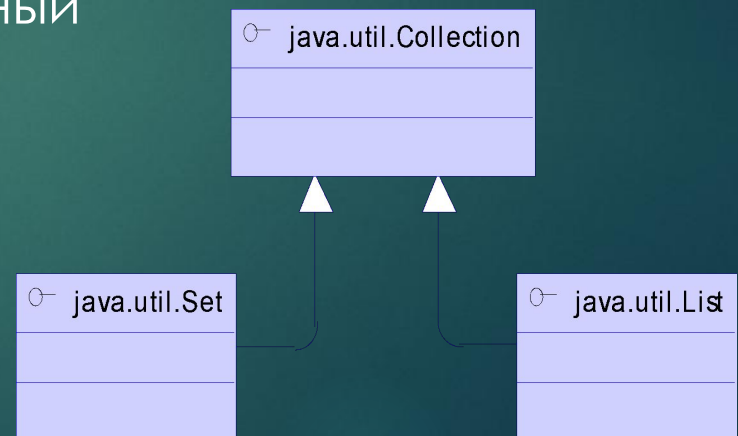


Коллекции

- **Коллекциями** называют структуры, предназначенные для хранения **однотипных данных**
- Все коллекции Java (до JDK 1.5) предназначены для хранения потомков класса **Object**.
- На вершине библиотеки контейнеров Java расположены два основных интерфейса, которые представляют два принципиально разных вида коллекций:
 - интерфейс **Collection** – группа объектов
 - интерфейс **Map** – ассоциативный массив объектов

Интерфейс Collection

- **Collection** представляет собой группу объектов
- Правила хранения элементов задаются нижележащими интерфейсами, сам же интерфейс **Collection** в JDK прямых реализаций не имеет.
- Интерфейс **Collection** расширяется двумя способами:
 - интерфейс **List** – упорядоченный список;
 - интерфейс **Set** – множество



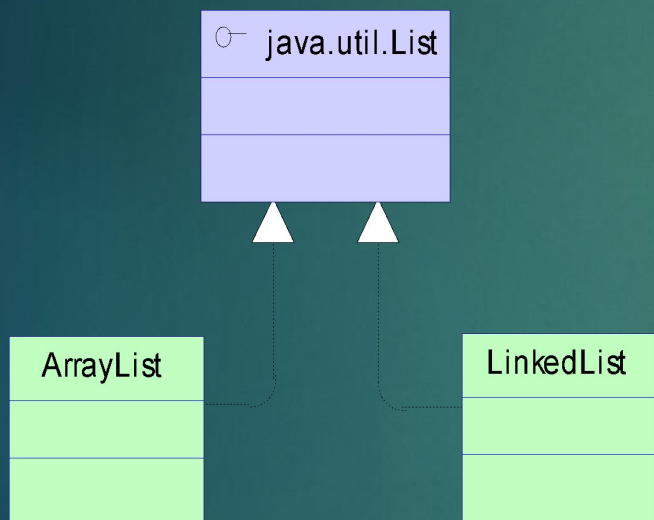
Интерфейс List

- **List** – это список объектов

- Объекты хранятся в порядке их добавления в список

- В пакете **java.util** имеется 2 класса, реализующих интерфейс List:

- **ArrayList** – в нем для хранения элементов используется массив
- **LinkedList** – для хранения элементов используется двусвязный список



Класс ArrayList

- Класс **ArrayList** представляет собой список динамической длины. Данные внутри класса хранятся во внутреннем массиве
- По умолчанию при создании нового объекта **ArrayList** создается внутренний массив длиной 10 элементов
 - `List l = new ArrayList();`
- Можно также создать **ArrayList**, задав его начальную длину
 - `List l = new ArrayList(100);`
- Если длины внутреннего массива не хватает для добавления нового объекта, внутри класса создается новый массив большего объема, и все элементы старого массива копируются в **новый**

Класс LinkedList

- Класс **LinkedList** также представляет собой список динамической длины. Данные внутри него хранятся в виде СВЯЗНОГО СПИСКА
- У **LinkedList** представлен ряд методов, не входящих в интерфейс **List**:
 - **addFirst()** и **addLast()** - добавить в начало и в конец списка
 - **removeFirst()** и **removeLast()** - удалить первый и последний элементы
 - **getFirst()** и **getLast()** - получить первый и последний элементы

Доступ к элементам списков

- Доступ к элементам списка возможен двумя способами:
 - по индексу
 - с помощью итератора (Iterator)
- Доступ по индексу:

```
for (int i = 0; i < list.size(); i++){  
    MyClass elem = (MyClass)list.get(i);  
    elem.doSome();  
}
```

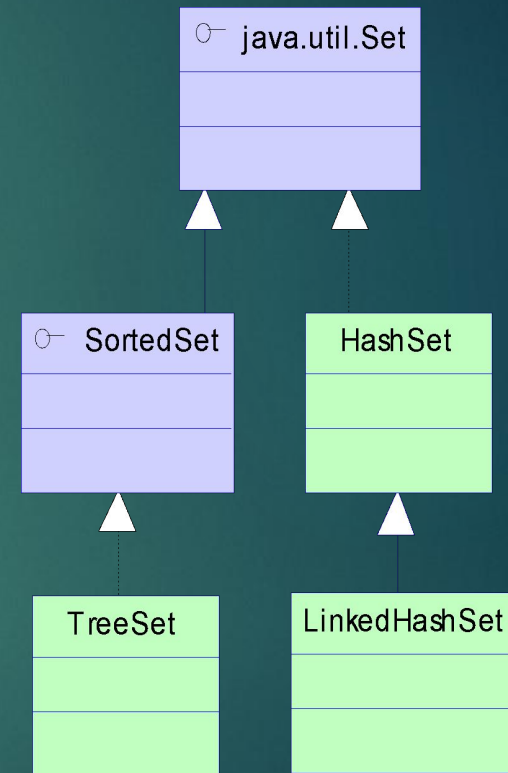
Итераторы (Iterator)

- Итератор – это вспомогательный объект, используемый для прохода по коллекции объектов
- Работа с итераторами производится через интерфейс `Iterator`, который специфицирует методы:
 - `boolean hasNext()` – проверяет есть ли еще элементы в коллекции
 - `Object next()` – выдает очередной элемент коллекции
 - `void remove()` – удаляет последний выбранный элемент из коллекции.
- Получить итератор для прохода коллекции можно с помощью метода `iterator()`, который определен у интерфейса `Collection`

```
for (Iterator iter = collection.iterator(); iter.hasNext();) {  
    MyClass element = (MyClass) iter.next();  
    element.doSome();  
}
```

Интерфейс Set

- **Set** – множество неповторяющихся объектов
- Добавление повторяющихся элементов в **Set** не вызывает исключений, но они не попадают в множество
- Для прохода по множеству используется интерфейс итератор

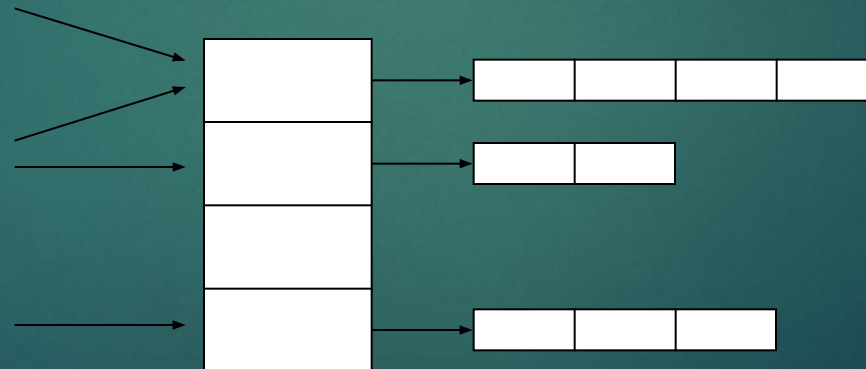


Классы HashSet и LinkedHashSet

- Классы **HashSet** и **LinkedHashSet** реализуют интерфейс **Set**
- Уникальность объектов в них обеспечивается благодаря использованию механизма хеширования
- В **HashSet** объекты хранятся в произвольном порядке
- **LinkedHashSet** является наследником класса **HashSet**. Он хранит объекты в порядке их добавления

Механизм хеширования

- **Хеширование** – такой способ хранения и доступа к данным, при котором пространство объектов вырожденно отображается на пространство адресов.
- Для вычисления адреса используется метод `hashCode()`.
- По сформированному адресу находится последовательность элементов. В пределах этой последовательности производится поиск с помощью `equals()`



Упорядоченные множества (SortedSet)

- Интерфейс **SortedSet** служит для спецификации упорядоченных множеств
- В JDK его реализация представлена в классе **TreeSet** (бинарное дерево)
- Объекты упорядоченного множества хранятся в порядке, заданном *функцией сравнения*.
- При добавлении нового объекта он становится на свое место в соответствии с его порядком в множестве.

```
Set sorted = new TreeSet();  
sorted.add(new Integer(2));  
sorted.add(new Integer(3));  
sorted.add(new Integer(1));  
System.out.println(sorted); // Распечатает [1, 2, 3]
```

Интерфейс Comparable

- В Java задача задания функции сравнения решается с использованием интерфейсов **Comparable** и **Comparator**
- Интерфейс **Comparable** предназначен для определения так называемого естественного порядка (**natural ordering**). Данный интерфейс содержит всего один метод

```
public int compareTo(Object o) // сравнивает объект с другим объектом.
```
- Метод **compareTo(Object o)** возвращает:
 - отрицательное число, если **this < other**;
 - ноль, если **this == other**;
 - положительное число, если **this > other**.
- Дополнительным условием является то, что метод **compareTo(other)** должен возвращать 0 тогда и только тогда, когда метод **equals(other)** возвращает true.

Пример с использованием Comparable

```
public class Employee implements Comparable{
    private String name; // имя
    private int salary; // зарплата
    ...
    public int compareTo(Object obj){
        // Задает функцию сравнения объектов по зарплате
        int otherSalary = ((Employee)obj).getSalary();
        if (salary == otherSalary)
            return 0;
        else
            return (salary > otherSalary) ? 1 : -1;
    }

    public static void main(String[] args) {
        Set emps = new TreeSet();
        emps.add(new Employee("Vasya", 500));
        emps.add(new Employee("Sanya", 1000));
        emps.add(new Employee("Petya", 300));
        System.out.println(emps); // Распечатает [Petya: 300, Vasya: 500, Sanya:
1000]
    }
}
```

Интерфейс Comparator

- Интерфейс **Comparator** используется, когда метод **compareTo()** уже переопределен, но необходимо задать еще какой-то порядок сортировки
- В этом случае создается отдельный вспомогательный класс, реализующий интерфейс **Comparator**, и уже на основании объекта этого класса будет производиться сортировка
- В этом классе нужно реализовать метод **compare(Object o1, Object o2)**. Правила работы этого метода такие же, как и у **compareTo(Object o)**

Пример работы с Comparator

```
public class Employee{
    ...
    public static final Comparator EMPLOYEE_NAME_COMPARATOR =
        new Comparator() {
        public int compare(Object o1, Object o2) {
            // Задает функцию сравнения по имени
            Employee e1 = (Employee)o1;
            Employee e2 = (Employee)o2;
            return e1.getName().compareTo(e2.getName());
        }
    };

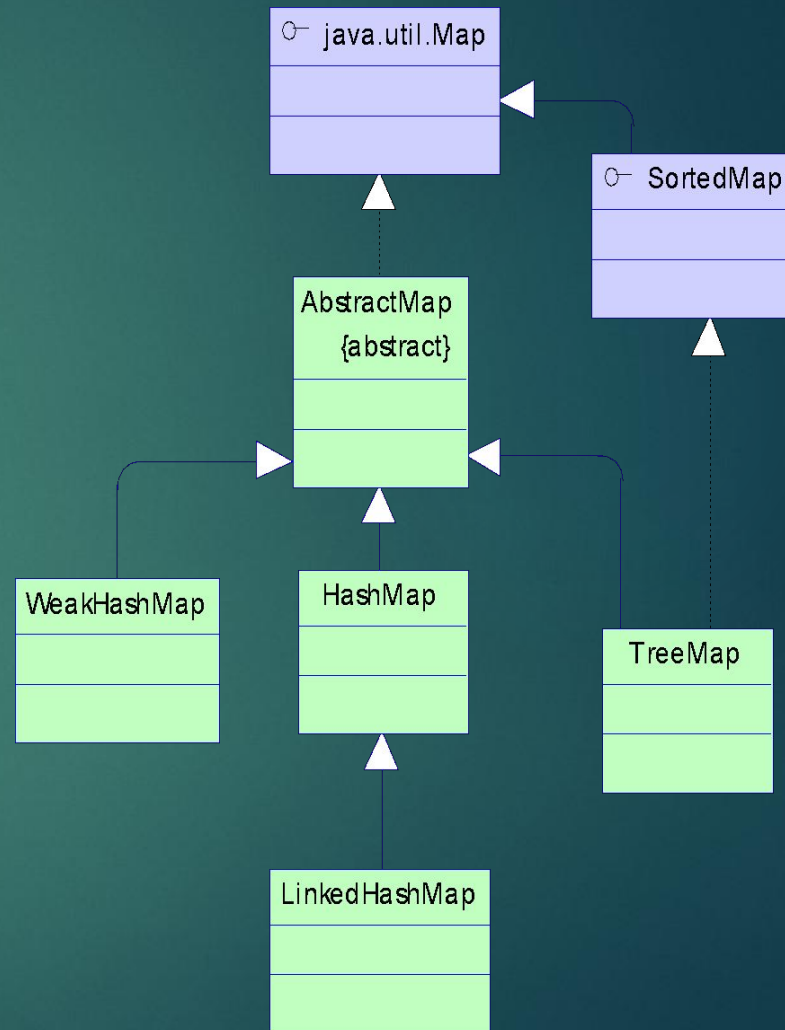
    public static void main(String[] args) {
        Set emps =
            new TreeSet(Employee.EMPLOYEE_NAME_COMPARATOR);
        emps.add(new Employee("Vasya", 500));
        emps.add(new Employee("Petya", 300));
        emps.add(new Employee("Sanya", 1000));
        System.out.println(emps);
    }
}
```

Класс Collections

- Класс **java.util.Collections** – это собрание статических методов для работы с коллекциями
- С его помощью можно заполнять коллекции, сортировать их, искать элементы в коллекциях и делать другие операции
- `public static void sort(List list)` – сортирует список. Элементы списка должны реализовывать **Comparable**
- `public static void sort(List list, Comparator c)` – сортирует список с использованием `Comparator`.
- `public static int binarySearch(List list, Object key)` – возвращает индекс найденного элемента. Список должен реализовывать `Comparable` и должен быть предварительно отсортирован
- `public static int binarySearch(List list, Object key, Comparator c)` – то же самое, но с использованием `Comparator`
- Остальные методы см. в API-документации

Интерфейс Map

- Интерфейс **Map** часто называют ассоциативным массивом
- **Map** осуществляет отображение (mapping) множества ключей на множество значений. Т.е. объекты хранятся в нем в виде пар <ключ, значение>
- Map позволяет получить значение по ключу.
- В Map не может быть 2-х пар с одинаковым ключом



Методы Map

- `public void put(Object key, Object value)` - добавляет новую пару <ключ, значение>
- `public Object get(Object key)` – возвращает `value` по заданному ключу, или **`null`**, если ничего не найдено
- `public Set keySet()` – возвращает множество ключей
- `boolean containsKey(Object key)` – возвращает `true`, если Map содержит пару с заданным ключем

Классы HashMap и LinkedHashMap

- **HashMap** формирует неупорядоченное множество ключей
- Для хранения ключей в **HashMap** и **LinkedHashMap** и используется механизм хеширования.
- Ключи в **HashMap** хранятся в произвольном порядке
- **LinkedHashMap** содержит ключи в порядке их добавления.

Пример с использованием HashMap

```
Map map = new HashMap();

// Заполнить его чем-нибудь
map.put("one", "111");
map.put("two", "222");
map.put("three", "333");
map.put("four", "333");

// Получить и вывести все ключи
System.out.println("Set of keys: " + map.keySet());

// Получить и вывести значение по ключу
String val = (String)map.get("one");
System.out.println("one=" + val);

// Получить и вывести все значения
System.out.println("Collection of values: " + map.values());

// Получить и вывести все пары
System.out.println("Set of entries: " + map.entrySet());
```

Внутренний интерфейс Entry

- Интерфейс **Map.Entry** позволяет работать с объектом, который представляет собой пару (ключ, значение).
- Интерфейс содержит такие методы как:
 - `boolean equals(Object o)` - проверяет эквивалентность двух пар
 - `Object getKey()` – возвращает ключ пары.
 - `Object getValue()` – возвращает значение пары.
 - `Object setValue(Object value)` – изменяет значение пары

Массивы и коллекции

Создание заполненного списка:

```
List myList = Arrays.asList(new String[] {"1", "2", "3"});
```

Добавление в коллекцию элементов типа, отличного от типа исходного массива, возбуждает **UnsupportedOperationException!**

Получение массива объектов списка:

```
String[] strings = (String[])myList.toArray();
```

- ▶ создает массив типа `Object[]` и копирует в него все элементы

```
String[] strings = (String [])myList.toArray(new String[myList.size()])
```

- ▶ копирует элементы в переданный массив и возвращает его
- ▶ если его размера недостаточно, создает новый массив того же типа

Создание немодифицируемых коллекций

При объявлении коллекции как

```
public static final myList = new ArrayList();
```

нельзя гарантировать, что данный объект не будет изменен извне

Обеспечить немодифицируемость списка после его инициализации можно с помощью метода `List Collections.unmodifiableList(List list)` после его инициализации

Пример:

```
ArrayList myList = new ArrayList();
```

```
myList.add("one");
```

```
myList.add("two");
```

```
myList = Collections.unmodifiableList(myList); // После этого попытка  
// изменения myList вызовет UnsupportedOperationException
```

То же самое

```
для Map: Map unmodifiableMap(Map m)
```

```
для Set: Set unmodifiableSet(Set s)
```

и т.д.

Синхронизированные коллекции

В CollectionsFramework большинство коллекций не синхронизировано

Кроме устаревших типа Vector

Чтобы сделать синхронизированную коллекцию, нужно воспользоваться методами класса

`Collections`

`List synchronizedList(List list)`

`Map synchronizedMap(Map m)`

`Set synchronizedSet(Set s)`

и т.д.

В этих методах создается **надстройка** над передаваемым объектом, реализующая соотв. интерфейс и выполняющая **синхронизацию** в **каждом из методов**