Программирование на Java
Тема 5.1 Коллекции

Способы хранения объектов в Java

Для хранения объектов используют:

1. Массивы
2. Контейнеры из пакета java.util.*

Дополнительные библиотеки:

org.apache.commons.collections
(http://commons.apache.org/collections/)

Способы хранения объектов в Java

- Для хранения объектов могут использоваться массивы, но они не всегда являются идеальным решением. Во-первых, длина массива задается заранее и в случае, если количество элементов заранее неизвестно, придется либо выделять память «с запасом», либо предпринимать сложные действия по переопределению массива. Во-вторых, элементы массива имеют жестко заданное размещение в его ячейках, поэтому, например, удаление элемента из массива не является простой операцией.
- Контейнеры (коллекции) обладают различными способами хранения объектов и содержат множество средств для работы с объектами в контейнере.

- До появления Java 5 тип объекта помещаемого в контейнер не контролировался.
- Параметризация – определяет тип объектов которые могут хранится в контейнере (коллекции или карте.)
- См. пример на следующем слайде.

Параметризация и типизация контейнеров

- **Пример:**

```
List list = new ArrayList(); (до Java 5)
list.add(100);
list.add("string");
//ошибка во время выполнения программы
Integer myInt = (Integer)list.get(1);

List<Integer> list = new ArrayList<Integer>();(Java 5)
list.add(100);
list.add("string");
//ошибка во время компиляции
```

```
The method add(Integer) in the type List<Integer> is
  not applicable for the arguments (String)
```

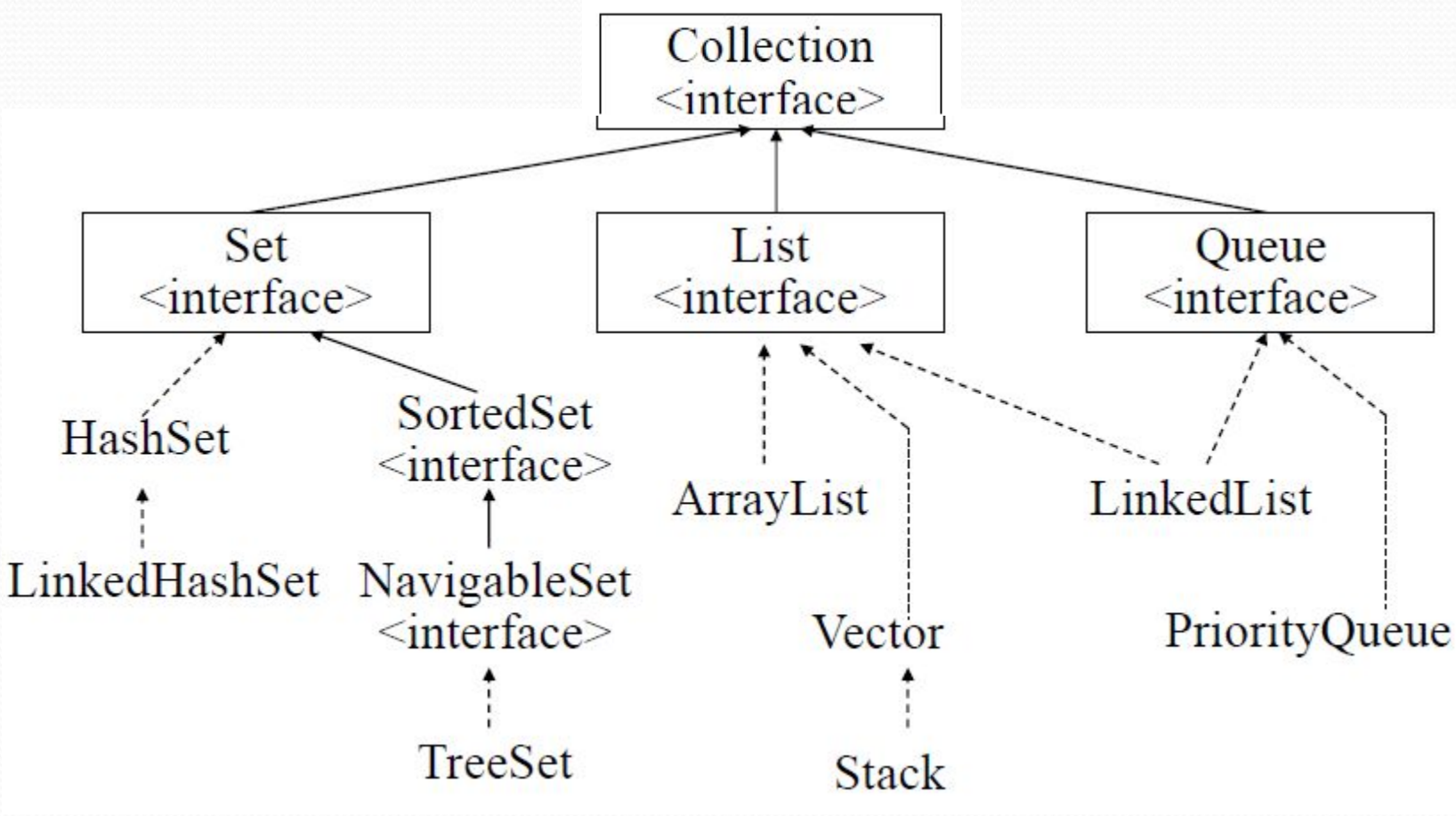
Интерфейсы Collection и Map

Контейнеры Java базируются на двух интерфейсах:

1. **Collection (Коллекция) – представляет концепцию «последовательности».**

2. **Map (Карта) – набор пар объектов «ключ-значение», выбор объекта-значения происходит по ключу.**

Иерархии контейнеров



- **Основные методы интерфейса Collection:**

```
boolean add (Object o);
void clear();
boolean contains(Object o);
boolean isEmpty();
boolean remove(Object o);
int size();

boolean addAll(Collection c);
boolean containsAll(Collection c);
boolean removeAll(Collection c);
Object[] toArray();
boolean equals(Object o);
int hashCode();
Iterator iterator();
```

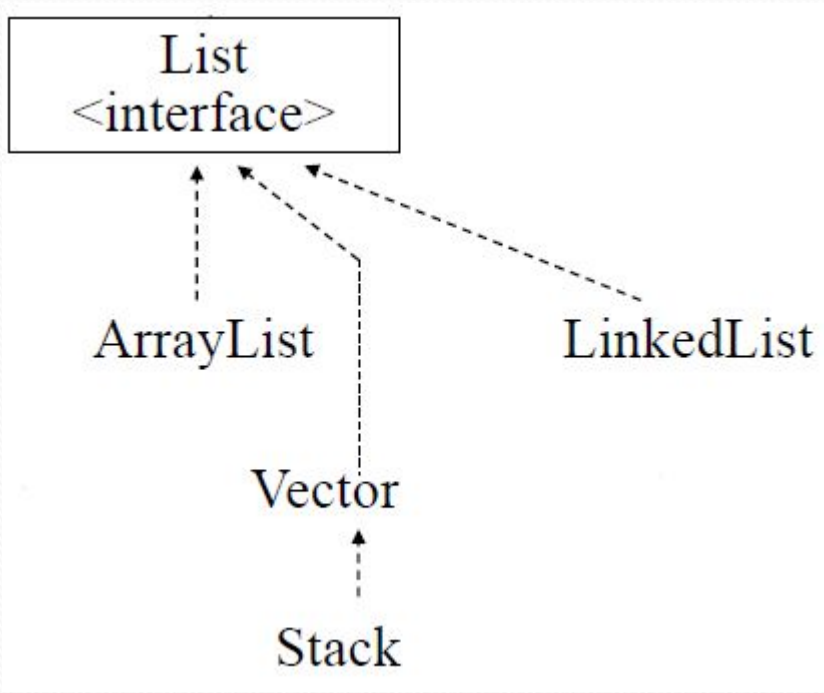
Иерархии контейнеров

- Основные методы интерфейса Collection:
  - *add(Object item)* — добавляет в коллекцию новый элемент. Метод возвращает true, если добавление прошло удачно и false — если нет*. Если элементы коллекции каким-то образом упорядочены, новый элемент добавляется в конец коллекции.
  - *clear()* — удаляет все элементы коллекции.
  - *contains(Object obj)* — возвращает true, если объект obj содержится в коллекции и false, если нет.
  - *isEmpty()* — проверяет, пуста ли коллекция.
  - *remove(Object obj)* — удаляет из коллекции элемент obj. Возвращает false, если такого элемента в коллекции не нашлось.
  - *size()* — возвращает количество элементов коллекции.
- Существуют разновидности перечисленных методов, которые в качестве параметра принимают любую другую коллекцию. Например, метод addAll(Collection coll) добавляет все элементы другой коллекции coll в конец данной, метод removeAll(Collection coll) удаляет из коллекции все элементы, которые присутствуют также в коллекции coll, а метод retainAll(Collection coll) поступает наоборот, удаляя все элементы, кроме содержащихся в coll. Метод toArray() возвращает все элементы коллекции в виде массива.

Интерфейс List

```
public interface List<E> extends Collection<E>
{
  E get(int index);
  E set(int index, E element);
  boolean add(E element);
  void add(int index, E element);
  E remove(int index);
  Iterator<E> iterator();
  int indexOf(Object o);
  int lastIndexOf(Object o);
  ListIterator<E> listIterator();

  boolean addAll(int index, Collection< extends E> c);
    ...
}
```



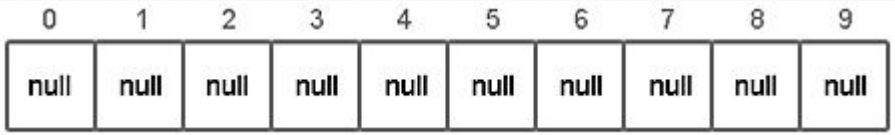
Интерфейс List

- Интерфейс List описывает упорядоченный список. Элементы списка пронумерованы, начиная с нуля и к конкретному элементу можно обратиться по целочисленному индексу.
- Объекты в этой коллекции хранятся в порядке вставки или в зависимости от индекса вставки, основное значение для этой коллекции имеет индекс, с помощью индекса объекты добавляются, удаляются и извлекаются.
- Интерфейс List является наследником интерфейса Collection, поэтому он наследует и переопределяет его методы и добавляет к ним несколько своих:
  - *add(int index, Object item)* — вставляет элемент item в позицию index, при этом список раздвигается (все элементы, начиная с позиции index, увеличивают свой индекс на 1);
  - *get(int index)* — возвращает объект, находящийся в позиции index;
  - *indexOf(Object obj)* — возвращает индекс первого появления элемента obj в списке;
  - *lastIndexOf(Object obj)* — возвращает индекс последнего появления элемента obj в списке;
  - *add(int index, Object item)* — заменяет элемент, находящийся в позиции index объектом item;
  - *subList(int from, int to)* — возвращает новый список, представляющий собой часть данного (начиная с позицииf rom до позиции to-1 включительно).
- Стандартные реализации интерфейса List: классы ArrayList, LinkedList, Vector, Stack.

Интерфейс List, реализация ArrayList

- **Создание объекта:**

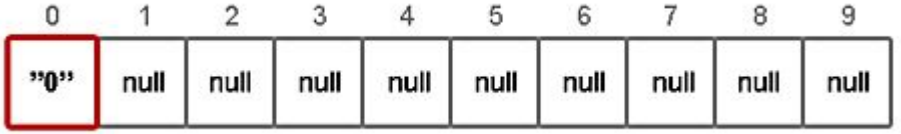
```
List<String> list = new ArrayList<String>();
```




```
elementData = (E[]) new Object[10];
```

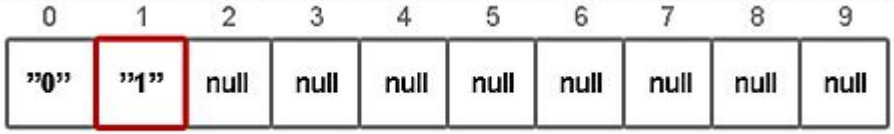
- **Добавление элементов:**

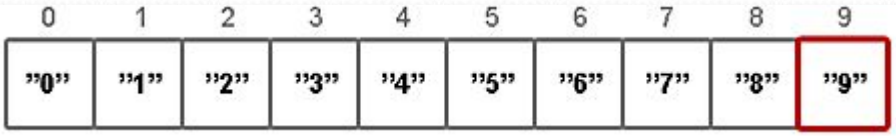
```
list.add("0");
```




```
ensureCapacity(size + 1);
elementData[size++] = element;
```

```
list.add("1") … list.add("9") ;
```





Интерфейс List, реализация ArrayList

- Класс ArrayList — реализует интерфейс List. Как известно, в Java массивы имеют фиксированную длину, и после того как массив создан, он не может расти или уменьшаться. ArrayList может менять свой размер во время исполнения программы, при этом не обязательно указывать размерность при создании объекта. Элементы ArrayList могут быть абсолютно любых типов в том числе и null.
- Созданный объект list (см. слайд), содержит свойства elementData и size. Хранилище значений elementData есть ни что иное как массив определенного типа, в нашем случае String[]. Если вызывается конструктор без параметров, то по умолчанию будет создан массив из 10-ти элементов типа Object (с приведением к типу, разумеется). Можно использовать конструктор ArrayList(capacity) и указать свою начальную емкость списка.
- Внутри метода **add(value)** происходит следующее:

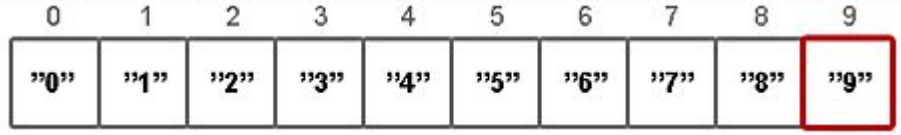
  1) проверяется, достаточно ли места в массиве для вставки нового элемента;

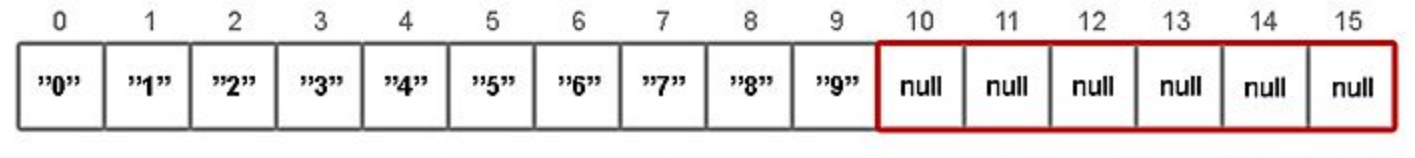
  2) добавляется элемент в конец (согласно значению size) массива.
- Весь метод **ensureCapacity(minCapacity)** рассматривать не будем, остановимся только на паре интересных мест. Если места в массиве не достаточно, новая емкость рассчитывается по формуле **(oldCapacity * 3) / 2 + 1**. Второй момент это копирование элементов. Оно осуществляется с помощью native метода **System.arraycopy()**.

Интерфейс List, реализация ArrayList

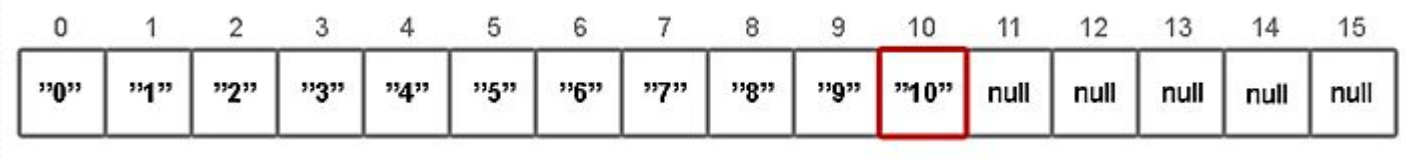
- **Добавление элементов:**

```
list.add("10");
```

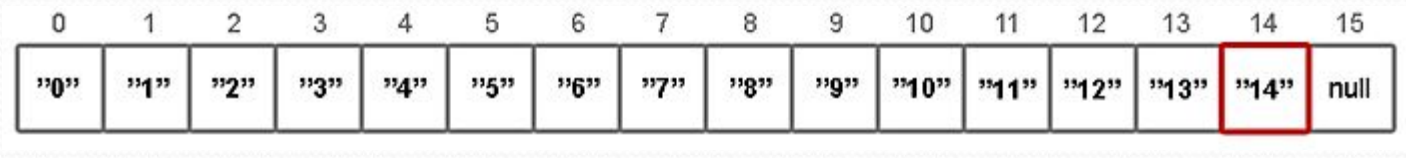





```
// newCapacity - новое значение емкости
elementData = (E[])new Object[newCapacity];
// oldData - временное хранилище текущего массива с данными
 System.arraycopy(oldData, 0, elementData, 0, size);
```




```
list.add("14");
```

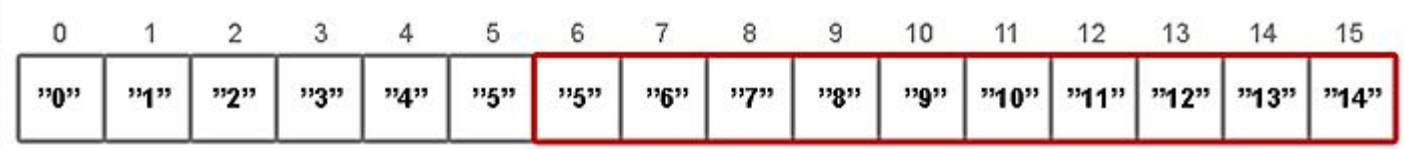


Интерфейс List, реализация ArrayList

- **Добавление в «середину» списка:**

```
list.add(5, "100");
```


```
ensureCapacity(size+1);
```




```
System.arraycopy(elementData, index, elementData, index +
1, size - index);
```


```
elementData[index] = element; size++;
```

Интерфейс List, реализация ArrayList

- **Удаление элементов по индексу:**

```
list.remove(5);
```

```
int numMoved = size - index - 1;
System.arraycopy(elementData, index + 1, elementData,
index, numMoved);
elementData[--size] = null;
```

- **Удаление элементов по значению:**

```
list.remove("100");
```

Интерфейс List, реализация ArrayList

- При добавлении 11-го элемента, проверка показывает что места в массиве нет. Соответственно создается новый массив и вызывается **System.arraycopy().** После этого добавление элементов продолжается.

- Добавление элемента на позицию с определенным индексом происходит в три этапа:
  1) проверяется, достаточно ли места в массиве для вставки нового элемента;
  2) подготавливается место для нового элемента с помощью **System.arraycopy();**
  3) перезаписывается значение у элемента с указанным индексом.
- Как можно догадаться, в случаях, когда происходит вставка элемента по индексу и при этом в вашем массиве нет свободных мест, то вызов **System.arraycopy()** случится дважды: первый в **ensureCapacity()**, второй в самом методе **add(index, value)**, что явно скажется на скорости всей операции добавления.
- В случаях, когда в исходный список необходимо добавить другую коллекцию, да еще и в «середину», стоит использовать метод **addAll(index, Collection)**. И хотя, данный метод скорее всего вызовет **System.arraycopy()** три раза, в итоге это будет гораздо быстрее поэлементного добавления.

- С удалением элемента по индексу всё достаточно просто. Сначала определяется какое количество элементов надо скопировать. Затем копируем элементы используя **System.arraycopy().** Уменьшаем размер массива и забываем про последний элемент.
- При удалении по значению, в цикле просматриваются все элементы списка, до тех пор пока не будет найдено соответствие. Удален будет лишь первый найденный элемент.
- При удалении элементов текущая величина capacity не уменьшается, что может привести к своеобразным утечкам памяти. Поэтому не стоит пренебрегать методом **trimToSize().**
- **Итоги.** **-** 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;
- Если вам необходимо часто добавлять или удалять элементы коллекции, то предпочтительней использовать LinkedList.

Интерфейс List, реализация LinkedList

- Элементы в LinkedList упорядочены по индексу и последний вставленный элемент в коллекцию занимает последнюю позицию в коллекции.
- Доступ к элементам в случайном порядке отсутствует, вызов get(int index) приводит к перебору элементов коллекции от первого до index.

Интерфейс List, реализация LinkedList

- LinkedList реализует дополнительно java.util.Queue, за счет этого коллекция имеет дополнительные методы.
  - peek() -получить первый элемент коллекции, без его удаления
  - poll() -получить и удалить первый элемент коллекции
  - offer() –добавить элемент в конец коллекции
  - addFirst()/addLast()–добавить элемент в начало/конец коллекции
- На базе LinkedList реализуют Stack

- **Интерфейс Iterator**

```
for (int i = 1; i < array.length; i++)
{// обрабатываем элемент array[i](array.get(i))}
```

- **Методы интерфейса Iterator:**
  - next()
  - hasNext()
  - remove()

```
Iterator iter = coll.iterator(); // coll – коллекция
while (iter.hasNext())
{ // обрабатываем объект, возвращаемый методом iter.next()
 }
```

Коллекции

- Преимущество использования массивов и коллекций заключается не только в том, что можно поместить в них произвольное количество объектов и извлекать их при необходимости, но и в том, что все эти объекты можно комплексно обрабатывать. В случае массива мы пользуемся циклом for:
- Имея дело со списком, мы можем поступить аналогичным образом, только вместо array[i] писать array.get(i). Но мы не можем поступить так с коллекциями, элементы которых не индексируются (например, очередью или множеством).
- В пакете java.util имеется интерфейс iterator , описывающий способ обхода всех элементов коллекции. В каждой коллекции есть метод iterator (), возвращающий реализацию интерфейса iterator для указанной коллекции. Получив эту реализацию, можно обходить коллекцию в некотором порядке, определенном данным итератором, с помощью методов, описанных в интерфейсе iterator и реализованных в этом итераторе. Интерфейс iterator имеет всего три метода:
  - next() возвращает очередной элемент коллекции, к которой «привязан» итератор (и делает его текущим). Порядок перебора определяет сам итератор.
  - hasNext() возвращает true, если перебор элементов еще не закончен
  - remove() удаляет текущий элемент
- Интерфейс Collection помимо рассмотренных ранее методов, имеет метод iterator(), который возвращает итератор для данной коллекции, готовый к ее обходу. С помощью такого итератора можно обработать все элементы любой коллекции простым способом: см. слайд.
-

- **Интерфейс ListIterator**
- **Методы интерфейса ListIterator:**
  - `previous()`
  - `add(Object item)`
  - `set(Object item)`
  - `nextIndex()` и `previousIndex()`
-

Коллекции

- Для коллекций, элементы которых проиндексированы, определен более функциональный итератор, позволяющий двигаться как в прямом, так и в обратном направлении, а также добавлять в коллекцию элементы. Такой итератор имеет интрефейс ListIterator, унаследованный от интерфейса Iterator и дополняющий его следующими методами:
  - *previous()* — возвращает предыдущий элемент (и делает его текущим);
  - *hasPrevious()* — возвращает true, если предыдущий элемент существует (т.е. текущий элемент не является первым элементом для данного итератора);
  - *add(Object item)* — добавляет новый элемент перед текущим элементом;
  - *set(Object item)* — заменяет текущий элемент;
  - *nextIndex() u previousIndex()* — служат для получения индексов следующего и предыдущего элементов соответственно.
- В интерфейсе List определен метод listIterator(), возвращающий итератор ListIterator для обхода данного списка.

- **Пример .**

```
import java.util.*;
public class LinkedListListIterator
{
  public static void main(String[] args)
   { //Создаем список на основе LinkedList
  List<String> grades = new LinkedList<String>();
     grades.add("A"); grades.add("B");
     grades.add("C"); grades.add("D");
  // Извлекаем объекты LinkedList исп-я интерфейс ListIterator
     ListIterator li = grades.listIterator();
     while(li.hasNext())
     { System.out.println("  Первый: " + li.next());
       System.out.println("Предыдущий: " +  li.hasPrevious());
       System.out.println("  Следующий: " + li.hasNext());
       System.out.println("Индекс предыдущего: " +
  li.previousIndex());
       System.out.println("  Индекс следующего: " +
  li.nextIndex());
      }
    }
}
```

```
  Первый: A
Предыдущий: true
  Следующий: true
Индекс предыдущего: 0
  Индекс следующего: 1
  Первый: B
Предыдущий: true
  Следующий: true
Индекс предыдущего: 1
  Индекс следующего: 2
  Первый: C
Предыдущий: true
  Следующий: true
Индекс предыдущего: 2
  Индекс следующего: 3
  Первый: D
Предыдущий: true
  Следующий: false
Индекс предыдущего: 3
  Индекс следующего: 4
```

- Цикл foreach
- Синтаксис :

```
for (тип итер-пер : коллекция)
    блок-операторов
```

```
Значение равно: 1
Значение равно: 2
Значение равно: 3
Значение равно: 4
Значение равно: 5
Значение равно: 6
Значение равно: 7
Значение равно: 8
Значение равно: 9
Значение равно: 10
Сумма равна: 55
```

```
class ForEach
{ public static void main(String args[])
   { int nums[] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
     int sum = 0;
     for(int x : nums)
      { System.out.println("Значение равно: " + x);
        sum+=x;
       }
      System.out.println("Сумма равна: " + sum) ;
   }
}
```

Коллекции

- Начиная с версии JDK 5 в Java можно использовать вторую форму цикла for, реализующую цикл в стиле "for-each" ("для каждого"). К
- Цикл в стиле "for-each" предназначен для строго последовательного выполнения повторяющихся действий по отношению к коллекции объектов, такой, например, как массив.
- Цикл for в стиле "for-each" называют также усовершенствованным циклом for.
- Общая форма версии "for-each" цикла for представлена на слайде: итер-пер — имя итерационной переменной, которая последовательно будет принимать значения из коллекции, от первого до последнего. Элемент коллекция указывает коллекцию, по которой должен выполняться цикл. С циклом for можно применять различные типы коллекций. На каждой итерации цикла программа извлекает следующий элемент коллекции и сохраняет его в переменной итер-пер. Цикл выполняется до тех пор, пока не будут получены все элементы коллекции.
- Поскольку итерационная переменная получает значения из коллекции, тип должен совпадать (или быть совместимым) с типом элементов, хранящихся в коллекции. Таким образом, при выполнении цикла по массивам тип должен быть совместим с базовым типом массива.
- Применение такого цикла позволяет не устанавливать значение счетчика цикла за счет указания его начального и конечного значений, и исключает необходимость индексации массива вручную. Вместо этого программа автоматически выполняет цикл по всему массиву, последовательно получая значения каждого из его элементов, от первого до последнего.
- При использовании цикла в стиле "for-each" необходимо помнить о следующем важном обстоятельстве. Его итерационная переменная является переменной "только для чтения", поскольку она связана только с исходным массивом. Операция присваивания значения итерационной переменной не оказывает никакого влияния на исходный массив. Иначе говоря, содержимое массива нельзя изменять, присваивая новое значение итерационной переменной.

Класс LinkedList

Пример . вставка элемента в середину списка:

```
Список содержит:
==================
Имя: Иванов
Имя: Петров
Имя: Юркин
Обновленный список содержит:
====================
Имя = Иванов
Имя = Петров
Имя = Сидоров
Имя = Юркин
```

```
import java.util.*;
public class LinkedListAddDemo
{ public static void main(String[] args)
  { // Создаем связанный список LinkedList
    List<String> names = new LinkedList<String>();
    names.add("Иванов");
    names.add("Петров");
    names.add("Юркин");
     // Выводим содержимое LinkedList
     System.out.println("Список содержит:");
     System.out.println("====================");
     for (String name : names)
        System.out.println("Имя: " + name);
     // Добавляем новый элемент в список на 2ю позицию.
     names.add(2, "Сидоров");
     System.out.println("Обновленный список содержит:");
     System.out.println("====================");
     for (String name : names)
        System.out.println("Имя = " + name);
    }
}
```