** - поддерживает связный список элементов набора в том порядке, в котором они вставлялись. Это позволяет организовать упорядоченную итерацию вставки в набор. То есть, когда идет перебор объекта класса LinkedHashSet с применением итератора, элементы извлекаются в том порядке, в каком они были добавлены.
* **TreeSet** - коллекция, которая хранит свои элементы в виде упорядоченного по значениям дерева. TreeSet реализует интерфейс **SortedSet**, поэтому хорошо подходит там, где нужны сортированые наборы.   
    
  TreeSet инкапсулирует в себе TreeMap, который в свою очередь использует сбалансированное бинарное красно-черное дерево для хранения элементов. TreeSet хорош тем, что для операций add(), remove() и contains() потребуется гарантированное время log(n).

# Реализации интерфейса Queue

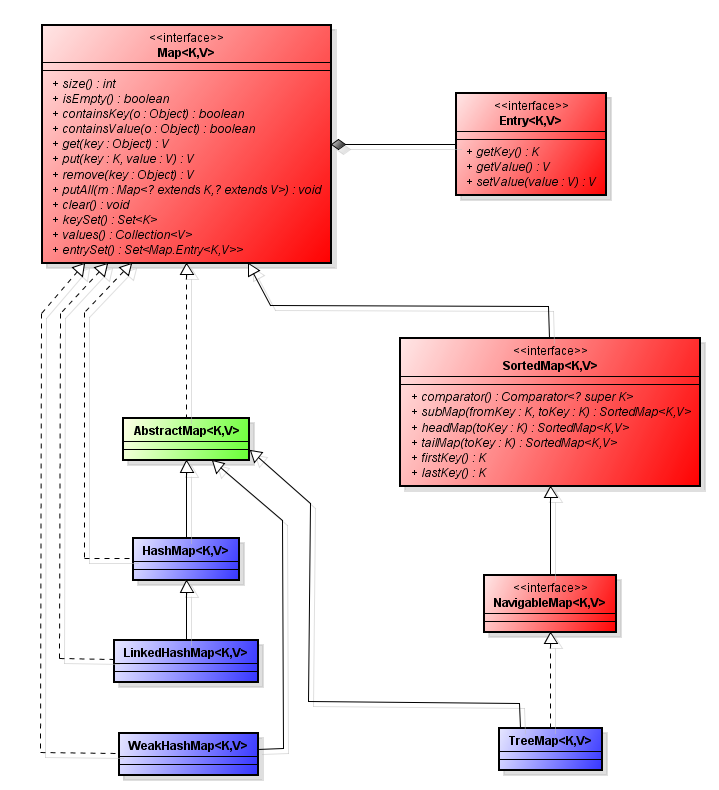
Упрощенная иерархия.



* **PriorityQueue -** единственная прямая реализация интерфейса Queue. Эта очередь упорядочивает элементы либо по их натуральному порядку (используя интерфейс Comparable), либо с помощью интерфейса Comparator, полученному в конструкторе.

# Реализации интерфейса Map

Интерфейс Map соотносит уникальные ключи со значениями. Ключ - это объект, который вы используете для последующего извлечения данных. Задавая ключ и значение, вы можете помещать значения в объект карты. После того как это значение сохранено, вы можете получить его по ключу.



* **HashMap** - основан на хэш-таблицах, реализует интерфейс Map (что подразумевает хранение данных в виде пар ключ/значение). Ключи и значения могут быть любых типов, в том числе и null. Данная реализация не дает гарантий относительно порядка элементов с течением времени.
* **LinkedHashMap -** расширяет класс HashMap. Он создает связный список элементов в карте, расположенных в том порядке, в котором они вставлялись. Это позволяет организовать перебор карты в порядке вставки. Также можете создать объект класса LinkedHashMap, возвращающий свои элементы в том порядке, в котором к ним в последний раз осуществлялся доступ.
* **TreeMap** - расширяет класс AbstractMap и реализует интерфейс NavigatebleMap. Он создает коллекцию, которая для хранения элементов применяет дерево. Объекты сохраняются в отсортированном порядке по возрастанию. Время доступа и извлечения элементов достаточно мало, что делает класс TreeMap блестящим выбором для хранения больших объемов отсортированной информации, которая должна быть быстро найдена.
* **WeakHashMap** - коллекция, использующая слабые ссылки для ключей (а не значений). Слабая ссылка (англ. weak reference) - специфический вид ссылок на динамически создаваемые объекты в системах со сборкой мусора. Отличается от обычных ссылок тем, что не учитывается сборщиком мусора при выявлении объектов, подлежащих удалению.

1. JAVA Methods Programming v2, 2015, Блинов, Романчик, с. 150, с. 253
2. Thinking in Java 4th Edition, Bruce Eckel, c. 222, с. 297
3. <http://www.quizful.net/interview/java/interface-vs-abstract-class>
4. <http://www.quizful.net/post/razlichie_v_primenenii_interfeysov_i_abstraktnih_klassov>
5. <https://ru.wikipedia.org/wiki/GRASP>
6. <http://www.quizful.net/post/Java-Collections>