БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
МЕХАНИКО-МАТЕМАТИЧЕСКИЙ ФАКУЛЬТЕТ
Кафедра веб-технологий и компьютерного моделирования

А. С. Кравчук, А. И. Кравчук, Е. В. Кремень

ЯЗЫК JAVA.
ОБЩИЕ СВЕДЕНИЯ О КОЛЛЕКЦИЯХ И РЕАЛИЗУЕМЫХ ИМИ ИНТЕРФЕЙСАХ. КОЛЛЕКЦИИ, РЕАЛИЗУЮЩИЕ ИНТЕРФЕЙС LIST

Учебные материалы
для студентов специальности 1-31 03 08
«Математика и информационные технологии
(по направлениям)»

МИНСК
2023

УДК 004.432.045:004.738.5Java (075.8)
ББК 32.973.2-018.1я73-1
К78

Утверждено на заседании кафедры
веб-технологий и компьютерного моделирования
8 ноября 2022 г., протокол № 3

Рецензент
кандидат физико-математических наук,
доцент *Г. А. Расолько*

Кравчук, А. С.

К78 Язык Java. Общие сведения о коллекциях и реализуемых ими интерфейсах. Коллекции, реализующие интерфейс List : учеб. материалы / А. С. Кравчук, А. И. Кравчук, Е. В. Кремень. – Минск : БГУ, 2023. – 80 с.

Представлены общие сведения об иерархии коллекций. Особое внимание уделено базовому для большинства коллекций интерфейсу Collection и интерфейсам, реализуемых коллекциями, таким как Iterator и Iterable, Comparable и Comparator. Подробно рассматриваются коллекции, реализующие интерфейс List: Vector, Stack и ArrayList. Для каждой коллекции даются пояснения, как пользоваться классами и методами. Приводится необходимый теоретический материал и код программ, что существенно ускоряет усваивание материала, а также способствует более квалифицированному подходу к программированию. Материал по теме тщательно подобран, хорошо структурирован и компактно изложен.

Издание ориентировано как на тех, кто не имеет опыта практического программирования на языке Java, так и на тех, кто хотел бы систематизировать и улучшить свои знания.

УДК 004.432.045:004.738.5Java (075.8)
ББК 32.973.2-018.1я73-1

Оглавление

Введение

Java Collection Framework — иерархия интерфейсов и их реализаций, которая является частью JDK и позволяет разработчику пользоваться большим количеством готовых структур данных.

Коллекции/контейнеры – это хранилища, 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 элемента в коллекцию;
- удаление элемента из коллекции;
- изменение элемента в коллекции.

Коллекции могут хранить любые ссылочные типы данных.

Практически все коллекции и интерфейсы являются обобщенными (параметризованными), это опирается на тот факт, что на этапе создания объектов коллекции, разработчик уже будет знать элементы какого типа ему нужны - `String`, `Integer` и т.д.

Стандартный набор коллекций Java служит для избавления программиста от необходимости самостоятельно создавать реализации различных динамических структур данных, существенно расширяют возможности разработчика по манипулированию этими данными, а также стандартизируют код, обрабатывающий коллекции.

Java Collection Framework состоит из 3-х частей:

- интерфейсы,
- классы,
- алгоритмы.

Общие сведения об иерархии коллекций и их интерфейсов

Для хранения наборов данных в Java предназначены массивы. Однако их не всегда удобно использовать, прежде всего потому, что они имеют фиксированную длину. Эту проблему в Java решают коллекции. Однако суть не только в гибких по размеру наборах объектов, но в и том, что классы коллекций реализуют различные алгоритмы и структуры данных, например, такие как стек, очередь, дерево и ряд других.

Классы коллекций располагаются в пакете `java.util`, поэтому перед применением любых коллекций следует подключить данный пакет:

```
import java.util.*;
```

Хотя в Java существует множество коллекций, но все они образуют стройную и логичную систему. Во-первых, в основе каждой из коллекций лежит применение того или иного *параметризованного* интерфейса, который определяет базовый функционал. Среди этих интерфейсов можно выделить следующие (Рисунок 1).

Интерфейс `Iterable<E>` позволяет объекту реализующего его класса быть параметром оператора цикла «для каждого» (цикл `for-each`).

Интерфейс `Collection<E>` расширяет интерфейс `Iterable<E>`, и традиционно именно он считается базовым интерфейсом для большинства коллекций, потому что именно в нем перечисляются абстрактные методы общие для большинства коллекций. Этот интерфейс обычно используется для передачи коллекций в качестве значений и управления ими там, где требуется максимальная универсальность.

Существует еще один базовый интерфейс `Map<K, V>`, предназначен для созданий структур данных в виде словаря, где каждый элемент имеет определенный ключ и значение.

Параметрические интерфейсы, расширяющие базовый `Collection<E>` (Рисунок 1):

- интерфейс `Set<E>` используется для хранения множеств уникальных объектов;
- интерфейс `List<E>` представляет функциональность простых списков;
- интерфейс `Queue<E>` представляет функционал для структур данных в виде очереди.

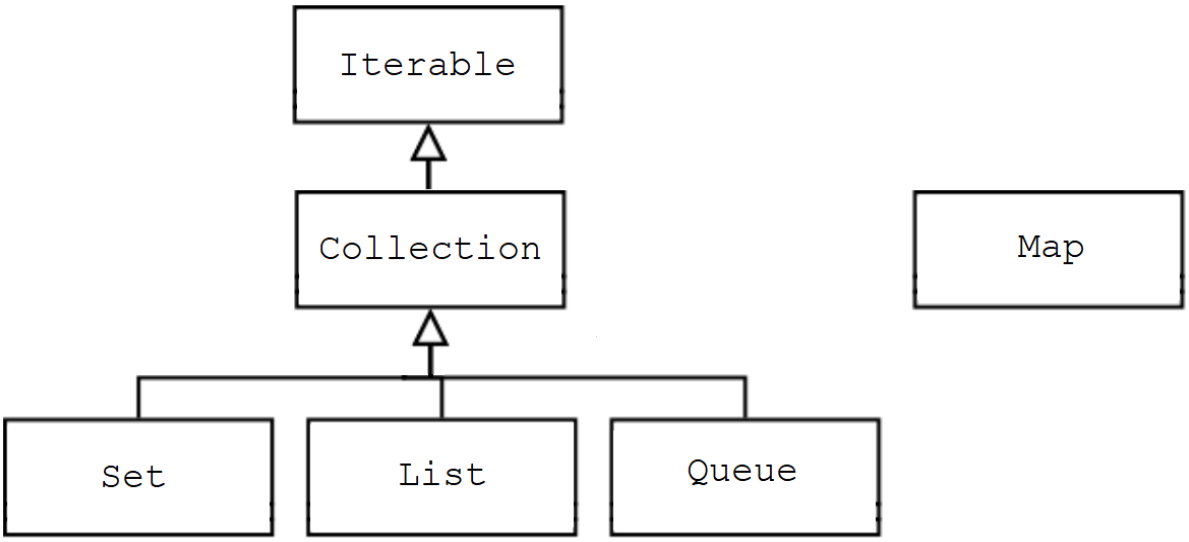


Рисунок 1 – Схема наследования базовых интерфейсов

Далее можно отметить, что:

- интерфейс `SortedSet<E>` - расширяет интерфейс `Set<E>` для создания сортированных коллекций;
- интерфейс `NavigableSet<E>` – в свою очередь расширяет интерфейс `SortedSet<E>` для создания коллекций, в которых можно осуществлять поиск по соответствию;
- интерфейс `Deque<E>` - наследует интерфейс `Queue<E>` и представляет функционал для двусвязных очередей.

Абстрактные классы, частично реализующие параметризованные интерфейсы

Эти интерфейсы частично реализуются абстрактными *параметризованными* классами:

- класс `AbstractCollection<E>` - базовый абстрактный класс (производный от `Object`) для других коллекций, который реализует интерфейс `Collection<E>`;
- класс `AbstractList<E>` - расширяет класс `AbstractCollection<E>` и реализует интерфейс `List<E>`, предназначен для создания коллекций в виде списков;
- класс `AbstractSet<E>` - расширяет класс `AbstractCollection<E>` и реализует интерфейс `Set<E>` для создания коллекций в виде множеств;
- класс `AbstractSequentialList<E>` - также расширяет класс `AbstractList<E>`. Используется для создания связанных списков;
- класс `AbstractQueue<E>` - расширяет класс `AbstractCollection<E>` и реализует интерфейс `Queue<E>`, предназначен для создания коллекций в виде очередей и стеков;
- класс `AbstractMap<K,V>` – расширяет класс `Object` и реализует параметризованный интерфейс `Map<K,V>`, предназначен для создания наборов по типу словаря с объектами в виде пары «ключ-значение».

Некоторые коллекции расширяющие абстрактные классы

С помощью применения вышеописанных интерфейсов и абстрактных классов в Java реализуется широкая палитра классов-коллекций - списки, множества, очереди, соответствия и другие, среди которых можно выделить следующие коллекции:

- ❖ расширяющие абстрактный класс `AbstractCollection<E>`:
  - `ArrayDeque` – является реализацией интерфейса `Deque`, имеет дополнительные методы, позволяющие реализовать конструкцию вида `LIFO` (`last-in-first-out` по-русски «стек»). Эта коллекция представляет собой реализацию с использованием массивов, подобно `ArrayList`, но не позволяет обращаться к элементам по индексу и хранение `null`. Как заявлено в документации, коллекция работает быстрее чем `Stack`, если используется как `LIFO` коллекция, а также быстрее чем `LinkedList`, если используется как FIFO (`first-in-first-out` по-русски «очередь»).
- ❖ расширяющие абстрактный класс `AbstractList<E>`:
  - `Vector` — реализует динамический массива объектов. Позволяет хранить любые данные, включая `null` в качестве элемента. В качестве альтернативы часто применяется ее аналог — `ArrayList`.
  - `ArrayList` - как и `Vector` является реализацией динамического массива объектов. Позволяет хранить любые данные, включая `null` в качестве элемента. Как можно догадаться из названия, его реализация основана на обычном массиве. Данную реализацию следует применять, если в процессе работы с коллекцией предполагается частое обращение к элементам по индексу. Из-за особенностей реализации индексное обращение к элементам выполняется за константное время `O(1)`. Но данную коллекцию рекомендуется избегать, если требуется частое удаление/добавление элементов в середину коллекции.

Замечание.

`Stack` — данная коллекция является расширением коллекции `Vector<E>`. После добавления в Java интерфейса `Deque`, рекомендуется использовать именно реализации этого интерфейса, например `ArrayDeque`.

- расширяющие абстрактный класс `AbstractSet<E>`:
  - `HashSet` – является реализацией интерфейса `Set`, базирующаяся на `HashMap`. Внутри использует объект `HashMap` для хранения данных. В качестве ключа используется добавляемый элемент, а в качестве значения — объект-пустышка (`new Object()`). Из-за особенностей реализации порядок элементов не гарантируется при добавлении.
  - `TreeSet` – представляет из себя набор отсортированных объектов в виде дерева. Предоставляет возможность управлять порядком элементов в коллекции при помощи объекта `Comparator`.

Замечание.

`LinkedHashSet` - связанное хеш-множество расширяет класс `HashSet<E>` и отличается от последнего только тем, что порядок элементов при обходе коллекции является идентичным порядку добавления элементов.

- расширяющие абстрактный класс `AbstractSequentialList<E>`:
  - `LinkedList` – является еще одной реализацией интерфейса `List`. Позволяет хранить любые данные, включая `null`. Особенностью реализации данной коллекции является то, что в ее основе лежит двунаправленный (двусвязный) список (каждый элемент имеет ссылку на предыдущий и следующий). Благодаря этому, добавление и удаление значения из середины коллекции происходит за линейное время `O(n)` при доступе из ее начала/конца, а доступ по индексу за константное `O(1)`. Так же, ввиду реализации, данную коллекцию можно использовать как стек или очередь. Для этого в ней реализованы соответствующие методы.
- расширяющие абстрактный класс `AbstractQueue<E>`:
  - `PriorityQueue` – особенностью данной очереди является возможность управления порядком элементов при помощи объекта `Comparator`, который задается при создании очереди.
- расширяющие абстрактный класс `AbstractMap<K,V>`:
  - `HashMap` – является структурой данных в виде словаря, в котором каждый объект имеет уникальный ключ и некоторое значение. Позволяет использовать `null` как в качестве ключа, так и значения. Данная коллекция не является упорядоченной: порядок хранения элементов зависит от значения хэш-функции

(понятие «функция» используется в литературе при описании хэширования в общеметодологическом, математическом смысле, а не в смысле конкретной реализации в рамках метода).

- `TreeMap` – является упорядоченной коллекцией. «По умолчанию», коллекция сортируется по ключам, но может настраиваться под конкретную задачу при помощи объекта `Comparator`.

Общие сведения о шаблонах проектирования

Шаблон проектирования или паттерн (англ. `design pattern`) в разработке программного обеспечения — повторяемая архитектурная конструкция, представляющая собой решение проблемы проектирования в рамках некоторого часто возникающего контекста (Рисунок 2).

Обычно шаблон не является законченным образцом, который может быть прямо преобразован в код; это лишь пример решения задачи, который можно использовать в различных ситуациях. Объектно-ориентированные шаблоны показывают отношения и взаимодействия между классами или объектами, без определения того, какие конечные классы или объекты приложения будут использоваться.

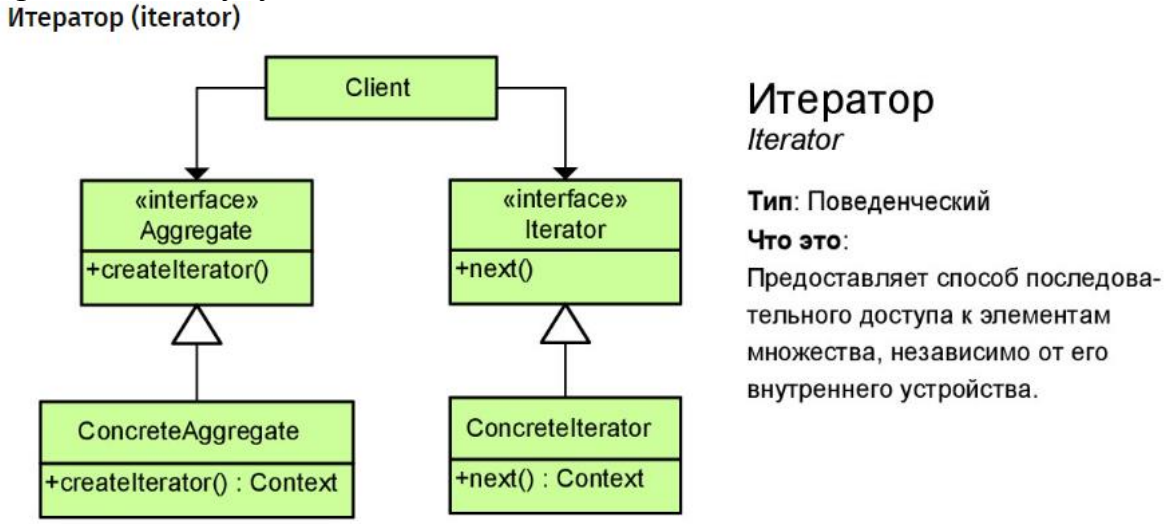


Рисунок 2 – Пример графического отображения с помощью UML-диаграмм шаблона проектирования

Параметризованный интерфейс `Iterator`

Дословный перевод слова «Iterator» - это «переборщик». То есть это некая сущность, способная перебрать все элементы в коллекции. При этом она позволяет это сделать без вникания во внутреннюю структуру и устройство коллекций.

Интерфейс `Iterator` в Java – частная реализация одноименного паттерна, применяемая как к готовым структурам (`List, Set, Queue, Map` и др.), так и к прочим, на усмотрение программиста.

Формат заголовка интерфейса:

```
Interface Iterator<E>
```

где `E` – тип элемента получаемого в качестве параметра и возвращаемого методом `next()`.

Интерфейс `Iterator` имеет четыре метода:

- `boolean` **`hasNext`**`()` - возвращает `true`, если в коллекции имеется текущий элемент, возвращаемый методом `next()`, на который указывает итератор, иначе возвращает `false`;
- `E` **`next`**`()` – возвращает *текущий* элемент и переходит к следующему;
- `default void` **`remove`**`()` - удаляет из базовой коллекции текущий элемент, который был получен последним вызовом `next()`;
- `Default void` **`forEachRemaining`**`(Consumer<? Super E> a)`- выполняет заданное действие для каждого оставшегося элемента, пока все элементы не будут обработаны или пока действие не вызовет исключение (аналог цикла `for each`).

Замечания.

- *первые два метода являются основными, а их реализации общеупотребительными для контейнеров, например в рамках использования конструкции цикла `while` и неявно в `for each`;*
- *несмотря на названия методы `hasNext()` и `next()` обрабатывают не следующий элемент, а* ***текущий****. Т. к. если бы они обрабатывали следующий, то в цикле `while()` не возможно было бы вывести на экран последний элемент коллекции, т.к. в этом случае метод `hasNext()` должен был*

бы вернуть `false`, а с помощью `next()` вывести на экран первый элемент коллекции;

- *можно сказать, что позиции итератора условно располагаются между элементами коллекции. Тогда в коллекции, позиций итератора всегда больше на единицу, чем количество элементов.*

Этот интерфейс является членом Java Collections Framework и заменил интерфейс `Enumeration`, ранее широко используемый для некоторых коллекций.

Расширяя или реализуя интерфейс `Iterator`, реализуется и соответствующий паттерн проектирования.

Замечание.

В целом по аналогии с языком C++ можно воспринимать значение объекта типа `Iterator` (или просто итератора) как ссылку (указатель) на некоторый элемент множества.

Существует интерфейс `ListIterator<E>` расширяющий `Iterator<E>` и применяющийся для обработки списков.

Параметризованный интерфейс `Iterable`

Формат заголовка интерфейса:

```
Interface Iterable<T>
```

где `T` – параметр типа элемента возвращаемый методом `iterator()`.

Из заголовка и описания этого интерфейса видно, что он не расширяет и не реализует интерфейс `Iterator`, но имеет метод `iterator()`, возвращающий значение типа `Iterator<T>`.

Имеет следующие методы:

- `Iterator<T> iterator()` – возвращает значение ссылки на некоторый элемент из множества элементов типа `T`.
- `Default Spliterator<T> spliterator()` - создает `Spliterator` над элементами, описанными этим `Iterable`;
- `default void forEach(Consumer<? Super T> ac)` - выполняет заданное действие для каждого элемента `Iterable` до тех пор, пока все элементы множества не будут обработаны или действие не вызовет исключение.

Замечание.

`Spliterator` - это еще один тип интерфейса `Iterator` для навигации по отдельным элементам. Предназначен для поддержки параллельного программирования. Однако можно использовать его не только для параллельной обработки элементов данных, но для последовательной.

`Iterable` говорит, что по объектам класса в принципе можно итерироваться, а `Iterator` задает сами методы для итерирования.

Но `Iterable` может в некий момент использовать несколько `Iterator` сразу. То есть, это интерфейсы, представляющие две разных сущности: коллекцию и обход коллекции.

Базовый интерфейс `Collection`

Интерфейс `Collection` является базовым для большинства коллекций и определяет их основной функционал:

```
public interface Collection<E> extends Iterable<E>{
    // определения методов
}
```

Интерфейс `Collection` является обобщенным (параметризованным) и расширяет интерфейс `Iterable`, поэтому все объекты коллекций можно перебирать в цикле по типу `for-each`.

Среди методов интерфейса Collection можно выделить следующие:

- `boolean` **`add`**`(E item)` - добавляет в коллекцию объект `item`. При удачном добавлении возвращает `true`, при неудачном – `false`;
- `boolean` **`addAll`**`(Collection<? Extends E> col)` - добавляет в коллекцию все элементы из коллекции `col`. При удачном добавлении возвращает `true`, при неудачном - `false`
- `void` **`clear`**`()` - удаляет все элементы из коллекции;
- `boolean` **`contains`**`(Object item)` - возвращает `true`, если объект `item` содержится в коллекции, иначе возвращает `false`;
- **`containsAll`**`(Collection<?> col)` - возвращает `true`, если эта коллекция содержит все элементы коллекции `col`.
- `boolean` **`isEmpty`**`()` - возвращает `true`, если коллекция пуста, иначе возвращает `false`;

- `Iterator<E>` **`iterator`**`()` - возвращает объект класса `Iterator` для обхода элементов коллекции;
- `Boolean` **`remove`**`(Object item)` - возвращает `true`, если объект `item` удачно удален из коллекции, иначе возвращается `false`;
- `boolean` **`removeAll`**`(Collection<?> col)` - удаляет все объекты коллекции `col` из текущей коллекции. Если текущая коллекция изменилась, возвращает `true`, иначе возвращается `false`;
- `boolean` **`retainAll`**`(Collection<?> col)` - удаляет все объекты из текущей коллекции, кроме тех, которые содержатся в коллекции `col`. Если текущая коллекция после удаления изменилась, возвращает `true`, иначе возвращается `false`;
- `int` **`size`**`()` - возвращает число элементов в коллекции
- `Object[]` **`toArray`**`()` - возвращает массив, содержащий все элементы коллекции.

Все эти и остальные методы, которые имеются в обобщенном (параметризованном) интерфейсе `Collection`, реализуются всеми коллекциями, поэтому в целом общие принципы работы с коллекциями будут одни и те же. Единообразный интерфейс упрощает понимание и работу с различными типами коллекций.

Так, добавление элемента будет всегда производиться с помощью метода `add`(), который принимает добавляемый элемент в качестве параметра. Для удаления вызывается метод `remove()`. Метод `clear()` будет очищать коллекцию, а метод `size()` возвращать количество элементов в коллекции.

Замечания:

- *в интерфейсе Collection также есть методы `stream()` и `parallelStream()` для получения из коллекции последовательного или параллельного потоков элементов соответственно;*
- *в настоящее время предпочтительным методом перебора коллекции является получение потока и выполнение над ним агрегатных операций, т.е. использования Stream API (тема рассмотривается позже);*
- *любые операции над множеством элементов коллекции изменяют базовую коллекцию, а агрегатные операции над потоком элементов коллекции не изменяют базовую коллекцию.*

Коллекции, реализующие интерфейс `List`

Интерфейс `List`

Реализации этого интерфейса представляют собой упорядоченные коллекции. Кроме того, разработчику предоставляется возможность доступа к элементам коллекции по индексу и по значению (так как реализации позволяют хранить дубликаты, результатом поиска по значению будет первое найденное вхождение).

В дополнение к операциям, унаследованным от `Collection`, интерфейс `List` включает в себя следующие методы:

- для манипулирования с элементами на основе их числового положения в списке (в эту группу входят такие методы, как `get()`, `set()`, `add()`, `addAll()` и `remove()`);
- для итерирования (методы `Iterator()` и `listIterator()`, первый предназначен для обхода списка только в прямом направлении, а второй в двух – прямом и обратном);
- для выделения диапазона и просмотра. Это метод `sublist()`, который выполняет произвольные операции с диапазоном в списке.

Интерфейс `List` имеет следующие основные методы:

1. `void` **`add`**`(int index, E element)` - вставляет указанный элемент в указанное положение в списке;
2. `boolean` **`add`**`(E elem)` - добавляет указанный элемент в конец списка;
3. `boolean` **`addAll`**`(int index, Collection<? extends E> col)` - добавляет в список по индексу `index` все элементы коллекции `col`. Если в результате добавления список был изменен, то возвращается `true`, иначе возвращается `false`;
4. `boolean` **`addAll`**`(Collection<? extends E> col)` - добавляет в конец списка все элементы коллекции `col`. Если в результате добавления список был изменен, то возвращается `true`, иначе возвращается `false`;
5. `void` **`clear`**`()` - удаляет все элементы из списка;
6. `void` **`sort`**`(Comparator<? super E> comp)`: сортирует список с помощью компаратора `comp`;
7. `boolean` **`contains`**`(Object elem)` – проверяет, является ли указанный объект компонентом в списке;

8. `boolean` **`containsAll`**`(Collection<?> col)` – возвращает `true`, если список содержит все элементы указанной коллекции `col`;
9. `boolean` **`equals`**`(Object o)` - сравнивает указанный объект с `Vector`-ом;
10. `E` **`get`**`(int index)` - возвращает элемент в указанной позиции в списке;
11. `int` **`hashCode`**`()` - возвращает значение хэш-кода для `Vector`-а;
12. `int` **`indexOf`**`(Object obj)` - возвращает индекс первого вхождения объекта `obj` в список. Если объект не найден, то возвращается -1;
13. `int` **`lastIndexOf`**`(Object obj)` - возвращает индекс последнего вхождения объекта `obj` в список. Если объект не найден, то возвращается -1;
14. `boolean` **`isEmpty`**`()` - проверяет список на отсутствие компонентов;
15. `ListIterator<E> listIterator()` - возвращает объект `ListIterator` для обхода элементов списка;
16. `static <E> List<E>` **`of`**`(элементы)` - создает из набора элементов объект `List;`
17. `E` **`remove`**`(int index)` - удаляет элемент в указанной позиции в списке;
18. `boolean` **`remove`**`(Object obj)` - удаляет первое упоминание указанного элемента в списке, если список не содержит элемент, он не изменяется;
19. `boolean` **`removeAll`**`(Collection col)` - удаляет из списка все его элементы, которые содержатся в указанной `Collection`;
20. `boolean` **`retainAll`**`(Collection<?> col)` — удаляет элементы, не принадлежащие переданной коллекции;
21. `E` **`set`**`(int index, E element)` - заменяет элемент в указанной позиции в списке указанным элементом;
22. `List<E>` **`subList`**`(int start, int end)`: получает набор элементов, которые находятся в списке между индексами `start` и `end`.

Коллекция `Vector`

Параметризованный класс `Vector`, реализующий интерфейс `List`, является динамическим массивом объектов. Позволяет хранить любые

данные, включая `null` в качестве элемента. Потому как в `Vector`, в отличии от других реализаций `List`, все операции с данными являются синхронизированными (потоко-безопасными в смысле многопоточного программирования).

Класс `Vector` синхронизирует каждую отдельную операцию. Это означает, что всякий раз, когда предполагается выполнение какой-либо операции над векторами, класс `Vector` автоматически применяет блокировку к этой операции. Это связано с тем, что когда один поток обращается к вектору, и в то же время другой поток пытается получить к нему доступ, генерируется вызываемое исключение `ConcurrentModificationException`. Следовательно, это постоянное использование блокировки для каждой операции делает векторы более медленными, менее эффективными, но обеспечивают безопасность от одновременного доступа из конкурирующих потоков. `ArrayList`, в отличие от `Vector`, метод, синхронизирующий список в целом. `Vector` до сих пор с успехом применяется в некоторых распределенных приложениях.

Замечание.

Коллекция считается устаревшей. Однако на ее базе очень удобно продемонстрировать реализованные в ней методы интерфейса `List`, которые будут совершенно аналогично применяться во всей группе коллекций, реализующих данный интерфейс.

В Java класс `Vector` оказывается очень полезным, если заранее не известен размер массива или нужен только тот, который может изменять размеры за время жизни программы.

Коллекция использует два понятия «размер» и «емкость»:

- размер массива – это текущее количество элементов с заданными значениями;
- емкость массива – размер ленты памяти, измеряемой в единицах ячеек выделенных под хранение элементов массива; эта память может быть пустой (ожидающей заполнения), частично заполненной и полностью заполненной; в последнем случае емкость совпадает с размером (в любом случае справедливо: емкость массива всегда больше или равна его размеру).

Коллекция поддерживает четыре конструктора со следующим синтаксисом:

- `Vector()` - первый конструктор, создает вектор по умолчанию, емкость элементов в массиве «по умолчанию» равно `10`, но размер (количество элементов) равен нулю

Пример.

```
Vector<ТипЗначения> v = new Vector();
```

- `Vector(int size)` - этот конструктор принимает целочисленный аргумент `size`, равный требуемому размеру (количеству элементов), и создает соответствующий емкостью `size` элементов, но размером (количество элементов) `0`.

Пример.

```
Vector<ТипЗначения> v = new Vector(5);
```

- `Vector(int size, int incr)` - этот конструктор создает вектор, чья начальная емкость задается `size` и инкремент которого определяется `incr`. Инкремент определяет количество элементов, на которое будет изменяться вектор при добавлении или удалении элементов.

Пример.

```
Vector<ТипЗначения> v = new Vector(5, 2);
```

- `Vector(Collection<? extends E> c)` - этот конструктор создает вектор, содержащий элементы коллекции `c`.

Пример.

```
Vector<ТипЗначения> v = new Vector(Vector c);
```

Замечание.

Параметр `<ТипЗначения>` необязательный, его можно явно не указывать (сохранение параметризации «по умолчанию»), тогда тип каждого хранимого значения в коллекции, будет определяться типом значения, указанного, например, в методе `add()`. Таким образом в отличие от обычных массивов в Java коллекция `Vector` может хранить разнотипные элементы если не указан явно `<ТипЗначения>`.

Пример работы с некоторыми методами коллекции

Методов в коллекции `Vector` существенно больше, чем определяет интерфейс `List`, т.к. эти дополнительные методы они переопределяют методы устаревшего интерфейса `Enumeration`.

Методы (устаревшего интерфейса) предлагается ***не*** использовать при разработке нового кода.

Приведем некоторые примеры использования методов интерфейса `List`, переопределенных в `Vector`-е.

Пример.

```
import java.util.*;
public class Main
{
   public static void main(String args[]) {
      //начальный размер 3, шаг 2
      Vector v = new Vector(3, 2);
      System.out.println("Начальный размер: " +
                         v.size());
      System.out.println("Начальная емкость: " +
                         v.capacity());

      v.add(new Integer(1));
      v.add(new Integer(2));
      v.add(new Integer(3));
      v.add(new Integer(4));
      System.out.println("Размер массива " +
                         "после четырех добавлений: " +
                         v.size());
      System.out.println("Емкость после четырех " +
                         "добавлений: " +
                         v.capacity());
      System.out.println("Массив: ");
      for(int i = 0 ; i < v.size() ; i++) {
         System.out.print(v.get(i) + " ");
      }
      System.out.println();
      v.set(0, new Integer(10));// допустимо v.set(0,10)
      //добавление строки
      v.add(new String("abv"));
      System.out.println("Текущая емкость после " +
                         "добавления объекта String: " +
                         v.capacity());
      v.add(new Double(6.08));
      v.add(new Integer(7));
      System.out.println("Текущая емкость после " +
                         "добавления объектов Integer " +
                         "и Double: " + v.capacity());
      System.out.println("Первый элемент: " + v.get(0));
      System.out.println("Последний элемент: " +
                         v.get(v.size() - 1));

      if(v.contains(new Integer(3))) {
         System.out.println("Вектор содержит число 3.");
      }
      //использование интерфейса Iterable
      System.out.println("Интерфейс Iterable:");
```

```
47          //цикл while
48          System.out.println("\tЦикл while:");
49          //напомним, что hasNext() - возвращает true, если
50          //итерация содержит текущий элемент коллекции;
51          //next()   возвращает текущий элемент и переходит к
52          //следующему;
53          Iterator iter = v.iterator();
54          while(iter.hasNext()) {
55             System.out.print(iter.next() + " ");
56          }
57          System.out.println();
58         //цикл for…each
59          System.out.println("\tЦикл while:");
60          for(var s : v) {
61             System.out.print(s.toString() + " ");
62          }
63       }
64    }
```

Результат работы программы:

```
Начальный размер: 0
Начальная емкость: 3
Размер массива после четырех добавлений: 4
Емкость после четырех добавлений: 5
Массив:
1 2 3 4
Текущая емкость после добавления объекта String: 5
Текущая емкость после добавления объектов Integer и Double: 7
Первый элемент: 10
Последний элемент: 7
Вектор содержит число 3.
Интерфейс Iterable:
    Цикл while:
10 2 3 4 abv 6.08 7
    Цикл while:
10 2 3 4 abv 6.08 7
```

Метод `clear()`

Метод `clear()` - один из самых ***не***обсуждаемых и на первый взгляд ненужных методов коллекции `Vector`. Согласно описанию, этот метод удаляет все элементы вектора, к которому он применяется.

Пример.

```
1  import java.util.*;
2  public class Main {
3     public static void main (String args[]) {
4        Vector array = new Vector();
```

```
5           array.add(1);
6           array.add(2);
7           array.add(3);
8           System.out.println("Исходный массив");
9           System.out.println(array.toString());
10          System.out.println("Массив после clear()");
11          array.clear();
12          System.out.println(array.toString());
13      }
14  }
```

Результаты работы программы:

```
Исходный массив
[1, 2, 3]
Массив после clear()
[]
```

Проблема с обоснованием необходимости использования этого метода заключается в том, что как было продемонстрировано выше любой объект типа `Vector` может изменять свою длину и значения элементов без промежуточной чистки памяти. Отсюда можно сделать вывод о ненужности метода `clear()`.

Обоснование необходимости его использования заключается в том, что если разработчик создает вектор с определенным набором значений, а потом (как он надеется) присваивает старой ссылочной переменной другой вектор с другим набором значений, то (если при создании нового вектора использовался метод `add()`) к старому вектору добавятся значения нового вектора.

Таким образом следует использовать метод `clear()` для промежуточной чистки значения ссылочной переменной перед присвоением значений новой коллекции.

Метод `removeAll()`

Данный метод удаляет из этого вектора все его элементы, которые содержатся в указанной `Collection`.

Пример.

```
1  import java.util.*;
2  public class Main {
3      public static void main (String args[]) {
4          Vector array = new Vector();
5          for(int i = 0; i < 7; i++) {
6              array.add(i);
7          }
8          System.out.println("Исходный массив");
9          System.out.println(array.toString());
10         Vector deletedArray = new Vector();
11         for(int i = 0; i < 3; i++) {
12             deletedArray.add(i + 2);
13         }
14         System.out.println("Удаляемый массив");
15         System.out.println(deletedArray.toString());
16         array.removeAll(deletedArray);
17         System.out.println("Результат удаления");
18         System.out.println(array.toString());
19     }
20 }
```

Результат работы программы:

```
Исходный массив
[0, 1, 2, 3, 4, 5, 6]
Удаляемый массив
[2, 3, 4]
Результат удаления
[0, 1, 5, 6]
```

Выводы:

- порядок перечисления элементов в удаляемом массиве неважен;
- удаляемый массив может содержать значения элементов, не существующих в массиве, из которого выполняется удаление;
- элементы удаляемого массива с значениями несовпадающими со значениями массива, из которого происходит удаление также могут иметь произвольные индексы;
- удаляемый массив может иметь большую размерность, чем массив, из которого будет происходить удаление; например,

полностью содержать в себе в произвольном порядке удаляемый массив; результатом подобной операции будет пустой массив.

Методы remove(). Примеры статических методов для работы с коллекцией `Vector`

Необходимо отметить, что коллекция `Vector` содержит два перегруженных метода `remove()` для работы с элементами коллекции по индексу или значению:

- `Object remove(int index);`
- `boolean remove(Object obj).`

Замечание.

Следует помнить, что существует еще один дефолтный метод интерфейса `Iterator` *(*`default void` ***`remove`****()) для работы с итератором. Но его использование будет продемонстрировано в других коллекциях.*

Первый удаляет элемент коллекции `Vector` по индексу, а второй удаляет первое упоминание указанного элемента `obj` в этом векторе. Если вектор не содержит элемент заданным значением `obj`, то он не изменяется.

В качестве примера использования метода `remove` в его первой версии приведем пример методов, осуществляющих сдвиг значений `Vector`-а вправо или лево на один элемент. Однако ***не*** будем использовать вариант с полной параметризацией, когда тип чисел, заполняющих вектор определяется типами констант MIN и MAX, передаваемых в метод для заполнения. Отказ от этой возможности очевидно сделан с целью упрощения конструкций языка и повышения читаемости примера.

В примере ниже создается `Vector`, заполненный случайными целыми числами.

Пример.

```
1  import java.util.*;
2  public class Program {
3      //заполнение случайными целыми числами
4      static Vector generationArray(final int MIN,
5                                    final int MAX,
6                                    int n) {
7          Vector array = new Vector(n);
8          for(int i = 0; i < n; i++) {
```

```
9           array.add((new Double(Math.random() *
10                          (MAX - MIN) + MIN)).intValue());
11      }
12      return array;
13  }
14  //циклический сдвиг влево на одно значение
15  static void shiftLeft(Vector a) {
16      var temp = a.get(0);
17      a.remove(0);
18      a.add(temp);
19  }
20  //циклический сдвиг вправо на одно значение
21  static void shiftRight(Vector a) {
22      var temp = a.get(a.size() - 1 );
23      a.remove(a.size() - 1);
24      a.add(0, temp);
25  }
26
27  public static void main (String args[]) {
28      //заполнение вектора целыми значениями
29      final int MIN1 = 0;  //нижняя граница генерации
30      final int MAX1 = 10; //верхняя граница генерации
31      Vector a = generationArray(MIN1, MAX1, 5);
32      System.out.println("Исходный массив");
33      System.out.println(a.toString());
34      System.out.println("Циклический сдвиг влево");
35      shiftLeft(a);
36      System.out.println(a.toString());
37      System.out.println("Циклический сдвиг вправо");
38      shiftRight(a);
39      System.out.println(a.toString());
40  }
41 }
```

Результат работы программы:

```
Исходный массив
[0, 4, 3, 9, 3]
Циклический сдвиг влево
[4, 3, 9, 3, 0]
Циклический сдвиг вправо
[0, 4, 3, 9, 3]
```

Замечания:

Сами методы, осуществляющие сдвиг `shiftLeft()` *и* `shiftRight()`*, полностью параметризованны и с их помощью можно осуществлять сдвиг не только в числовом* `Vector`*-е, но и в* `Vector`*-е содержащем объекты пользовательского типа.*

Применение второго варианта метода `remove()` *настолько очевидно, что не будет сопровождаться примером.*

Статические методы для работы с вектором, содержащим значения базовых типов

Рассмотрим пример определения статических пользовательских методов для работы с коллекцией `Vector`.

Пример.

```
1  import java.util.*;
2  public class Program {
3     //Первая реализация метода заполнения Vector-а
4     //случайными целыми числами
5     static void generationArray(final int MIN,
6                                 final int MAX,
7                                 Vector array,
8                                 int n) {
9       for(int i = 0; i < n; i++) {
10        array.add((new Double(Math.random() *
11                    (MAX - MIN) + MIN)).intValue());
12      }
13    }
14    //Перегрузка метода: создание + заполнение массива
15    //из случайных чисел с плавающей точкой
16    static Vector generationArray(final int MIN,
17                                  final int MAX,
18                                  int n) {
19      Vector array = new Vector();
20      for(int i = 0; i < n; i++) {
21        array.add(Math.random() * (MAX - MIN) + MIN);
22      }
23      return array;
24    }
25
26    public static void main (String args[]) {
```

```
27          int n = 5;
28          //выделение памяти под коллекцию
29          Vector a = new Vector();
30          final int MIN = 0;  //нижняя граница генерации
31          final int MAX = 10; //верхняя граница генерации
32          //вызов первой реализации метода
33          //generationArray()
34          generationArray(MIN, MAX, a, n);
35          System.out.println(a.toString() + "\n");
36          //вызов перегруженного метода, возвращающего в
37          //качестве значения заполненную коллекцию Vector
38          String strArray =
39                  generationArray(MIN, MAX, n).toString();
40          System.out.println(strArray);
41      }
42  }
```

Результат работы программы:

```
[4, 6, 0, 2, 0]

[0.9205124711057533, 5.680497940533917, 7.13172800287661, 9.063366355237086, 0.13035299246728305]
```

Предыдущий пример является очень узким в демонстрационном смысле. Поскольку параметризованный вектор может принимать различные типы значений, но в перегруженных методах `generationArray()` типы заполняемых чисел указывается жестко использованием (или неиспользованием в перегруженном методе) метода `intValue()`, приводящего значение объекта любого типа наследника `Number` к целому числу.

Отсутствие достаточной методологической полноты демонстрации использования параметризации коллекции объясняется тем, что в перегруженных методах `generationArray()` какие-либо параметры отсутствуют (это не `Generic`-и) и тип коллекции `Vector` алгоритмически жестко предопределен.

Этой жесткой предопределенностью типа значений коллекции `Vector` и объясняется узость приведенного примера.

Продемонстрируем пример использования параметризации `Vector`-а, т.е. предусмотрим возможность заполнения `Vector`-а значениями, типы которых *могут* определяться не на этапе компиляции, а во время выполнения программы.

Замечание.

Слово «могут» указывает на то, что в примере приведенном ниже нет ввода с консоли, а следовательно все определяется на этапе компиляции. Но если разработчик будет использовать консоль и предусмотрит возможность ввода разных типов чисел (через их строковое представление), то приведенные примеры все равно будут работоспособны.

Например, продемонстрируем самую простую (без использования параметризации) возможность реализации подобного подхода (изменения типа заполняющих значений) с перегрузкой второго варианта метода `generationArray()`, приведенного выше.

Эта перегрузка позволит заполнить созданный `Vector` различными типами значений, определимых типами введенных значений `MIN` и `MAX` диапазона, генерируемых величин.

Пример.

```
1  import java.util.*;
2  public class Program {
3     //первый вариант метода
4     static Vector generationArray(final int MIN,
5                                   final int MAX,
6                                   int n) {
7       Vector array = new Vector();
8       for(int i = 0; i < n; i++) {
9         array.add((new Double(Math.random() *
10                      (MAX - MIN) + MIN)).intValue());
11      }
12      return array;
13    }
14    //перегрузка с другими типами входных параметров
15    static Vector generationArray(final double MIN,
16                                  final double MAX,
17                                  int n) {
18      Vector array = new Vector();
19      for(int i = 0; i < n; i++) {
20        array.add(Math.random() * (MAX - MIN) + MIN);
21      }
22      return array;
23    }
24
25    public static void main (String args[]) {
26      //заполнение вектора целыми значениями
```

```
27          final int MIN1 = 0;  //нижняя граница генерации
28          final int MAX1 = 10; //верхняя граница генерации
29          String strArray =
30                  generationArray(MIN1, MAX1, 5).toString();
31          System.out.println(strArray);
32          //заполнение вектора значениями с плав. точкой
33          final double MIN2 = 0;
34          final double MAX2 = 10;
35          System.out.println(generationArray(MIN2,MAX2,5));
36      }
37  }
```

Пример работы программы:

```
[7, 6, 9, 5, 2]
[2.3044985699729414, 5.870416314624483, 0.8609444635608143, 1.668204632717033, 5.068389173835172]
```

Замечание.

В Java все объекты преобразуются к строковому типу при выводе на экран. При этом для объектов неявно вызывается метод `toString()` *родительского класса* `Object`*, от которого наследуются все классы. В случае необходимости метод можно вызывать явно. Следующие две строки равносильны:*

```
System.out.println(ob);
System.out.println(ob.toString());
```

Однако продемонстрированный вариант – это больше силовое решение вопроса с требованием универсальности заполнения `Vector`-а любыми типами данных.

Казалось бы, все можно просто решить с помощью параметризации входных параметров, но это не совсем так. Если ввести параметризацию параметров `MIN` и `MAX`, то возникнет проблема с тем, что с параметризованными типами не переделаны арифметические операции и соответствующее выражение не может быть вычислено (выражение: `Math.random() * (MAX - MIN) + MIN`) после этого также выясниться, что метод `add()` коллекции `Vector` также не может работать с параметризованными типами.

Соответственно необходимо организовать преобразование параметризованного типа данного к ***не***параметризованному с учетом *естественного* вариативного изменения типа входного параметра.

Также следует предусмотреть возможность возникновения исключительной ситуации при которой типы параметров `MIN` и `MAX` не совпадают.

Пример.

```
1  import java.util.*;
2  //класс-обертка для приведения параметра типа к
3  //числовому типу double
4  class CastDouble<T extends Number> {
5     private double x;
6     //параметризованный конструктор
7     CastDouble (T x) {
8         this.x = x.doubleValue();
9     }
10    //геттер
11    double get() { return x; }
12 }
13
14 public class Program
15 {
16     static <T extends Number> Double valueRandom(T a,
17                                                  T b) {
18           CastDouble aCast = new CastDouble(a);
19           CastDouble bCast = new CastDouble(b);
20           return Math.random() * (bCast.get() -
21                                   aCast.get()) + aCast.get();
22     }
23
24    static <T extends Number>
25              Vector generationArray(final T MIN,
26                                     final T MAX,
27                                     int n) {
28       Vector array = new Vector();
29       //исключение: типы MAX и MIN не совпадают
30       try {
31          if(MIN instanceof Double &
32                                    MAX instanceof Double ) {
33          for(int i = 0; i < n; i++) {
34             array.add(valueRandom(MIN, MAX));
35          }
36          }
37          else if (MAX instanceof Integer &
38                                   MIN instanceof Integer ) {
39          for(int i = 0; i < n; i++) {
40             array.add(valueRandom(MIN,
```

```
41                                          MAX).intValue());
42                  }
43              }
44              else {
45                  throw new Exception("типы MIN и MAX " +
46                                      "не совпадают");
47              }
48          }//конец try
49          catch(Exception ex) {
50              System.out.println(ex.getMessage());
51          }
52          return array;
53      }
54
55      public static void main (String args[]) {
56          //заполнение вектора целыми значениями
57          final int MIN1 = 0;  //нижняя граница генерации
58          final int MAX1 = 10; //верхняя граница генерации
59          System.out.println(generationArray(MIN1, MAX1, 5));
60          //заполнение вектора значениями с плав. точкой
61          final double MIN2 = 0;
62          final double MAX2 = 10;
63          System.out.println(generationArray(MIN2, MAX2, 3));
64          //вызов исключительной ситуации
65          System.out.println(generationArray(0.5, 10, 4));
66      }
67  }
```

Результат работы программы:

```
[9, 2, 7, 8, 2]
[2.0924065204873554, 4.790745862940501, 3.9743869580450273]
типы MIN и MAX не совпадают
[]
```

Пользовательские классы в параметрах `vector`

Следует отметить, что исходя из примера, приведенного ниже, универсальность алгоритмов обработки достигается только тогда, когда для класса используемого в качестве параметра в коллекции переопределены все необходимые методы (ниже такими методами являются методы `toString()` записи и собственно самого `Vector`-а).

Пример.

```
1  import java.util.*;
2
3  record Person(String name, int age) { }
4
5  public class Program
6  {
7      static <T> Vector generationArray(T[] aInit) {
8        Vector array = new Vector();
9        for(int i = 0; i < aInit.length; i++) {
10         array.add(aInit[i]);
11       }
12       return array;
13     }
14
15     static void displayColumn(Vector array) {
16       Iterator iter = array.iterator();
17       while(iter.hasNext()) {
18          System.out.println(iter.next().toString());
19       }
20     }
21
22     public static void main (String args[]) {
23       //создание инициализирующего массива объектов
24       Person[] a = {new Person("Том", 36),
25                     new Person("Боб", 36),
26                     new Person("Сергей", 36) };
27       //вывод вектора, содержащего объекты
28       //пользовательского класса
29       displayColumn(generationArray(a));
30       System.out.println();
31       //второй вариант вывода Vector-а,
32       //сформированного из массива Person
33       String strArray =
34                         generationArray(a).toString();
35       System.out.println(strArray);
36     }
37 }
```

Результат работы программы:

```
Person[name=Том, age=36]
Person[name=Боб, age=36]
Person[name=Сергей, age=36]

[Person[name=Том, age=36], Person[name=Боб, age=36], Person[name=Сергей, age=36]]
```

Создание многомерных векторов и их использование в методах

Элементами вектора могут быть и другие вектора. Собственно, как и объекты других коллекций. Например, можно сделать вектор, каждый элемент которого представляет собой вектор целых чисел:

```
Vector<Vector> имяМатрицы = new Vector();
```

В Jave при работе с векторами нельзя использовать операцию индексирования (например, [i]), однако есть возможность использования методов. С непривычки это очень затрудняет работу с двумерными массивами, но если разобраться, то гибкость синтаксических конструкций, используемых в Java достаточна для освоения работы с многомерными массивами.

Для первичного и упрощенного знакомства с правилами работы с двумерными массивами перепишем предыдущий пример следующим образом: массив из трех записей преобразуем в двумерный массив из трех строк и двух столбцов, т.е. объект класса `record` преобразуем к двум значениям имени (тип `String`) и возрасту (тип `int`). Таким образом создаваемый двумерный массив будет иметь один столбец строк, а второй столбец целых чисел.

Пример.

```
1  import java.util.*;
2
3  record Person(String name, int age) { }
4
5  public class Program
6  {
7      static Vector<Vector> generationArray(Person[] aInit){
8          Vector<Vector> array = new Vector();
9          for(int i = 0; i < aInit.length; i++) {
10             //построчная инициализация двумерного массива
11             Vector temporary = new Vector();
```

```
12              temporary.add(aInit[i].name());;
13              temporary.add(aInit[i].age());
14              //построчное присвоение двумерному массиву
15              array.add(temporary);
16          }
17          return array;
18      }
19
20      static void displayColumn(Vector<Vector> array) {
21          for(int i = 0; i < array.size(); i++) {
22             //построчный вывод двумерного массива
23              System.out.println(array.get(i).toString());
24           }
25      }
26
27      public static void main (String args[]) {
28          //создание массива объектов
29          Person[] a = {new Person("Том", 36),
30                        new Person("Боб", 36),
31                        new Person("Сергей", 36) };
32          //построчный вывод двумерного вектора
33          displayColumn(generationArray(a));
34          System.out.println();
35          //второй вариант вывода двумерного Vector-а,
36          String strArray =
37                              generationArray(a).toString();
38          System.out.println(strArray);
39      }
40  }
```

Результат работы программы:

```
[Том, 36]
[Боб, 36]
[Сергей, 36]

[[Том, 36], [Боб, 36], [Сергей, 36]]
```

Замечание.

Также можно создавать вектор из стеков, очередей, деков, можно создавать трехмерные векторы (трехмерные массивы) и т.д.

Следующий пример демонстрирует то, что в Java можно формировать двумерный массив как построчно (метод `initArray()`), так и работать в

привычном стиле поэлементно (метод `transposition()`) с помощью методов `get()` и `set()`.

Пример.

```
1  import java.util.*;
2  public class Program {
3     //метод для создания и заполнение двум. массива
4     static Vector<Vector> initArray(int n) {
5        Vector<Vector> array = new Vector(n);
6        for(int i = 0; i < n; i++) {
7           //вспомагательная строка
8           Vector tempVector = new Vector();
9           //построчное обнуление
10          for(int j = 0; j < n; j++) {
11             tempVector.add(0);
12          }
13          tempVector.set(n - 1 - i, i + 1);
14          array.add(tempVector);
15       }
16       return array;
17    }
18    //метод для вывода на экран в виде матрицы
19    static void display(Vector<Vector> a) {
20       for(int i = 0; i < a.size(); i++) {
21          System.out.println(a.get(i));
22       }
23    }
24    //транспонирование матрицы
25    static void transposition(Vector<Vector> a) {
26       for(int i = 0; i < a.size(); i++) {
27          for(int j = i + 1; j < a.get(i).size(); j++) {
28             var temp = a.get(i).get(j);
29             a.get(i).set(j, a.get(j).get(i));
30             a.get(j).set(i, temp);
31          }
32       }
33    }
34
35    public static void main (String args[]) {
36       int n = 3;
37       System.out.println("Исходная матрица");
38       Vector<Vector> array = initArray(n);
39       display(array);
40       System.out.println("Результат транспонирования");
```

```
41          transposition(array);
42          display(array);
43      }
44  }
```

Результат работы программы:

```
Исходная матрица
[0, 0, 1]
[0, 2, 0]
[3, 0, 0]
Результат транспонирования
[0, 0, 3]
[0, 2, 0]
[1, 0, 0]
```

Вектора как свойства классов

Рассмотрим случай, когда полем класса являются `Vector`. В рассмотренном ниже примере в классе в качестве полей создается два `Vector`-а:

- имена городов;
- почтовые индексы.

Эти два вектора связаны между собой посредством соответствия индексов. Методы класса позволяют организовать поиск индекса города по его названию в создаваемом объекте.

Пример.

```
1   import java.util.*;
2   class Matcher {
3       private Vector<String> name = new Vector();
4       private Vector<Integer> postCode = new Vector();
5       //блок инициализации
6       {
7           name.add("Минск");
8           name.add("Брест");
9           name.add("Витебск");
10          postCode.add(220000);
11          postCode.add(224000);
12          postCode.add(210000);
13      }
14      //методы
```

```
15      int finedIndex(String cityName){
16         for(int i = 0; i < name.size(); i++) {
17            if(cityName.equals(this.name.get(i))) {
18               return i;
19            }
20         }
21         return -1;
22      }
23      int finedPostCode(String cityName) {
24         int i = finedIndex(cityName);
25         if (i < 0) return i;
26         return this.postCode.get(i);
27      }
28   }
29
30   public class Main
31   {
32      public static void main(String[] args) {
33         Matcher data = new Matcher ();
34         System.out.print(data.finedPostCode("Минск"));
35      }
36   }
```

Результат работы программы:

```
220000
```

Коллекция Stack

Параметризованный класс `Stack` является расширением коллекции `Vector`. Была добавлена в Java как реализация стека `LIFO` (last-in-first-out – последний-вошёл-первый-вышел). Является частично синхронизированной коллекцией (кроме метода добавления `push()`). После добавления интерфейса `Deque`, рекомендуется использовать именно реализации этого интерфейса, например `ArrayDeque`.

Замечания:

- *`Stack` также как и `Vector` сейчас практически не используется в силу недостаточно высокой гибкости. Тем не менее, изучить его стоит, поскольку может именно он может пригодиться при разработке;*

- *более полный и непротиворечивый набор операций стека LIFO предоставляется интерфейсом Deque и его реализациями (например, ArrayDeque<E>), которые следует использовать вместо этого класса. Класс ArrayDeque<E> быстрее, чем Stack<E>, если используется как стек.*

Класс `Stack` является подклассом (классом-наследником) класса `Vector`. Коллекция поддерживает только один конструктор «по умолчанию» `Stack()` (без параметров), с помощью которого создается пустой `Stack`:

Пример.

```
Stack<ТипЗначения> v = new Stack();
```

Для работы с коллекцией используются пять специфических методов, определенных в данной коллекции:

- `boolean` **`empty`**`()` - проверяет, является ли стек пустым: возвращает `true`, если стек пустой; возвращает `false`, если стек содержит элементы;
- `Object` **`peek`**`()` - возвращает последний элемент стека, но ***не*** удаляет его.
- `Object` **`pop`**`()` - возвращает последний элемент стека, удаляя его;
- `Object` **`push`**`(Object element)` - вставляет элемент в конец стека; значение вставляемого элемента также возвращается и по желанию может быть использовано;
- `int` **`search`**`(Object element)` - ищет элемент в стеке. Если элемент найден, возвращается его смещение от вершины стека. В противном случае возвращается `-1`.

Пример.

```
1  import java.util.*;
2  public class Main
3  {
4      public static void main(String args[]) {
5          Stack stack = new Stack();
6          stack.push(0);
7          stack.push(1);
8          stack.push(2);
9
10         System.out.println("Текущий стек: " + stack);
11         System.out.println("Последний эл. стека: " +
```

```
                                stack.peek());
        System.out.println("Стек не изменился: " + stack);
        System.out.println("Результат поиска 7 в стеке: " +
                                stack.search(7));
        System.out.println("Удаляем: " + stack.pop());
        System.out.println("Удаляем: " + stack.pop());
        System.out.println("Удаляем: " + stack.pop());
        System.out.println("Текущий стек: " + stack);
        System.out.println("Стек пуст? " + stack.empty());
    }
}
```

Результат работы программы:

```
Текущий стек: [0, 1, 2]
Последний эл. стека: 2
Стек не изменился: [0, 1, 2]
Результат поиска 7 в стеке: -1
Удаляем: 2
Удаляем: 1
Удаляем: 0
Текущий стек: []
Стек пуст? true
```

Некоторые методы коллекции, унаследованные от `Vector`

Коллекция `Stack` непосредственно не реализует интерфейс `List`, но является потомком `Vector`-а и получает через механизм наследования переопределённые в `Vector`-е методы `List`-а.

Продемонстрируем, что унаследованные `Stack`-ом от `Vector`-а методы работают совершенно аналогично.

Замечание.

В конце примера также используется метод pop() и демонстрируется как с его помощью можно вывести элементы стека на экран в обратном порядке.

Пример.

```
import java.util.*;
public class Main
{
    public static void main(String args[]) {
```

```
5         Stack s = new Stack();
6         System.out.println("Начальный размер: " +
7                            s.size());
8         System.out.println("Начальная емкость: " +
9                            s.capacity());
10
11        s.add(new Integer(1));
12        s.add(new Integer(2));
13        s.add(3);
14        s.add(4);
15        System.out.println("Размер стека " +
16                           "после четырех добавлений: " +
17                           s.size());
18        System.out.println("Емкость после четырех " +
19                           "добавлений: " +
20                           s.capacity());
21        System.out.println("Текущий стек: " + s);
22        System.out.println("Вывод стека как вектора " +
23                           " (цикл for): ");
24        for(int i = 0 ; i < s.size() ; i++) {
25           System.out.print(s.get(i) + " ");
26        }
27        System.out.println();
28        s.set(0, new Integer(10));// допустимо v.set(0,10)
29        //добавление в стек строки
30        System.out.println("Текущий стек: " + s);
31        s.add(new String("abv"));
32        System.out.println("Текущий стек: " + s);
33        System.out.println("Текущая емкость после " +
34                           "добавления объекта String: " +
35                           s.capacity());
36        s.add(new Double(6.08));
37        s.add(new Integer(7));
38        System.out.println("Текущий стек: " + s);
39        System.out.println("Текущая емкость после " +
40                           "добавления объектов Integer " +
41                           "и Double: " + s.capacity());
42        System.out.println("Первый элемент: " + s.get(0));
43        System.out.println("Последний элемент: " +
44                           s.get(s.size() - 1));
45
46        if(s.contains(new Integer(3))) {
47           System.out.println("Стек содержит число 3.");
48        }
49        //использование интерфейса Iterable
50        System.out.println("Интерфейс Iterable.");
51      //цикл for…each
52        System.out.println("\tЦикл for..each:");
```

```
53          for(var t : s) {
54              System.out.print(t.toString() + " ");
55          }
56          //цикл while
57          System.out.println("\n\tЦикл while. " +
58                                  "Прямой порядок");
59          //напомним, что hasNext() - возвращает true, если
60          //итерация содержит текущий элемент коллекции;
61          //next()   возвращает текущий элемент и переходит к
62          //следующему;
63          Iterator iter = s.iterator();
64          while(iter.hasNext()) {
65              System.out.print(iter.next() + " ");
66          }
67          System.out.println("\n\tWhile. Обратный " +
68                                  "порядок с удалением");
69          iter = s.iterator();
70          while(iter.hasNext()) {
71              System.out.print(s.pop() + " ");
72          }
73          if(s.isEmpty()) {
74              System.out.println("\nСтек пуст.");
75          }
76      }
77  }
```

Результат работы программы:

```
Начальный размер: 0
Начальная емкость: 10
Размер стека после четырех добавлений: 4
Емкость после четырех добавлений: 10
Текущий стек: [1, 2, 3, 4]
Вывод стека как вектора  (цикл for):
1 2 3 4
Текущий стек: [10, 2, 3, 4]
Текущий стек: [10, 2, 3, 4, abv]
Текущая емкость после добавления объекта String: 10
Текущий стек: [10, 2, 3, 4, abv, 6.08, 7]
Текущая емкость после добавления объектов Integer и Double: 10
Первый элемент: 10
Последний элемент: 7
Стек содержит число 3.
Интерфейс Iterable.
    Цикл for..each:
10 2 3 4 abv 6.08 7
    Цикл while. Прямой порядок
```

```
10 2 3 4 abv 6.08 7
    While. Обратный порядок с удалением
7 6.08 abv 4 3 2 10
Стек пуст.
```

Метод `removeAll()` удаляет из стека все элементы, которые содержатся в указанной `Collection`.

Пример.

```
1  import java.util.*;
2  public class Main {
3      public static void main (String args[]) {
4          Stack s = new Stack();
5          for(int i = 0; i < 7; i++) {
6              s.push(i);
7          }
8          System.out.println("Исходный стек");
9          System.out.println(s.toString());
10         Stack deletedStack = new Stack();
11         for(int i = 0; i < 3; i++) {
12             deletedStack.push(i + 2);
13         }
14         System.out.println("Удаляемый массив");
15         System.out.println(deletedStack.toString());
16         s.removeAll(deletedStack);
17         System.out.println("Результат удаления");
18         System.out.println(s.toString());
19     }
20 }
```

Результат работы программы:

```
Исходный стек
[0, 1, 2, 3, 4, 5, 6]
Удаляемый массив
[2, 3, 4]
Результат удаления
[0, 1, 5, 6]
```

Выводы:

- порядок перечисления элементов в удаляемом стеке неважен;
- удаляемый стек может содержать значения элементов, не существующих в стеке, из которого выполняется удаление;
- элементы удаляемого стека со значениями несовпадающими со значениями стека, из которого происходит удаление также могут располагаться в стеке в произвольном порядке;
- удаляемый стек может иметь большую размерность, чем стек, из которого будет происходить удаление; например, полностью содержать в себе в произвольном порядке удаляемый стек; результатом подобной операции будет пустой стек.

Метод `remove()` интерфейса `Iterator`

В коллекции `Stack` появились свои собственные методы `peek()`, `pop()`, `push()`. В дополнение к ним можно использовать интерфейс `Iterator` и его методы такие как `hasNext()`, `next()` и дефолтный метод `remove()`.

Представляется интересным продемонстрировать алгоритмы обработки `Stack`-а с помощью перечисленных выше методов.

Приведем пример методов, осуществляющих циклический сдвиг значений `Stack`-а вправо и лево на один элемент, а также метода осуществляющего реверс стека.

Пример.

```
1  import java.util.*;
2  public class Program {
3      //заполнение целыми числами числами по возрастанию
4      static Stack intGenerStack(int n) {
5          Stack s = new Stack();
6          for(int i = 0; i < n; i++) {
7              s.push(i + 1);
8          }
9          s.push(12.56);
10         s.push(new Boolean(false));
11         return s;
12     }
13     //циклический сдвиг влево на одно значение
14     static void shiftLeft(Stack s) {
15         Iterator iter = s.iterator();
16         var temp = iter.next();
```

```
17          iter.remove();
18          s.push(temp);
19      }
20      //циклический сдвиг вправо на одно значение
21      static void shiftRight(Stack s) {
22          Stack temporary = (Stack) s.clone();
23          s.clear();
24          s.push(temporary.pop());
25          Iterator iter = temporary.iterator();
26          while(iter.hasNext()) {
27              s.push(iter.next());
28          }
29      }
30      //реверс стека
31      static void reverse(Stack s) {
32          Stack temporary = (Stack) s.clone();
33          s.clear();
34          Iterator iter = temporary.iterator();
35          while(iter.hasNext()) {
36              s.push(temporary.pop());
37          }
38      }
39
40      public static void main (String args[]) {
41          //заполнение вектора целыми значениями
42          Stack s = intGenerStack(5);
43          System.out.println("Исходный стек");
44          System.out.println(s.toString());
45          System.out.println("Циклический сдвиг влево");
46          shiftLeft(s);
47          System.out.println(s.toString());
48          System.out.println("Циклический сдвиг вправо");
49          shiftRight(s);
50          System.out.println(s.toString());
51          System.out.println("Реверс стека");
52          reverse(s);
53          System.out.println(s.toString());
54      }
55  }
```

Результат работы программы:

```
Исходный стек
[1, 2, 3, 4, 5, 12.56, false]
```

```
Циклический сдвиг влево
[2, 3, 4, 5, 12.56, false, 1]
Циклический сдвиг вправо
[1, 2, 3, 4, 5, 12.56, false]
Реверс стека
[false, 12.56, 5, 4, 3, 2, 1]
```

Замечания:

Методы, осуществляющие сдвиг и реверс (`shiftLeft()`, `shiftRight()`, `reverse()`) полностью параметризованны и с их помощью можно осуществлять сдвиг не только в `Stack`-е, содержащем базовые типы, но и объекты пользовательских классов.

Статические методы для работы со стеком, содержащим значения базовых типов

Рассмотрим пример определения статических пользовательских методов для работы с коллекцией `Stack`.

Пример.

```
1  import java.util.*;
2  public class Program {
3     //Первая реализация метода заполнения Stack-а
4     //случайными целыми числами
5     static void generStack(Stack s,
6                            final int MIN,
7                            final int MAX,
8                            int n) {
9       for(int i = 0; i < n; i++) {
10        s.push((new Double(Math.random() *
11                  (MAX - MIN) + MIN)).intValue());
12      }
13    }
14    //Перегрузка метода: создание + заполнение стека
15    //случайными числами с плавающей точкой
16    static Stack generStack(final int MIN,
17                            final int MAX,
18                            int n) {
19      Stack s = new Stack();
20      for(int i = 0; i < n; i++) {
21        s.add(Math.random() * (MAX - MIN) + MIN);
22      }
```

```
23          return s;
24      }
25
26      public static void main (String args[]) {
27          int n = 5;
28          //выделение памяти под коллекцию
29          Stack s = new Stack();
30          final int MIN = 0;  //нижняя граница генерации
31          final int MAX = 10; //верхняя граница генерации
32          //вызов первой реализации метода
33          //generationArray()
34          generStack(s, MIN, MAX, n);
35          System.out.println(s.toString() + "\n");
36          //вызов перегруженного метода, возвращающего в
37          //качестве значения заполненную коллекцию Stack
38          System.out.println(generStack(MIN, MAX, n));
39      }
40  }
```

Результат работы программы:

```
[8, 4, 7, 3, 0]

[4.9722391126959, 8.44201953420851, 1.6736508060605915, 7.698923841443715, 6.720785208541729]
```

В последнем примере, параметризованный стек может принимать различные типы значений. Однако в перегруженных методах `generStack()` типы заполняемых чисел указывается жестко. Это делается на этапе заполнения использованием (или неиспользованием в перегруженном методе) метода `intValue()`, приводящего значение объекта любого типа наследника `Number` к целому числу.

По сути, как и в случае с `Vector`-ом предыдущий пример не использует параметризацию `Stack`-а, что существенно снижает полноту изложения.

Продемонстрируем пример более профессионального использования параметризации `Stack`-а, т.е. создадим возможность заполнения `Stack`-а значениями, типы которых *могут* определяться не на этапе компиляции, а во время выполнения программы типами значений `MIN` и `MAX`.

Замечание.

Слово «могут» указывает, как и раньше, на возможность реализации с приведенными ниже методами (без их изменения) работы с разными типами данных в зависимости от типа вводимых с консоли значений.

Например, продемонстрируем самую простую (без использования параметризации) возможность реализации подобного подхода (изменения типа заполняющих значений) с перегрузкой второго варианта метода `generStack()`, приведенного выше.

Пример.

```
1  import java.util.*;
2  public class Program {
3     //первый вариант метода с возвращаемым типом void
4     static void generStack(Stack s,
5                            final int MIN,
6                            final int MAX,
7                            int n) {
8       for(int i = 0; i < n; i++) {
9         s.push((new Double(Math.random() *
10                    (MAX - MIN) + MIN)).intValue());
11      }
12    }
13    //перегрузка с другими типами входных параметров
14    static void generStack(Stack s,
15                           final double MIN,
16                           final double MAX,
17                           int n) {
18      for(int i = 0; i < n; i++) {
19        s.add(Math.random() * (MAX - MIN) + MIN);
20      }
21    }
22
23    public static void main (String args[]) {
24       Stack s = new Stack();
25      //заполнение стека целыми значениями
26      final int MIN1 = 0;  //нижняя граница генерации
27      final int MAX1 = 10; //верхняя граница генерации
28      generStack(s, MIN1, MAX1, 5);
29      System.out.println(s);
30      //очищение стека
31      s.clear();
32      //заполнение стека значениями с плав. точкой
33      final double MIN2 = 0;
34      final double MAX2 = 10;
35      generStack(s, MIN2, MAX2, 5);
36      System.out.println(s);
37    }
38 }
```

Пример работы программы:

```
[8, 4, 8, 6, 5]
[7.580837994536839, 4.364983032917705, 0.3493584986855647, 6.6364798682916, 2.5084715681571246]
```

Однако продемонстрированный вариант – это больше «силовое» решение вопроса с требованием универсальности заполнения `Stack`-а любыми типами данных.

Как и ранее в коллекции `Vector`, если ввести параметризацию на параметры `MAX` и `MIN`, то возникнет проблема с тем, что с параметризованными типами не переделены арифметические операции (имеется в виду выражение `Math.random() * (MAX – MIN) + MIN`) после этого выясниться, что методы `push()` и `add()` коллекции `Stack` также не могут работать с параметризованными типами.

Соответственно необходимо организовать преобразование параметризованного типа хранимого в `Stack` данного к ***не***параметризованному. Это выполнить с учетом естественной вариативности изменения типа входного параметра.

Пример.

```
1  import java.util.*;
2  //класс-обертка для приведения параметра типа к типу
3  //double
4  class CastDouble<T extends Number> {
5     private double x;
6     //параметризованный конструктор
7     CastDouble (T x) {
8         this.x = x.doubleValue();
9     }
10    //геттер
11    double get() { return x; }
12 }
13
14 public class Program
15 {
16     static <T extends Number> Double valueRandom(T a,
17                                                  T b) {
18            CastDouble aCast = new CastDouble(a);
19            CastDouble bCast = new CastDouble(b);
20            return Math.random() * (bCast.get() -
21                                    aCast.get()) + aCast.get();
22     }
```

```
23
24      static <T extends Number>
25                  void generStack(Stack s,
26                                  final T MIN,
27                                  final T MAX,
28                                  int n) {
29          if(MIN instanceof Double &
30                                      MAX instanceof Double ) {
31              for(int i = 0; i < n; i++) {
32                  s.push(valueRandom(MIN, MAX));
33              }
34          }
35          else if (MAX instanceof Integer &
36                                     MIN instanceof Integer ) {
37              for(int i = 0; i < n; i++) {
38                  s.add(valueRandom(MIN, MAX).intValue());
39              }
40          }
41          else {
42              //исключение: типы MAX и MIN не совпадают
43              try {
44                  throw new Exception("типы MIN и MAX " +
45                                      "не совпадают");
46              }
47              catch(Exception ex) {
48                  System.out.println(ex.getMessage());
49              }
50          } //конец else
51      }
52
53      public static void main (String args[]) {
54          Stack s = new Stack();
55          //заполнение стека целыми значениями
56          final int MIN1 = 0;  //нижняя граница генерации
57          final int MAX1 = 10; //верхняя граница генерации
58          generStack(s, MIN1, MAX1, 5);
59          System.out.println(s);
60          //очистка стека
61          s.clear();
62          //заполнение стека значениями с плав. точкой
63          final double MIN2 = 0;
64          final double MAX2 = 10;
65          generStack(s, MIN2, MAX2, 5);
66          System.out.println(s);
```

```
67          //вызов исключительной ситуации
68          generStack(s, 0.5, 10, 4);
69      }
70  }
```

Результат работы программы:

```
[4, 4, 2, 6, 9]
[5.513570381845386, 4.012510283440935, 7.581400964255193, 6.7178567887399225, 2.8911384991817113]
типы MIN и MAX не совпадают
```

Замечания:

- *в отличие от метода приведенного в разделе Vector в данном случае стек приходит в качестве одного из параметров для заполнения, а сама коллекция создается в методе* `main();`
- *обращает на себя внимание синтаксис параметра* `s`*, являющегося в методе объектом коллекции* `Stack`*. Хотя метод имеет параметр, но тип объекта* `s` *явно не параметризованн и если попытаться выполнить явную параметризацию (*`Stack<T> s`*), то компилятор воспримет его как второй параметр* `T2`*.*

Пользовательские классы в параметрах `Stack`

Следует отметить, что исходя из примера, приведенного ниже, универсальность алгоритмов обработки достигается только тогда, когда для класса используемого в качестве параметра в коллекции переопределены все необходимые методы (ниже таким методами является метод `toString()` для записи).

Пример.

```
1   import java.util.*;
2
3   record Person(String name, int age) { }
4
5   public class Program
6   {
7       static <T> Stack generStack(T[] aInit) {
8           Stack s = new Stack();
9           for(int i = 0; i < aInit.length; i++) {
10              s.push(aInit[i]);
11          }
```

```
12          return s;
13      }
14
15      static void displayColumn(Stack s) {
16          for(var element : s) {
17              System.out.println(element.toString());
18          }
19      }
20
21      public static void main (String args[]) {
22          //создание инициализирующего массива объектов
23          Person[] a = {new Person("Том", 36),
24                        new Person("Боб", 36),
25                        new Person("Сергей", 36) };
26          //вывод вектора, содержащего объекты
27          //пользовательского класса
28          displayColumn(generStack(a));
29          System.out.println();
30          //второй вариант вывода Vector-а,
31          //сформированного из массива Person
32          System.out.println(generStack(a));
33      }
34  }
```

Результат работы программы:

```
Person[name=Том, age=36]
Person[name=Боб, age=36]
Person[name=Сергей, age=36]

[Person[name=Том, age=36], Person[name=Боб, age=36], Person[name=Сергей, age=36]]
```

Замечание.

Если для любого объекта определен метод `toString()`*, то при консольном выводе на экран при помощи методов* `print()` *и* `println()` *компилятор «по умолчанию» дополняет параметр передаваемый в эти методы вызовом соответствующего данному объекту метода* `toString()`.

Создание стека из стеков

Элементами стека могут быть любые коллекции, в частности сами стеки. Например, можно сделать стек, каждый элемент которого представляет собой стек целых чисел:

```
Stack<Stack> имяОбъекта = new Stack();
```

Следует отметить, что при работе со стеком стеков можно целиком постеково заполнять стек стеков.

Для первичного и упрощенного знакомства с правилами работы со стеком стеков внесем очевидные исправления в уже рассмотренный аналогичный пример для двумерных векторов.

Пример.

```
1  import java.util.*;
2
3  record Person(String name, int age) { }
4
5  public class Program
6  {
7      static Stack<Stack> generStack(Person[] aInit){
8          Stack<Stack> s = new Stack();
9          for(int i = 0; i < aInit.length; i++) {
10                 //постековая инициализация стека стеков
11                 Stack temporary = new Stack();
12                 temporary.push(aInit[i].name());;
13                 temporary.push(aInit[i].age());
14                 //постековое присвоение стеку стеков
15                 s.add(temporary);
16         }
17         return s;
18     }
19
20     static void displayColumn(Stack<Stack> s) {
21         Iterator iter = s.iterator();
22         while(iter.hasNext()) {
23             System.out.println(iter.next().toString());
24         }
25     }
26
```

```
27      public static void main (String args[]) {
28        //создание массива объектов
29        Person[] a = {new Person("Том", 36),
30                      new Person("Боб", 36),
31                      new Person("Сергей", 36) };
32        //построчный вывод двумерного вектора
33        displayColumn(generStack(a));
34        System.out.println();
35        //второй вариант вывода двумерного Vector-a,
36        System.out.println(generStack(a).toString());
37      }
38  }
```

Результат работы программы:

```
[Том, 36]
[Боб, 36]
[Сергей, 36]

[[Том, 36], [Боб, 36], [Сергей, 36]]
```

Следующий пример демонстрирует насколько ***не***удобно в Java работать со стеком стеков только с использованием методов `push()`, `peek()` и `pop()` без привлечения методов базового класса `Vector`.

Замечание.

В связи с использованием только методов `push()`, `peek()` и `pop()` в нижеприведенном примере транспонирование двумерного массива, хранимого в стеке из стеков сведено к двум операциям:

- *копирование в обратном порядке строк в столбцы вспомогательного массива;*
- *реверс с удалением объектов стеков-строк из вспомогательного стека при копировании в предварительно очищенный исходный стек.*

Пример.

```
1   import java.util.*;
2   public class Program {
3      //создания и заполнение стека стеков
4      static Stack<Stack> initStackOfStacks(int n) {
5        Stack<Stack> s = new Stack();
6        for(int i = 0; i < n; i++) {
7          //формирование вспомогательного стека
8          Stack tempStack = new Stack();
```

```
 9            int j;
10            for(j = 0; j < n - i - 1; j++) {
11               tempStack.push(j + 1);
12            }
13            tempStack.push(i + 1);
14            for(j = j + 1; j < n; j++) {
15               tempStack.push(0);
16            }
17            //создание основного стека (из вспом. стеков)
18            s.push(tempStack);
19         }
20         return s;
21      }
22      //вывод на экран в виде матрицы стека стеков
23      static void display(Stack<Stack> stackOfStacks) {
24         Iterator iter = stackOfStacks.iterator();
25         while(iter.hasNext()) {
26            System.out.println(iter.next().toString());
27         }
28      }
29      //транспонирование матрицы, хранимой в виде стека
30      //стеков
31      static void transposition(Stack<Stack> stackOfStacks){
32         //создание вспомогательного стека стеков для
33         //хранения результата транспонирования
34         Stack<Stack> result = new Stack();
35         //создание итератора по стеку нижнего уровня
36         //элементам в последней строке, чтобы было без
37         //for()
38         Iterator iter = stackOfStacks.peek().iterator();
39         while(iter.hasNext()) {
40            Stack temporary = new Stack();
41            //перебор стеков верхнего уровня (строки)
42            for(var stack : stackOfStacks) {
43               temporary.push(stack.pop());
44            }
45            result.push(temporary);
46         }
47         //предварительная очистка памяти, где хранился
48         //исходный стек стеков. После применения метода
49         //pop() он уже пуст, но имеет вид стека пустых
50         //стеков и если его начать заполнять стеками, то
51         //пустые стеки будут дополняться, что не
52         //соответствует целям задачи
53         stackOfStacks.clear();
```

```
54          //создание зеркальной копии result по ссылке
55          //stackOfStacks
56          Iterator it = result.iterator();
57          while(it.hasNext()) {
58              stackOfStacks.push(result.pop());
59          }
60      }
61
62      public static void main (String args[]) {
63          int n = 3;
64          System.out.println("Исходная матрица");
65          Stack<Stack> ss = initStackOfStacks(n);
66          display(ss);
67          System.out.println("Результат транспонирования");
68          transposition(ss);
69          display(ss);
70      }
71  }
```

Результат работы программы:

```
Исходная матрица
[1, 2, 1]
[1, 2, 0]
[3, 0, 0]
Результат транспонирования
[1, 1, 3]
[2, 2, 0]
[1, 0, 0]
```

Замечание.

В связи с тем, что в последнем примере ставилась цель отказаться от методов коллекции `Vector`*, а использовать лишь методы* `push()`*,* `peek()` *и* `pop()`*, то было установлено, что в этом случае в дополнение к последним перечисленным методам необходимо использовать еще и методы* `clear()`*,* `iterator()` и `hasNext()`*.*

Кроме того целенаправленное игнорирование методов базового класса `Vector` *приводит к тому, что внутри метода* `transposition()` *приходится заводить дополнительный объект стека стеков, чтобы выполнить требуемые алгоритмические преобразования.*

Очень часто при решении конкретной задачи самое главное - выбор наиболее подходящей для этого структуры данных. При выборе стека в качестве инструмента при решении следует помнить, что отказавшись от методов родительского класса `Vector()` таких как `elementAt()` и `contains()` нельзя будет ни получить доступ к произвольному элементу

стека, ни проверить входит ли элемент в стек, как у списков. Пользуясь исключительно методами этой коллекции можно только добавлять или удалять элементы в определенном порядке.

Стеки как поля классов

Рассмотрим случай, когда свойством класса являются `Stack`. В рассмотренном ниже примере в классе в качестве свойств создается два `Stack`-а:

- имена городов;
- почтовые индексы.

Эти два стека связаны между собой посредством соответствия порядков перечисления. Методы класса позволяют организовать поиск индекса города по его названию в создаваемом объекте.

Замечание.

По возможности не будем использовать методы, представляемые суперклассом `Vector`, а постараемся обойтись только методами `Stack`-а и интерфейса `Iterator`, также будет использоваться метод `equals()` класса `String` из-за проверки на совпадение строк.

Пример.

```
1  import java.util.*;
2  class Matcher {
3      private Vector<String> name = new Vector();
4      private Vector<Integer> postCode = new Vector();
5      //блок инициализации
6      {
7          name.add("Минск");
8          name.add("Брест");
9          name.add("Витебск");
10         postCode.add(220000);
11         postCode.add(224000);
12         postCode.add(210000);
13     }
14     //методы
15     int finedIndex(String cityName){
16         for(int i = 0; i < name.size(); i++) {
17             if(cityName.equals(this.name.get(i))) {
18                 return i;
19             }
20         }
```

```
21          return -1;
22      }
23      int finedPostCode(String cityName) {
24          int i = finedIndex(cityName);
25          if (i < 0) return i;
26          return this.postCode.get(i);
27      }
28  }
29
30  public class Main
31  {
32      public static void main(String[] args) {
33          Matcher data = new Matcher ();
34          System.out.print(data.finedPostCode("Минск"));
35      }
36  }
```

Результат работы программы:

```
220000
```

Коллекция `ArrayList`

В качестве альтернативы для `Vector` часто применяется аналог — `ArrayList` также реализующий интерфейс `List`.

`ArrayList` — как и `Vector` является реализацией динамического массива объектов. Позволяет хранить ссылки на любые объекты, включая `null` в качестве элемента. Как можно догадаться из названия, его реализация основана на обычном массиве. Данную реализацию следует применять, если в процессе работы с коллекцией предполагается частое обращение к элементам по индексу (например, применение сортировок). Но данную коллекцию рекомендуется избегать, если требуется частое удаление/добавление элементов в середину коллекции.

Свойства коллекции:

- быстрый доступ к элементам *по индексу* за время `O(1)`;
- доступ к элементам *по значению* за линейное время `O(n)`;
- медленный, когда вставляются и удаляются элементы из «середины» списка;
- позволяет хранить любые значения в том числе и `null`;
- не синхронизирован (для многопоточности, т.е. разрешен доступ из разных потоков).

Коллекция поддерживает три конструктора со следующим синтаксисом:

- `ArrayList()` - создает список «по умолчанию». Хотя коллекция позволяет добавлять элементы в середину объекта этой коллекции, однако в реальности `ArrayList` использует для хранения объектов опять же массив. «По умолчанию» данный массив предназначен для хранения `10` объектов, проинициализированных значением `null`.

Пример.

```
ArrayList<ТипЗначения> a = new ArrayList();
```

- `ArrayList(int capacity)` - создает список (массив) с указанной начальной `capacity`. Если в процессе программы добавляется больше, чем `10`, то создается новый массив, который может вместить в себя все количество. Подобные перераспределения памяти уменьшают производительность. Поэтому если точно известно, что список не будет содержать больше определенного количества элементов, например, `25`, то можем сразу же явным образом установить это количество, либо в конструкторе.

Пример.

```
ArrayList<ТипЗначения> a = new ArrayList(25);
```

- `ArrayList(Collection<? extends E> c)` - создает список массивов, который инициализируется элементами коллекции c.

Пример.

```
ArrayList<ТипЗначения> a =
                          new ArrayList(Vector c);
```

Пример.

```
1  import java.util.*;
2  public class Main
3  {
4      public static void main(String args[]) {
5          ArrayList a =
6              new ArrayList(Arrays.asList(1, "два", true, 4));
7          System.out.println(a);
8          //массив для инициализации списка
```

```
        String[] aStr =
                {"раз", (new Integer(2)).toString(), "три"};
        a = new ArrayList(Arrays.asList(aStr));
        System.out.println(a);

    }
}
```

Результат работы программы:

```
[1, два, true, 4]
[раз, 2, три]
```

Замечание.

*Напомним емкость – это текущий размер памяти (необязательно заполненной значениями), выделенной под хранение массива с текущим количеством хранимых элементов. Пустая коллекция тоже имеет **не**нулевую емкость, хотя нулевой размер. Емкость растет автоматически по мере добавления элементов в список массивов.*

Некоторые методы коллекции

Коллекция `ArrayList` реализует методы интерфейса `List`. Кроме перечисленных ранее можно указать, что существует еще ряд интересных экземплярных методов. Например, метод `Object[] toArray()` - возвращает в качестве значения массив, содержащий все элементы в этом списке в правильном порядке. И метод `public <T> T[] toArray(T[] array)`, который возвращает массив элементов того типа, массив элементов какого тип был передан в метод. Он может вызывать исключительную ситуацию `NullPointerException`, если массив `array`, передаваемый в метод имеет значение `null`, или `ArrayStoreException` - если тип массива (в момент выполнения программы) не является супертипом типа каждого элемента в этом списке.

Особенностью этого метода является то, что можно не использовать возвращаемое значение, т.к. массив вернется по ссылке `array`, передаваемой в метод.

Пример.

```
import java.util.*;
public class Main
{
    public static void main(String args[]) {
        ArrayList a = new ArrayList();
```

```
6       System.out.println("Начальный размер: " +
7                          a.size());
8
9       a.add(new Integer(1));
10      a.add(new Double(2));
11      a.add("Строка");
12      a.add(4.56);
13      System.out.println("Размер ArrayList " +
14                         "после четырех добавлений: " +
15                         a.size());
16      System.out.println("Массив: ");
17      for(int i = 0 ; i < a.size() ; i++) {
18         System.out.print(a.get(i) + " ");
19      }
20      System.out.println();
21      a.set(2, new Integer(10));
22      a.set(3, 0);
23      a.set(1, 20);
24      //измененный массив
25      System.out.println("Измененый массив");
26      System.out.println(a);
27      System.out.println("Последний элемент: " +
28                         a.get(a.size() - 1));
29
30      if(a.contains(new Integer(10))) {
31         System.out.println("Вектор содержит число 10");
32      }
33      //вспомогательный массив для преобразования
34      //коллекции с помощью метода toArray()
35      Integer[] resultArray = new Integer[a.size()];
36      //первый вариант синтаксиса использования метода
37      //toArray()
38      a.toArray(resultArray);
39      for (var element : resultArray) {
40            System.out.print(element + " ");
41      }
42      //второй вариант синтаксиса использования метода
43      //toArray()
44      System.out.println();
45      Object[] obj = a.toArray(resultArray);
46      System.out.print(Arrays.toString(obj));
47      //или с преобразованием типов
48      System.out.println();
49      resultArray = (Integer[]) a.toArray(resultArray);
```

```
        for (var element : resultArray) {
            System.out.print(element + " ");
        }
    }
}
```

Результат работы программы:

```
Начальный размер: 0
Размер ArrayList после четырех добавлений: 4
Массив:
1 2.0 Строка 4.56
Измененый массив
[1, 20, 10, 0]
Последний элемент: 0
Вектор содержит число 10
1 20 10 0
[1, 20, 10, 0]
1 20 10 0
```

Кроме того, существует *защищенный* метод коллекции `ArrayList`. Это **`protected`** `void` **`removeRange`**`(int fromIndex, int toIndex)` - удаляет из этого списка все элементы, индекс которых находится между `fromIndex`, включительно, и `toIndex`, исключительно.

Очевидно, у разработчика возникает вопрос, а как можно применить данный метод в его программе. Ответ достаточно прост: класс, создаваемый пользователем и содержащий программу разработчика должен *наследовать* `ArrayList`.

Пример.

```
import java.util.*;
public class Main extends ArrayList {
    public static void main(String[] args)
    {
        Main arr = new Main();
        arr.add(1.5);
        arr.add("Строка");
        arr.add(3);
        arr.add(new Boolean("true"));
        System.out.println("Список до удаления " +
                           arr);
        //использование protected метода
        arr.removeRange(0, 2);
```

```
14          System.out.println("Список после удаления " +
15                                  arr);
16      }
17  }
```

Результат работы программы:

```
Список до удаления [1.5, Строка, 3, true]
Список после удаления [3, true]
```

Методы интерфейса `ListIterator`

Интерфейс `ListIterator` расширяет интерфейс `Iterator` и определяет ряд дополнительных методов коллекциям, реализующих интерфейс `List`:

- `void` **`add`**`(E obj)` - вставляет объект `obj` после элемента, на который указывает последний вызов `next()`;
- `boolean` **`hasNext`**`()` - возвращает `true`, если в коллекции имеется текущий элемент, возвращаемый методом `next()`, на который указывает итератор, иначе возвращает `false`;
- `boolean` **`hasPrevious`**`()` - возвращает `true`, если в коллекции имеется текущий элемент, возвращаемый методом `previous()`, на который указывает итератор, иначе возвращает `false`;
- `E` **`next`**`()` - возвращает текущий элемент и переходит к следующему, если такого нет, то генерируется исключение `NoSuchElementException`;
- `E` **`previous`**`()` - возвращает текущий элемент и переходит к предыдущему, если такого нет, то генерируется исключение `NoSuchElementException`;
- `int` **`nextIndex`**`()` - возвращает индекс текущего элемента, на который смотрит итератор (результат последнего обращения к списку методом `next()`). Если такого нет, то возвращается размер списка;
- `int` **`previousIndex`**`()` - возвращает индекс текущего элемента, на который смотрит итератор (результат последнего обращения к списку методом `previous()`). Если такого нет, то возвращается число `-1`;

- `void` **`remove`**`()` - удаляет текущий элемент из списка. Таким образом, этот метод должен быть вызван после методов `next()` или `previous()`, иначе будет сгенерировано исключение `IlligalStateException`;
- `void` **`set`**`(E obj)` - присваивает текущему элементу, выбранному вызовом методов `next()` или `previous()`, ссылку на объект `obj`.

Замечание.

*Наличие методов `E` **`previous`**`()`, `int` **`previousIndex`**`()` и `boolean` **`hasPrevious`**`()` обеспечивает обратную навигацию по списку, поэтому интерфейс `ListIterator` предоставляет методы работы с двусвязным списком, тогда как интерфейс `Iterator` с односвязным.*

Пример.

```
1  import java.util.*;
2  public class Main
3  {
4      public static void main(String[] args) {
5          ArrayList states = new ArrayList();
6          states.add("Germany");
7          states.add("France");
8          states.add("Italy");
9          states.add("Spain");
10         //использование методов интерфейса ListIterator
11         //для объекта states типа ArrayList
12         ListIterator listIter = states.listIterator();
13         while(listIter.hasNext()) {
14             System.out.println(listIter.next());
15         }
16         System.out.println();
17         // сейчас текущий – последний элемент Spain
18         // изменим значение этого элемента
19         listIter.set("Португалия");
20         System.out.println(listIter.nextIndex());
21        //добавим еще одни элемент
22         listIter.add("Беларусь");
23         //выведем результирующий список на экран
24         System.out.println(states);
25         System.out.println();
26         // пройдемся по элементам в обратном порядке
```

```
27          while(listIter.hasPrevious()) {
28              System.out.println(listIter.previous());
29          }
30      }
31  }
```

Результат работы программы:

```
Germany
France
Italy
Spain

4
[Germany, France, Italy, Португалия, Беларусь]

Беларусь
Португалия
Italy
France
Germany
```

Статические методы для работы с коллекцией `ArrayList`

Рассмотрим пример определения статических пользовательских методов для работы с коллекцией `ArrayList`.

Сразу перейдем к примеру использования параметризации `ArrayList`-а, т.е. предусмотрим возможность заполнения `ArrayList`-а значениями, типы которых *могут* определяться не на этапе компиляции, а во время выполнения программы.

Замечание.

Слово «могут» указывает на то, что если в примере использовать консоль и предусмотреть ввод разных типов чисел, то приведенные примеры все равно будут работоспособны.

Продемонстрируем реализацию для объектов коллекции `ArrayList` самой простой реализации подобного подхода (изменения типа заполняющих значений) с перегрузкой метода, заполняющего объект коллекции. Эта перегрузка позволит заполнить созданный `ArrayList`

различными типами значений, определимых типами введенных значений MIN и MAX диапазона, генерируемых величин.

Дополнительно приведем статический метод, позволяющий работать с итераторами интерфейса listIterator.

Пример.

```
import java.util.*;
public class Program {
   //первый вариант метода
   static void listArrayGen(ArrayList arrList,
                            final int MIN,
                            final int MAX,
                            int n) {
     for(int i = 0; i < n; i++) {
       arrList.add((new Double(Math.random() *
                   (MAX - MIN) + MIN)).intValue());
     }
   }
   //перегрузка с другими типами входных параметров
   static void listArrayGen(ArrayList arrList,
                            final double MIN,
                            final double MAX,
                            int n) {
     for(int i = 0; i < n; i++) {
        arrList.add(Math.random() * (MAX - MIN) + MIN);
     }
   }
   //метод работающие с итераторами listIterator
   static void display(ArrayList arrList,
                       StringBuffer directEnumer,
                       StringBuffer reverseEnumer) {
      ListIterator it = arrList.listIterator();
      while(it.hasNext()) {
         directEnumer.append(it.next().toString() +
                             " ");
      }
      while(it.hasPrevious()) {
         reverseEnumer.append(it.previous().toString() +
                              " ");
      }
   }

   public static void main (String args[]) {
     ArrayList arrList = new ArrayList();
     StringBuffer directEnumer = new StringBuffer(" ");
     StringBuffer reverseEnumer = new StringBuffer(" ");
```

```
41          //заполнение вектора целыми значениями
42          final int MIN1 = 0;  //нижняя граница генерации
43          final int MAX1 = 10; //верхняя граница генерации
44          System.out.println("Целочисленный ArrayList");
45          listArrayGen(arrList, MIN1, MAX1, 5);
46          System.out.println(arrList);
47          System.out.println("Работа с ListIterator");
48          display(arrList, directEnumer, reverseEnumer);
49          System.out.println(directEnumer);
50          System.out.println(reverseEnumer);
51          //участок чистки от старых значений строк-буферов и
52          //объекта arrList коллекции ArrayList, чтобы к
53          //старым значениям не добавились новые
54          directEnumer.delete(0, directEnumer.capacity());
55          reverseEnumer.delete(0, directEnumer.capacity());
56          arrList.clear();
57          //заполнение вектора значениями с плав. точкой
58          final double MIN2 = 0;
59          final double MAX2 = 10;
60          System.out.println("ArrayList с плавающей точкой");
61          listArrayGen(arrList, MIN2, MAX2, 5);
62          System.out.println(arrList);
63          System.out.println("Работа с listIterator");
64          display(arrList, directEnumer, reverseEnumer);
65          System.out.println(directEnumer);
66          System.out.println(reverseEnumer);
67      }
68  }
```

Пример работы программы:

```
Целочисленный ArrayList
[1, 1, 5, 6, 1]
Работа с ListIterator
 1 1 5 6 1
 1 6 5 1 1
ArrayList с плавающей точкой
[5.74638408508581, 6.242267486434539, 9.018940465622, 3.493759957990795, 0.4082497331028945]
Работа с listIterator
5.74638408508581 6.242267486434539 9.018940465622 3.493759957990795 0.4082497331028945
0.4082497331028945 3.493759957990795 9.018940465622 6.242267486434539 5.74638408508581
```

Замечания:

- *метод* `display()` *является полностью параметризованным, но явно параметр типа не указывается. Этот метод выводит значения любых типов, для которых переопределен метод* `toString()` *из коллекции* `ArrayList`*;*

- *пример демонстрирует, что метод может возвращать через параметры-ссылки сколько угодно значений (в частности, в примере - два). Тип `String` является `Immutable` и каждое добавление в строку приводит к созданию новой строки поэтому более удобно использовать `StringBuffer`.*

Приведенный пример – это больше силовое решение вопроса с требованием универсальности заполнения `ArrayList`-а любыми типами данных.

Возможность параметризации перегруженного метода `listArrayGen()` уже очевидна исходя из материала, приведенного для предыдущих коллекций.

Поэтому читатель может совершенно самостоятельно с помощью замены коллекции `Vector` (или `Stack`) на коллекцию `ArrayList` предложить вариант параметризации метода `listArrayGen` в зависимости от типов передаваемых параметров в этот метод, и приводить данный пример еще раз не имеет смысла.

Примеры экземплярных методов обработки `ArrayList`

Поэтому представляется интересным продемонстрировать алгоритмы обработки `ArrayList`-а с их помощью методов представляемых интерфейсом `ListIterator`.

В качестве примера использования метода `remove()` интерфейса `ListIterator` приведем пример экземплярных методов, осуществляющих циклический сдвиг значений `ArrayList`-а влево на один элемент.

Замечание.

В примере ниже создается `ArrayList`, заполненный случайными целыми числами. Сдвиг влево осуществляется методами интерфейса `ListIterator`. Использование данных методов еще неудобнее, чем методы `push()`, `pop()` и `peek()` коллекции `Stack`.

Пример.

```
1  import java.util.*;
2  public class Program {
3      //заполнение целыми числами числами по возрастанию
4      void intGenerArrayList(ArrayList a, int n) {
```

```
5       for(int i = 0; i < n; i++) {
6           a.add(i + 1);
7       }
8   }
9   //циклический сдвиг влево на одно значение
10  void shiftLeft(ArrayList a) {
11      ListIterator iter = a.listIterator();
12      var temp = iter.next();
13      iter.remove();
14      while(iter.hasNext()){
15          iter.next();
16      }
17      iter.add(temp);
18  }
19
20  public static void main (String args[]) {
21      //заполнение вектора целыми значениями
22      Program ex = new Program();
23      ArrayList a =  new ArrayList();
24      ex.intGenerArrayList(a, 5);
25      System.out.println("Исходный стек");
26      System.out.println(a.toString());
27      System.out.println("Циклический сдвиг влево");
28      ex.shiftLeft(a);
29      System.out.println(a.toString());
30  }
31  }
```

Результат работы программы:

```
Исходный стек
[1, 2, 3, 4, 5]
Циклический сдвиг влево
[2, 3, 4, 5, 1]
```

Пример хранения двумерного массива в простой коллекции `ArrayList`

В данном пункте демонстрируется как можно двумерный массив (как, собственно, и массив произвольной размерности) сохранить в одномерном. Однако в случае размерности больше двух существует вероятность запутаться с индексами при сложных преобразованиях.

Пример основан на использовании медленного и быстрого индекса. Если известно, что двумерный массив в виде `ArrayList`-а поступает на вход и является квадратным, то тогда количество строк и количество столбцов совпадают. Будем считать, что количество строк будет храниться в переменной `n`. Эта величина будет определяться по формуле:

$$n = \sqrt{a.size()},$$

где коллекция типа `ArrayList` имеет имя `a`.

Пусть, как и раньше индекс `i` определяет индекс строки, а индекс `j` – номер столбца.

Будем считать, что все действия с массивом производятся по срокам, т.е. цикл по `i` является внешним, а переменная - медленной, тогда цикл по `j` является внутренним, а переменная - быстрой. В этом случае результирующий индекс элемента в одномерной коллекции `ArrayList` симулирующей двумерную коллекцию определяется по формуле (Рисунок 3):

$$i \cdot n + j.$$

Сама идея хранения двумерного массива в одномерном состоит в том, что двумерный массив сохраняется построчно в одномерном массиве (Рисунок 3).

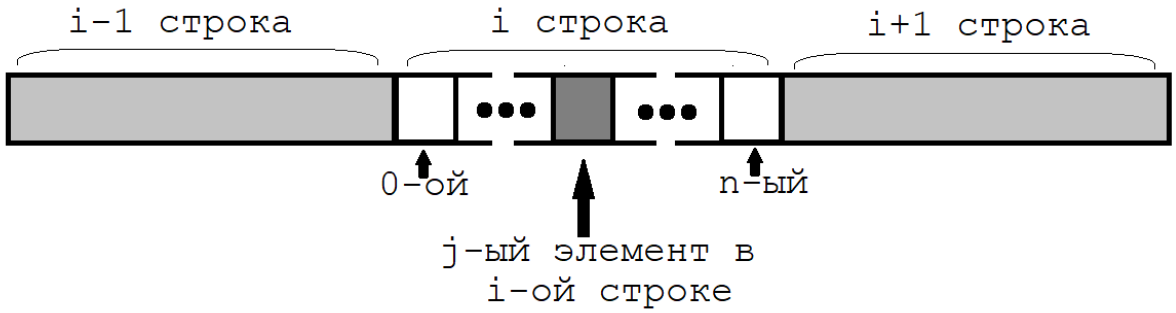


Рисунок 3 – Схема хранения двумерного массива в одномерном

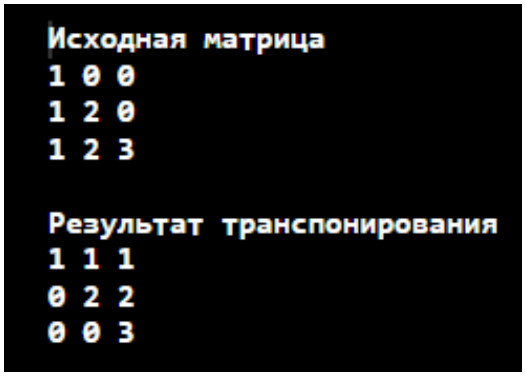
Таким образом обратиться к элементу матрицы $m[i, j]$ сохраненному в виде объекта `a` одномерной коллекции можно с помощью синтаксической конструкции $a[i \cdot n + j]$.

Пример.

```
1  import java.util.*;
2  public class Program
3  {
4      void setMatrixSimulation(ArrayList a, int n) {
5          for(int i = 0; i < n; i++) {
6              for(int j = 0; j < n; j++) {
```

```
            if(i >= j) a.add(i * n + j, j + 1);
            else a.add(i * n + j, 0);
         }
      }
   }

   String outString(ArrayList a){
      String result ="";
      int n =
            (new Double(Math.sqrt(a.size()))).intValue();
      for(int i = 0; i < n; i++) {
         for(int j = 0; j < n; j++) {
            result = result + a.get(i * n + j) + " ";
         }
         result = result + "\n";
      }
      return result;
   }
   //транспонирование двумерного массива
   void transposition(ArrayList a){
      int n =
            (new Double(Math.sqrt(a.size()))).intValue();
      for(int i = 0; i < n; i++) {
         for(int j = i + 1; j < n; j++) {
            var temp = a.get(i * n + j);
            a.set(i * n + j, a.get(j * n + i));
            a.set(j * n + i, temp);
         }
      }
   }

   public static void main (String args[]) {
      int n = 3;
      Program ex = new Program();
      ArrayList arrListArrList = new ArrayList();
      System.out.println("Исходная матрица");
      ex. setMatrixSimulation(arrListArrList, n);
      System.out.println(ex.outString(arrListArrList));
      System.out.println("Результат транспонирования");
      ex.transposition(arrListArrList);
      System.out.println(ex.outString(arrListArrList));
   }
}
```

Результат работы программы:



Интерфейс `RandomAccess`

Откроем описание коллекции `ArrayList<E>`:

```
public class ArrayList<E> extends AbstractList<E>
implements List<E>, RandomAccess, Cloneable, Serializable
```

Обращает на себя внимание, что данная коллекция реализует `RandomAccess`. Естественно, возникает вопрос, а какие методы этот интерфейс содержит и что конкретно определяет само название.

Собственно, если посмотреть далее текст описания, то сразу выясняется, что никаких методов данный интерфейс не определяет. Интерфейсы без методов называются интерфейсами маркеров в Java.

Таким образом **`RandomAccess`** - это просто интерфейс-маркер (или интерфейс тегов).

Общее значение текста, приведенного в описании, заключается в том в том, что `RandomAccess` является интерфейсом, указывающим, что коллекция `ArrayList<E>`, реализующая этот интерфейс, поддерживает быстрый произвольный (последовательный, как у списка, и по индексу, как у вектора) доступ. Другими словами, коллекция, реализующая этот интерфейс реализует стратегию быстрого произвольного доступа.

В то же время, можно найти в интернете следующее заявление о том, что если этот интерфейс реализован, то лучше использовать цикл `for()` для получения данных (т.е. по индексу), чем использовать итератор в этих же целях.

Выполним тестирование с помощью статического метода `nanoTime()` класса `System`. Он имеет точность до одной миллионной секунды (наносекунды) и возвращает текущее значение наиболее точного доступного системного таймера.

Пример.

```
1  import java.util.*;
2  public class Main
3  {
4      //статический метод инициализации коллекции ArrayList
5      public static ArrayList initArray(int n) {
6          ArrayList list = new ArrayList();
7          for (int i = 0; i < n; i++) {
8              list.add(i + 1);
9          }
10         return list;
11     }
12     //метод измеряющий время для for()
13     public static long timeGettingFor(ArrayList a) {
14         //время начала
15         long startTime = System.nanoTime();
16         int n = a.size();
17         //тестовая работа
18         for (int j = 0; j < n; j++) {
19           Object num = a.get(j);
20         }
21         //время окончания
22         long endTime = System.nanoTime();
23         //длительность
24         return endTime - startTime;
25     }
26     //метод измеряющий время для итератора
27     public static long timeGettingIterator(ArrayList a) {
28         long startTime = System.nanoTime();
29         Iterator iterator = a.iterator();
30         while (iterator.hasNext()){
31             Object next = iterator.next();
32         }
33         long endTime = System.nanoTime();
34         return endTime - startTime;
35     }
36
37    public static void main(String[] args) {
38      ArrayList list = initArray(1000000);
39      System.out.println("Коллекция ArrayList. " +
40                         "Время обращения к " +
41                          "значению с помощью:");
42      System.out.println("\t- индекса и цикла for(): " +
43                         timeGettingFor(list));
```

```
44          System.out.println("\t- итератора: " +
45                                  timeGettingIterator(list));
46      }
47  }
```

Результат работы программы:

```
Коллекция ArrayList. Время обращения к значению с помощью:
    - индекса и цикла for(): 8520358
    - итератора: 11702609
```

Можно видеть, что цикл `for()` проходит элементы за меньшее время, чем итераторы, доказывая, что интерфейс `RandomAccess` действительно может проявлять этот эффект (при числе обращений один миллион `1000000`).

Поскольку тестирование проводилось на сайтах, реализующих интегрированные среды разработки программного обеспечения на языке Java (т.е. на многопользовательских системах), то была проведена серия испытаний.

Повторные запуски данного тестовой программы указали на то, что этот эффект проявляется достаточно устойчиво. Однако отношение времени обработки коллекции `ArrayList` по индексу (с помощью `for()`) и итератора не представляет собой константу и постоянно меняется (и иногда может быть даже больше единицы). Вероятно, в момент работы теста сказывается загруженность серверов массовым запуском программ пользователями.

В любом случае, производительность вычислительной техники настолько высока, что в случае незначительного количества данных (до `1000`) разрыв в производительности между двумя способами обхода данных незначителен.

Таким образом в случае незначительного размера коллекция `ArrayList` поддерживает быстрый произвольный доступ к элементам ее объекта.

Пользовательские классы в параметрах `ArrayList`

Для классов используемых в качестве параметров в коллекциях должны быть переопределены все необходимые методы (ниже такими методами являются методы `toString()` записи и самой коллекции `ArrayList`).

Замечание.

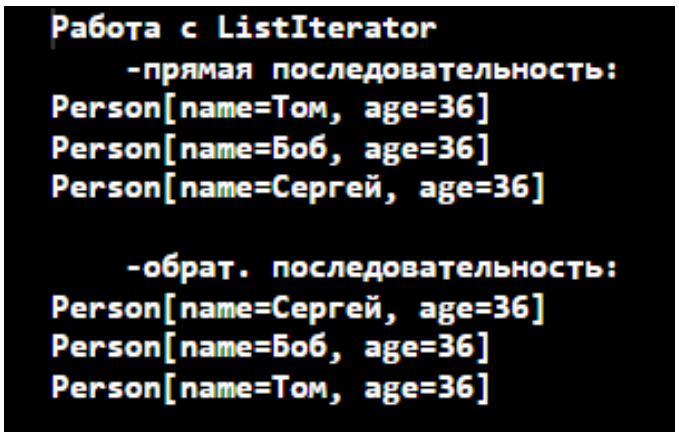
Будет использовать исключительно интерфейс интерфейс `ListIterator`.

Пример.

```
import java.util.*;

record Person(String name, int age) { }

public class Program
{
   static <T> void arrListGener(ArrayList a, T[] aInit){
      ListIterator it = a.listIterator();
      for(int i = 0; i < aInit.length; i++) {
         it.add(aInit[i]);
      }
   }

   static void display(ArrayList a,
                       StringBuffer directEnumer,
                       StringBuffer reverseEnumer) {
      ListIterator it = a.listIterator();
      while(it.hasNext()) {
         directEnumer.append(it.next().toString() +
                             "\n");
      }
      while(it.hasPrevious()) {
         reverseEnumer.append(it.previous().toString() +
                              "\n");
      }
   }

   public static void main (String args[]) {
      //создание инициализирующего массива объектов
      Person[] p = {new Person("Том", 36),
                    new Person("Боб", 36),
                    new Person("Сергей", 36) };
      ArrayList arrList = new ArrayList();
      arrListGener(arrList, p);
      StringBuffer directEnumer = new StringBuffer("");
      StringBuffer reverseEnumer = new StringBuffer("");
      System.out.println("Работа с ListIterator");
      display(arrList, directEnumer, reverseEnumer);
      System.out.println("\t-прямая последовательность:");
      System.out.println(directEnumer);
      System.out.println("\t-обрат. последовательность:");
```

```
42          System.out.println(reverseEnumer);
43      }
44  }
```

Результат работы программы:



Создание объекта коллекции `ArrayList` содержащего `ArrayList`-ы

Элементами `ArrayList`-а могут быть и другие `ArrayList`-ы. Собственно, как и объекты других коллекций. Например, можно сделать `ArrayList`, каждый элемент которого представляет собой `ArrayList` целых чисел (аналогично двумерному массиву):

```
ArrayList<ArrayList> имяОбъекта = new ArrayList();
```

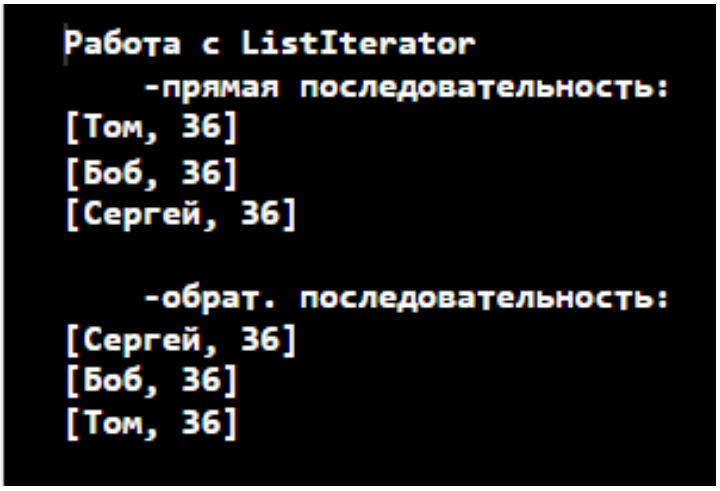
Будем использовать исключительно интерфейс `ListIterator` для внесения изменений в предыдущий пример.

Инициализированный массив из трех записей преобразуем в `ArrayList` состоящий также из трех объектов `ArrayList`. Т.е. каждый (из трех) внутренних объектов типа `ArrayList` будет содержать `record` типа `Person` с полями типа `String` (имя) и типа `int` (возраст).

Пример.

```
import java.util.*;

record Person(String name, int age) { }

public class Program
{
   static void arrListGener(ArrayList<ArrayList> a,
                            Person[] aInit) {
     for(int i = 0; i < aInit.length; i++) {
        ArrayList temporary = new ArrayList();
        temporary.add(aInit[i].name());;
        temporary.add(aInit[i].age());
        a.add(temporary);
     }
   }

   static void display(ArrayList a,
                       StringBuffer directEnumer,
                       StringBuffer reverseEnumer) {
      ListIterator it = a.listIterator();
      //toString() дополняется компилятором
      while(it.hasNext()) {
         directEnumer.append(it.next() + "\n");
      }
      while(it.hasPrevious()) {
         reverseEnumer.append(it.previous() + "\n");
      }
   }

   public static void main (String args[]) {
     //создание инициализирующего массива объектов
     Person[] p = {new Person("Том", 36),
                   new Person("Боб", 36),
                   new Person("Сергей", 36) };
     ArrayList<ArrayList> arrList = new ArrayList();
     arrListGener(arrList, p);
     StringBuffer directEnumer = new StringBuffer("");
     StringBuffer reverseEnumer = new StringBuffer("");
     System.out.println("Работа с ListIterator");
     display(arrList, directEnumer, reverseEnumer);
     System.out.println("\t-прямая последовательность:");
     System.out.println(directEnumer);
     System.out.println("\t-обрат. последовательность:");
     System.out.println(reverseEnumer);
   }
}
```

Результат работы программы:



Замечание.

Также можно создавать объект `ArrayList` *из стеков, очередей, деков, можно создавать трехмерные векторы (трехмерные массивы) и т.д.*

Приведение примера транспонирования двумерного массива, хранящегося в объекте коллекции `ArrayList` *из* `ArrayList`*-ов нецелесообразно, т.к. без привлечения методов интерфейса* `List` *уже известных по их реализациям в коллекции* `Vector` *выполнить эту манипуляцию практически невозможно.*

Объекты коллекции `ArrayList` *как свойства классов*

Рассмотрим случай, когда полем класса является `ArrayList`-ом. В рассмотренном ниже примере в классе, как и для других коллекций в качестве полей создается два `ArrayList`-а:

- имена городов;
- почтовые индексы.

Эти два `ArrayList`-а связаны между собой посредством соответствия индексов. Методы класса позволяют организовать поиск индекса города по его названию в создаваемом объекте.

Замечание.

Пример приведен с целью демонстрации, что в рамках общей концепции коллекций, реализующих интерфейс `List` *практически любые алгоритмы обработки данных (если не ставить целью применить специфические методы какой-либо отдельной коллекции) повторяются с точностью до замены типов.*

Пример.

```
1  import java.util.*;
2  class Matcher {
3      private ArrayList<String> name =
4                                   new ArrayList();
5      private ArrayList<Integer> postCode =
6                                   new ArrayList();
7      //блок инициализации
8      {
9          name.add("Минск");
10         name.add("Брест");
11         name.add("Витебск");
12         postCode.add(220000);
13         postCode.add(224000);
14         postCode.add(210000);
15     }
16     //методы
17     int finedIndex(String cityName){
18         for(int i = 0; i < name.size(); i++) {
19             if(cityName.equals(this.name.get(i))) {
20                 return i;
21             }
22         }
23         return -1;
24     }
25     int finedPostCode(String cityName) {
26         int i = finedIndex(cityName);
27         if (i < 0) return i;
28         return this.postCode.get(i);
29     }
30 }
31
32 public class Main
33 {
34     public static void main(String[] args) {
35         Matcher data = new Matcher ();
36         System.out.print(data.finedPostCode("Минск"));
37     }
38 }
```

Результат работы программы:

```
220000
```

Интерфейсы Comparable и Comparator

Интерфейс Comparable

Для того чтобы объекты можно было сравнить и сортировать, они должны реализовать параметризованный интерфейс Comparable.

Интерфейс Comparable содержит один единственный экземплярный метод int compareTo(T item), который сравнивает текущий объект с объектом, переданным в качестве параметра.

Если этот метод возвращает отрицательное число, то текущий объект располагается перед тем, который передается через параметр. Если метод вернет положительное число, то, наоборот, после второго объекта. Если метод возвращает ноль, значит, оба объекта равны.

Замечание.

Обратите внимание, что null не является экземпляром какого-либо класса, и вызов метода для объекта obj с параметром метода null, например obj.compareTo(null), должен вызывать исключение NullPointerException.

Интерфейс Comparator

Если класс по какой-то причине не может реализовать интерфейс Comparable, или же просто нужен другой вариант сравнения, используется интерфейс Comparator.

Среди множества методов интерфейс Comparator содержит один из основных методов, а именно метод int **compare**(T obj1, T obj2), который должен быть реализован классом, имплементирующим интерфейс Comparator.

Метод compare() возвращает числовое значение - если оно отрицательное, то объект obj1 предшествует объекту obj2, иначе – наоборот, а если метод возвращает ноль, то объекты равны.

Для переопределения метода compare(). Вначале надо создать класс имплементирующий Comparator. Формат класса:

```
class ИмяКласса implements Comparator<ТипОбъектовСравн> {
    @Override
    int compare(ТипОбъектовСравн obj1,
                ТипОбъектовСравн obj2) {
```

```
        //тело переопределенного метода
        return целочисленРезультат;
    }
}
```

Для дальнейшего примирения компаратора необходимо:

- создать ссылку типа `Comparator` и проинициализировать ее объектом класса `ИмяКласса`;
- передать эту ссылку в качестве параметра в конструктор, создающий коллекцию.

Естественно, что один из конструкторов коллекции должен принимать подобный объект в качестве параметра.

Замечание.

Интерфейс `Comparable` можно использовать для разрабатываемых классов. Если разработчик получает классы, созданные другим разработчиком и у получаемого класса либо вообще отсутствует переопределенная версия метода `compareTo()` или и его не устраивают признаки положенные в основу сравнения, то разработчиком создается класс реализующий интерфейс `Comparator`, в котором с помощью метода `compare()` реализуется новая логика сравнения. Классов с разными логиками сравнения может быть несколько.

Литература

1. Блинов, И.Н. Java from EPAM / И.Н. Блинов, В.С. Романчик. - Минск: Четыре четверти, 2020. - 560 с.
2. Руководство по языку программирования Java/ METANIT.COM - [Электронный ресурс], URL: https://metanit.com/java/tutorial/ (дата обращения: 08.06.22)
3. Самоучитель по Java с нуля/ Vertex Academy - [Электронный ресурс], URL: https://vertex-academy.com/tutorials/ru/samouchitel-po-java-s-nulya/ (дата обращения: 08.06.22)
4. javascopes.com - [Электронный ресурс], URL: http://javascopes.com (дата обращения: 08.06.22)
5. JavaRush - [Электронный ресурс], URL: https://javarush.ru/ (дата обращения: 08.06.22)
6. Справочник по Java Collections Framework / Хабр - [Электронный ресурс], URL: https://habr.com/ru/post/237043/ (дата обращения: 08.06.22)
7. Collections in Java / GeeksforGeeks - [Электронный ресурс], URL: https://www.geeksforgeeks.org/collections-in-java-2/?ref=lbp (дата обращения: 08.06.22)
8. Java Collections / Baeldung - [Электронный ресурс], URL: https://www.baeldung.com/java-collections (дата обращения: 08.06.22)

Учебное издание

Кравчук Александр Степанович
Кравчук Анжелика Ивановна
Кремень Елена Васильевна

Язык Java.
Общие сведения о коллекциях и реализуемых ими интерфейсах. Коллекции, реализующие интерфейс List

Учебные материалы
для студентов специальности 1-31 03 08
«Математика и информационные технологии
(по направлениям)»

В авторской редакции

Ответственный за выпуск *Е. В. Кремень*

Подписано в печать 22.12.2022. Формат 60×84/16. Бумага офсетная.
Усл. печ. л. 4,65. Уч.- изд. л. 4,29. Тираж 50 экз. Заказ

Белорусский государственный университет.
Свидетельство о государственной регистрации издателя, изготовителя,
распространителя печатных изданий № 1/270 от 03.04.2014.
Пр. Независимости 4, 220030, Минск.

Отпечатано с оригинал-макета заказчика
на копировально-множительной технике
механико-математического факультета
Белорусского государственного университета.
Пр. Независимости 4, 220030, Минск.