Java school

Generics, Collections, Внутренние классы

Класс Object

- Является суперклассом для всех классов (включая массивы).
- Переменная этого типа может ссылаться на любой объект (но не на переменную примитивного типа).
- Его методы наследуются всеми классами.
- Реализует базовые операции с объектами.

Методы класса Object

- Получение строкового представления объекта **String toString()**
- Получение ссылки на описание класса объекта **final Class getClass()**
- Клонирование объекта (получение копии) **protected Object clone()**
- Проверка равенства объектов **boolean equals(Object obj)**
- Получение хэш-кода объекта **int hashCode()**
- Метод завершения работы с объектом **protected void finalize()**

Обертки примитивных типов

- Значения примитивных типов не могут быть непосредственно использованы в контексте, где требуется ссылка.
- Ссылочное представление значений примитивных типов является основной задачей классов-оберток.
- Экземпляр такого класса хранит внутри значение примитивного типа и предоставляет доступ к этому значению.

Generics - это типы с параметром

```
public class Main {
    public static void main(String[] args) {
        List<String> myList1 = new ArrayList<>();
        myList1.add("Test String 1");
        myList1.add("Test String 2");
    }
}
```

Применение обобщений (Generics)

- Создание экземпляров обобщенных классов

new ArrayList<Book>();

- Объявление и инициализация переменных, имеющих обобщенный тип

List<Book> bookList = new ArrayList<Book>();

- Объявление и вызов методов, которые принимают значения обобщенных типов

void get(List<Book> books)

shelf.get(bookList);

Зачем?

```
public class MyListClass {

    private Object[] data;
    private int count;

    public MyListClass() {
        this.data = new Object[10];
        this.count = 0;
    }

    public void add(Object o) {
        this.data[count] = o;
        count++;
    }

    public Object[] getData() {
        return data;
    }
}
```

Какая-то ерунда

```
public class Main {

    public static void main(String[] args) {

        MyListClass list = new MyListClass();
        list.add(100);
        list.add(200);
        list.add("Lolkek");
        list.add("Shalala");

        Integer sum1 = (Integer) list.getData()[0] + (Integer) list.getData()[1];
        System.out.println(sum1);

        Integer sum2 = (Integer) list.getData()[2] + (Integer) list.getData()[3];
        System.out.println(sum2);
    }
}
```

Что получаем?

- *300 Exception in thread "main" java.lang.ClassCastException: java.lang.String cannot be cast to java.lang.Integer at Main.main*
- Неправильный код попал в важное место нашей программы и успешно скомпилировался
- Увидим ошибку не на этапе написания кода, а только на этапе тестирования

Можно и свой класс не создавать

```
public class Main {

    public static void main(String[] args) {

        List list = new ArrayList();

        list.add(100);
        list.add(200);
        list.add("Lolkek");
        list.add("Shalala");

        System.out.println((Integer) list.get(0) + (Integer) list.get(1));
        System.out.println((Integer) list.get(2) + (Integer) list.get(3));
    }
}
```

IDE предупреждает

“Unchecked call to add(E) as a member of raw type of java.util.List”

Raw type — это класс-дженерик, из которого удалили его тип.

Нужен чтобы сохранялась обратная совместимость и старый код работал на новых версии Java.

Generic methods

```
public class TestClass {

    public static <T> void fill(List<T> list, T val) {
        for (int i = 0; i < list.size(); i++)
            list.set(i, val);
    }

    public static void main(String[] args) {

        List<String> strings = new ArrayList<>();
        strings.add("Старая строка 1");
        strings.add("Старая строка 2");
        strings.add("Старая строка 3");

        fill(strings, "Новая строка");

        System.out.println(strings);

        List<Integer> numbers = new ArrayList<>();
        numbers.add(1);
        numbers.add(2);
        numbers.add(3);

        fill(numbers, 888);
        System.out.println(numbers);
    }
}
```

https://docs.oracle.com/javase/tutorial/java/generics/methods.html

Собственный Generic class

```
public class Box<T> {

    private T t;

    public void set(T t) {
        this.t = t;
    }

    public T get() {
        return t;
    }

    public static void main(String[] args) {

        Box<String> stringBox = new Box<>();

        stringBox.set("Старая строка");
        System.out.println(stringBox.get());
        stringBox.set("Новая строка");

        System.out.println(stringBox.get());

        stringBox.set(12345);//ошибка компиляции!
    }
}
```

Стирание типов

Внутри класса-generic'а не хранится никакой информации о его типе-параметре.

```
public static class Util {
    public static <T> T getValue(Object obj) {
        return (T) obj;
    }
}

public static void main(String []args) {
    List list = Arrays.asList(2, 3);
    for (Object element : list) {
        System.out.println(Util.<Integer>getValue(element) + 1); //работает успешно
        System.out.println(Util.getValue(element) + 1); //ошибка компиляции
    }
}
```

Еще пример

```
public static void main(String []args) {
    //SomeType<String> st = new SomeType<>(); // отработает успешно
    SomeType st = new SomeType<>();
    List<String> list = Arrays.asList("test");
    st.test(list);
}

public static class SomeType<T> {
    public <E> void test(Collection<E> collection) {
        for (E element : collection) {
            System.out.println(element);
        }
    }
    public void test(List<Integer> collection) {
        for (Integer element : collection) {
            System.out.println(element);
        }
    }
}
```

Использование типовых параметров

Код **ArrayList<String> books = new ArrayList<String>();**

означает, что ArrayList

public class ArrayList<E> extends AbstractList<E> ... {

public boolean add(E o);

}

будет воспринят компилятором как:

public class ArrayList<String> extends AbstractList<String>

public boolean add(String o);

}

Использование обобщенных методов

- Использование типового параметра, описанного при объявлении класса:

public class ArrayList<E> extends AbstractList<E> ... {

public boolean add(E o)

- Использование типового параметра, который не был описан при объявлении класса:

public <T extends Musician> void add(ArrayList<T> list)

Однако,

public <T extends Musician> void add(ArrayList<T> list)

не то же самое, что:

public void add (ArrayList<Musician> list)

Оба выражения корректны, но они разные!

Wildcards

```
class Warrior {
}

class MagicWarrior extends Warrior {
}

class ArcherWarrior extends Warrior {
}

public static boolean fight(List<Warrior> w1, List<Warrior> w2) {
    return …
}
```

```
ArrayList<MagicWarrior> magi = new ArrayList<MagicWarrior>();
for (int i = 0; i < 10; i++)
    magi.add(new MagicWarrior());

ArrayList<ArcherWarrior> archers = new ArrayList<ArcherWarrior>();
for (int i = 0; i < 10; i++)
    archers.add(new ArcherWarrior());
boolean isMagicCooler = WarriorManager.fight(magi, archers); //ошибка компиляции!
```

Почему ошибка?

MagicWarrior – это наследник Warrior

List<MagicWarrior> — это не наследник List<Warrior>

https://docs.oracle.com/javase/tutorial/java/generics/wildcards.html

Решение проблемы:

```
public static boolean fight(List<? extends Warrior> w1, List<? extends Warrior> w2)
{
   return …
}
```

«? extends Warrior» обозначает «любой тип, унаследованный от Warrior ».

`List<?> - для передачи любого листа.`

Добавление в список

```
public void doSomething(List<? extends MyClass> list)
{
    for(MyClass object : list)
    {
        System.out.println(object.getState()); //тут все работает отлично.
    }
}
```

```
public void doSomething(List<? extends MyClass> list)
{
    list.add(new MyClass()); //ошибка!
}
```

Почему и что делать?

Дело в том, что в общем случае в метод doSomething можно передать List с типом элементов не MyClass, а любой из наследников MyClass. А в такой список заносить объекты MyClass уже нельзя!

List<? **super** MyClass> list

«? extends T» обозначает, что класс должен быть наследником T

«? super T» обозначает, что класс должен быть предком T или сам T, т.е. может быть даже List<Object>

T extends Class

public static class NumberContainer<T **extends** Number>

public static class NumberContainer<T extends Number & Comparable>

https://docs.oracle.com/javase/tutorial/java/generics/bounded.html

Какую конструкцию выбрать?

Все зависит от того, хотите ли вы использовать «T» где-нибудь еще.

Если есть метод, принимающий два списка, то проще написать:

public <T extends Musician> void view(ArrayList<T> one,
ArrayList<T> two)

вместо:

public void view(ArrayList<? extends Musician> one,
ArrayList<? extends Musician> two)

Принцип PECS - Producer Extends Consumer Super

- Use an `extends` wildcard when you only **get** values out of a structure.
- Use a `super` wildcard when you only **put** values into a structure.
- And don't use a wildcard when you both want to get and put from/to a structure.

```
     *         list-iterator does not support the <tt>set</tt> operation.
     */
    @Contract(mutates = "param1")
    public static <T> void copy( @NotNull List<? super T> dest,  @NotNull List<? extends T> src) {
        int srcSize = src.size();
        if (srcSize > dest.size())
            throw new IndexOutOfBoundsException("Source does not fit in dest");

        if (srcSize < COPY_THRESHOLD ||
            (src instanceof RandomAccess && dest instanceof RandomAccess)) {
            for (int i=0; i<srcSize; i++)
                dest.set(i, src.get(i));
        } else {
            ListIterator<? super T> di=dest.listIterator();
            ListIterator<? extends T> si=src.listIterator();
            for (int i=0; i<srcSize; i++) {
                di.next();
                di.set(si.next());
            }
        }
    }
```

Объявление метода sort()

public static <T extends Comparable<? super T>> void sort

(List<T> list)

T extends Comparable - любой T должен соответствовать типу Comparable

? super T - типовой параметр для Comparable должен быть типа T, или одним из его подтипов.

compareTo()

int compareTo(T o)

Возвращает:

- отрицательное целое число, если объект меньше указанного
- ноль, если объекты равны
- положительное число, если объект больше указанного

Интерфейс Comparator

int compare(T o1, T o2)

Возвращает те же значения, что и метод **compareTo()**.

Равенство ссылок и равенство объектов - не одно и то же. Но если у объектов равны ссылки, то объекты равны, так как по сути это один и тот же объект.

Правила сравнения объектов

- Если два объекта равны, то они должны иметь одинаковые идентификаторы.
- Если два объекта равны, то вызов **equals()** для каждого из них должен вернуть **true**:

a.equals(b) == b.equals(a)

- Если два объекта содержат одинаковые идентификаторы (хэши), то они не обязательно должны быть равны. Но если они равны, то идентификаторы должны совпадать.
- Если переопределяется метод **equals()**, то надо переопредить и **hashCode()**.

Правила сравнения объектов

- По умолчанию **hashCode()** генерирует уникальное целое число для каждого объекта в куче. Поэтому если не переопределить этот метод, никакие два объекта не смогут рассматриваться как равные.
- По умолчанию **equals()** проверяет, указывают ли две ссылки на один объект в куче. Поэтому если не переопределить этот метод, то два объекта не будут рассматриваться как равные, потому что ссылки на них всегда будут разными.
- **a.equals(b) == true** должно означать, что

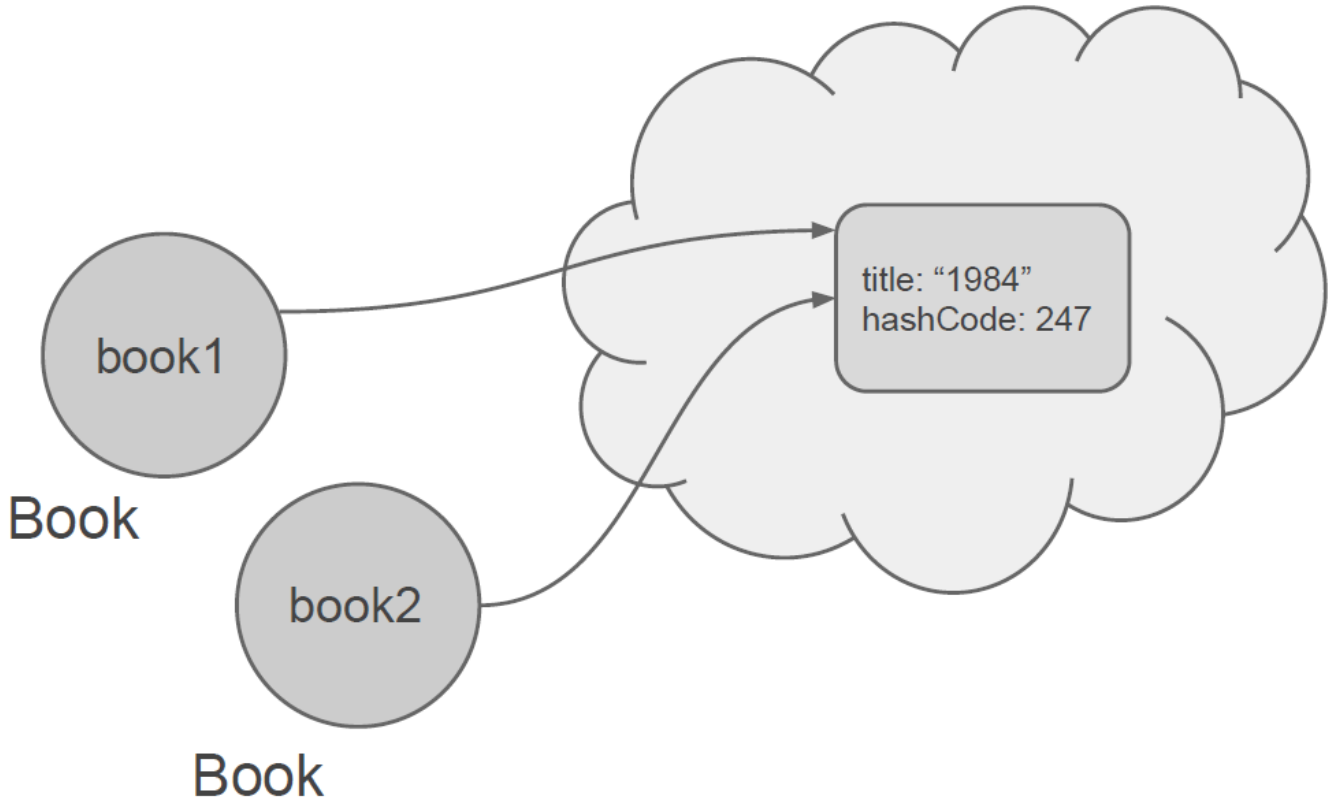
a.hashCode() == b.hashCode()

- Но если **a.hashCode() == b.hashCode()** , это не значит,

что **a.equals(b) == true**

Равенство ссылок

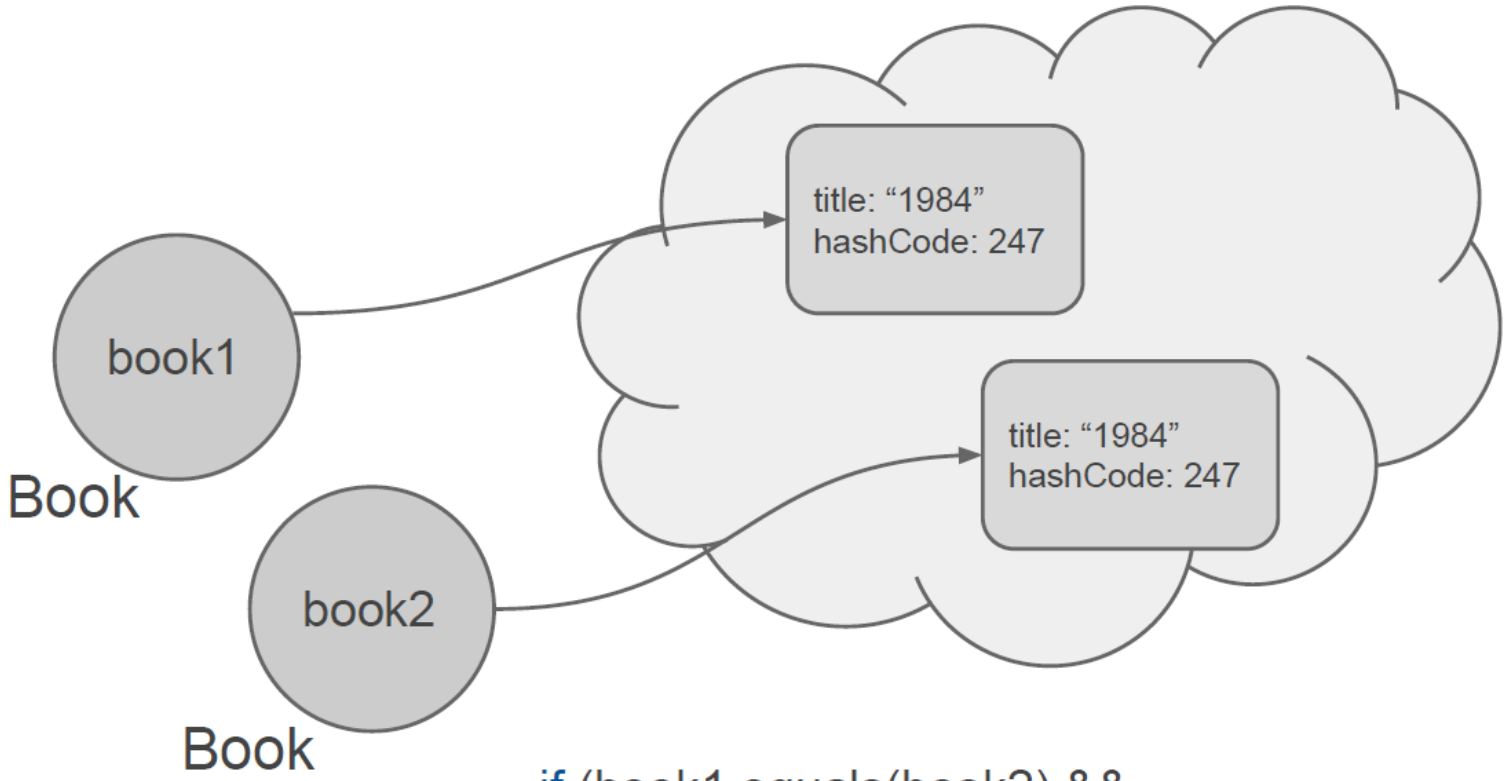
Две ссылки, связанные с одним объектом в куче, равны:




```
if (book1 == book2) { …
// Обе ссылки указывают на один объект
```

Равенство объектов

Чтобы воспринимать два объекта как одинаковые, нужно переопределить методы **hashCode()** и **equals()**.




```
if (book1.equals(book2) &&
    book1.hashCode() == book2.hashCode())
```

Iterator

Итераторы придумали практически тогда, когда и коллекции. Основная задача коллекций была – хранить элементы, а основная задача итератора – выдавать эти элементы по одному.

Итератор – это специальный внутренний объект в коллекции, который с одной стороны имеет доступ ко всем ее private данным и знает ее внутреннюю структуру, с другой – реализует общедоступный интерфейс Iterator, благодаря чему все знают, как с ним работать.

Iterator and Iterable

Методы интерфейса Iterator<E>	Описание
`boolean hasNext()`	Проверяет, есть ли еще элементы
`E next()`	Возвращает текущий элемент и переключается на следующий.
`void remove()`	Удаляет текущий элемент

Методы interface Iterable<T>	Описание
`Iterator<T>iterator()`	Возвращает объект-итератор

Пример использования

```
TreeSet<String> set = new TreeSet<String>();
Iterator<String> iterator = set.iterator();

while (iterator.hasNext())
{
    String item = iterator.next();
    System.out.println(item);
}
```

Сейчас можно проще

```
TreeSet<String> set = new TreeSet<String>();

for(String item : set)
{
    System.out.println(item);
}
```

ListIterator

Метод	Описание
`boolean hasNext()`	Проверяет, есть ли еще элементы впереди.
`E next()`	Возвращает следующий элемент.
`int nextIndex()`	Возвращает индекс следующего элемента
`void set(E e)`	Меняет значение текущего элемента
`boolean hasPrevious()`	Проверяет, есть ли элементы позади.
`E previous()`	Возвращает предыдущий элемент
`int previousIndex()`	Возвращает индекс предыдущего элемента
`void remove()`	Удаляет текущий элемент
`void add(E e)`	Добавляет элемент в список.

Недостатки массивов

- При создании необходимо задать длину
- Размер массива нельзя изменить
- Элементы массива привязаны к номерам ячеек

Коллекции

Коллекция — это группа элементов с операциями добавления, извлечения и поиска элемента.

В Java коллекции разделены на интерфейсы, абстрагирующие общие принципы работы с элементами, и классы, реализующие конкретную функциональность.

Механизмы работы коллекций существенно отличаются в зависимости от их типов.

Интерфейс Collection

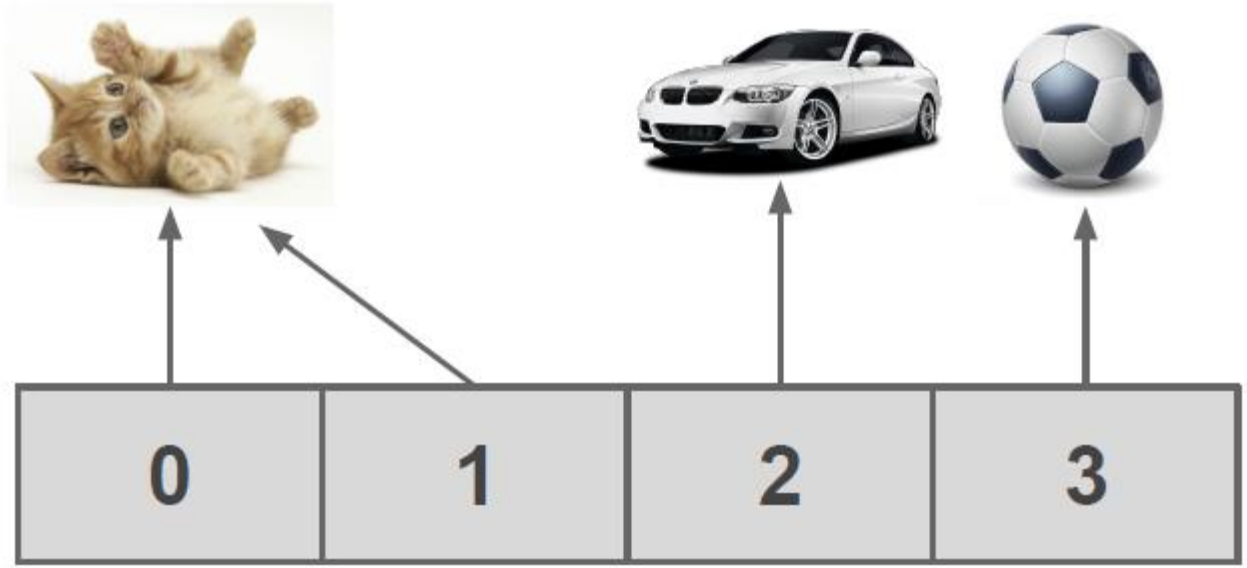
Методы	Описание
`boolean add(E e);`	Добавляет элемент в коллекцию
`boolean addAll(Collection<? extends E> c);`	Добавляет элементы в коллекцию
`void clear();`	Удаляет все элементы из коллекции
`boolean contains(Object o);`	Проверяет – есть ли в коллекции элемент?
`boolean containsAll(Collection<?> c);`	Проверяет – есть ли в коллекции элементы?
`boolean equals(Object o);`	Сравнивает коллекции
`int hashCode();`	Возвращает хэш-код
`boolean isEmpty();`	Проверяет – пуста ли коллекция?
`Iterator<E> iterator();`	Возвращает объект-итератор
`boolean remove(Object o);`	Удаляет элемент из коллекции
`boolean removeAll(Collection<?> c);`	Удаляет элементы из коллекции
`boolean retainAll(Collection<?> c);`	Удаляет все элементы, которых нет «c»
`int size();`	Возвращает размер коллекции
`Object[] toArray();`	Преобразовывает коллекцию к массиву
`<T> T[] toArray(T[] a);`	Преобразовывает коллекцию к массиву

Интерфейса **List<E>**

Методы	Описание
`void add(int index, E element);`	Добавляет элементы в середину коллекции
`boolean addAll(int index, Collection<? extends E> c);`	Добавляет элементы в коллекцию
`E get(int index);`	Возвращает элемент по номеру
`int indexOf(Object o);`	Возвращает индекс(номер) элемента
`int lastIndexOf(Object o);`	Возвращает последний индекс элемента.
`ListIterator<E> listIterator();`	Возвращает итератор для списка
`ListIterator<E> listIterator(int index);`	Возвращает итератор для списка
`E remove(int index);`	Удаляет элемент по индексу
`E set(int index, E element);`	Устанавливает новое значение по индексу
`List<E> subList(int fromIndex, int toIndex);`	Возвращает подколлекцию

Список (List)

- Список применяется, когда имеет значение порядок следования элементов.
- Учитывается индекс (позиция) элемента.
- Несколько элементов могут ссылаться на один объект.

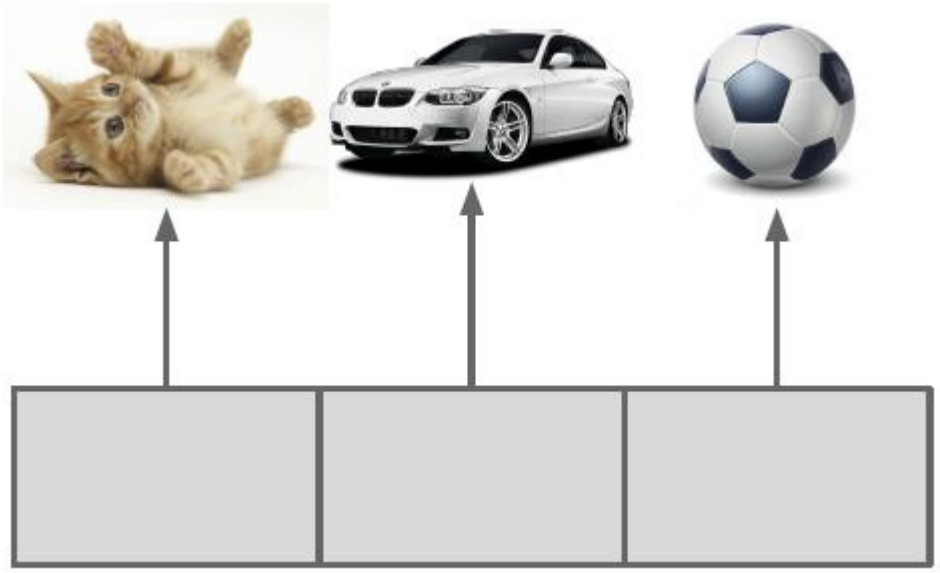


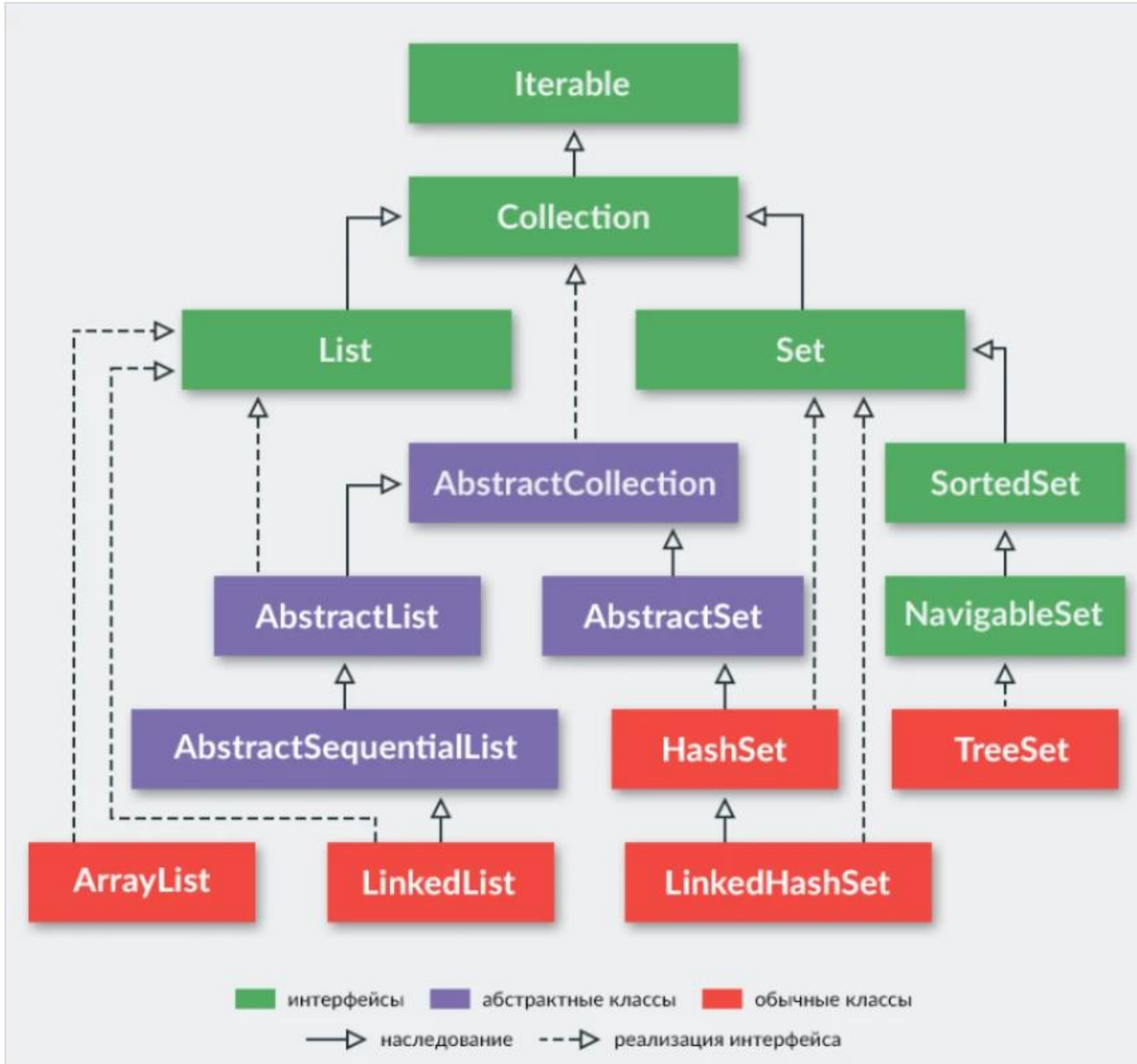
Set<E>

Методы	Описание
нет методов	

Множество (Set)

- Множество используется, когда нужно обеспечить уникальность элементов.
- Не допускает дублирования.
- Не может содержать элементы, ссылающиеся на один и тот же объект.



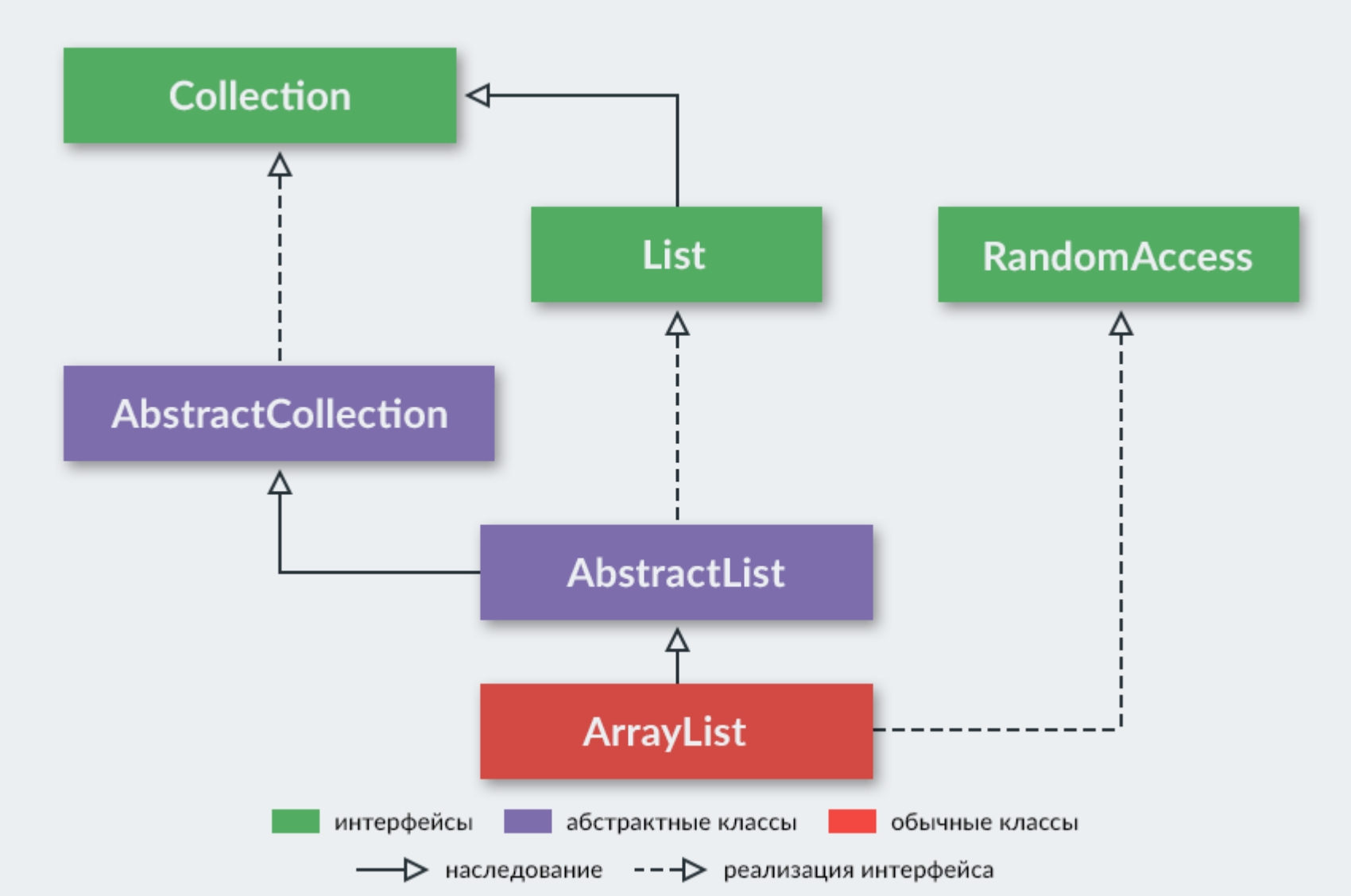

Iterable
Collection
List
Set
AbstractCollection
SortedSet
AbstractList
AbstractSet
NavigableSet
AbstractSequentialList
HashSet
TreeSet
ArrayList
LinkedList
LinkedHashSet
интерфейсы
абстрактные классы
обычные классы
наследование
реализация интерфейса

Устаревшие коллекции

Vector, Stack, Dictionary, Hashtable – они являются синхронизированными (потокобезопасными) версиями обычных коллекций.

Не нужны так как появилась специальная библиотека **concurrency.** Коллекции в данной библиотеке более эффективные.

ArrayList


Collection
List
RandomAccess
AbstractCollection
AbstractList
ArrayList
интерфейсы
абстрактные классы
обычные классы
наследование
реализация интерфейса

ArrayList

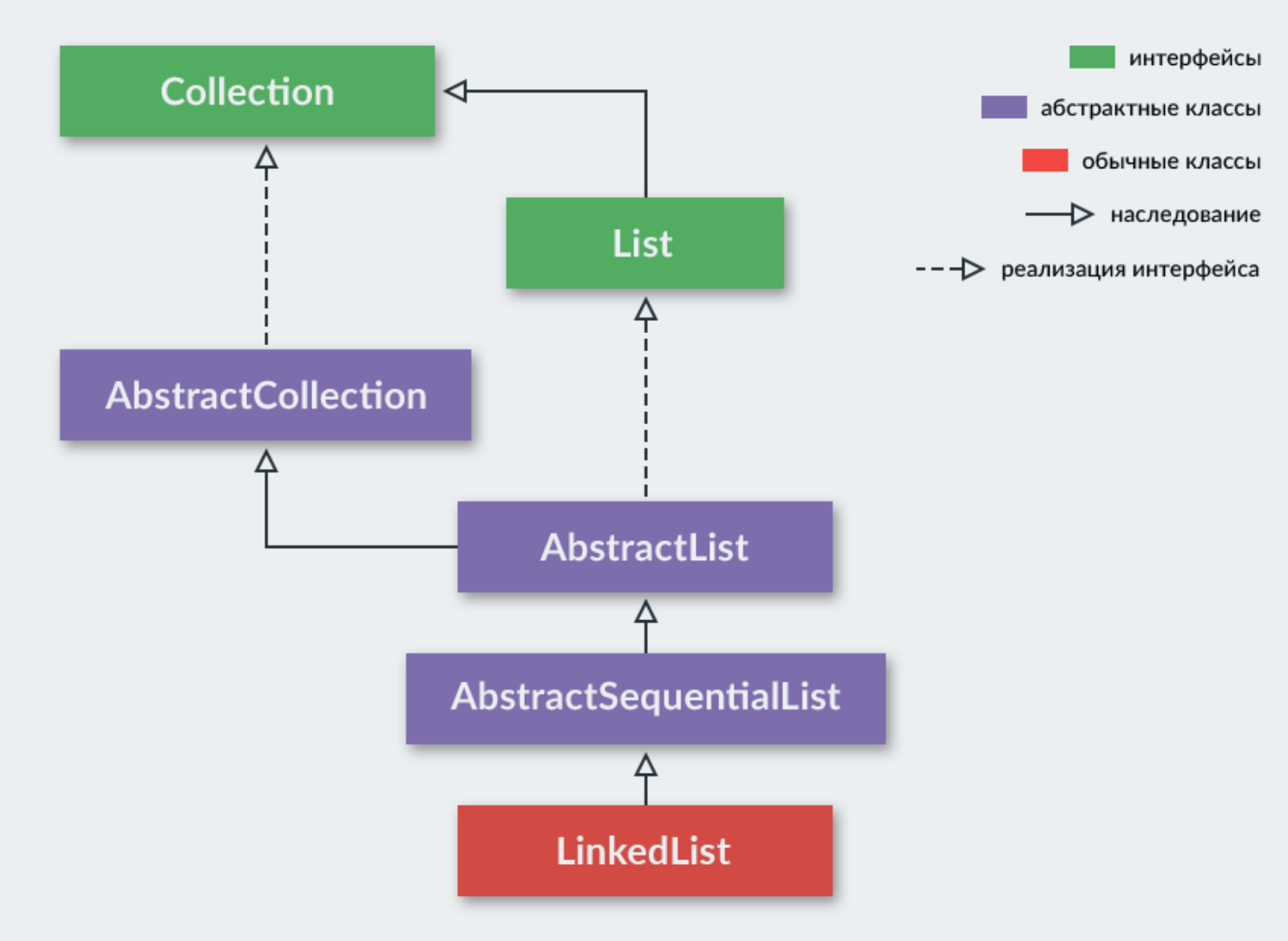
Список, аналог массива.

Основное отличие от массивов – обладает определенным поведением (содержит методы), тогда как в массиве доступно лишь свойство length.

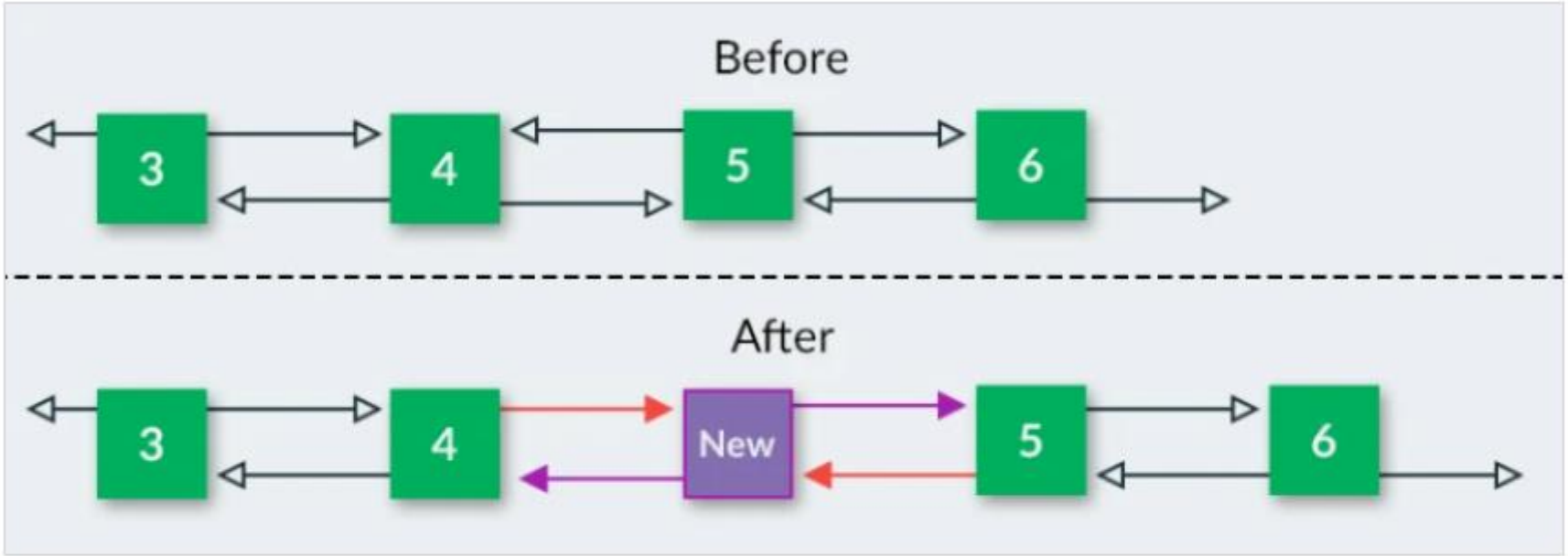
Сравнение ArrayList с массивами

В момент создания массива нужно указать размер.	Размер указывать необязательно. Контейнер увеличивается или уменьшается в зависимости от количества содержащихся в нем объектов. При желании размер можно задать.
При добавлении объекта в обычный массив нужно присвоить ему конкретный индекс. Возникает угроза переполнения.	add(Object) add(index, Object)
Может хранить примитивы.	-
Элементы добавляются в произвольном порядке.	Элементы хранятся в том порядке, в котором их добавляли.

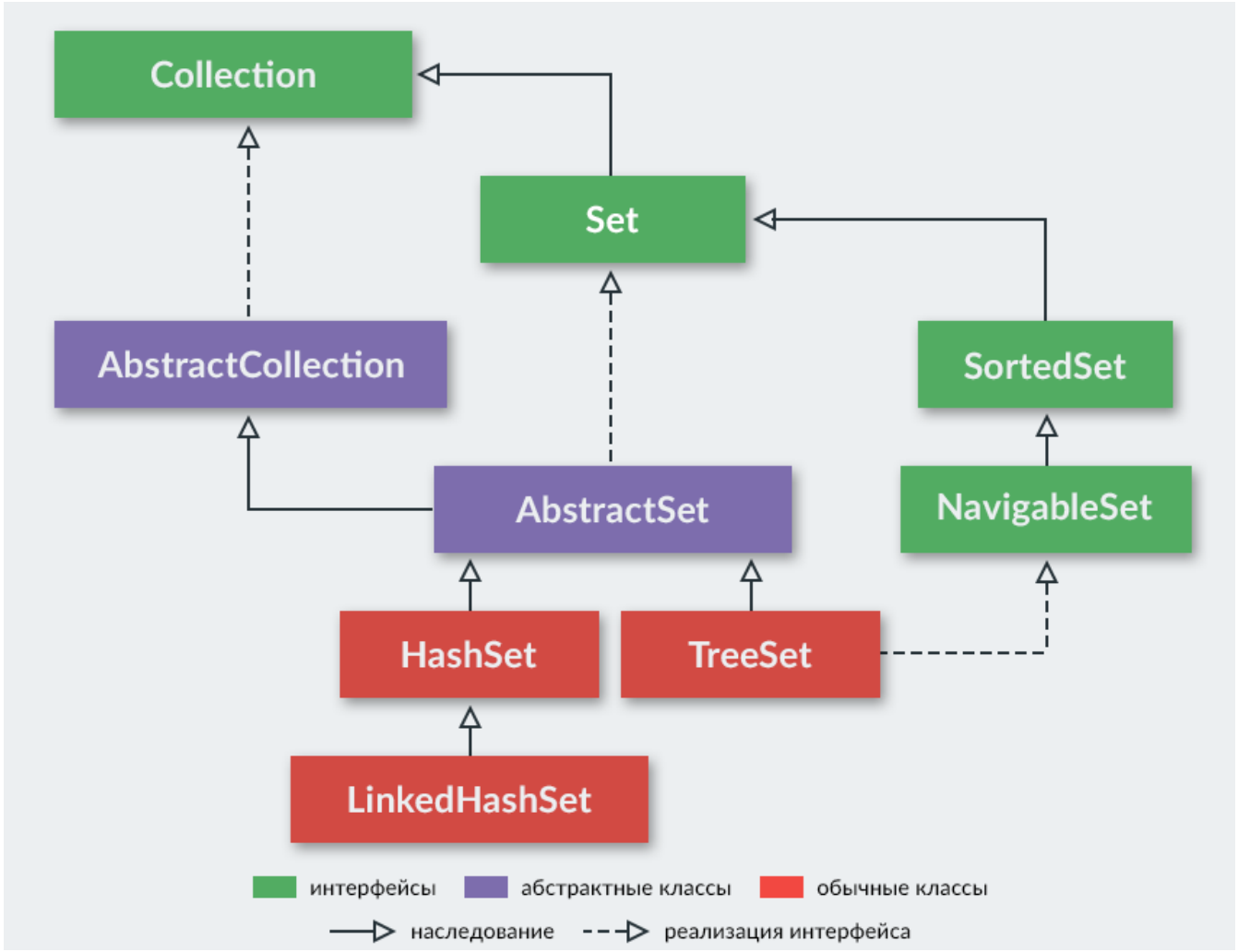
LinkedList


Collection
List
AbstractCollection
AbstractList
AbstractSequentialList
LinkedList
интерфейсы
абстрактные классы
обычные классы
наследование
реализация интерфейса

Вставка в середину



Set


Collection
Set
AbstractCollection
SortedSet
AbstractSet
NavigableSet
HashSet
TreeSet
LinkedHashSet
интерфейсы
абстрактные классы
обычные классы
наследование
реализация интерфейса

HashSet

HashSet – это коллекция, которая для хранения элементов внутри использует их хэш-значения, которые возвращает метод **hashCode**().

Для простоты внутри HashSet<E> хранится объект HashMap<E, Object>, который и хранит в качестве ключей значения HashSet.

Поиск в HashSet (и в HashMap) гарантированно работает правильно, только если объекты – **immutable**.

LinkedHashSet

LinkedHashSet – это HashSet, в котором элементы хранятся еще и в связном списке. Обычный HashSet не поддерживает порядок элементов. Во-первых, официально его просто нет, во-вторых, даже внутренний порядок может сильно поменяться при добавлении всего одного элемента.

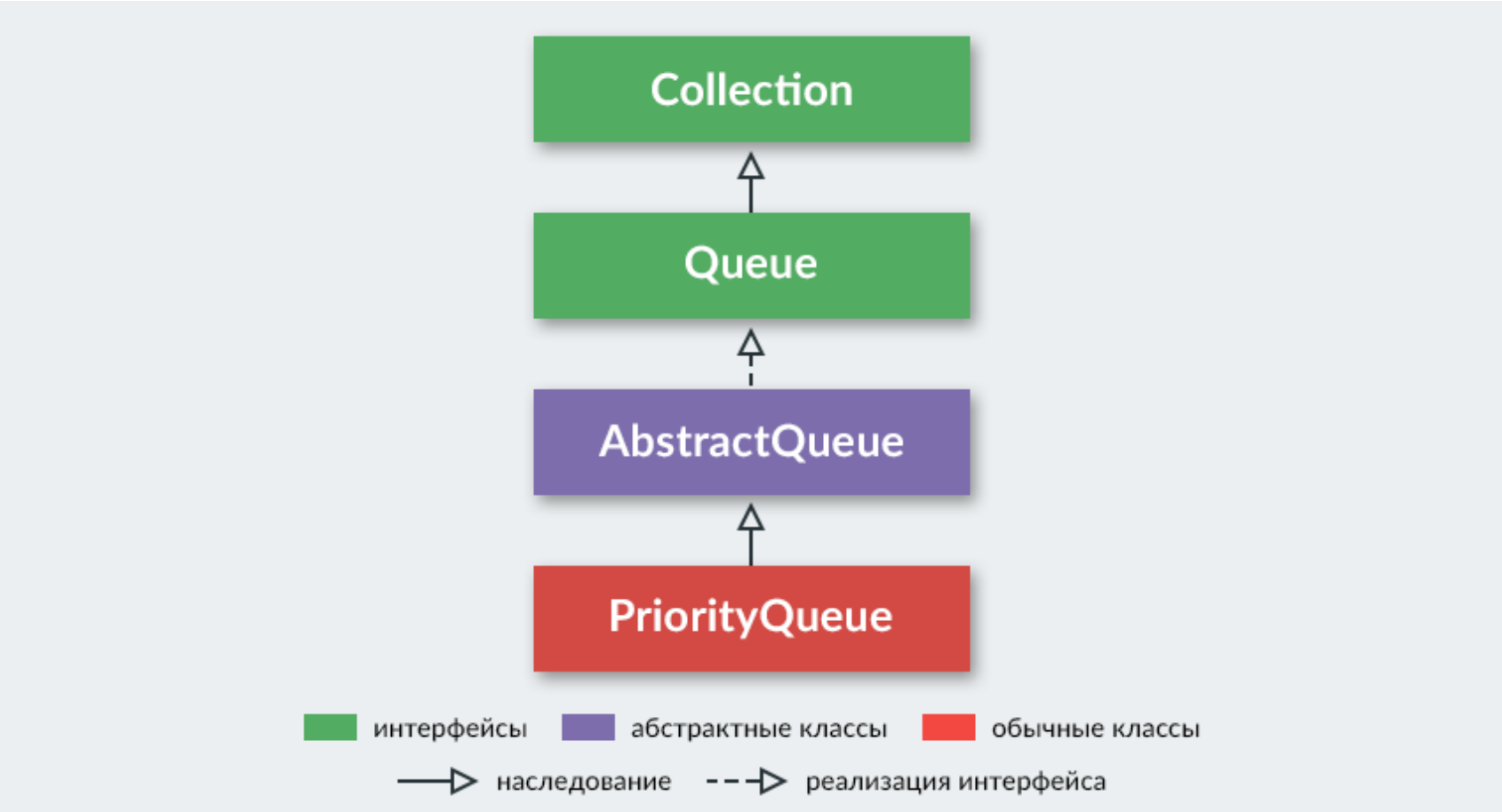
А у **LinkedHashSet** можно получить итератор и с его помощью обойти все элементы именно в том порядке, в котором они добавлялись в **LinkedHashSet**. Не часто, но иногда это может очень понадобится.

TreeSet

TreeSet – это коллекция, которая хранит элементы в виде упорядоченного по значениям дерева. Внутри **TreeSet**<E> содержится **TreeMap**<E, Object> который и хранит все эти значения. А этот **TreeMap** использует **красно—черное сбалансированное бинарное дерево** для хранения элементов. Поэтому у него очень быстрые операции **add, remove, contains.**

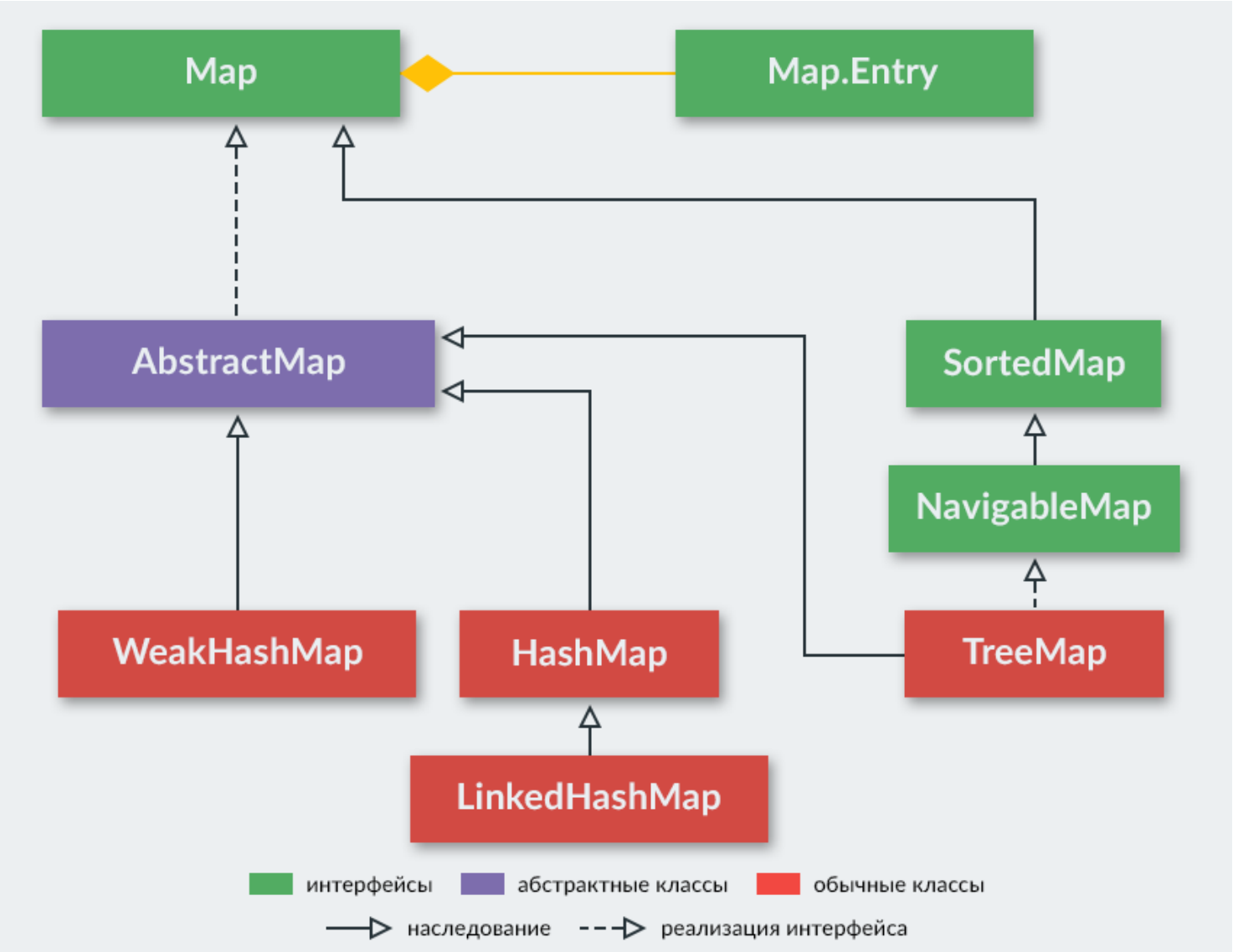
Что такое красно-черное сбалансированное бинарное дерево можно прочесть тут: https://habr.com/ru/post/330644/

Queue



Queue – это очередь. Элементы добавляются в конец очереди, а выбираются из ее начала.

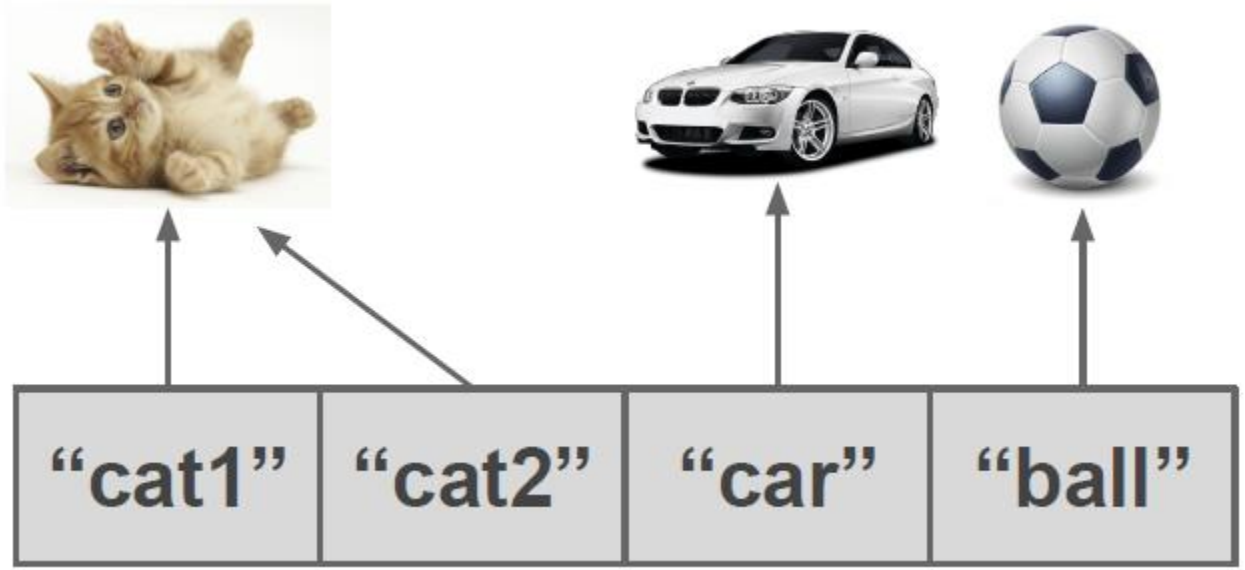
Map



Желтым отмечено, что **Entry** – это вложенный интерфейс в интерфейсе **Map**.

Отображение (Map)

- Отображение применяется для доступа к элементу по ключу.
- Ключи должны быть уникальными.
- Любой объект может быть ключом.
- Могут существовать два ключа, ссылающихся на один и тот же объект.



Основные методы

Методы	Описание
`int size()`	Возвращает количество пар в map.
`boolean isEmpty()`	Проверяет, пустой ли map.
`boolean containsKey(Object key)`	Содержит ли map заданный ключ?
`boolean containsValue(Object value)`	Содержит ли map заданное значение?
`V get(Object key)`	Возвращает значение по ключу.
`V put(K key, V value)`	Устанавливает новое значение ключу. Метод возвращает старое значение или null
`putAll(Map<? extends K, ? extends V> m)`	Добавляет пары из другого map.
`void clear()`	Очищает map – удаляет все пары.
`Set<K>keySet()`	Возвращает Set из ключей.
`Collection<V>values()`	Возвращает коллекцию из значений.
`Set<Map.Entry<K, V>>entrySet()`	Возвращает множество (Set) пар.

Entry

Методы	Описание
`K getKey()`	Возвращает значение «ключа пары».
`V getValue()`	Возвращает значение «значения пары».
`V setValue(V value)`	Устанавливает новое значение «значения пары».

HashMap

Она использует хэш-таблицы для хранения элементов. Ключи и значения могут быть любых типов, а также могут быть null. Порядок элементов может меняться при изменении коллекции.

Элементы хранятся внутри HashMap в виде набора групп – корзин (bucket). В какую корзину попадет элемент — зависит от значения его hashCode().

Более подробно: https://habr.com/ru/post/128017/

LinkedHashMap

Дополнительно хранит элементы в виде связного списка. У обычной HashMap порядок элементов неопределён и, строго говоря, может меняться со временем.

Более подробно: https://habr.com/ru/post/129037/

TreeMap

Хранит свои элементы отсортированными по возрастанию. Это достигается благодаря тому, что TreeMap для их хранения использует сбалансированное красно-черное бинарное дерево.

Благодаря этому там очень низкое время вставки и поиска элементов. Этот класс – отличный выбор при использовании очень больших объемов данных.

Более подробно: http://www.quizful.net/post/Java-TreeMap

Arrays - утилитный класс

Методы	Пояснение
`List<T> asList(T... a)`	Возвращает неизменяемый список, заполненный переданными элементами.
`int binarySearch(int[] a, int fromIndex, int toIndex, int key)`	Ищет элемент (key) в массиве a или подмассиве, начиная с fromIndex и до toIndex. Массив должен быть отсортирован! Возвращает номер элемента или fromIndex-1, если элемент не найдет.
`int[] copyOf(int[] original, int newLength)`	Возвращает подмассив original массива, newLength элементов, начиная с нулевого.
`int[] copyOfRange(int[] original, int from, int to)`	Возвращает под массив original массива, начиная с from и до to.
`boolean deepEquals(Object[] a1, Object[] a2)`	Глубокое сравнение массивов. Массивы считаются равными, если равны их элементы. Если элементы сами являются массивами, для них тоже

`int deepHashCode(Object a[])`	Глубокий хэшкод на основе всех элементов. Если элемент является массивом, для него также вызывается deepHashCode.
`String deepToString(Object[] a)`	Глубокое преобразование к строке. Для всех элементов вызывается toString(). Если элемент является массивом, для него тоже выполняется глубокое преобразование к строке.
`boolean equals(int[] a, int[] a2)`	Сравнивает два массива поэлементно.
`void fill(int[] a, int fromIndex, int toIndex, int val)`	Заполняет массив (или подмассив) заданным значением.
`int hashCode(int a[])`	Вычисляет общий hashCode всех элементов массива.
`void sort(int[] a, int fromIndex, int toIndex)`	Сортирует массив (или подмассив) по возрастанию.
`String toString(int[] a)`	Преобразовывает массив к строке. Для всех элементов вызывается toString();

Collections

`void copy(List<? super T> dest, List<? extends T> src)`	Копирует список «src»в список «dest».
`T max(Collection<? extends T> coll)`	Ищет максимальное число/ значение в коллекции. Как найти максимум из 6-и чисел? Collections.max(Arrays.asList(51, 42, 33, 24, 15, 6));
`T min(Collection<? extends T> coll)`	Ищет минимальное значение в коллекции.
`boolean replaceAll(List<T> list, T oldVal, T newVal)`	Заменяет в коллекции list все элементы oldVal на newVal
`void reverse(List<?> list)`	Разворачивает список задом наперед.

null vs empty

Старайтесь из методов возвращать пустые коллекции вместо **null**. Это избавляет от лишних проверок.

В классе Collections для этого есть несколько методов:

Collections.emptyList();

Collections.emptyMap();

Collections.emptySet();

Для самостоятельного изучения

Сложность алгоритмов: https://tproger.ru/articles/computational-complexity-explained/

Красно-черное дерево: https://habr.com/ru/post/330644/