# Программирование на Java

# 4. Библиотека коллекций

Глухих Михаил Игоревич
mailto: glukhikh@mail.ru

# Интерфейс Map<K,V>

- Ассоциативный массив, хранящий в себе пары ключ-значение (Key-Value)
- Ключи и значения неупорядочены, но каждое значение жестко привязано к своему ключу
- Совпадение ключей не допускается
- Интерфейс-помощник **Entry** – одна пара ключ-значение

# Возможности интерфейса Map<K,V>

```
public interface Map<K,V> {
    int size();
    boolean isEmpty();
    boolean containsKey(Object key);
    boolean containsValue(Object value);
    V get(Object key);
    V put(K key, V value);
    V remove(Object key);
    void putAll(Map<? extends K, ? extends V> m);
    void clear();
    Set<K> keySet();
    Collection<V> values();
    Set<Entry<K,V>> entrySet();
}
```

# Доп. возможности интерфейса Map<K,V> – Java 1.8

```
public interface Map<K,V> {
    V compute(K key, BiFunction/*(K,V)-> V*/ remapping);
    V computeIfAbsent(K key,
                      Function/*(K)-> V*/ mapping);
    V computeIfPresent(K key,
                       BiFunction/*(K,V)-> V*/ remapping);
    void forEach(BiConsumer<? super K, ? super V> action);
    V getOrDefault(Object key, V defaultValue);
    V merge(K key, V value,
            BiFunction/*(K,V)-> V*/ remapping);
    V putIfAbsent(K key, V value);
    boolean remove(Object key, Object value);
    V replace(K key, V value);
    boolean replace(K key, V oldValue, V newValue);
    void replaceAll(BiFunction/*(K,V)->V*/ function);
}
```

# Возможности интерфейса Entry<K,V>

```
public interface Entry<K,V> {
    K getKey();
    V getValue();
    V setValue();
}
```

# Реализации интерфейса Map<K,V>

- Имеется частичная реализация – AbstractMap (скелет, на базе **одной** функции entrySet)
- Реализация на основе хэш-таблицы – HashMap
  - используется хэш-поиск для обращения к элементам
  - при удачно написанной хэш-функции время выполнения put, remove, get, containsKey не зависят от размера массива
  - HashSet<E> – на самом деле HashMap<E, Object>

# Реализации интерфейса Map<K,V>

- Реализация на основе бинарного дерева – TreeMap<K, V>
  - реализует интерфейс SortedMap
  - порядок либо на основе Comparable<K>, либо на основе Comparator<V>
  - используется бинарный поиск для обращения к элементам
  - логарифмическое время поиска
  - несколько проще добраться до соседних элементов, чем в HashSet

# Реализации интерфейса Map<K,V>

- Реализация с перечислением-ключом: EnumMap
- В основе используется массив, индексом которого является номер элемента перечисления

```
Map<Planet, Set<Properties>> planetProperties =
  new EnumMap<Planet, Set<Properties>>(Planet.class);
```

- Зачем аргумент Planet.class??

# Перечисления в Java

- Предназначены для реализации типов с ограниченным количеством значений
- Пример:
  тип "планета солнечной системы"
  - Меркурий, Венера, Земля, ...

# Простое определение перечисления

```
public enum Planet {
    // Элементы перечисления
    MERCURY,
    VENUS,
    EARTH,
    MARS,
    JUPITER,
    SATURN,
    URANUS,
    NEPTUN;
}
```

# Использование перечисления

```
Planet p1 = Planet.EARTH;
// или
switch (p1) {
    case MERCURY:
        // ...
        break;
    case MARS:
        // ...
        break;
    default:
        break;
}
```

# Общие методы перечислений

- На самом деле перечисления – это классы, наследующие класс Enum

```
// Перебрать все планеты
for (Planet p: Planet.values()) {
    // Вывести числовой код
    System.out.println(p.ordinal());
}
// Определить планету по строке-названию
Planet p = Planet.valueOf("EARTH");
```

# Дополнительные методы перечислений

```
public enum Planet /* See part2.planet example */ {
    MERCURY(3.302e+23, 2.439e+06),
    // ...
    NEPTUNE(1.024e+26, 2.477e+07);
    private final double mass;
    private final double radius;
    private final double gravity;

    private static final double G = 6.67300e-11;
    Planet(double mass, double radius) {
        this.mass = mass;
        this.radius = radius;
        this.gravity = G * mass / (radius * radius);
    }
}
```

# Что такое перечисление и его элементы в Java?

- Перечисление – это тоже класс, наследник класса Enum
- Элемент перечисления – это статическое поле соответствующего класса, тип поля совпадает с типом перечисления

# Интерфейс Queue<E>

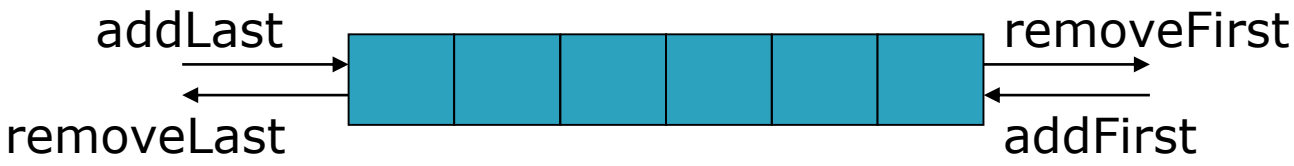
- Коллекция, с которой можно работать, как с очередью (FIFO)
- Добавлены методы
  - add(e)/offer(e) – добавление элемента в хвост очереди
  - remove()/poll() – удаление элемента из головы очереди
  - element()/peek() – просмотр элемента из головы очереди
- Контракты прочих методов не меняются

# Реализация интерфейса Queue

- Частичная реализация – AbstractQueue
- Одна из полных реализаций – LinkedList (обратите внимание, что данный класс реализует как интерфейс List, так и интерфейс Queue)

# Интерфейс Deque<E>

- Коллекция, с которой можно работать, как с двухсторонней очередью (можно добавлять/удалять элементы как из головы, так и из хвоста)
- Расширяет Collection, добавлены методы
  - addFirst(e)/offerFirst(e)/addLast(e)/offerLast(e) – добавление элемента в начало/конец очереди
  - removeFirst()/pollFirst()/removeLast()/pollLast() – удаление элемента из начала/конца очереди
  - getFirst()/peekFirst()/getLast()/peekLast() – просмотр элемента из начала/конца очереди
  - push(e)/pop() – работа с очередью как со стеком
- Контракты прочих методов не меняются



# Реализация интерфейса Deque

- ArrayDeque – реализация на основе массива
- LinkedList – реализация на основе списка (да, и этот интерфейс LinkedList тоже реализует)

# Пример на Map и Deque – вычислитель выражений

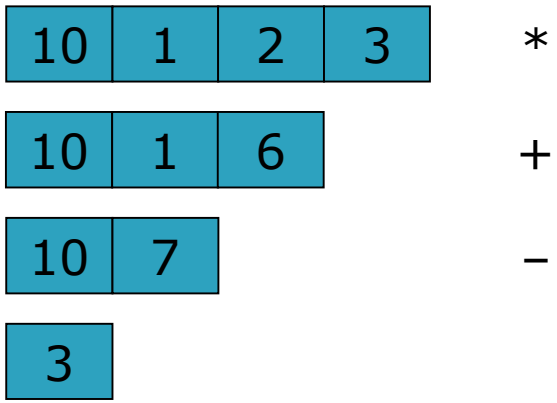
- Выражения представлены в обратной польской записи (знак операции следует за операндами) – например
  - 10 1 2 3 * + –
- Необходимо реализовать класс для вычисления подобных выражений и протестировать его

# Вычислитель выражений – основные классы

- Собственно вычислитель
  - разбивает строку на отдельные элементы и передает их в арифметический стек для расчета

# Вычислитель выражений – основные классы

- Собственно вычислитель
  - разбивает строку на отдельные элементы и передает их в арифметический стек для расчета
- Арифметический стек



# Реализация класса "Арифметический стек"

- Стек, хранящий вещественные числа и умеющий делать операции вида "взять два числа с вершины, сложить, положить обратно"

# Реализация класса "Арифметический стек"

- Стек, хранящий вещественные числа и умеющий делать операции вида "взять два числа с вершины, сложить, положить обратно"
- Можно унаследоваться от ArrayDeque или LinkedList

# Реализация класса "Арифметический стек"

- Стек, хранящий вещественные числа и умеющий делать операции вида "взять два числа с вершины, сложить, положить обратно"
- Можно сделать наследника ArrayDeque или LinkedList
- Или использовать приём "композиция" – сделаем Deque закрытым членом класса и открытые функции push, pop, top

# Операция = обозначение + бинарная функция

- Выполняемые операции можно описать в виде перечисления
- К каждому элементу перечисления привязывается некоторая бинарная функция

# Класс «операция» (внутри стека)

```
public enum Operation {
    PLUS,
    MINUS,
    TIMES,
    DIV;

    public BiFunction<Double, Double, Double> getFunction() {
        switch (this) {
            case PLUS: return (x, y) -> x + y;
            case MINUS: return (x, y) -> y - x;
            case TIMES: return (x, y) -> x * y;
            case DIV: return (x, y) -> y / x;
            default: throw new AssertionError("");
        }
    }
}
```

# Преобразование строка – операция

```
public class PolishNotationCalculator {
    static Map<String, Operation> operationMap =
          new HashMap<>();
    static {
        operationMap.put("+", Operation.ADD);
        operationMap.put("-", Operation.SUB);
        operationMap.put("*", Operation.MUL);
        operationMap.put("/", Operation.DIV);
    }
    // ...
}
```

# Класс "арифметический стек"

```
public class ArithmeticStack {
    private final Deque<Double> stack = new LinkedList<>();
    public void push(double x) {
        stack.push(x);
    }

    public double top() {
        return stack.peek();
    }

    public void execute(Operation op) {
        stack.push(op.getFunction().apply(
            stack.pop(), stack.pop()));
    }
}
```

# Функциональный класс "вычислитель польской записи"

```
public class PolishNotationCalculator {
    static public double calc(String expr) {
        final ArithmeticStack stack = new ArithmeticStack();
        for (String arg: expr.split(" ")) {
            Operation op = operationMap.get(arg);
            if (op == null) {
                double x = Double.parseDouble(arg);
                stack.push(x);
            }
            else {
                stack.execute(op);
            }
        }
        return stack.top();
    }
}
```

# Тестирование

```
public class PolishTest {
    @Test
    public void test2() {
        assertEquals(10.0,
        PolishNotationCalculator.calc("6 -4 -"), 1e-10);
    }
    @Test
    public void test3() {
        assertEquals(3.0,
           PolishNotationCalculator.calc("10 1 2 3 * + -"),
           1e-10);
    }
    @Test(expected=IllegalArgumentException.class)
    public void test4() {
        Polish.calc("1 -");
    }
}
```

# Демонстрация

- См. part2.stack

# Коллекции из старых версий JDK

- Vector – примерно повторяет ArrayList
  - Расширение – Stack, реализация стека
- Hashtable – примерно повторяет HashMap
- Данные коллекции поддерживаются, но использование их не рекомендуется

# Итоги

- Рассмотрено
  - Map, Enum, Queue, Deque, Stack, …
- Далее
  - Пример с наследованием
  - Обзор стандартной библиотеки Java