Глава 11

КОЛЛЕКЦИИ И STREAM API

Программирование заставило дерево зацвести.

Алан Дж. Перлис

Общие определения

Коллекции — это хранилища или контейнеры, поддерживающие различные способы накопления и упорядочения объектов с целью обеспечения возможно­стей эффективного доступа к ним. Они представляют собой реализацию аб­страктных структур данных, поддерживающих три основные операции: • добавление нового элемента в коллекцию;

* удаление элемента из коллекции;
* изменение элемента в коллекции.

В качестве других операций могут быть реализованы следующие: заменить, просмотреть элементы, подсчитать их количество и др.

Для работы с коллекциями разработчиками был создан **Collection Framework**. Применение коллекций обусловливается возросшими объемами обрабатываемой информации. Когда счет используемых объектов идет на сотни тысяч или миллио­нов, массивы не обеспечивают ни должной скорости, ни экономии ресурсов.

Примером коллекции является стек (структура LIFO — *Last In First Out*), в ко­тором всегда удаляется объект, вставленный последним. Для очереди (структура FIFO — *First In First Out*) используется другое правило удаления: всегда уда­ляется элемент, вставляемый первым. В абстрактных типах данных существует несколько видов очередей: двусторонние очереди, кольцевые очереди, обобщен­ные очереди, в которых запрещены повторяющиеся элементы. Стеки и очереди могут быть реализованы как на базе массива, так и на базе связного списка.

Коллекции в языке Java объединены в библиотеке классов **java.util** и представ­ляют собой контейнеры для хранения и манипулирования объектами. До появ­ления Java 2 эта библиотека содержала классы только для работы с простейши­ми структурами данных: **Vector**, **Stack**, **Hashtable**, **BitSet**, а также интерфейс **Enumeration** для работы с элементами этих классов. Коллекции, появившиеся в Java 2, представляют собой общую технологию хранения и доступа к объектам. Скорость обработки коллекций повысилась по сравнению с предыдущей версией языка за счет отказа от их потокобезопасности. Поэтому, если объект коллекции может быть доступен из различных потоков, что наиболее естественно для распре­деленных приложений, возможно использование коллекции из Java 1.

В Java 5 в новом пакете **java.util.concurrent** появились ограниченно потоко­безопасные коллекции, гарантирующие более высокую производительность в многопоточной среде для конкурирующих потоков.

Так как в коллекциях при практическом программировании хранится набор ссылок на объекты одного типа, следует обезопасить коллекцию от появления ссылок на другие, не разрешенные логикой приложения типы. Такие ошибки при использовании нетипизированных коллекций выявляются на стадии выпол­нения, что повышает трудозатраты на исправление и верификацию кода. Поэтому, начиная с версии Java SE 5, коллекции стали типизированными или *generic*.

Более удобным стал механизм работы с коллекциями, а именно:

* предварительное сообщение компилятору о типе ссылок, которые будут хра­ниться в коллекции, при этом проверка осуществляется на этапе компиляции;
* отсутствие необходимости постоянно преобразовывать возвращаемые по ссылке объекты (тип **Object**) к требуемому типу.

Структура коллекций характеризует способ, с помощью которого програм­мы Java обрабатывают группы объектов. Так как **Object** — суперкласс для всех классов, то в коллекции можно хранить объекты любого типа, кроме базовых. Коллекции — это динамические массивы, связные списки, деревья, множест­ва, хэш-таблицы, стеки, очереди.

Интерфейсы коллекций:

**Map<K, V>** — карта отображения вида «ключ-значение»;

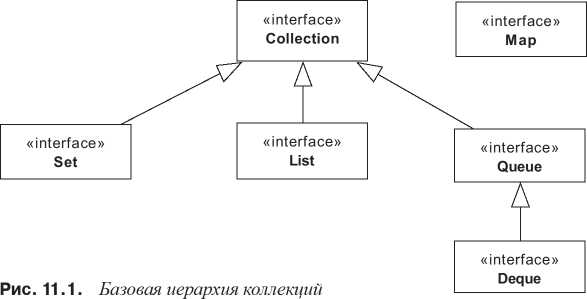
**Collection<E>** — основной интерфейс коллекций, вершина иерархии кол­лекций **List**, **Set**. Также наследует интерфейс **Iterable<E>**;

**List<E>** — специализирует коллекции для обработки упорядоченного набо­ра элементов;

**Set<E>** — множество, содержащее уникальные элементы;

**Queue<E>** — очередь, где элементы добавляются в один конец списка, а из­влекаются из другого конца.

Все классы коллекций реализуют интерфейсы **Serializable**, **Cloneable** (кро­ме **WeakHashMap**).



В интерфейсе **Collection<E>** определены методы, которые работают на всех коллекциях:

**boolean add(E obj)** — добавляет **obj** к вызывающей коллекции и возвращает **true**, если объект добавлен, и **false**, если **obj** уже элемент коллекции;

**boolean remove(Object obj)** — удаляет **obj** из коллекции;

**boolean addAll(Collection<? extends E> c)** — добавляет все элементы кол­лекции к вызывающей коллекции;

**void clear()** — удаляет все элементы из коллекции;

**boolean contains(Object obj)** — возвращает **true**, если вызывающая коллек­ция содержит элемент **obj**;

**boolean equals(Object obj)** — возвращает **true**, если коллекции эквивалентны;

**boolean isEmpty()** — возвращает **true**, если коллекция пуста;

**int size()** — возвращает количество элементов в коллекции;

**Object[] toArray()** — копирует элементы коллекции в массив объектов;

**<T> T[] toArray(T a[])** — копирует элементы коллекции в массив объектов определенного типа.

Появление **Stream API** обусловило возникновение методов для создания потоков объектов и работы с функциональными интерфейсами:

**default Stream<E> stream()** — преобразует коллекцию в *stream* объектов;

**default Stream<E> parallelStream()** — преобразует коллекцию в парал­лельный *stream* объектов. Повышает производительность при работе с очень большими коллекциями на многоядерных процессорах;

**default boolean removeIf(Predicate<? super E> filter)** — удаляет все эле­менты коллекции в зависимости от условия.

Методы **void forEach(Consumer<? super T> action)**, **Iterator<T> iterator()**, **Spliterator<T> spliterator()** унаследованы от интерфейса **Iterable<T>**.

При работе с элементами коллекции применяются интерфейсы: **Iterator<E>**, **ListIterator<E>**, **Map.Entry<K, V>** — для перебора коллекции и доступа к объектам коллекции.

Интерфейс **Iterator<E>** строит объект, обеспечивающий доступ к элемен­там коллекции. К этому типу относится объект, возвращаемый методом **iterator()**. Такой объект позволяет осуществлять навигацию по содержимому коллекции последовательно, элемент за элементом. Позиции итератора услов­но располагаются в коллекции между элементами. В коллекции, состоящей из N элементов, существует N+1 позиций итератора.

Методы интерфейса **Iterator<E>**, представляющего собой одну из реализа­ций дизайн-паттерна с одноименным названием:

**boolean hasNext()** — проверяет наличие следующего элемента, а в случае его отсутствия (завершения коллекции) возвращает **false**. Итератор при этом остается неизменным;

**E next()** — возвращает ссылку на объект, на который указывает итератор, и передвигает текущий указатель на следующий, предоставляя доступ к следующему элементу. Если следующий элемент коллекции отсутствует, то метод **next()** генерирует исключение **NoSuchElementException**;

**void remove()** — удаляет объект, возвращенный последним вызовом метода **next()**. Если метод **next()** до вызова **remove()** не вызывался, то будет сгенериро­вано исключение **IllegalStateException**;

**void forEachRemaining(Consumer<? super E> action)** — выполняет дейст­вие над каждым оставшимся необработанным элементом коллекции.

Интерфейс **Map.Entry** предназначен для извлечения ключей и значений кар­ты с помощью методов **K getKey()** и **V getValue()** соответственно. Вызов метода **V setValue(V value)** заменяет значение, ассоциированное с текущим ключом.

ArrayList

Интерфейс **List<E>** реализует класс **ArrayList<E>** — динамический мас­сив объектных ссылок.

Иерархия наследования следующая:

java.util.AbstractCollection<E>

java.util.AbstractList<E>

java.util.ArrayList<E>

В классе объявлены конструкторы:

ArrayList()

ArrayList(Collection <? **extends** E> c)

ArrayList(**int** capacity)

Практически все методы класса являются реализацией абстрактных мето­дов из суперклассов и интерфейсов или дефолтными методами интерфейсов **Collection**, **List**, **Iterable**. Методы интерфейса **List<E>** позволяют вставлять и удалять элементы из позиций, указываемых через отсчитываемый от нуля индекс:

**E get(int index)** — возвращает элемент в виде объекта из позиции **index**, пред­ставляет собой одно из главных достоинств класса из-за скорости выполнения;

**void add(int index, E element)** — вставляет **element** в позицию, указанную в **index**;

**E remove(int index)** — удаляет объект из позиции **index**;

**E set(int index, E element)** — заменяет объект в позиции **index**, возвращает при этом удаляемый элемент;

**boolean addAll(int index, Collection<? extends E> c)** — вставляет в вызыва­ющий список все элементы коллекции **с**, начиная с позиции **index**;

**int indexOf(Object ob)** — возвращает индекс указанного объекта;

**default void sort(Comparator<? super E> c)** — сортирует список на основе компаратора;

**List<E> subList(int fromIndex, int toIndex)** — извлекает часть коллекции в указанных границах;

**static <E> List<E>copyOf(Collection <? extentds E> coll)** — создает немо­дифицируемый список на основе передаваемой коллекции.

Удаление и добавление элементов для такой коллекции представляет ресур­соемкую задачу, поэтому объект **ArrayList<E>** лучше всего подходит для хра­нения списков с малым числом подобных действий. С другой стороны, навига­ция по списку осуществляется очень быстро, поэтому операции поиска производятся за более короткое время.

/\* # 1 # создание параметризованной коллекции # ListMain.java \*/

**package** by.epam.learn.collection;

**import** java.util.ArrayList;

**import** java.util.List;

**public class** ListMain {

**public static void** main(String[] args) {

List<Integer> list = **new** ArrayList<>();

*// List<int> integers = new ArrayList<>(); // compile error*

list.add(5);

list.add(7);

list.add(42);

list.add(**null**);*// you can add null to the list*

list.add(50);

list.add(77);

System.*out*.println(list);

list.add(5, 87); *// adding to position number 5*

System.*out*.println(list);

list.remove(2); *// removal from position number 2*

System.*out*.println(list);

}

}

В результате будет выведено:

**[5, 7, 42, null, 50, 77]**

**[5, 7, 42, null, 50, 87, 77]**

**[5, 7, null, 50, 87, 77]**

В данной ситуации не создается новый класс для каждого конкретного типа, и сама коллекция не меняется, просто компилятор снабжается информацией о типе элементов, которые могут храниться в **list**. При этом параметром коллек­ции может быть только объектный тип. Попытка добавления\удаления элемен­та с номером, выходящим за пределы текущего размера списка, приведет к исключительной ситуации.

Указывать тип следует при создании ссылки, иначе будет позволено добав­лять объекты всех типов. На этом основан принцип типобезопасности, обеспе­чиваемый параметризацией коллекций. Пусть в системе онлайн-торговли используется понятие **Order**. Заказы могут собираться в списки, например, список заказов за день. Возможна ситуация, когда по каким-либо причинам в список заказов будет добавлен экземпляр товара или просто строка. В этой ситуации даже свести баланс действительно сделанных заказов простым опре­делением размера списка будет невозможно. Придется извлекать каждый объект, определять его тип и проч.

/\* # 2 # некорректная(raw) и корректная коллекции # UncheckMain.java \*/

**package** by.epam.learn.collection;

**import** by.epam.learn.entity.Order;

**import** java.util.ArrayList;

**import** java.util.List;

**public class** UncheckMain {

**public static void** main(String[] args) {

*// raw type*

List raw = List.*of*(**new** Order(231, 12.f),

**new** ArrayList(),

**new** Order(217, 17.f), "Unitas");

*// type casting required*

Order or1 = (Order) raw.get(0);

ArrayList or2 = (ArrayList)raw.get(1);

Order or3 = (Order) raw.get(2);

String or4 = (String) raw.get(3);

raw.forEach(System.*out*::println);

List<Order> orders = List.*of*(**new** Order(231, 12.f),

**new** Order(389, 29.f),

*//* ***new*** *ArrayList(), compile error*

**new** Order(217, 17.f));

orders.forEach(System.*out*::println);

}

}

В результате будет выведено:

**Order{orderId=231, amount=12.0}**

**[]**

**Order{orderId=217, amount=17.0}**

**Unitas**

**Order{orderId=231, amount=12.0}**

**Order{orderId=389, amount=29.0}**

**Order{orderId=217, amount=17.0}**

где класс **Order** представляет собой сущность в следующем виде:

/\* # 3 # класс-сущность используется здесь и далее для наполнения коллекций # Order.java \*/

**package** by.epam.learn.entity;

**public class** Order **extends** Entity {

**private long** orderId;

**private double** amount;

**public** Order() {

}

**public** Order(**long** orderId, **double** amount) { **this**.orderId = orderId;

**this**.amount = amount;

}

**public long** getOrderId() { **return** orderId;

}

**public void** setOrderId(**long** orderId) { **this**.orderId = orderId;

}

**public double** getAmount() { **return** amount;

}

**public void** setAmount(**double** amount) { **this**.amount = amount;

}

**public boolean** equals(Object o) {

**if** (**this** == o) **return true**;

**if** (o == **null** || getClass() != o.getClass()) **return false**;

Order order = (Order) o;

**if** (orderId != order.orderId) **return false**;

**return** Double.*compare*(order.amount, amount) == 0;

}

**public int** hashCode() {

**int** result;

**long** temp;

result = (**int**) (orderld A (orderld >>> 32));

temp = Double.*doubleToLongBits*(amount);

result = 31 \* result + (**int**) (temp A (temp >>> 32));

**return** result;

}

**public** String toString() {

**final** StringBuilder sb = **new** StringBuilder("Order{");

sb.append("orderId=").append(orderId).append(", amount=").append(amount); sb.append('}');

**return** sb.toString();

}

}

/\* # 4 # супер-класс всех сущностей системы # Entity.java \*/

**package** by.epam.learn.entity;

**import** java.io.Serializable;

**public abstract class** Entity **implements** Serializable, Cloneable { }

Чтобы параметризация коллекции была полной, необходимо указывать пара­метр и при объявлении ссылки, и при создании объекта оператором даймонд «<>».

Интерфейс **List<E>** был дополнен в Java 8 несколькими методами: **void sort(Comparator<? super E> c)**, методом **copyOf()**, серией статических пере­груженных методов **of()** с параметрами и без. Метод **of()** выполняет задачу со­здания коллекции, с которой нельзя выполнить операции по добавлению\уда- лению элементов. Этот метод добавляет еще один способ создания немодифицируемой коллекции, в частности, для поддержки принципа инкап­суляции. Метод **copyOf()** создает немодифицируемую коллекцию на основе модифицируемой исходной.

Пусть существует класс с полем в виде списка:

/\* # 5 # класс, агрегирующий список объектов # OrderList.java \*/

**package** by.epam.learn.collection;

**import** by.epam.learn.entity.Order;

**import** java.util.ArrayList;

**import** java.util.List;

**public class** OrderList {

**private** List<Order> orders;

**public** OrderList() {

**this**.orders = **new** ArrayList<Order>();

}

**public** OrderList(List<Order> orders) { **this**.orders = orders;

}

**public** List<Order> getOrders() { **return** orders;

}

}

Метод **getOrders()** дает пользователю доступ к объекту **orders**. Но есть одно «но». Тот, кто получит список, может спокойно добавлять, удалять объек­ты в список, и эти изменения коснутся списка внутри объекта **OrderList**. После этого нет смысла говорить об инкапсуляции — ее просто не существует в этом варианте. Решение довольно простое: метод **getOrders()** должен возвращать неизменяемый список:

**public** List<Order> getOrders() {

**return** List.*copyOf*(orders);

}

или

**public** List<Order> getOrders() {

Order[] array = {};

**return** List.*of*(orders.toArray(array));

}

Список, полученный обеими реализациями метода **getOrders()**, нельзя мо­дифицировать.

Итераторы

Интерфейс **Иега^г<Е>** используется для последовательного доступа к эле­ментам коллекции. Классическое использование итератора выглядит так:

/\* # 6 # классическое итерирование коллекции # IteratorMain.java \*/

**package** by.epam.learn.collection;

**import** by.epam.learn.entity.Order;

**import** java.util.ArrayList;

**import** java.util.Iterator;

**import** java.util.List;

**public class** IteratorMain {

**public static void** main(String[] args) {

List<Order> orders = **new** ArrayList<Order>() {

{

add(**new** Order(231, 12.));

add(**new** Order(389, 29.));

add(**new** Order(747, 32.));

add(**new** Order(517, 18.));

add(**new** Order(414, 17.));

add(**new** Order(777, 10.));

}

};

Iterator<Order> iterator = orders.iterator();

**while** (iterator.hasNext()) {

Order order = iterator.next();

System.*out*.println(order);

}

}

}

Альтернативное применение итератора с применением метода **foreach()** ин­терфейса **Iterable** намного проще:

orders.forEach(System.*out*::println);

Ниже приведен пример поиска заказов с помощью **while**, **if** и итератора, сумма которых превышает заданную, с удалением всех объектов, не удовлетво­ряющих условию, и назначением всем оставшимся заказам скидки.

/\* # 7 # поиск, удаление, изменение элементов списка. Классический способ # \*/

**final int** controlAmount = 20;

**final int** discountPercent = 10;

Iterator<Order> iterator = orders.iterator();

**while** (iterator.hasNext()) {

Order current = iterator.next();

**if** (current.getAmount() < controlAmount) {

iterator.remove();

**continue**;

}

current.setAmount(current.getAmount() \* (100 - discountPercent) / 100.0);

}

System.*out*.println(orders);

где **orders** уже наполненный список заказов.

Методы **removeIf(Predicate<? super T> filter)** и **foreach(Consumer<? super T > action)** позволяют переписать код уже без ветвлений.

/\* # 8 # поиск, удаление, изменение элементов списка. Функциональные интерфейсы # \*/

orders.removeIf(o -> o.getAmount() <= controlAmount);

orders.forEach(o -> o.setAmount(o.getAmount() \* (100 - discountPercent)/100.0)); orders.forEach(System.*out*::println);

Для доступа к элементам списка может также использоваться специализи­рованный интерфейс **ListIterator<E>**, который позволяет получить доступ сразу в необходимую позицию списка вызовом метода **listIterator(int index)** на объекте списка. Интерфейс **ListIterator<E>** расширяет интерфейс **Iterator<E>**, предназначен для обработки списков и их вариаций. Наличие методов **E previous(), int previousIndex()** и **boolean hasPrevious()** обеспечивает обрат­ную навигацию по списку. Метод **int nextIndex()** возвращает номер следующе­го итератора. Метод **void add(E e)** позволяет вставлять элемент в список теку­щей позиции. Вызов метода **void set(E e)** производит замену текущего элемента списка на объект, передаваемый методу в качестве параметра.

Внесение изменений в коллекцию методами коллекции после извлечения ите­ратора гарантирует генерацию исключения **ConcurrentModificationException** из пакета **java.util** при попытке использовать итератор позднее модификации коллекции, а именно:

List<Order> orders = **new** ArrayList<>();

*// list filling*

Iterator<Order> iterator = orders.iterator();

orders.add(**new** Order(12, 12.1f)); *// or orders.remove(0);*

**while** (iterator.hasNext()) { *// generation exception*

System.*out*.println(iterator.next());

}

Параллельная или конкурентная модификация коллекции методами самой коллекции и ее итератором приводит к неразрешимой проблеме и заканчивает­ся генерацией исключения практически всегда. Проблема заключается в том, что итератор извлек N число позиций, а коллекция изменила число своих эк­земпляров, и итератор перестал соответствовать коллекции. Если модифика­цию коллекции и работу с итератором нужно выполнять из различных потоков, то для решения такой задачи используются ограниченно потокобезопасные ре­шения для коллекций из пакета **java.util.concurrent**.

При создании класса, содержащего коллекцию элементов, возможны два способа: агрегация коллекции в качестве поля класса или отношение **HAS-A** и наследование от класса, представляющего коллекцию, или отношение **К-А**.

Последнее встречается на практике значительно реже.

Класс **ОгйегТуре** тогда может выглядеть следующим образом:

/\* # 9 # класс, агрегирующий список, с реализацией интерфейса Iterable # OrderType.java\*/

**package** by.epam.learn.collection;

**import** java.util.ArrayList;

**import** java.util.Iterator;

**import** java.util.List;

**public class** OrderType **implements** Iterable<String> {

**private int** orderId;

**private** List<String> currencyNames = **new** ArrayList<>();*/\* SEK, DKK, NOK, CZK,*

*GBP, EUR, PLN \*/*

**public** OrderType(**int** orderId) {

**this**.orderId = orderId;

}

**public** List<String> getCurrencyNames() {

**return** List.*copyOf*(currencyNames);

}

*// delegated method*

**public boolean** add(String e) {

**return** currencyNames.add(e);

}

@Override

**public** Iterator<String> iterator() {

**return** currencyNames.iterator();

}

}

Отношение **IS-A** записывается еще проще, но теряется имя **currencyNames**. Класс **OrderType** теперь сам является списком.

/\* # 10 # класс, наследующий список # OrderType.java\*/

**package** by.epam.learn.collection;

**import** java.util.ArrayList;

**public class** OrderType **extends** ArrayList<String> {

**private int** orderId;

**public** OrderType(**int** orderId) {

**this**.orderId = orderId;

} }

Stream API

Преобразование коллекции в *stream* для последующей обработки позволяет не только избавиться от циклов, условных переходов и прямого извлечения итератора, но все действия, выполненные в примере #8, можно произвести в одном операторе. Код становится более коротким, но менее интуитивно по­нятным, особенно на первый взгляд.

/\* # 11 # поиск, удаление, изменение элементов списка с помощью stream # \*/

List<Order> orderList = orders.stream()

.filter(o -> o.getAmount() <= controlAmount)

.map(o -> { o.setAmount(o.getAmount() \* (100 - discountPercent) / 100.0);

**return** o;})

.collect(Collectors.*toList*());

Выполнение данного оператора никак не изменит исходный список **orders**, в то время как два предыдущих способа изменяют внутреннее состояние спис­ка **orders**. Отсутствие влияния на исходную коллекцию является еще одной дополнительной возможностью, предоставляемой **Stream API**.

Интерфейс **java.util.stream.Stream<T>** — поток объектов для преобразова­ния коллекций, массивов. В потоке не хранятся элементы операции, он не мо­дифицирует источник, а формирует в ответ на действие новый поток. Операции в потоке не выполняются до момента, пока не потребуется получить конечный результат выполнения.

**Stream** нельзя воспринимать как просто поток ввода\вывода. Этот поток создается на основе коллекции\массива, элементы которой переходят в состоя­ние информационного ожидания действия, переводящего поток в следующее состояние до достижения терминальной цели, после чего поток прекращает свое существование.

Невозможно на одном и том же состоянии объекта *stream* вызвать метод его обработки повторно:

/\* # 12 # ошибочное использование stream # \*/

Stream<String> stream = strings.stream();

stream.findFirst();

stream.filter(String::isBlank).findAny();

В результате будет сгенерировано исключение **IllegalStateException**.

Получить *stream* из коллекции можно методом **stream()** интерфейса **Collection**, а из массива методом **Stream.of(T...values)**.

List<String> strings = List.*of*("as a the d on and".split("\\s+")); strings.stream();

Некоторые *промежуточные* методы преобразования потоков объектов:

**filter(Predicate<? super T> predicate)** *—* выбор элементов из потока на ос­новании работы предиката в новый поток. Отбрасываются все элементы, не удовлетворяющие условию предиката:

/\* # 13 # фильтрация элементов списка # \*/

strings.stream()

.filter(s -> s.length() < 2)

.forEach(s -> System.*out*.print(s + " "));

В результате останутся две строки, чья длина меньше чем 2 символа:

**a d**

**map(Function<? super T, ? extends R> mapper)** *—* изменение всех элементов потока, применяя к каждому элементу функцию **mapper**. Тип параметризации по­тока может изменяться, если типизация **T** и **R** относится к различным классам:

/\* # 14 # изменение всех элементов списка # \*/

strings.stream()

.map(s -> s.length())

.forEach(s -> System.*out*.print(s + " "));

Функция в методе **map()** получит строку и возвратит ее длину:

**2 1 3 1 2 3**

strings.stream()

.map(String::toUpperCase)

.forEach(s -> System.*out*.print(s + " "));

Функция в методе **map()** получит строку и возвратит ее, преобразовав все символы в верхний регистр:

**AS A THE D ON AND**

**flatMap(Function<T, Stream<R>> mapper)** — преобразовывает один объект, как правило составной, в объект более простой структуры, например, массив в строку, список в объект, список списков в один список:

/\* # 15 # преобразование списка # \*/

OrderType type1 = **new** OrderType(771);

type1.add("SEK");

type1.add("DKK");

type1.add("NOK");

type1.add("EUR");

OrderType type2 = **new** OrderType(779);

type2.add("SEK");

type2.add("PLN");

type2.add("CZK");

type2.add("EUR");

List<OrderType> orderTypes = List.*of*(type1, type2);

List<String> currencyList =

orderTypes.stream()

.map(o -> o.getCurrencyNames()) *// ■> Stream<List<String>>*

.flatMap(o -> o.stream()) *// ■> Stream<String>* .collect(Collectors.*toList*());

System.*out*.println(currencyList);

В результате будет получен объединенный список на основе двух списков:

**[SEK, DKK, NOK, EUR, SEK, PLN, CZK, EUR]**

**peek(Consumer<T> consumer)** — возвращает поток, содержащий все эле­менты исходного потока. Используется для просмотра элементов в текущем состоянии потока. Можно использовать для записи логов:

strings.stream()

.map(String::length)

.peek(s -> System.*out*.print(s + "-"))

.map(n -> n + 100)

.forEach(s -> System.*out*.print(s + " "));

Объект в *stream* обрабатывается сразу через все промежуточные и терми­нальные функции:

**2-102 1-101 3-103 1-101 2-102 3-103**

**sorted(Comparator<T> comparator)** и **sort()** — сортировка в новый поток:

/\* # 16 # сортировка списка # \*/

strings.stream()

.sorted()

.forEach(s -> System.*out*.print(s + " "));

Сортировка с использованием собственного компаратора класса **String**:

**a and as d on the**

strings.stream()

.sorted((s1, s2) -> s1.length() - s2.length())

*//.sorted(Comparator.comparingInt(String::length)) // identically*

.forEach(s -> System.*out*.print(s + " "));

Сортировка с использованием заданного компаратора, сортирующего по длине строк:

**a d as on the and**

**limit(long maxSize)** — ограничивает выходящий поток заданным в параме­тре значением;

strings.stream()

.limit(3)

.forEach(s -> System.*out*.print(s + " "));

Выведет в поток только первые три элемента:

**as a the**

**skip(long n)** — не включает в выходной поток первые **n** элементов исходно­го потока;

strings.stream()

.skip(4)

.forEach(s -> System.*out*.print(s + " "));

Пропустит первые четыре элемента:

**on and**

**distinct()** — удаляет из потока все идентичные элементы;

List<String> stringsDouble = List.*of*("the and the and the and".split("\\s+")); stringsDouble.stream()

.distinct()

.forEach(s -> System.*out*.print(s + " "));

Будет выведено:

**the and**

Некоторые *терминальные* методы сведения потока к результату. Результатом может быть новая коллекция, объект некоторого класса, число. Промежуточные операции обязательно должны завершаться терминальными, иначе они не вы­полнятся, так как просто не имеют смысла.

**void forEach(Consumer<T> action)** — выполняет действие над каждым эле­ментом потока. Чтобы результат действия сохранялся, реализация *action* должна предусматривать фиксацию результата в каком-либо объекте или потоке вывода;

**Optional<T> findFirst()** — находит первый элемент в потоке;

String firstStr = strings.stream()

.filter(s -> s.matches("a\\w\*"))

.findFirst()

.orElse("none");

System.*out*.println(firstStr);

Результат: **as**

Фильтр выбрал в потоке все элементы, удовлетворяющие условию предика­та, а метод **findFirst()** нашел первый элемент.

**Optional<T> findAny()** — находит элемент в потоке;

String anyStr = strings.stream()

.filter(s -> s.matches("an[a-z]"))

.findAny()

.orElse("none");

System.*out*.println(anyStr);

Результат: **and**

Фильтр выбрал в потоке все элементы, удовлетворяющие условию предика­та, а метод **findAny()** нашел один из элементов.

**long count()** — возвращает число элементов потока;

**long** count = strings.stream()

.filter(s -> s.matches("a\\w\*"))

.count();

System.*out*.println(count);

Результат: **3**

Фильтр выбрал в потоке все элементы, удовлетворяющие условию предика­та, а метод **count()** вернул их число.

**boolean allMatch(Predicate<T> predicate)** — возвращает истину, если все элементы *stream* удовлетворяют условию предиката;

**boolean** res1 = strings.stream()

.allMatch(s -> s.length() < 5); *// true*

**boolean anyMatch(Predicate<T> predicate)** — возвращает истину, если хотя бы один элемент *stream* удовлетворяет условию предиката;

**boolean** res2 = strings.stream()

.anyMatch(s -> s.startsWith("a")); *// true*

**boolean noneMatch(Predicate<T> predicate)** — возвращает истину, если ни один элемент *stream* не удовлетворяет условию предиката;

**boolean** res3 = strings.stream()

.noneMatch(s -> s.endsWith("z")); *// true*

**Optional<T> reduce(T identity, BinaryOperator<T> accumulator)** — сво­дит все элементы потока к одному результирующему объекту, завернутому в оболочку **Optional**;

**int** sumLength = strings.stream()

.map(s -> s.length())

.reduce(0, (n1, n2) -> n1 + n2);

Результат: **12**

Поток строк преобразуется в поток их длин и метод **reduce()** вычисляет сум­му всех длин строк.

**int** sum = strings.stream() .reduce(0, (identity, v) -> v.length() + identity, Integer::*sum*);

То же самое действие, но без промежуточного преобразования в *stream* це­лых чисел.

**<R, A> R collect(Collector<? super T, A, R> collector)** — собирает элементы в коллекцию или объект другого типа;

Map<String, Integer> map = Arrays.*stream*("as a the d on and".split("\\s+")) .collect(Collectors.*toMap*(s -> s, s -> s.length()));

В результате будет получен объект **map** вида:

**{the=3, a=1, as=2, d=1, and=3, on=2}**

Разбиение строки на *stream* подстрок. Создание **Map**, где ключом является сама подстрока, а значением ее длина.

**Optional<T> min(Comparator<T> comparator**) — находит минимальный элемент;

String min = strings.stream() .min(Comparator.*comparingInt*(s -> s.charAt(s.length() - 1))) .orElse("none");

System.*out*.println(min);

Результат: **a**

Поиск строки с минимальным значением кода его последнего символа.

**Optional<T> max(Comparator<T> comparator**) — находит максимальный элемент.

String max = strings.stream() .max(Comparator.*comparingInt*(Action::*sumCharCode*)) .orElse("empty");

System.*out*.println(max);

Поиск строки с максимальной суммой кодов его символов.

Результат: **the**

где класс **Action** и его метод подсчета суммы кодов — символов строки:

**class** Action **{**

**public static int** sumCharCode(String str) {

**return** str.codePoints().reduce(0, (v1, v2) -> v1 + v2); }

}

Источники Stream API:

**collection.stream(),**

**Arrays.stream(int[] array),**

**Files.walk(Path path),**

**Files.list(Path path),**

**bufferedReader.lines(),**

**Stream.iterate(T seed, UnaryOperator<T> f),**

**Stream.generate(Supplier<? extends T> s),**

**Stream.of(T...t),**

**Stream.ofNullable(T t),**

**Stream.empty(),**

**Stream.concat(Stream<? extends T> a, Stream<? extends T> b), string.lines(), string.codePoints(), string.chars(), random.ints(), random.doubles(), random.longs()** и другие.

Алгоритмы сведения Collectors

Методы класса **java.util.stream.Collectors** представляют основные возможно­сти, позволяя произвести обработку *stream* к результату, будь то число, строка или коллекция. Каждый статический метод класса создает экземпляр **Collector** для пе­редачи методу **collect()** интерфейса **Stream**, который и выполнит действия по пре­образованию *stream* алгоритмом, содержащимся в экземпляре **Collector**.

List<String> strings = List.*of*("as a the d on and".split("\\s+"));

**toCollection(Supplier**<**C> collectionFactory), toList(), toSet()** — преобразо­вание в коллекцию;

*// accumulate lengths of strings into a List*

List<Integer> listLengths = strings.stream()

.map(String::length)

.collect(Collectors.*toList*());

System.*out*.println(listLengths);

Результат: **[2, 1, 3, 1, 2, 3]**

Преобразование в список с применением ссылки на конструктор:

List<Character> listFirstSymb = strings.stream()

.map(s -> s.charAt(0))

.collect(Collectors.*toCollection*(ArrayList::**new**));

System.*out*.println(listFirstSymb);

Результат: **[a, a, t, d, o, a]**

**joining(CharSequence delimiter)** — обеспечивает конкатенацию строк с за­данным разделителем;

*// convert strings to upperCase and concatenate them, separated by colon*

String result = strings.stream()

.map(String::toUpperCase).collect(Collectors.*joining*(" : "));

System.*out*.println(result);

Результат: **AS : A : THE : D : ON : AND**

**mapping(Function<? super T,? extends U> mapper,Collector<? super U,A,R> downstream)** — позволяет преобразовать элементы одного типа в эле­менты другого типа;

*// converting a list of strings to a list of codes of their first characters*

List<Integer> listCode = strings.stream()

.collect(Collectors.*mapping*(s -> (**int**) s.charAt(0), Collectors.*toList*())); System.*out*.println(listCode);

Результат: **[97, 97, 116, 100, 111, 97]**

*// converting a list of strings to a list of codes of their first characters and*

*// finding the maximum value*

**int** max = strings.stream()

.collect(Collectors.*mapping*(s -> (**int**) s.charAt(0),

Collectors.*maxBy*(Integer::compareTo))).orElse(-1);

System.*out*.println(max);

Результат: **116**

**minBy(Comparator<? super T> c)/maxBy(Comparator<? super T> c)** — кол­лектор для нахождения минимального или максимального элемента в потоке;

*// search for the first line with the minimum length*

String minLex = strings.stream()

.collect(Collectors.*minBy*(String::compareTo)).orElse("none");

System.*out*.println(minLex);

Результат: **a**

**filtering(Predicate<? super T> predicate, Collector<? super T,A, R> downstream)** — выполняет фильтрацию элементов;

*// selection of flow elements with a length greater than 1*

List<String> lists = strings.stream()

.collect(Collectors.*filtering*(s -> s.length() > 1, Collectors.*toList*())); System.*out*.println(lists);

Результат: **[as, the, on, and]**

**counting()** — позволяет посчитать количество элементов потока;

*// counting the number of stream elements* **long** counter = strings.stream()

.collect(Collectors.*counting*());

System.*out*.println(counter);

Результат: **6**

**summingInt(ToIntFunction<? super T> mapper)** — выполняет суммирова­ние элементов. Существуют версии для **Long** и **Double**;

*// compute sum of lengths of all strings*

**int** length = strings.stream()

.collect(Collectors.*summingInt*(String::length));

System.*out*.println(length);

Результат: **12**

**averagingInt(ToIntFunction<? super T> mapper)** — вычисляет среднее арифметическое элементов потока. Существуют версии для **Long** и **Double**;

*// average value of stream line lengths*

Double averageLength = strings.stream()

.collect(Collectors.*averagingDouble*(String::length));

System.*out*.println(averageLength);

Результат: **2.0**

**reducing(T identity, BinaryOperator<T> op)** — коллектор, осуществляю­щий редукцию (сведение) элементов на основании заданного бинарного опера­тора;

*// compute sum of code first chars each strings*

**int** sumCodeFirstChars = strings.stream()

.map(s -> (**int**)s.charAt(0))

.collect(Collectors.*reducing*(0, (a, b) -> a + b));

System.*out*.println(sumCodeFirstChars);

Результат: **618**

**reducing(U identity, Function<? super T,? extends U> mapper, BinaryOperator<U> op)** — аналогичное предыдущему действие. В примере ниже производится умножение всех длин строк, но без предварительного пре­образования *stream* к другому типу;

**int** productLength = strings.stream()

.collect(Collectors.*reducing*(1, s -> s.length(), (s1, s2) -> s1 \* s2)); System.*out*.println(productLength);

Результат: **36**

**groupingBy(Function<? super T, ? extends K> classifier)** — коллектор груп­пировки элементов потока;

*// group strings by length*

Map<Integer, List<String>> byLength = strings.stream()

.collect(Collectors.*groupingBy*(String::length));

System.*out*.println(byLength);

Результат: **{1=[a, d], 2=[as, on], 3=[the, and]}**

*// compute sum of common length by string length*

Map<Integer, Integer> sumChars = strings.stream()

.collect(Collectors.*groupingBy*(String::length, Collectors.*summingInt*(String::length)));

System.*out*.println(sumChars);

Результат: **{1=2, 2=4, 3=6}**

**partitioningBy(Predicate<? super T> predicate)** — коллектор разбиения элементов потока;

*// partition strings into length less than 2 and all the rest*

Map<Boolean, List<String>> boolLength = strings.stream()

.collect(Collectors.*partitioningBy*(s -> s.length() < 2));

System.*out*.println(boolLength);

Результат: **{false=[as, the, on, and], true=[a, d]}**

Метасимвол в коллекциях

Метасимволы используются при параметризации коллекций для расшире­ния возможностей самой коллекции и обеспечения ее типобезопасности. Например, если параметр метода предыдущего примера изменить с **List<Order>** на **List<? extends Order>**, то в метод можно будет передавать коллекции, пара­метризованные любым допустимым типом, а именно классом **Order** и любым его подклассом, что невозможно при записи без анонимного символа.

Но в методе нельзя будет добавить к коллекции новый элемент, пусть даже и допустимого типа, так как компилятору неизвестен заранее тип параметриза­ции списка.

List<Order> action(List<? **extends** Order> orders) {

*// orders.add(new Order(231, 12.f)); // compile error* orders.remove(0); *// can be deleted*

}

Объясняется это тем, что список, передаваемый в качестве параметра мето­да, может быть параметризован классом **DiscountOrder**, а в методе будет со­вершаться попытка добавления экземпляра самого класса **Order**, как показано выше, что недопустимо определением параметризации самого списка, переда­ваемого в метод, а именно:

action(**new** ArrayList<DiscountOrder>());

где

**public class** DiscountOrder **extends** Order {

**public** DiscountOrder(**int** orderId, **float** amount) {

**super**(orderId, amount);

}

}

Поэтому добавление к спискам, параметризованным метасимволом с при­менением **extends**, запрещено всегда.

Параметризация **List<? super Order>** утверждает, что параметр метода или возвращаемое значение может получить список типа **Order** или любого из его суперклассов, в то же время разрешает добавлять туда экземпляры класса **Order** и любых его подклассов.

List<? **super** Order> action(List<? **super** Order> orders) {

orders.add(**new** Order(231, 12.f));

**return** orders;

}

В этом случае к списку, возвращенному методом, можно будет добавлять экземпляры класса **Order** и его подклассов.

Класс LinkedList и интерфейс Queue

Коллекция **LinkedList<E>** реализует возможности связанного списка.

java.util.AbstractCollection<E>

java.util.AbstractList<E>

java.util.AbstractSequentialList<E> **java.util.LinkedList<E>**

Реализует, кроме интерфейсов, указанных при описании **ArrayList**, также интерфейсы **Queue<E>** и **Deque<E>**.

Связанный список хранит ссылки на объекты отдельно вместе со ссылками на следующее и предыдущее звенья последовательности, поэтому часто назы­вается двунаправленным списком.

Самый быстрый метод класса **add(E element)**. Главным же достоинством класса является скорость работы метода **remove()** на **Iterator**, после получения его из **LinkedList**. Также очень быстро работает метод **add(E element)** на **ListIterator**.

Операция удаления из начала и конца списка выполняется достаточно быс­тро, в отличие от операций поиска и извлечения.

При тестировании на списке из десяти тысяч элементов **LinkedList** быстрее, чем **ArrayList**, при добавлении в середину списка методом **add()** в 2 раза, а в начало или конец примерно в 40 раз. Вставки и удаления элементов из **LinkedList** происходят за постоянное время, в том числе и с использованием итераторов, в тоже время вставка\удаление элемента в **ArrayList** приводит к сдвигу всех эле­ментов после позиции добавления\удаления, а в случае, если базовый массив хранения переполняется, то еще и сам массив увеличивается в полтора раза с ко­пированием старого массива в новый. Список **ArrayList**, в свою очередь, быст­рее при вызове метода **get(index)** примерно в 50 раз. Происходит это вследствие того, что определение позиции в списке производится за конкретный интервал времени, не зависящий от размера списка, при поиске же индекса в **LinkedList** время поиска пропорционально размеру списка.

Список **LinkedList** занимает больший объем памяти за счет необходимости хранения ссылок на соседние объекты, что следует учитывать при создании списков больших размеров. Список **LinkedList** занимает от 3,5 до 5 раз больше памяти нежели аналогичный список **ArrayList**.

В этом классе объявлены методы, позволяющие манипулировать им как очередью, двунаправленной очередью и т.д. Двунаправленный список, кроме обычного, имеет особый «нисходящий» итератор, позволяющий двигаться от конца списка к началу, и извлекается методом **descendingIterator()**.

Для манипуляций с первым и последним элементами списка в **LinkedList<E>** реализованы методы:

**void addFirst(E ob), void addLast(E ob)** — добавляющие элементы в нача­ло и конец списка;

**E getFirst(), E getLast()** — извлекающие элементы;

**E removeFirst(), E removeLast()** — удаляющие и извлекающие элементы;

**E removeLastOccurrence(E elem), E removeFirstOccurrence(E elem)** — удаляющие и извлекающие элемент, первый или последний раз встречаемый в списке.

Класс **LinkedList<E>** реализует интерфейс **Queue<E>**, что позволяет спи­ску придать свойства очереди. В компьютерных науках очередь — структура данных, в основе которой лежит принцип FIFO (*first in, first out*). Элементы добавляются в конец и вынимаются из начала очереди. Но существует возмож­ность не только добавлять и удалять элементы, также можно просмотреть, что находится в очереди. К тому же методы интерфейса **Queue<E>** по манипуля­ции первым и последним элементами такого списка **E element(), boolean offer(E o), E peek(), E poll(), E remove()** работают немного быстрее, чем соот­ветствующие методы класса **LinkedList<E>**.

Методы интерфейса **Queue<E>**:

**boolean add(E o)** — вставляет элемент в очередь, но если же очередь пол­ностью заполнена, то генерирует исключение **IllegalStateException**;

**boolean offer(E o)** — вставляет элемент в очередь, если возможно;

**E element()** — возвращает, но не удаляет головной элемент очереди;

**E peek()** — возвращает, но не удаляет головной элемент очереди, возвраща­ет **null**, если очередь пуста;

**E poll()** — возвращает и удаляет головной элемент очереди, возвращает **null**, если очередь пуста;

**E remove()** — возвращает и удаляет головной элемент очереди.

Методы **element()** и **remove()** отличаются от методов **peek()** и **poll()** тем, что генерируют исключение **NoSuchElementException**, если очередь пуста.

Queue<Order> queue = **new** LinkedList<>();

Создается очередь простым присваиванием списка **LinkedList** ссылке типа **Queue**.

/\* # 17 # двунаправленный список и очередь # QueueMain.java \*/

**package** by.epam.learn.collection;

**import** java.util.LinkedList;

**import** java.util.Queue;

**public class** QueueMain {

**public static void** main(String[] args) {

Queue<String> queue = **new** LinkedList<>(){

{

**this**.offer("Jeans");

}

};

queue.add("Dress");

queue.offer("Gloves");

queue.offer("T-Shirt");

queue.stream().filter(s -> !s.endsWith("s")).forEach(System.*out*::println);

System.*out*.println(" ");

queue.forEach(System.*out*::println);

} }

В результате исходная очередь не изменится и будет выведено:

**T-Shirt**

**Jeans**

**Dress**

**Gloves**

**T-Shirt**

Если заменить применение *stream* на

queue.removeIf(s -> s.endsWith("s"));

то действие будет произведено с изменением исходной коллекции.

При всей схожести списков **ArrayList** и **LinkedList** существуют серьезные отличия, которые необходимо учитывать при использовании коллекций в кон­кретных задачах. Если необходимо осуществлять быструю навигацию по спи­ску, то следует применять **ArrayList**, так как перебор элементов в **LinkedList** осуществляется на порядок медленнее. С другой стороны, если требуется ча­сто добавлять и удалять элементы из списка, то уже класс **LinkedList** обеспе­чивает значительно более высокую скорость переиндексации. То есть если кол­лекция формируется в начале процесса и в дальнейшем используется только для доступа к информации, то применяется **ArrayList**, если же коллекция под­вергается изменениям на всем протяжении функционирования приложения, то выгоднее **LinkedList**.

Интерфейс **Queue<E>** также реализует класс **PriorityQueue<E>**. Иерархия наследования представлена в виде:

java.util.AbstractCollection<E>

java.util.AbstractQueue<E>

**java.util.PriorityQueue<E>**

В такую очередь элементы добавляются не в порядке «живой очереди», а в порядке, определяемом в классе, экземпляры которого содержатся в очереди. Сам порядок следования задан реализацией интерфейсов **Сотрага(;ог<Е>** или **СотрагаЬ1е<Е>**.

По умолчанию компаратор размещает элементы в очереди в порядке возраста­ния, который можно изменить на убывание, если компаратору задать **reverseOrder()**.

/\* # 18 # создание приоритетной очереди # PriorityMain.java \*/

**package** by.epam.learn.collection;

**import** java.util.Comparator;

**import** java.util.PriorityQueue;

**import** java.util.Queue;

**public class** PriorityMain {

**public static void** main(String[] args) {

Queue<String> prior = **new** PriorityQueue<>(Comparator.*reverseOrder*());

prior.offer("J");

prior.offer("A");

prior.offer("V");

prior.offer("A");

prior.offer("1");

prior.offer("4");

**while**(!prior.isEmpty()) {

System.*out*.println(prior.poll());

}

}

}

Результатом будет:

**V**

**J**

**A**

**A**

**4**

**1**

Если порядок в классе не определен, а именно, не реализован ни один из указан­ных интерфейсов, то генерируется исключительная ситуация **ClassCastException** при добавлении второго элемента в очередь. При добавлении первого элемента компаратор не вызывается, так как сравнивать объект не с кем.

PriorityQueue<Order> orders = **new** PriorityQueue();

orders.add(**new** Order(546, 53.)); *// ok*

orders.add(**new** Order(146, 17.)); *// runtime error*

С другой стороны, класс **String** реализует интерфейс **Comparable<E>**, тог­да будет:

PriorityQueue<String> orders = **new** PriorityQueue<String>();

orders.add(**new** String("Oracle")); *// ok*

orders.add(**new** String("Google")); *// ok*

Если попытаться заменить тип **String** на **StringBuilder** или **StringBuffer**, то создать очередь **PriorityQueue** так просто не удастся, как и в случае добавле­ния объектов класса **Order**. Решением такой задачи будет создание необходи­мого класса-компаратора или класса-оболочки с полем типа **StringBuilder** и реализацией интерфейса **Comparable<T>**.

/\* # 19 # пользовательский класс, объект которого может быть добавлен в очередь PriorityQueue и множество TreeSet # StringBuilderWrapper.java \*/

**package** by.epam.learn.collection;

**public class** StringBuilderWrapper **implements** Comparable<StringBuilder> {

**private** StringBuilder content = **new** StringBuilder();

@Override

**public int** compareTo(StringBuilder o) {

**return** content.toString().compareTo(o.toString());

} }

Предлагаемое решение универсально для любых пользовательских типов. Применимо для создания упорядоченных множеств вида **TreeSet<T>**, которые используют **Comparable** или **Comparator** для сортировки при добавлении эк­земпляра в множество.

Интерфейс Deque и класс ArrayDeque

Интерфейс **Deque** определяет «двунаправленную» очередь и, соответствен­но, методы доступа к первому и последнему элементам двусторонней очереди. Реализацию этого интерфейса можно использовать для моделирования стека. Методы обеспечивают удаление, вставку и обработку элементов. Каждый из этих методов существует в двух формах. Одни методы создают исключитель­ную ситуацию в случае неудачного завершения, другие возвращают какое-ли­бо из значений (**null** или **false** в зависимости от типа операции). Вторая форма добавления элементов в очередь сделана специально для реализаций **Deque**, имеющих ограничение по размеру. В большинстве реализаций операции до­бавления заканчиваются успешно. Методы **addFirst()**, **addLast()** вставляют элементы в начало и в конец очереди соответственно. Метод **add()** унаследован от интерфейса **Queue** и абсолютно аналогичен методу **addLast()** интерфейса **Deque**. Объявить двуконечную очередь на основе связанного списка можно, например, следующим образом:

Deque<String> deque = **new** LinkedList<>();

Класс **ArrayDeque<E>** быстрее, чем **Stack<E>**, если используется как стек, и быстрее, чем **LinkedList<E>**, если используется в качестве очереди.

Простое применение в виде стека и очереди:

/\* # 20 # двуконечная очередь как стек и как очередь # ArrayDequeMain.java \*/

**package** by.epam.learn.collection;

**import** java.util.ArrayDeque;

**import** java.util.Deque;

**public class** ArrayDequeMain {

**public static void** main(String[] args) {

Deque<Integer> stack = **new** ArrayDeque<>();

stack.push(1);

stack.push(2);

stack.push(3);

**while** (!stack.isEmpty()){

System.*out*.println(stack.pop()); *// like as stack*

}

Deque<Integer> queue = **new** ArrayDeque<>();

queue.offer(11);

queue.offer(22);

queue.offer(33);

**while**(!queue.isEmpty()){

System.*out*.println(queue.poll()); *// like as queue*

}

}

}

Будет выведено:

**3**

**2**

**1**

**11**

**22**

**33**

Множества

Интерфейс **Set<E>** объявляет поведение коллекции, не допускающей дуб­лирования элементов. Интерфейс **SortedSet<E>** наследует **Set<E>** и объявляет поведение набора, отсортированного в возрастающем порядке, заранее опреде­ленном для класса. Интерфейс **NavigableSet<E>** существенно облегчает поиск элементов, например, расположенных рядом с заданным.

Класс **HashSet<E>** наследуется от абстрактного суперкласса **AbstractSet<E>** и реализует интерфейс **Set<E>**, используя хэш-таблицу для хранения коллекции.

java.util.AbstractCollection<E>

java.util.AbstractSet<E>

**java.util.HashSet<E>**

Ключ хэш-код используется в качестве индекса хэш-таблицы для доступа к объектам множества, что значительно ускоряет процессы поиска, добавления и извлечения элемента. Скорость указанных процессов становится заметной для коллекций с большим количеством элементов. Множество **HashSet** не яв­ляется сортированным. В таком множестве могут храниться элементы с одина­ковыми хэш-кодами в случае, если эти элементы не эквивалентны при сравне­нии. Для грамотной организации **HashSet** стоит следить, чтобы реализации методов **equals()** и **hashCode()** соответствовали правилам.

Конструкторы класса:

HashSet()

HashSet(Collection <? **extends** E> c)

HashSet(**int** capacity)

HashSet(**int** capacity, **float** loadFactor),

где **capacity** — число ячеек для хранения хэш-кодов.

/\* # 21 # создание множества # HashSetMain.java \*/

**package** by.epam.learn.collection;

**import** java.util.HashSet;

**public class** HashSetMain {

**public static void** main(String[] args) {

HashSet<String> hashSet = **new** HashSet<>();

hashSet.add("8Y");

hashSet.add("2Y");

hashSet.add("2Y");

hashSet.add("8Y");

hashSet.add("6Y");

hashSet.add("5Y");

hashSet.add("Y-");

hashSet.stream()

.peek(System.*out*::print)

.forEach(s -> System.*out*.println(" " + s.hashCode()));

}

}

При выводе видно, что объекты в **HashSet** хранятся в произвольном поряд­ке относительно значений хэш-кодов.

**2Y 1639**

**5Y 1732**

**6Y 1763**

**Y- 2804**

**8Y 1825**

/\* # 22 # использование множества для вывода всех уникальных слов из файла # HashSetWordsMain.java \*/

**package** by.epam.learn.collection;

**import** java.io.File;

**import** java.io.FileNotFoundException;

**import** java.util.HashSet;

**import** java.util.Scanner;

**import** java.util.TreeSet;

**public class** HashSetWordsMain {

**public static void** main(String[] args) {

HashSet<String> words = **new** HashSet<>(100\_000);

Scanner scan;

**try** {

scan = **new** Scanner(**new** File("data\\uladzimir\_arlou.txt"));

scan.useDelimiter("[AA-H]+");

**while** (scan.hasNext()) {

String word = scan.next();

words.add(word.toLowerCase());

}

} **catch** (FileNotFoundException e) {

e.printStackTrace();

}

System.*out*.println(words);

TreeSet<String> treeSet = **new** TreeSet<>(words);

System.*out*.println(treeSet);

}

}

Класс **TreeSet<E>** для хранения объектов использует бинарное (красно­черное) дерево. С этим алгоритмом желательно ознакомиться самостоятельно. Иерархия наследования **TreeSet**:

java.util.AbstractCollection<E>

java.util.AbstractSet<E>

**java.util.TreeSet<E>**

При добавлении объекта в дерево он сразу же размещается в необходимую позицию с учетом сортировки. Сортировка происходит благодаря тому, что класс реализует интерфейс **SortedSet**, где правило сортировки добавляемых элементов определяется в самом классе, сохраняемом в множестве, который в большинстве случаев реализует интерфейс **Comparable**. Обработка опера­ций удаления и вставки объектов происходит несколько медленнее, чем в хэш- множествах, где при любом числе элементов время этих операций постоянно.

Конструкторы класса:

TreeSet()

TreeSet(Collection <? **extends** E> c)

TreeSet(Comparator <? **super** E> c)

TreeSet(SortedSet <E> s)

Класс **TreeSet<E>** содержит методы по извлечению первого и последнего (наименьшего и наибольшего) элементов **E first()** и **E last()**. Методы **subSet(E from, E to)**, **tailSet(E from)** и **headSet(E to)** предназначены для извлечения определенной части множества. Метод **Comparator <? super E> comparator()** возвращает объект **Comparator**, используемый для сортировки объектов мно­жества или **null**, если выполняется обычная сортировка.

/\* # 23 # создание и преобразование множеств и их методы # HashSetMain.java \*/

**package** by.epam.learn.collection;

**import** java.util.Set;

**import** java.util.TreeSet;

**public class** HashSetMain {

**public static void** main(String[] args) {

Set<String> set = Set.*of*("2Y", "8Y", "6Y", "5Y", "Y-");

System.*out*.println(set);

TreeSet<String> treeSet = **new** TreeSet<>(set);

treeSet.add("Y-");

System.*out*.println(treeSet);

System.*out*.println(treeSet.last() + " " + treeSet.first());

} }

В результате может быть выведено:

**[2Y, 5Y, 6Y, Y-, 8Y]**

**[2Y, 5Y, 6Y, 8Y, Y-]**

**Y- 2Y**

Множество инициализируется списком и сортируется сразу же в процессе создания. С помощью итератора элемент может быть найден и удален из мно­жества. Для множества, состоящего из обычных строк, используется лексико­графическая сортировка, задаваемая реализацией интерфейса **Comparable**, поэтому метод **comparator()** возвращает **null**.

EnumSet

Абстрактный класс **EnumSet<E extends Enum<E>>** наследуется от аб­страктного класса **AbstractSet**.

java.util.AbstractCollection<E>

java.util.AbstractSet<E>

**java.util.EnumSet<E>**

Класс специально реализован для работы с типами **enum**. Все элементы та­кой коллекции должны принадлежать единственному типу **enum**, определенно­му явно или неявно. Внутренне множество представимо в виде вектора битов, обычно единственного **long**. Множества нумераторов поддерживают перебор по диапазону из нумераторов. Скорость выполнения операций над таким множест­вом очень высока, даже если в ней участвует большое количество элементов.

Основное назначение множества **EnumSet<E>** — это выделение подмноже­ства из полного набора элементов перечисления. Сами способы создания объекта этого множества указывают на эту особенность. Создать объект этого класса можно только с помощью статических методов. Метод **EnumSet<E> noneOf(Class<E> elemType)** создает пустое множество нумерованных констант с указанным типом элемента, метод **allOf(Class<E> elementType)** создает мно­жество нумерованных констант, содержащее все элементы указанного типа. Метод **of(E first, E... rest)** создает множество, первоначально содержащее ука­занные элементы. С помощью метода **complementOf(EnumSet<E> s)** создает­ся множество, содержащее все элементы, которые отсутствуют в указанном множестве. Метод **range(E from, E to)** создает множество из элементов, содер­жащихся в диапазоне, определенном двумя элементами. При передаче указан­ным методам в качестве параметра **null** будет сгенерирована исключительная ситуация **NullPointerException**.

/\* # 24 # класс-множество # Country.java \*/

**package** by.epam.learn.collection;

**public enum** Country {

*ARMENIA*, *BELARUS*, *INDIA*, *KAZAKHSTAN*, *POLAND*, *UKRAINE* }

/\* # 25 # использование множества enum-типов # EnumSetCountryMain.java \*/

**package** by.epam.learn.collection;

**import** java.util.EnumSet;

**import static** by.epam.learn.collection.Country.\*;

**public class** EnumSetCountryMain {

**public static void** main(String[] args) {

EnumSet<Country> asiaCountries = EnumSet.*of*(*ARMENIA*, *INDIA*, *KAZAKHSTAN*);

String nameCountry = "Belarus";

Country current = Country.*valueOf*(nameCountry.toUpperCase());

**if** (asiaCountries.contains(current)) {

System.*out*.print(current + " is in Asia");

} **else** {

System.*out*.print(current + " is not in Asia");

}

} }

Карты отображений

Карта отображений — это объект, который хранит пару «ключ-значение». Поиск объекта (значения) облегчается по сравнению с множествами за счет того, что его можно найти по уникальному ключу. Уникальность объектов-клю­чей должна обеспечиваться переопределением методов **hashCode()** и **equals()** или реализацией интерфейсов **Comparable**, **Comparator** пользовательским классом. Классы карт отображений:

**AbstractMap<K, V>** — реализует интерфейс **Map<K, V>**, является супер­классом для всех перечисленных карт отображений;

**HashMap<K, V>** — использует хэш-таблицу для работы с ключами;

**TreeMap<K, V>** — использует дерево, где ключи расположены в виде дере­ва поиска в определенном порядке;

**WeakHashMap<K, V>** — позволяет механизму сборки мусора удалять из карты значения по ключу, ссылка на который вышла из области видимости приложения;

**LinkedHashMap<K, V>** — образует дважды связанный список ключей. Этот механизм эффективен, только если превышен коэффициент загруженно­сти карты при работе с кэш-памятью и др.

Для класса **IdentityHashMap<K, V>** хэш-коды объектов-ключей вычис­ляются методом **System.identityHashCode()** по адресу объекта в памяти, в от­личие от обычного значения **hashCode()**, вычисляемого сугубо по содержимо­му самого объекта.

Интерфейсы карт:

**Map<K, V>** — отображает уникальные ключи и значения;

**Map.Entry<K, V> —** описывает пару «ключ-значение»;

**SortedMap<K, V>** — содержит отсортированные ключи и значения;

**NavigableMap<K, V>** — добавляет новые возможности навигации и пои­ска по ключу.

Интерфейс **Map<K, V>** содержит следующие методы:

**V get(Object obj)** — возвращает значение, связанное с ключом **obj**. Если эле­мент с указанным ключом отсутствует в карте, то возвращается значение **null**;

**V put(K key, V value)** — помещает ключ **key** и значение **value** в вызываю­щую карту. При добавлении в карту элемента с существующим ключом, прои­зойдет замена текущего элемента новым. При этом метод возвратит заменяе­мый элемент;

**default V putIfAbsent(K key, V value)** — помещает ключ **key** и значение **value** в вызывающую карту. При добавлении в карту элемента с существую­щим ключом, замена не произойдет;

**default V compute(K key, BiFunction<? super K, ? super V, ? extends V> remappingFunction)** — помещает ключ **key** и вычисляет значение **value** при добавлении в вызывающую карту;

**default V computeIfAbsent(K key, Function<? super K, ? super V> mappingFunction)** — помещает ключ **key** и значение **value** в вызывающую карту, если пары с таким ключом не существует, если ключ существует, то за­мена не производится;

**default V computeIfPresent(K key, BiFunction<? super K, ? super V, ? extends V> remappingFunction)** — заменяет значение **value** в вызывающей карте, если ключ с таким значением существует, если же пары с таким ключом не существует, то вставка пары не производится;

**void putAll(Map <? extends K, ? extends V> m)** — помещает карту **m** в вы­зывающую карту;

**V remove(Object key) —** удаляет пару «ключ-значение» по ключу **key**;

**void clear()** — удаляет все пары из вызываемой карты;

**boolean containsKey(Object key)** — возвращает **true**, если вызывающая карта содержит **key** как ключ;

**boolean containsValue(Object value)** — возвращает **true**, если вызывающая карта содержит **value** как значение;

**Set<K> keySet()** — возвращает множество ключей;

**Set<Map.Entry<K, V>> entrySet()** — возвращает множество, содержащее значения карты в виде пар «ключ-значение»;

**Collection<V> values()** — возвращает коллекцию, содержащую значения карты;

**static <K, V> Map<K, V> copyOf(Map <? extends K,? extends V> map)** — копирует исходную карту в немодифицируемую новую карту;

**static <K, V> Map<K, V> of(*parameters*)** — перегруженный метод для со­здания неизменяемых карт на основе переданных в метод параметров;

**default void forEach(BiConsumer<? super K, ? super V> action)** — выпол­няет действие над каждым элементом **Map**.

В коллекциях, возвращаемых тремя последними методами, можно только удалять элементы, добавлять нельзя. Данное ограничение обуславливается па­раметризацией возвращаемого методами значения.

Интерфейс **Map.Entry<K, V>** представляет пару «ключ-значение» и содер­жит следующие методы:

**K getKey()** — возвращает ключ текущего входа;

**V getValue()** — возвращает значение текущего входа;

**V setValue(V obj)** — устанавливает значение объекта **obj** в текущем входе.

В примере показаны способы создания хэш-карты и доступа к ее элементам.

/\* # 26 # создание hashmap # HashMapMain.java \*/

**package** by.epam.learn.collection;

**import** java.util.HashMap;

**import** java.util.Map;

**public class** HashMapMain {

**public static void** main(String[] args) { Map<String, Integer> map = **new** HashMap<>(); map.put("Jeans", 40); *// adding a pair* map.put("T-Shirt", 35);

map.put("Gloves", 42);

map.compute("Shoes", (k,v) -> 77); *// adding a pair* System.*out*.println(map);

*// replacing value if key exists*

map.computeIfPresent("Shoes", (k,v) -> v + k.length());

System.*out*.println(map);

map.computeIfAbsent("Shoes", v -> 11);

*// adding a pair if the key is missing* map.computeIfAbsent("Shoes\_2", v -> 11); System.*out*.println(map);

}

}

**{Gloves=42, T-Shirt=35, Jeans=40, Shoes=77}**

**{Gloves=42, T-Shirt=35, Jeans=40, Shoes=82}**

**{Shoes\_2=11, Gloves=42, T-Shirt=35, Jeans=40, Shoes=82}**

/\* # 27 # создание хэш-карты и замена элемента по ключу # MapEntryMain.java \*/

**package** by.epam.learn.collection;

**import** java.util.\*;

**public class** MapEntryMain {

**public static void** main(String[] args) {

HashMap<String, Integer> hashMap = **new** HashMap<String, Integer>();

hashMap.put("Пряник", 5);

hashMap.put("Кефир", 1);

hashMap.put('^e6", 1);

hashMap.putIfAbsent('^e6", 2); *// replacement will not happen* hashMap.putIfAbsent("Молоко", 5);

hashMap.computeIfAbsent("CbipoK", v -> 3); *// adding a pair* hashMap.computeIfPresent("CbipoK", (k, v) -> 4); *// replacement a value* hashMap.computeIfAbsent("CbipoK", v -> 144);*// replacement will not happen* System.*out*.println(hashMap);

hashMap.put("Пряник", 4); *// replacement or addition in the absence of a key* System.*out*.println(hashMap + "after replacing the element");

System. *out* .println(hashMap.get('^e6") + " - found by key 'Хлеб'");

Set<Map.Entry<String, Integer>> entrySet = hashMap.entrySet();

System.*out*.println(entrySet);

entrySet.stream()

.forEach(e -> System.*out*.println(e.getKey() + " : " + e.getValue()));

Set<Integer> values = **new** HashSet<Integer>(hashMap.values());

System.*out*.println(values);

}

}

**{Хлеб=1, Сырок=4, Пряник=5, Кефир=1, Молоко=5}**

**{Хлеб=1, Сырок=4, Пряник=4, Кефир=1, Молоко=5}аЙ«г replacing the element**

**1 - found by key 'Хлеб'**

**[Хлеб=1, Сырок=4, Пряник=4, Кефир=1, Молоко=5]**

**Хлеб : 1**

**Сырок : 4**

**Пряник : 4**

**Кефир : 1**

**Молоко : 5 [1, 4, 5]**

Метод **put(K key, V value)** не проверяет наличия пары с таким же ключом, как и у добавляемой пары, а просто заменяет значение по ключу на новое. Если критично наличие всех ранее добавленных элементов, следует использовать метод **putIfAbsent(K key, V value)**, который может добавлять пару ключ-зна­чение только в том случае, если в карте не содержится ключа со значением, идентичным ключу в параметре метода, или же перед добавлением пары вы­полнять проверку на наличие идентичных ключей вида:

**if**(!hashMap.containsKey("Пряник") ) {

*// replacement will not happen if the key exists*

hashMap.put(«Пряник», 4);

}

Класс **EnumMap<K extends Enum<K>, V>** в качестве ключа может прини­мать только объекты, принадлежащие одному типу **enum**, который должен быть определен при создании коллекции. Специально организован для обеспе­чения максимальной скорости доступа к элементам коллекции.

/\* # 28 # создание EnumMap # EnumMapCountryMain.java \*/

**package** by.epam.learn.collection;

**import** java.util.EnumMap;

**public class** EnumMapCountryMain {

**public static void** main(String[] args) {

EnumMap<Country, Integer> map = **new** EnumMap<Country, Integer>(Country.**class**);

map.put(Country.*POLAND*, 8);

map.put(Country.*UKRAINE*, 1);

map.put(Country.*BELARUS*, 0);

map.forEach((k, v) -> System.*out*.println(k + " " + v));

}

}

**BELARUS 0**

**POLAND 8**

**UKRAINE 1**

Класс **WeakHashMap<K, V>** хорошо работает в ситуациях, когда в процес­се работы с коллекцией некоторые объекты должны из нее гарантированно удаляться, но моменты необходимости удаления и само удаление пары из кол­лекции могут отстоять друг от друга по времени.

Пусть существует некоторый набор заказов. Заказ может находиться в сос­тоянии «обработан/необработан». Исходно коллекция содержит необработанные заказы. Как только заказ изменяет статус на «обработан», его следует удалить из коллекции. Разрешать всем пользователям коллекции удалять заказы, то есть модифицировать коллекцию не очень разумно по причинам безопасности. Коллекция **WeakHashMap** сама будет заботиться об удалении неактуальных объектов, утративших ссылку на внешний ключ.

/\* # 29 # карта со слабыми ссылками # CurrentOrders.java \*/

**package** by.epam.learn.collection;

**import** by.epam.learn.entity.Key;

**import** by.epam.learn.entity.Order;

**import** java.util.WeakHashMap;

**public class** CurrentOrders {

**private** WeakHashMap<Key, Order> orders = **new** WeakHashMap<>();

**public** Order put(Key key, Order value) {

**return** orders.put(key, value);

}

**public** Order get(Object key) {

**return** orders.get(key);

}

**public int** size() {

**return** orders.size();

}

}

где класс **Key** содержит информацию об уникальном номере заказа и текущем статусе заказа.

/\* # 30 # ключ для карты # Key.java \*/

**package** by.epam.learn.entity;

**public class** Key {

**private int** keyUnique;

**private boolean** isProcessed;

**public** Key(**int** keyUnique) {

**this**.keyUnique = keyUnique;

}

**public boolean** isProcessed() {

**return** isProcessed;

}

**public void** setProcessed(**boolean** processed) { isProcessed = processed;

}

}

Чтобы сохранять ссылки на ключи, для них будет создан список. После это­го значение **isProcessed** у ключа будет изменено, он будет удален из списка ключей и, как следствие, ключ из **WeakHashMap** утратит ссылку и станет целью для удаления «сборщиком мусора».

/\* # 31 # демонстрация автоматического удаления из карты слабых ссылок # CurrentOrdersMain.java \*/

**package** by.epam.learn.collection;

**import** by.epam.learn.entity.Key;

**import** by.epam.learn.entity.Order;

**import** java.util.ArrayList;

**import** java.util.Iterator;

**import** java.util.List;

**public class** CurrentOrdersMain {

**public static void** main(String[] args) **throws** InterruptedException {

CurrentOrders orders = **new** CurrentOrders();

List<Key> keys = **new** ArrayList<>();

keys.add(**new** Key(100));

keys.add(**new** Key(220));

keys.add(**new** Key(770));

orders.put(keys.get(0), **new** Order(77, 10d));

orders.put(keys.get(1), **new** Order(65, 54d));

orders.put(keys.get(2), **new** Order(41, 93d));

keys.get(1).setProcessed(**true**);

**final int** size = keys.size();

Iterator<Key> iterator = keys.iterator();

**while** (iterator.hasNext()) {

Key ordersKey = iterator.next();

**if** (ordersKey.isProcessed()) {

iterator.remove();

}

}

System.*out*.println(orders.size());

System.*gc*();

Thread.*sleep*(1\_000);

System.*out*.println(orders.size());

}

}

Будет выведено:

**3**

**2**

Удаление происходит не мгновенно, а только когда сработает «сборщик му­сора». И после этого число заказов уменьшится с **3** до **2**.

Унаследованные коллекции

В ряде распределенных приложений, например, с использованием сервле­тов, до сих пор применяются коллекции более медленные в обработке, но при этом потокобезопасные (*thread-safety*), существовавшие в языке Java с момента его создания, а именно карта **Hashtable<K, V>**, список **Vector<E>** и итератор **java.util.Enumeration<E>**. Все они также были параметризованы и могут быть обработаны с помощью *stream*, но сохраняют безопасность от одновременного доступа из конкурирующих потоков.

Класс **Hashtable<K, V>** реализует интерфейс **Map**, обладает также несколь­кими специфичными, по сравнению с другими коллекциями, методами:

**Enumeration<V> elements()** — возвращает итератор для значений карты;

**Enumeration<K> keys()** — возвращает итератор для ключей карты.

/\* # 32 # создание и обработка legacy хэш-таблицы # HashtableMain.java \*/

**package** by.epam.learn.collection;

**import** java.util.Enumeration;

**import** java.util.Hashtable;

**public class** HashtableMain {

**public static void** main(String[] args) {

Hashtable<String, Integer> table = **new** Hashtable<>();

table.put("Jeans", 40); *// adding a pair*

table.put("T-Shirt", 35);

table.put("Gloves", 42);

table.compute("Shoes", (k,v) -> 77); *// adding a pair*

System.*out*.println(table);

Enumeration<String> keys = table.keys();

**while** (keys.hasMoreElements()) {

System.*out*.println(keys.nextElement());

}

Enumeration<Integer> values = table.elements();

**while** (values.hasMoreElements()) {

System.*out*.println(values.nextElement());

}

} }

В результате в консоль будет выведено:

**{Jeans=40, Gloves=42, Shoes=77, T-Shirt=35}**

**Jeans**

**Gloves**

**Shoes**

**T-Shirt**

**40**

**42**

**77**

**35**

Принципы работы с коллекциями, в отличие от их структуры, со сменой версий языка существенно не изменились. Внутренняя организация **HashMap** в Java 8 претерпела важные изменения, необходимые к самостоятельному изучению.

/\* # 33 # создание и обработка legacy списка # VectorMain.java \*/

**package** by.epam.learn.collection;

**import** java.util.Enumeration;

**import** java.util.Vector;

**public class** VectorMain {

**public static void** main(String[] args) {

Vector<String> vector = **new** Vector<>(777);

vector.add("java");

vector.add("epam");

vector.add(1, **null**);

vector.addAll(vector);

System.*out*.println(vector);

vector.removeIf(e -> e==**null**);

vector.replaceAll(String::toUpperCase);

System.*out*.println(vector);

**long** size = vector.stream().count();

System.*out*.println(size);

Enumeration<String> enumeration = vector.elements();

**while**(enumeration.hasMoreElements()) {

System.*out*.printf("%s ", enumeration.nextElement());

}

}

}

**[java, null, epam, java, null, epam]**

**[JAVA, EPAM, JAVA, EPAM]**

**4**

**JAVA EPAM JAVA EPAM**

Класс **Properties** предназначен для хранения карты свойств, где и имена, и значения являются экземплярами класса **String**. Объект этого класса исполь­зуется многими технологиями для хранения и передачи свойств конфигурации и пр. Значения пары можно загружать из файла и сохранять в файле.

/\* # 34 # создание экземпляра и файла properties # PropertiesStoreMain.java \*/

**package** by.epam.learn.collection;

**import** java.io.FileWriter;

**import** java.io.IOException;

**import** java.util.Properties;

**public class** PropertiesStoreMain {

**public static void** main(String[] args) {

Properties props = **new** Properties();

**try** {

props.setProperty("db.driver", "com.mysql.cj.jdbc.Driver");

*//props.setProperty("db.url", "jdbc:mysql://127.0.0.1:3306/testphones");*

props.setProperty("user", "root");

props.setProperty("password", "pass");

props.setProperty("poolsize", "5");

props.store(**new** FileWriter("datares/base.properties"), "No Comment’s");

} **catch** (IOException e) { e.printStackTrace();

}

} }

Директория **datares** должна существовать. Объект **FileWriter** не создает промежуточные директории. В результате в файле **base.properties** будет разме­щена следующая информация:

***#No Comment’s***

***#Wed July 15 18:32:32 MSK 2020***

**db.driver=com.mysql.cj.jdbc.Driver**

**user=root**

**password=pass**

**poolsize=5**

Символ «=» служит по умолчанию разделителем ключа и значения в файле **properties**, также в этом качестве можно использовать символ «:». Эти два спе­циальных символа при записи в файл в качестве части ключа или значения получают впереди символ «\», чтобы в дальнейшем при чтении не быть вос­принятым как разделитель. Тогда для вызова

props.setProperty("db.url", "jdbc:mysql://127.0.0.1:3306/testphones");

в файл будет записано:

**db.url=jdbc\:mysql\://127.0.0.1\:3306/testphones**

При чтении символ «\» будет проигнорирован.

Извлечь информацию из файла достаточно просто.

/\* # 35 # загрузка файла properties в экземпляр # PropertiesLoadDemo.java \*/

**package** by.epam.learn.collection;

**import** java.io.FileReader;

**import** java.io.IOException;

**import** java.util.Properties;

**public class** PropertiesLoadDemo {

**public static void** main(String[] args) {

Properties props = **new** Properties();

**try** {

props.load(**new** FileReader("datares\\base.properties"));

} **catch** (IOException e) {

e.printStackTrace();

}

String dbUrl = props.getProperty("db.url");

*// following two names are missing in the file*

String maxIdle = props.getProperty("maxIdle"); *// maxIdle = null*

*// value "20" will be assigned to the key if it is not found in the file*

String maxActive = props.getProperty("maxActive", "20");

System.*out*.println("dbUrl: " + dbUrl);

System.*out*.println("maxIdle: " + maxIdle );

System.*out*.println("maxActive: " + maxActive);

}

}

В результате будет выведено:

**dbUrl: jdbc:mysql://127.0.0.1:3306/testphones**

**maxIdle: null**

**maxActive: 20**

В веб-приложении поток ввода для чтения файла **properties** создается сле­дующим образом:

**this**.getClass().getClassLoader().getResourceAsStream("datares//base.properties")

Алгоритмы класса Collections

Класс **java.util.Collections** содержит большое количество статических мето­дов, предназначенных для манипулирования коллекциями.

**<T> void copy(List<? super T> dest, List<? extends T> src)** — копирует все элементы из одного списка в другой;

**boolean disjoint(Collection<?> c1, Collection<?> c2)** — возвращает **true**, если коллекции не содержат одинаковых элементов;

**<T> List <T> emptyList(), <K, V> Map <K, V> emptyMap(), <T> Set <T> emptySet()** — возвращают пустой список, карту отображения и множество со­ответственно;

**<T> void fill(List<? super T> list, T obj)** — заполняет список заданным эле­ментом;

**int frequency(Collection<?> c, Object o)** — возвращает количество вхожде­ний в коллекцию заданного элемента;

**<T> boolean replaceAll(List<T> list, T oldVal, T newVal)** — заменяет все заданные элементы новыми;

**void reverse(List<?> list)** — «переворачивает» список;

**void rotate(List<?> list, int distance)** — сдвигает список циклически на за­данное число элементов;

**void shuffle(List<?> list)** — перетасовывает элементы списка;

**singleton(T o), singletonList(T o), singletonMap(K key, V value)** — создают множество, список и карту отображения, позволяющие добавлять только один элемент;

**<T extends Comparable<? super T>> void sort(List<T> list)**,

**<T> void sort(List<T> list, Comparator<? super T> c)** — сортировка списка естественным порядком и с использованием **Comparable** или **Comparator** со­ответственно;

**void swap(List<?> list, int i, int j)** — меняет местами элементы списка, стоя­щие на заданных позициях;

**<T> List<T> unmodifiableList(List<? extends T> list)** — возвращает ссыл­ку на список с запрещением его модификации. Аналогичные методы есть для всех коллекций.

/\* # 36 # применение некоторых алгоритмов # AlgorithmMain.java \*/

**package** by.epam.learn.collection;

**import** java.util.ArrayList;

**import** java.util.Collections;

**import** java.util.List;

**public class** AlgoritmMain {

**public static void** main(String[] args) {

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

Collections.*addAll*(list, 1, 2, 3, 4, 5);

Collections.*shuffle*(list);

System.*out*.println(list);

Collections.*sort*(list);

System.*out*.println(list);

Collections.*reverse*(list);

System.*out*.println(list);

Collections.*rotate*(list, 3);

System.*out*.println(list);

System.*out*.println("min: " + Collections.*min*(list));

System.*out*.println("max: " + Collections.*max*(list));

List<Integer> singletonList = Collections.*singletonList*(777);

System.*out*.println(singletonList);

*//singletonList.add(21); // runtime error*

}

}

В результате будет выведено:

**[5, 3, 4, 1, 2]**

**[1, 2, 3, 4, 5]**

**[5, 4, 3, 2, 1]**

**[3, 2, 1, 5, 4]**

**min: 1**

**max: 5**

**[777]**

Вопросы к главе 11

1. Назвать основные интерфейсы коллекций. Какие бывают коллекции?
2. В чем особенности разных видов коллекций? Когда и какие коллекции сле­дует применять?
3. Сравнить **ArrayList** и **LinkedList**.
4. Сравнить **HashMap** и **Hashtable**.
5. Как устроены **HashSet**, **TreeMap**, **TreeSet**.
6. Принцип работы и реализации **HashMap**. Изменения **HashMap** в java 8.
7. Чем отличается **ArrayList** от **Vector**?
8. Особенности интерфейса **Set**.
9. Как добавляются объекты в **HashSet**?
10. Какими способами можно отсортировать коллекцию? Привести три способа.
11. Как правильно удалить элемент из коллекции при итерации в цикле?
12. Как правильно удалить элемент из **ArrayList** (или другой коллекции) при поиске этого элемента в цикле?
13. Коллекции из пакета **java.util.concurrent**. Их особенности.
14. Что происходит при добавлении в **ArrayList** нового элемента и как это реа­лизовано?
15. Метод для преобразования потоконебезопасной коллекции в потокобезопасную.
16. Написать метод, в котором проверяется **HashMap** на наличие в нем некото­рого значения, и его извлечения, если такого значения нет, надо добавить значение с пустой строкой и ее вернуть. Написать код, чтобы он был как можно более эффективным.
17. Какие коллекции более «быстрые» — *legacy* (**Vector**, **Hashtable**) или из па­кета **java.util.concurrent**?
18. Если в коллекцию часто добавлять элементы и удалять, какую лучше ис­пользовать? Почему? Как они устроены?
19. Как быстро получить копию коллекции? Записать код преобразования.
20. Чем **Stream** отличается от коллекции?
21. Промежуточные и терминальные операции в *stream*.
22. Методы: **map()** *vs* **flatMap()** в *stream*.
23. Что такое потоковая обработка данных?

Задания к главе 11

Вариант A

1. Ввести строки из файла, записать в список. Вывести строки в файл в обрат­ном порядке.
2. Ввести число, занести его цифры в стек. Вывести число, у которого цифры идут в обратном порядке.
3. Создать список из элементов каталога и его подкаталогов.
4. Занести стихотворения одного автора в список. Провести сортировку по возрастанию длин строк.
5. Задать два стека, поменять информацию местами.
6. Определить множество на основе множества целых чисел. Создать методы для определения пересечения и объединения множеств.
7. Списки, стеки или очереди T(1..n) и U(1..n) содержат результаты n-измере­ний тока и напряжения на неизвестном сопротивлении R. Найти прибли­женное число R методом наименьших квадратов.
8. С использованием множества выполнить попарное суммирование произ­вольного конечного ряда чисел по следующим правилам: на первом этапе суммируются попарно рядом стоящие числа, на втором этапе суммируются результаты первого этапа и т.д. до тех пор, пока не останется одно число.
9. Сложить два многочлена заданной степени, если коэффициенты многочле­нов хранятся в объекте **HashMap**.
10. Умножить два многочлена заданной степени, если коэффициенты многоч­ленов хранятся в различных списках.
11. Не используя вспомогательных объектов, переставить отрицательные эле­менты данного списка в конец, а положительные — в начало списка.
12. Ввести строки из файла, записать в список **ArrayList**. Выполнить сорти­ровку строк, используя метод **sort()** из класса **Collections**.
13. Задана строка, состоящая из символов «(», «)», «[», «]», «{», «}». Проверить правильность расстановки скобок. Использовать стек.
14. Задан файл с текстом на английском языке. Выделить все различные слова. Слова, отличающиеся только регистром букв, считать одинаковыми. Использовать класс **HashSet**.
15. Задан файл с текстом на английском языке. Выделить все различные слова. Для каждого слова подсчитать частоту его встречаемости. Слова, отличаю­щиеся регистром букв, считать различными. Использовать класс **HashMap**.
16. Заполнить **HashMap** 10 объектами <**Integer**, **String**>. Найти строки у кото­рых *ключ*>5. Если *ключ* = 0, вывести строки через запятую. Перемножить все ключи, где длина строки>5.
17. Написать функцию, которая получала бы итераторы на начало и конец от­сортированного **List** и заданный символ. Возвращать функция должна на­чало и конец диапазона, строки в котором начинаются с заданного символа.

Вариант B

1. В кругу стоят N человек, пронумерованных от 1 до N. При ведении счета по кругу вычеркивается каждый второй человек, пока не останется один. Составить две программы, моделирующие процесс. Одна из программ должна использовать класс **ArrayList**, а вторая — **LinkedList**. Какая из двух программ работает быстрее? Почему?
2. Задан список целых чисел и некоторое число X. Не используя вспомога­тельных объектов и методов сортировки и не изменяя размера списка, пе­реставить элементы списка так, чтобы сначала шли числа, не превосходя­щие X, а затем числа, больше X.
3. Написать программу, осуществляющую сжатие английского текста. Построить для каждого слова в тексте оптимальный префиксный код по алгоритму Хаффмена. Использовать класс **PriorityQueue**.
4. Реализовать класс **Graph**, представляющий собой неориентированный граф. В конструкторе класса передается количество вершин в графе. Методы должны поддерживать быстрое добавление и удаление ребер.
5. На базе коллекций реализовать структуру хранения чисел с поддержкой следующих операций:

* добавление/удаление числа;
* поиск числа, наиболее близкого к заданному (т.е. модуль разницы мини­мален).

1. Реализовать класс, моделирующий работу N-местной автостоянки. Машина подъезжает к определенному месту и едет вправо, пока не встретится сво­бодное место. Класс должен поддерживать методы, обслуживающие приезд и отъезд машины.
2. Во входном файле хранятся две разреженные матрицы — А и В. Построить циклически связанные списки СА и СВ, содержащие ненулевые элементы соответственно матриц А и В. Просматривая списки, вычислить: а) сумму S = A + B; б) произведение P = A х B.
3. Во входном файле хранятся наименования некоторых объектов. Построить список Z, элементы которого содержат наименования и шифры данных объектов, причем элементы списка должны быть упорядочены по возраста­нию шифров. Затем «сжать» список Z, удаляя дублирующие наименования объектов.
4. Во входном файле расположены два набора положительных чисел; между наборами стоит отрицательное число. Построить два списка C1 и С2, эле­менты которых содержат соответственно числа 1-го и 2-го набора таким образом, чтобы внутри одного списка числа были упорядочены по возра­станию. Затем объединить списки C1 и С2 в один упорядоченный список, изменяя только значения полей ссылочного типа.
5. Во входном файле хранится информация о системе главных автодорог, свя­зывающих г. Полоцк с другими городами Беларуси. Используя эту инфор­мацию, построить дерево, отображающее систему дорог республики, а за­тем, продвигаясь по дереву, определить минимальный по длине путь из г. Полоцка в другой заданный город. Предусмотреть возможность сохране­ния дерева в виртуальной памяти.
6. Один из способов шифрования данных, называемый «двойным шифрова­нием», заключается в том, что исходные данные при помощи некоторого преобразования последовательно шифруются на некоторые два ключа — K1 и K2. Разработать и реализовать эффективный алгоритм, позволяющий находить ключи K1 и K2 по исходной строке и ее зашифрованному вариан­ту. Проверить, оказался ли разработанный способ действительно эффек­тивным, протестировав программу для случая, когда оба ключа являются 20-битными (время ее работы не должно превосходить одной минуты).
7. На плоскости задано *N* точек. Вывести в файл описания всех прямых, кото­рые проходят более чем через одну точку из заданных. Для каждой прямой указать, через сколько точек она проходит. Использовать класс **HashMap**.
8. На клетчатой бумаге нарисован круг. Вывести в файл описания всех клеток, целиком лежащих внутри круга, в порядке возрастания расстояния от клет­ки до центра круга. Использовать класс **PriorityQueue**.
9. На плоскости задано *N* отрезков. Найти точку пересечения двух отрезков, имеющую минимальную абсциссу. Использовать класс **TreeMap**.
10. На клетчатом листе бумаги закрашена часть клеток. Выделить все различ­ные фигуры, которые образовались при этом. Фигурой считается набор за­крашенных клеток, достижимых друг из друга при движении в четырех направлениях. Две фигуры являются различными, если их нельзя совме­стить поворотом на угол, кратный 90 градусам, и параллельным перено­сом. Используйте класс **HashSet**.
11. Дана матрица из целых чисел. Найти в ней прямоугольную подматрицу, состоящую из максимального количества одинаковых элементов. Исполь­зовать класс **ArrayDeque**.
12. Реализовать структуру «черный ящик», хранящую множество чисел и имею­щую внутренний счетчик K, изначально равный нулю. Структура должна поддерживать операции добавления числа в множество и возвращение *K*-го по минимальности числа из множества.
13. На прямой гоночной трассе стоит *N* автомобилей, для каждого из которых известны начальное положение и скорость. Определить, сколько произой­дет обгонов.
14. На прямой гоночной трассе стоит *N* автомобилей, для каждого из которых известны начальное положение и скорость. Вывести первые *K* обгонов.

Тестовые задания к главе 11

Вопрос 11.1.

Дан класс:

**public record** Order(**long** orderId, **double** amount) {

}

и фрагмент кода:

List<Order> orders = List.*of*(**new** Order(1, 50), **new** Order(5, 70), **new** Order(7, 50)); orders.stream()

.collect(Collectors.*groupingBy*(Order::amount))

.forEach((source, r) -> System.*out*.print(source + " "));

Что будет результатом? (выбрать один)

1. 70.0 50.0
2. 50.0 70.0 50.0
3. 1, 50.0 5, 70.0 7, 50.0
4. 5, 70.0 7, 50.0
5. ничего не будет выведено

Вопрос 11.2.

Дан класс:

**class** Order {

**long** orderId;

**double** amount;

**public** Order(**long** orderId, **double** amount) {

**this**.orderId = orderId;

**this**.amount = amount;

}

**public** String toString() {

**return** orderId + ", " + amount ;

}

}

и фрагмент кода:

List<Order> orders = *Arrays.asList*(**new** Order(1, 50), **new** Order(5, 70),

**new** Order(7, 70));

Order order = orders.stream()

.reduce(**new** Order(4, 0), (p1, p2) ->

**new** Order(p1.orderId, p1.amount += p2.amount));

System.*out*.print(order);

Что будет результатом? (выбрать один)

1. 4, 0.0
2. 4, 190.0
3. 17, 0.0
4. 17, 190.0

Вопрос 11.3.

Дан класс:

**class** Order {

**long** orderId;

**double** amount;

**public** Order(**long** orderId, **double** amount) { **this**.orderId = orderId;

**this**.amount = amount;

}

**public** String toString() {

**return** orderId + ", " + amount ;

}

}

и фрагмент кода:

List<Order> orders = *Arrays.asList*(**new** Order(1, 50), **new** Order(5, 70), **new** Order(7, 70));

orders.stream()

.reduce((p1, p2) -> p1.amount > p2.amount ? p1 : p2) .ifPresent(System.*out*::println);

Что будет результатом? (выбрать один)

1. ничего не будет выведено
2. 5, 70.0
3. 1, 50.0
4. 7, 70.0

Вопрос 11.4.

Дан класс:

**public record** Item(String name, **double** price) {

}

и фрагмент кода:

List<Item> items = List.*of*(**new** Item("Jeans", 20), **new** Item("Socks", 10), **new** Item("Jacket", 30));

**var** value = items.stream()

.filter(s -> s.name().endsWith("s"))

.mapToDouble(Item::price)

.average()

.getAsDouble();

System.*out*.print(value);

Что будет результатом? (выбрать один)

1. compilation fails
2. 0.0
3. 15.0
4. 20.0

Вопрос 11.5.

Дан фрагмент кода:

Map<String, Integer> map = **new** HashMap<>();

map.compute("y", (k, v) -> v==**null** ? 1 : 0);

map.compute("z", (k, v) -> v==**null** ? 2 : 0);

map.computeIfPresent("z", (k, v) -> v!=**null** ? 3 : 0);

map.computeIfAbsent("y", v -> v!=**null** ? 4 : 0);

System.*out*.println(map.values());

Что будет результатом? (выбрать один)

1. [4, 3]
2. [1, 3]
3. [2, 4]
4. [2, 4]
5. runtime error

Вопрос 11.6.

Дан фрагмент кода:

List<Integer> integers = List.*of*(1, 2, 3, 1, 7);

**boolean** res = integers.stream()

.*// line 1*

System.*out*.print(res);

Какой фрагмент кода следует вставить вместо комментария *line 1*, чтобы в консоль было выведено значение **true**? (выбрать один)

1. noneMatch(i -> i == 2);
2. allMatch(i -> i == 2);
3. anyMatch(i -> i == 2);

findFirst().