

# Программирование на Java

## 3. Библиотека коллекций

Глухих Михаил Игоревич  
mailto: [glukhikh@mail.ru](mailto:glukhikh@mail.ru)

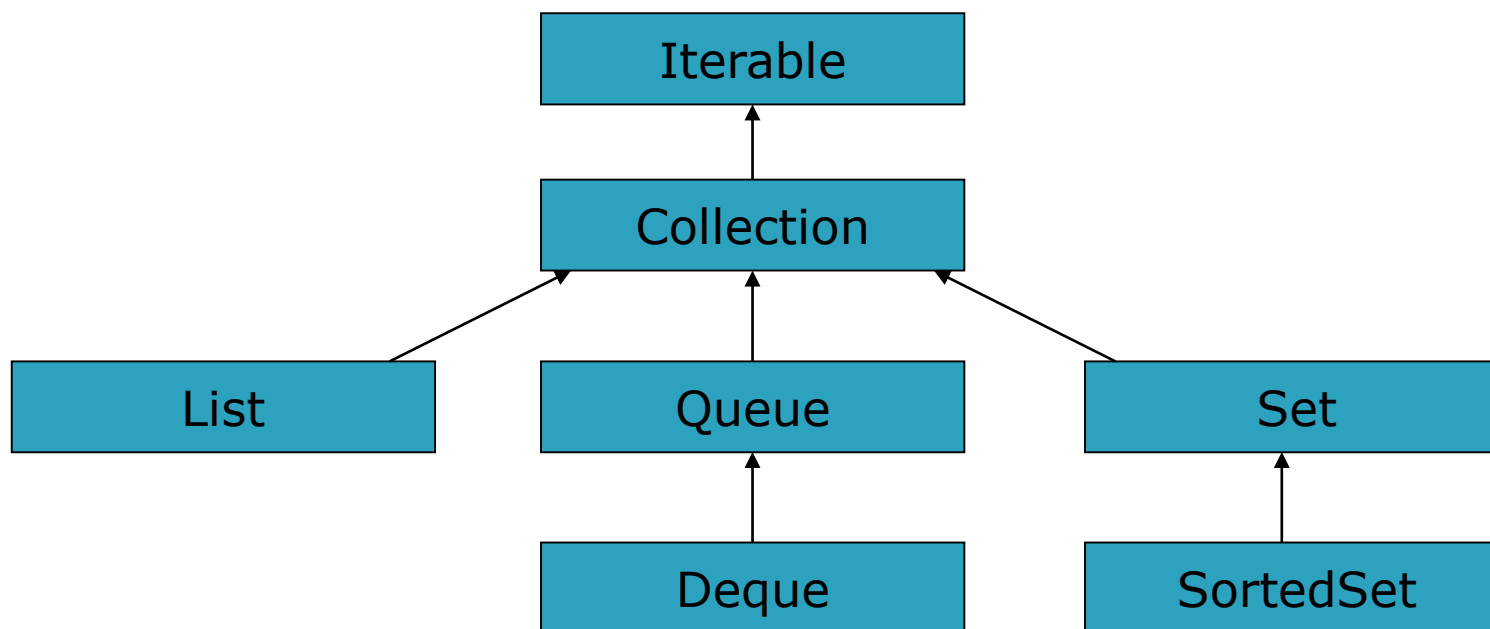
# Назначение

- ▶ Работа с контейнерами различного типа:
  - списки
  - множества
  - очереди
  - ассоциативные массивы
  - ...

# Организация

- ▶ Интерфейс  
(что мы умеем делать)
  - Абстрактный класс  
(частичная реализация действий)
    - Класс  
(полная реализация действий)

# Иерархия интерфейсов коллекций



# Принцип наследования

- ▶ Iterable ← Collection  
всё, что является Collection, является Iterable  
Collection is Iterable

# Неизменяемые vs Изменяемые

- ▶ Котлин: compile time
  - Collection / MutableCollection
  - List / MutableList
  - Set / MutableSet
- ▶ Java: run time
  - Collection (mutable?)
  - List (mutable?)
  - Set (mutable?)

# Интерфейс Iterable<T>

- ▶ Означает перебираемый
- ▶ Содержит внутри себя элементы T, которые можно перебирать с помощью цикла for-each

```
Iterable<Integer> container =  
    new ArrayList<Integer>();  
// ...  
for (Integer element: container) {  
}
```

# Интерфейсы Iterable<T> и Iterator<T>

```
public interface Iterable<T> {  
    Iterator<T> iterator();  
}  
// Класс, реализующий итератор,  
// перебирает чьи-то элементы  
// Класс-помощник  
public interface Iterator<T> {  
    boolean hasNext();  
    T next();  
    void remove();  
}
```



# Порядок использования итератора

- ▶ Проверить, есть ли следующий элемент (`hasNext`)
- ▶ Получить следующий элемент (`next`)
- ▶ Если нам это требуется, удалить его (`remove`)
- ▶ Повторить
- ▶ **НЕЛЬЗЯ** во время работы итератора изменять содержимое перебираемой коллекции (за исключением метода `it.remove()`)

# Пример использования итератора

```
Iterable<Integer> container =  
    new ArrayList<Integer>();  
// ...  
// Аналог цикла for-each  
Iterator<Integer> it =  
    container.iterator();  
while (it.hasNext()) {  
    Integer element = it.next();  
    // ...  
}
```

# Интерфейс Collection<E>

- ▶ Корневой интерфейс иерархии коллекций (реализуется всеми типами из данной иерархии)
- ▶ Коллекция состоит из элементов (типа E)
- ▶ Элементы (по умолчанию) могут дублироваться
- ▶ Коллекции (по умолчанию) неупорядочены, то есть, неизвестно, какой элемент на какой позиции стоит
- ▶ Расширяет Iterable<E>

# Методы интерфейса Collection<E>

```
public interface Collection<E> extends Iterable<E> {  
    boolean contains(Object obj);  
    boolean containsAll(Collection<?> c);  
    boolean isEmpty();  
    int size();  
    Object[] toArray();  
    <T> T[] toArray(T[] arr);  
    // Mutable-only!  
    boolean add(E obj);  
    boolean addAll(Collection<? extends E> c);  
    void clear();  
    boolean remove(Object obj);  
    boolean removeAll(Collection<?> c);  
    boolean retainAll(Collection<?> c);  
}
```

# Использование Collection

- ▶ Интерфейс Collection напрямую не реализуется классами из JDK
- ▶ Реализуются его расширения: List, Queue, Set
- ▶ Интерфейс Collection часто используется в аргументах функций (когда нам требуется передать какую-нибудь коллекцию, все равно какую)

# Пример использования Collection

```
static public int calcSum(  
    Collection<Integer> c) {  
    int sum = 0;  
    for (Integer i: c)  
        sum += i;  
    return sum;  
}
```

# Методы интерфейса Collection<E> (Java 8)

```
public interface Collection<E> extends Iterable<E> {  
    default Stream<E> parallelStream();  
    default Spliterator<E> spliterator();  
    default Stream<E> stream();  
    // Mutable-only!  
    default boolean removeIf(  
        Predicate<? super E> filter);  
}
```

# Stream<E> (Java 8)

- ▶ Описывает последовательность элементов (в широком смысле)



# Stream<E> (Java 8)

- ▶ Описывает последовательность элементов (в широком смысле)
- ▶ Ленивые (ничего не вычисляется без необходимости)

# Stream<E> (Java 8)

- ▶ Описывает последовательность элементов (в широком смысле)
- ▶ Ленивые (ничего не вычисляется без необходимости)
- ▶ Содержат преобразующие методы (Stream → Stream, например filter, map)

# Stream<E> (Java 8)

- ▶ Описывает последовательность элементов (в широком смысле)
- ▶ Ленивые (ничего не вычисляется без необходимости)
- ▶ Содержат преобразующие методы (Stream → Stream, например filter, map)
- ▶ А также терминальные методы (Stream → ..., например count, anyMatch)

# Stream<E> (Java 8)

- ▶ Описывает последовательность элементов (в широком смысле)
- ▶ Ленивые (ничего не вычисляется без необходимости)
- ▶ Содержат преобразующие методы (Stream → Stream, например filter, map)
- ▶ А также терминальные методы (Stream → ..., например count, anyMatch)
- ▶ Бывают последовательные и параллельные: см. isParallel

# Частичная реализация Collection – класс AbstractCollection

- ▶ Абстрактный класс, в отличие от интерфейса, может содержать члены-данные и реализации методов
- ▶ В данном случае все методы коллекции реализованы на основе других (skeletal implementation):
  - size()
  - iterator()
  - add()
- ▶ Некоторые методы (например, clear) реализованы заведомо неэффективно
- ▶ Класс нужен для того, чтобы можно было быстро создать свою коллекцию

# Методы интерфейса Stream<E> (Java 8) – примеры

```
public interface Stream<E> {  
    // Builder  
    static <E> Stream<E> of(T... values);  
    // Intermediate  
    Stream<E> distinct();  
    Stream<E> filter(Predicate<? super E> predicate);  
    Stream<R> map(Function<? super E, ? extends R>  
                  mapper);  
    // Terminal  
    boolean allMatch(Predicate<? super E> predicate);  
    boolean anyMatch(Predicate<? super E> predicate);  
    long count();  
    void forEach(Consumer<? super E> action);  
}
```

# SAM-интерфейсы и лямбды

- ▶ SAM = Single Abstract Method

```
public interface Predicate<T> {  
    boolean test(T t);  
    // + some default methods  
}
```

# SAM-интерфейсы и лямбды

- ▶ SAM = Single Abstract Method

```
public interface Predicate<T> {  
    boolean test(T t);  
    // + some default methods  
}
```

- ▶ Использование напрямую

```
list.stream()  
    .filter(new Predicate<Integer>() {  
        @Override  
        boolean test(Integer arg) {  
            return arg > 0;  
        }  
    }) .sum();
```



# SAM-интерфейсы и лямбды

- ▶ SAM = Single Abstract Method

```
public interface Predicate<T> {  
    boolean test(T t);  
    // + some default methods  
}
```

- ▶ Использование в виде лямбды

```
list.stream()  
    .filter { e -> e > 0 }  
    .sum();
```

# Splitter<E> (Java 8)

- ▶ Напоминает итератор, вспомогательный класс для потоков
- ▶ Главное отличие – умеет делиться надвое (split)

# Частичная реализация Collection – класс AbstractCollection

```
public abstract class AbstractCollection<E>
    implements Collection<E> {
    boolean isEmpty() { return size()==0; }
    boolean addAll(Collection<? extends E> c) {
        for (E e: c)
            add(e);
    }
    boolean contains(Object obj) {
        for (E e: this) {
            if (obj==null ? e==null : obj.equals(e))
                return true;
        }
        return false;
    }
}
```

# Абстрактные классы VS интерфейсы

- ▶ Могут иметь нестатические-члены данные
- ▶ Могут включать реализацию некоторых функций
- ▶ Могут включать абстрактные abstract функции
- ▶ Могут иметь конструкторы
- ▶ Могут использоваться другими классами как базовые:  
`public class ArrayList extends AbstractList`
- ▶ Класс может иметь только один базовый класс (абстрактный или нет)

# Интерфейсы VS абстрактные классы

- ▶ Могут иметь только статические члены–данные
- ▶ Могут включать только абстрактные функции
- ▶ Не могут иметь конструкторы
- ▶ Могут реализовываться другими классами:  
`public class` AbstractList `implements` List
- ▶ Класс может реализовывать любое количество интерфейсов

# Абстрактные классы и интерфейсы

Feature	Interface	abstract class
Non-static fields	No	Yes
Method implementations	No (except Java 8)	Yes
Constructors	No	Yes
Class inherits by...	“implements”, N times	“extends”, one time
Interface inherits by...	“extends”	Not possible
Member visibility	public	any

# Типичное использование абстрактных классов и интерфейсов

- ▶ Когда требуется указать некоторый перечень возможностей объекта, используется интерфейс
- ▶ Когда требуется реализовать некоторые общие для разных объектов функции, используется абстрактный класс

# Интерфейс List<E>

- ▶ Список – упорядоченная коллекция
- ▶ У каждого элемента списка (в отличие от обобщенной коллекции) есть своя позиция
- ▶ Могут быть дублированные элементы
- ▶ Добавлены методы, связанные с конкретными позициями
- ▶ Кроме этого, изменены контракты некоторых методов Collection



# Методы интерфейса List<E>

```
public interface List<E> extends Collection<E> {  
    E get(int index);  
    int indexOf(Object obj);  
    int lastIndexOf(Object obj);  
    List<E> subList(int fromIndex, int toIndex);  
    ListIterator<E> listIterator(); // extends simple one  
    // Mutable-only! add(obj) inserts element to the end  
    boolean add(int index, E obj);  
    boolean addAll(int index, Collection<? extends E> c);  
    void clear();  
    E remove(int index);  
    E set(int index, E obj);  
    // Java 8  
    default void replaceAll(UnaryOperator<E> operator);  
    default void sort(Comparator<? super E> c);  
}
```

# Интерфейс ListIterator<E> – списочный итератор

- ▶ Расширяет интерфейс Iterator<E> (перебор в две стороны, вставка и замена элементов, работа с индексами)
- ▶ Методы

```
boolean hasPrevious();  
E previous();  
int nextIndex();  
int previousIndex();  
void set(E elem);  
void add(E elem);
```

# Частичная реализация List – класс `AbstractList`

- ▶ Также скелетальная реализация на основе методов
  - `add(index, element)`
  - `get(index)`
  - `set(index, element)`
  - `remove(index)`
  - `size()`
- ▶ Расширяет класс `AbstractCollection`

# Пример: реализация списка на основе заданного массива

```
static List<Integer> intArrayAsList(final int[] a) {  
    if (a==null) throw new NullPointerException();  
    return new AbstractList<Integer>() {  
        @Override  
        public Integer get(int i) { return a[i]; }  
        @Override  
        public Integer set(int i, Integer val) {  
            int oldVal = a[i];  
            a[i] = val;  
            return oldVal;  
        }  
        @Override  
        public int size() { return a.length; }  
    }  
}
```

# Частичная реализация List – класс `AbstractSequentialList`

- ▶ Также скелетальная реализация на основе **одного** метода
  - `listIterator(index)` – получение списочного итератора, указывающего на заданный элемент
- ▶ Расширяет класс `AbstractList`

# Реализации интерфейса List

- ▶ Реализация на основе массива – ArrayList, расширяет AbstractList

```
// Java 7+
```

```
List<Integer> c = new ArrayList<>();
```

```
List<Integer> d = new ArrayList<>(c);
```

```
// Java 6
```

```
List<Integer> c = new ArrayList<Integer>();
```

- ▶ Реализация на основе линейного списка – LinkedList, расширяет AbstractSequentialList
  - похожие конструкторы

# Сравнение реализаций

- ▶ ArrayList – быстрее выполняется доступ в произвольное место массива, медленнее – удаление и вставка элементов в заданное место
- ▶ LinkedList – быстрее выполняется удаление и вставка элементов в заданное место, медленнее – доступ в произвольное место

# Сравнение реализаций

Operation	ArrayList	LinkedList
Access by index	$O(1)$	$O(\text{index})$
Add to the end	$O(1)$	$O(1)$
Add to the beginning	$O(\text{size})$	$O(1)$
Add into the middle	$O(\text{size})$	$O(1) / O(\text{size}) (*)$
Delete from the end	$O(1)$	$O(1)$
Delete from the beginning	$O(\text{size})$	$O(1)$
Delete from the middle	$O(\text{size})$	$O(1) / O(\text{size}) (*)$
Iterate through	$O(\text{size})$	$O(\text{size})$



# Интерфейсы как типы

- ▶ Везде, где возможно, лучше задавать тип интерфейсом – это позволяет легко менять реализацию

```
List<Integer> c = new ArrayList<>();
```

- ▶ НО!

```
ArrayList<Integer> c =  
new ArrayList<>();
```

# Интерфейс Set<E>

- ▶ Множество, не содержащее равных элементов
- ▶ Неупорядочено – неизвестно, какой элемент на какой позиции
- ▶ **Не содержит новых** по сравнению с коллекцией методов, однако модифицирует контракты некоторых существующих методов:
  - add – не добавляет уже присутствующий во множестве элемент
  - equals – множества равны, если равны их размеры, и каждый элемент одного содержится в другом

# Вопрос

- ▶ Как правильно сформировать хэш-код множества?

# Вопрос

- ▶ Как правильно сформировать хэш-код множества?
- ▶ Основная проблема – хэш-код должен быть одинаковым независимо от порядка, в котором перебираются элементы

# Реализации интерфейса Set

- ▶ Имеется частичная реализация – `AbstractSet` (расширяет `AbstractCollection`, базовые функции те же)
- ▶ Реализация на основе хэш-таблицы – `HashSet`
  - используется хэш-поиск для обращения к элементам
  - при удачно написанной хэш-функции время выполнения `add`, `remove`, `contains` не зависит от размера множества
- ▶ Та же хэш-таблица, но упорядоченная – `LinkedHashSet`

# Реализации интерфейса Set

- ▶ Реализация на основе бинарного дерева – TreeSet
  - реализует интерфейс SortedSet
  - порядок либо на основе Comparable<T>, либо на основе Comparator<T>
  - используется бинарный поиск для обращения к элементам
  - логарифмическое время поиска

# Реализации интерфейса Set

- ▶ Реализация на основе битового поля – EnumSet
  - только для перечислений
  - каждому элементу перечисления ставится в соответствие один бит
  - нет традиционных конструкторов, вместо этого

```
Set<Planet> set = EnumSet.of(  
    Planet.MERCURY, Planet.EARTH)
```

# ИТОГИ

- ▶ Рассмотрены:  
Iterable, Collection, Stream, List, Set
- ▶ Далее:  
Deque, Map, ...