Занятие № 11, 12.

# Collection Framework

Коллекции в Java: List, Map, Set, Queue
Основные реализации коллекций

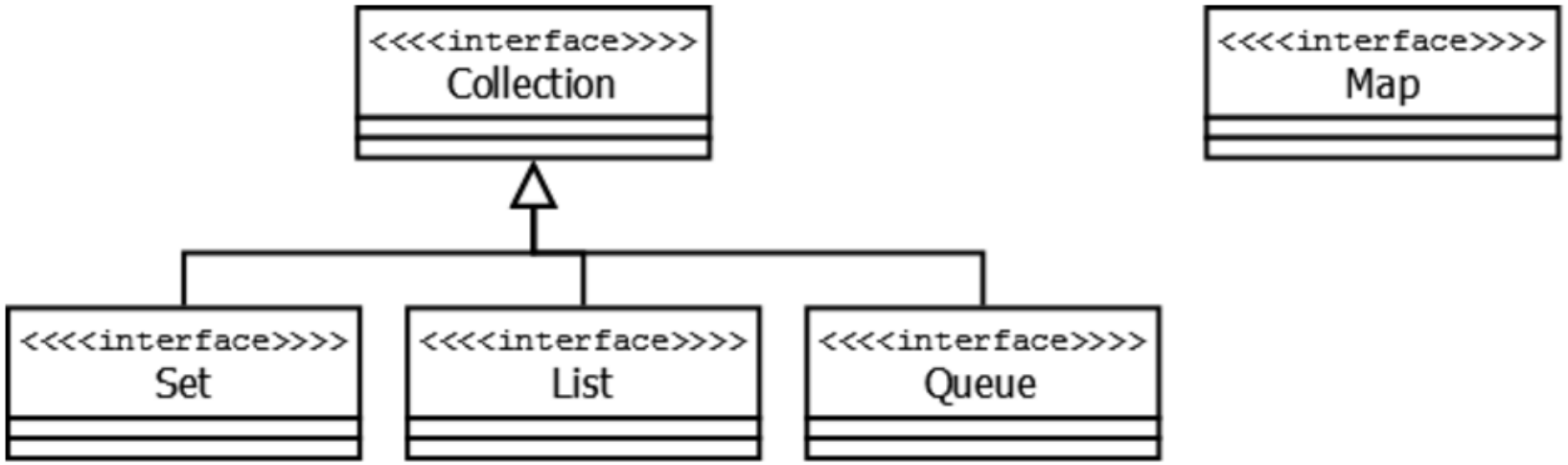
Основные приемы использования коллекций
Проход по элементам коллекции, шаблон проектирования Iterator

Сравнение и Сортировка элементов коллекции

# Коллекции в Java

**Java Collection Framework** — иерархия интерфейсов и их реализаций, которая является частью JDK и позволяет разработчику пользоваться большим количесвом структур данных из «коробки».

На вершине иерархии в Java Collection Framework располагаются 2 интерфейса: **Collection и Map.** Эти интерфейсы разделяют все коллекции, входящие во фреймворк, на две части по типу хранения данных: простые последовательные наборы элементов и наборы пар «ключ — значение» (словари).

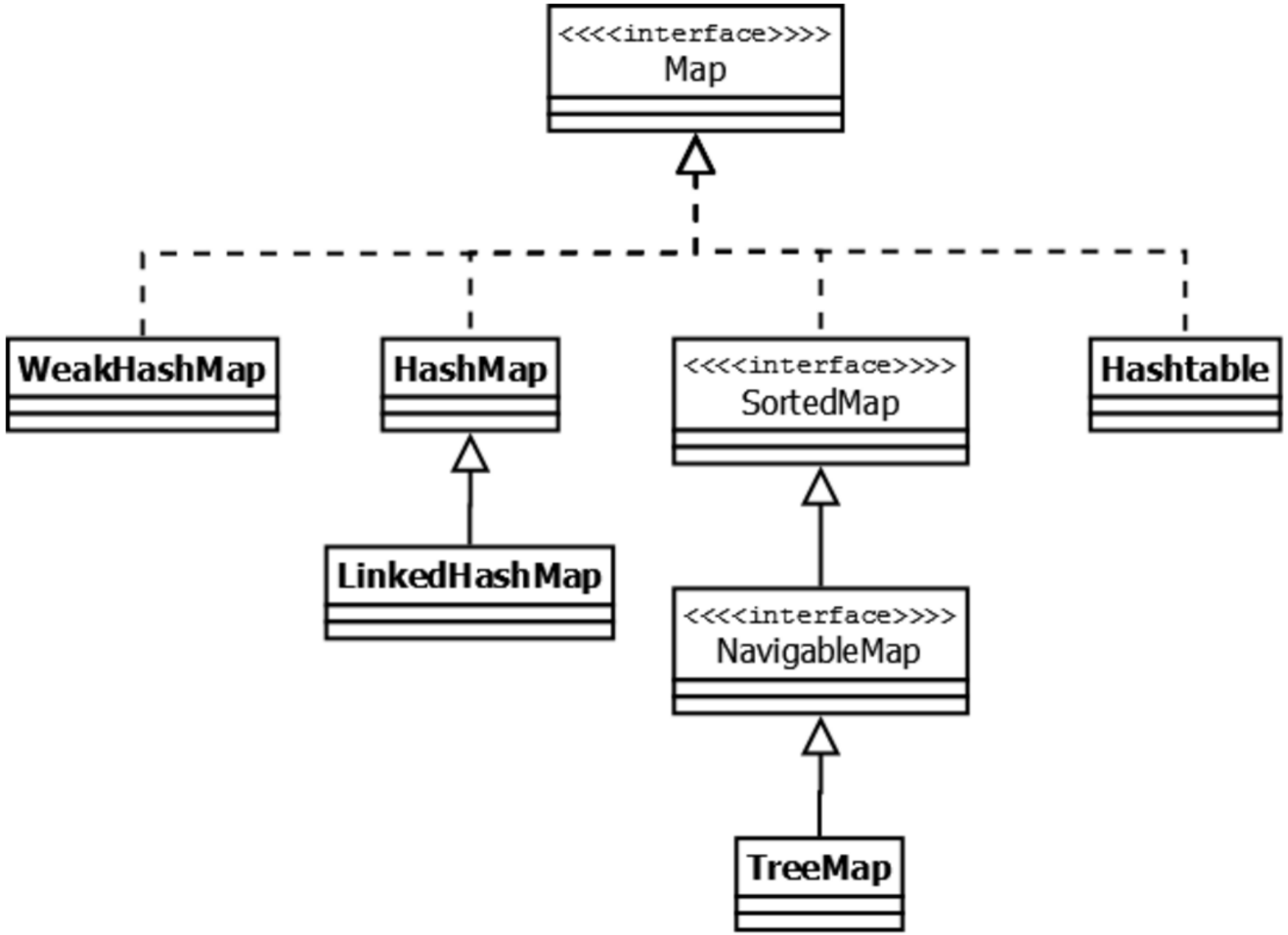


**Collection** — этот интерфейс находится в составе JDK с версии 1.2 и определяет основные методы работы с простыми наборами элементов, которые будут общими для всех его реализаций (например size(), isEmpty(), add(E e) и др.). Интерфейс был слегка доработан с приходом дженериков в Java 1.5. Так же в версии Java 8 было добавлено несколько новых методов для работы с лямбдами (такие как stream(), parallelStream(), removeIf(Predicate<? super E> filter) и др.).

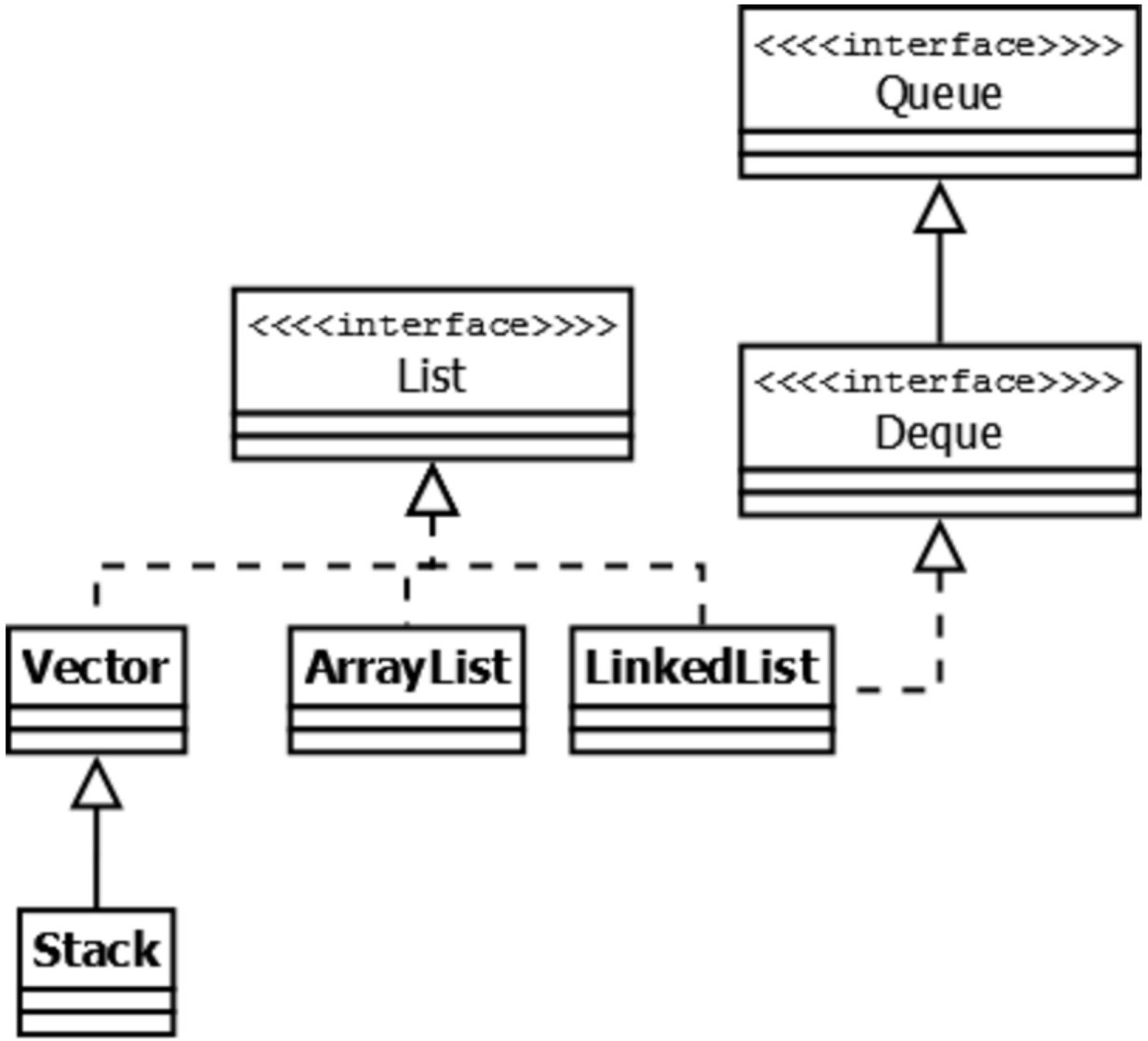
Важно также отметить, что эти медоды были реализованы непосредственно в интерфейсе как default-методы.

**Map**
Данный интерфейс также находится в составе JDK с версии 1.2 и предоставляет разработчику базовые методы для работы с данными вида «ключ — значение».Также как и Collection, он был дополнен дженериками в версии Java 1.5 и в версии Java 8 появились дополнительные методы для работы с лямбдами, а также методы, которые зачастую реализовались в логике приложения (getOrDefault(Object key, V defaultValue), putIfAbsent(K key, V value)).

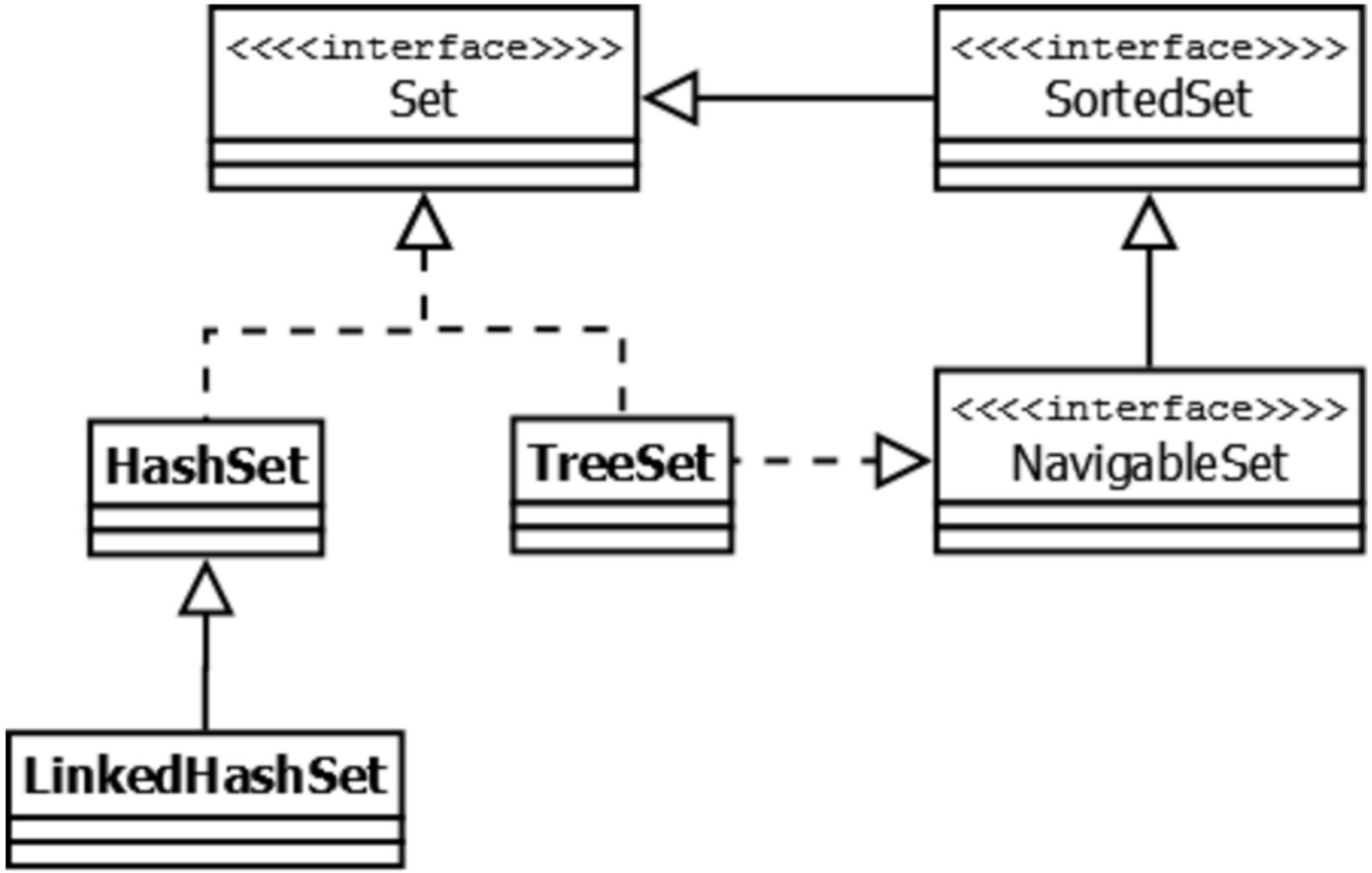
# *Интерфейс Map*



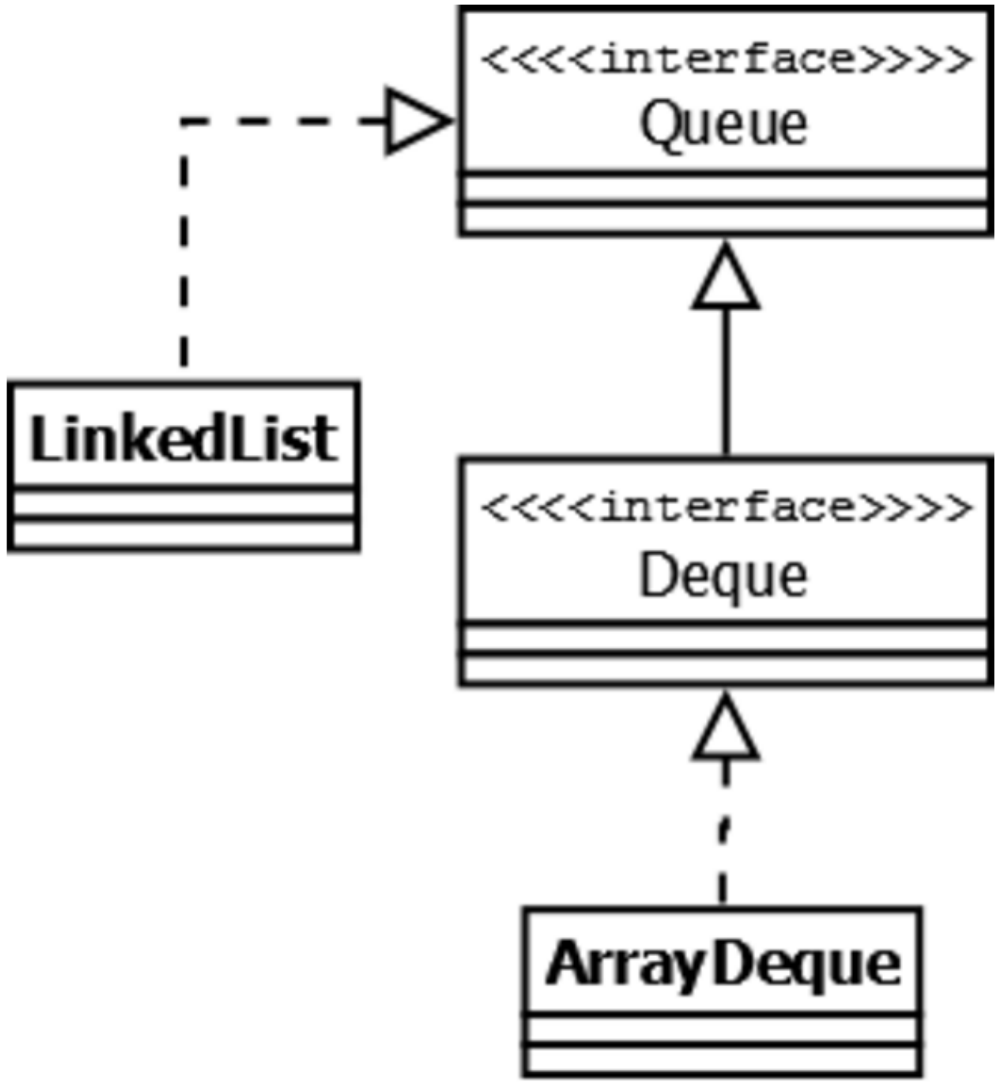
# *Интерфейс List*



# *Интерфейс Set*



# *Интерфейс Queue*



# Основные реализации коллекций

## Реализации интерфейса Map

**Hashtable** — реализация такой структуры данных, как хэш-таблица. Она не позволяет использовать null в качестве значения или ключа. Эта коллекция была реализована раньше, чем Java Collection Framework, но впоследствии была включена в его состав. Как и другие коллекции из Java 1.0, Hashtable является синхронизированной (почти все методы помечены как synchronized). Из-за этой особенности у неё имеются существенные проблемы с производительностью и, начиная с Java 1.2, в большинстве случаев рекомендуется использовать другие реализации интерфейса Map ввиду отсутствия у них синхронизации.

**HashMap** — коллекция является альтернативой Hashtable. Двумя основными отличиями от Hashtable являются то, что HashMap не синхронизирована и HashMap позволяет использовать null как в качестве ключа, так и значения. Так же как и Hashtable, данная коллекция не является упорядоченной: порядок хранения элементов зависит от хэш-функции. Добавление элемента выполняется за константное время O(1), но время удаления, получения зависит от распределения хэш-функции. В идеале является константным, но может быть и линейным O(n).

**HashMap** состоит из «корзин» (bucket`ов). С технической точки зрения «корзины» — это элементы массива, которые хранят ссылки на списки элементов. При добавлении новой пары ключ-значение  вычисляется хеш-код ключа, на основании которого вычисляется номер корзины (номер ячейки массива), в которую попадет новый элемент. Если корзина пустая, то в нее сохраняется ссылка на вновь добавляемый элемент, если же там уже есть элемент, то происходит последовательный переход по ссылкам между элементами в цепочке, в поисках последнего элемента, от которого и ставится ссылка на вновь добавленный элемент. Если в списке был найден элемент с таким же ключом, то он заменяется. Добавление, поиск и удаление элементов выполняется за константное время. Но стоит отметить, что хеш-функция должна равномерно распределять элементы по корзинам, в этом случае временная сложность для этих 3 операций будет не ниже lg N, а в среднем случае как раз константное время.
***Начальное количество корзин в HashMap — 16.*** Можно использовать конструкторы с параметрами: через параметр capacity задавать свое начальное количество корзин.

**Как и когда происходит увеличение количества корзин в HashMap?**
Помимо capacity в HashMap есть еще параметр loadFactor, на основании которого вычисляется предельное количество занятых корзин (capacity*loadFactor). По умолчанию loadFactor = 0,75. По достижению предельного значения  число корзин увеличивается в 2 раза. Для всех хранимых элементов вычисляется новое «местоположение» с учетом нового числа корзин.

**LinkedHashMap** — это упорядоченная реализация хэш-таблицы. Здесь, в отличии от HashMap, порядок итерирования равен порядку добавления элементов. Данная особенность достигается благодаря двунаправленным связям между элементами (аналогично LinkedList). Но это преимущество имеет также и недостаток — увеличение памяти, которое занимет коллекция.

**TreeMap** — реализация Map, основанная на красно-чёрных деревьях. Как и LinkedHashMap, является упорядоченной. По умолчанию, коллекция сортируется по ключам с использованием принципа "natural ordering", но это поведение может быть настроено под конкретную задачу при помощи объекта Comparator, который указывается в качестве параметра при создании объекта TreeMap.

**WeakHashMap** — реализация хэш-таблицы, которая организована с использованием weak references. Другими словами, Garbage Collector автоматически удалит элемент из коллекции при следующей сборке мусора, если на ключ этого элеметна нет жёстких ссылок.

# Реализации интерфейса 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).

**Vector** — реализация динамического массива объектов. Позволяет хранить любые данные, включая null в качестве элемента. Vector появился в JDK версии Java 1.0, но как и Hashtable, эту коллекцию не рекомендуется использовать, если не требуется достижения потокобезопасности. Потому как в Vector, в отличии от других реализаций List, все операции с данными являются синхронизированными. В качестве альтернативы часто применяется аналог — **ArrayList**.

**Stack** — данная коллекция является расширением коллекции Vector. Была добавлена в Java 1.0 как реализация стека LIFO (last-in-first-out). Является частично синхронизированной коллекцией (кроме метода добавления push()). После добавления в Java 1.6 интерфейса Deque, рекомендуется использовать именно реализации этого интерфейса, например **ArrayDeque**.

**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 поиндексное обращение к элементам выполняется за константное время O(1). Но данную коллекцию рекомендуется избегать, если требуется частое удаление/добавление элементов в середину коллекции.

**LinkedList** — ещё одина реализация List. Позволяет хранить любые данные, включая null. Особенностью реализации данной коллекции является то, что в её основе лежит двунаправленный связный список (каждый элемент имеет ссылку на предыдущий и следующий). Благодаря этому, добавление и удаление из середины, доступ по индексу, значению происходит за линейное время O(n), а из начала и конца за константное O(1). Так же, ввиду реализации, данную коллекцию можно использовать как стек или очередь. Для этого в ней реализованы соответсвующие методы. **LinkedList** предпочтительно применять, когда происходит активная работа (вставка/удаление) с серединой списка или в случаях, когда необходимо гарантированное время добавления элемента в список.

# Реализации интерфейса Set

Интерфейс Set представляет собой неупорядоченную коллекцию, которая не может содержать дублирующиеся данные. Является программной моделью математического понятия «множество».

**HashSet** — реализация 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.

**LinkedHashSet** — отличается от HashSet только тем, что в основе лежит LinkedHashMap вместо HashSet. Благодаря этому отличию порядок элементов при обходе коллекции является идентичным порядку добавления элементов.

**TreeSet** — аналогично другим классам-реализациям интерфейса Set содержит в себе объект NavigableMap, что и обуславливает его поведение. Предоставляет возможность управлять порядком элементов в коллекции при помощи объекта Comparator, либо сохраняет элементы с использованием "natural ordering".

# Реализации интерфейса Queue

Этот интерфейс описывает коллекции с предопределённым способом вставки и извлечения элементов, а именно — очереди FIFO (first-in-first-out). Помимо методов, определённых в интерфейсе Collection, определяет дополнительные методы для извлечения и добавления элементов в очередь. Большинство реализаций данного интерфейса находится в пакете java.util.concurrent и подробно рассматриваются при работе с многопоточностью.

**PriorityQueue** — является единственной прямой реализацией интерфейса Queue (была добавлена, как и интерфейс Queue, в Java 1.5), не считая класса LinkedList, который так же реализует этот интерфейс, но был реализован намного раньше. Особенностью данной очереди является возможность управления порядком элементов. По-умолчанию, элементы сортируются с использованием «natural ordering», но это поведение может быть переопределено при помощи объекта Comparator, который задаётся при создании очереди. Данная коллекция не поддерживает null в качестве элементов.

**ArrayDeque** — реализация интерфейса Deque, который расширяет интерфейс Queue методами, позволяющими реализовать конструкцию вида LIFO (last-in-first-out). Интерфейс Deque и реализация ArrayDeque были добавлены в Java 1.6. 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.

**Java Collections Framework** содержит большое количество различных структур данных, доступных в JDK «из коробки», которые в большинстве случаев покрывают все потребности при реализации логики приложения.

При необходимости, разработчик может создать собственную реализацию, расширив или переопределив существующую логику, либо создав свою собственную реализацию подходящего интерфейса с нуля. Также существует некоторое количество готовых решений, которые являются альтернативой или дополнением к Java Collections Framework. Наиболее популярными являются Google Guava и Commons Collections.

# Основные приемы использования коллекций

Массив имеет ряд недостатков. Один из самых существенных - размер массива фиксируется до начала его использования. Т.е. мы должны заранее знать или подсчитать, сколько нам потребуется элементов массива до начала работы с ним. Зачастую это неудобно, а в некоторых случаях — невозможно.

Коллекции обладают одним важным свойством — их размер не ограничен. Выделение необходимых для коллекции ресурсов спрятано внутри соответствующего класса. Работа с коллекциями облегчает и упрощает разработку приложений. Отсутствие же подобного механизма в составе средств разработки вызывает серьезные проблемы, которые приводят либо к различным ограничениям в реализации либо к самостоятельной разработке адекватных средств для хранения и обработки массивов информации заранее неопределенного размера.

В Java средства работы с коллекциями весьма развиты и имеют много приятных и полезных особенностей, облегчающих жизнь разработчикам.

Одним из широко используемых классов коллекций является **ArrayList.**

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

public class ArrayListTest {
     ArrayList lst = new ArrayList();
     Random generator = new Random();

     void addRandom() {
           lst.add(new Integer(generator.nextInt()));
     }


     public String toString() {
          return lst.toString();
     }

     public static void main(String args[]) {
         ArrayListTest tst = new ArrayListTest();
         for(int i = 0; i < 100; i++ )
                tst.addRandom();
          System.out.println("Сто случайных чисел: "+tst.toString());
     }
}
```

Этот пример не демонстрирует особых преимуществ коллекций, а лишь технику их использования. Из него видно, что добавить элемент в коллекцию можно методом add(...) класса ArrayList и при этом мы нигде не указываем размер коллекции.

Отображения в Java представлены несколькими классами. Базовым классом для всех отображений является абстрактный класс **AbstractMap**, который реализует большую часть методов интерфейса **Map**. Наиболее распространенным классом отображений является **HashMap**, который реализует интерфейс  **Map** и наследуется от класса **AbstractMap**.

Пример использования класса:

```
import java.util.*;

public class CollectionApp {

    public static void main(String[] args) {

        Map<Integer, String> states = new HashMap<Integer, String>();
        states.put(1, "Германия");
        states.put(2, "Испания");
        states.put(4, "Франция");
        states.put(3, "Италия");

        // получим объект по ключу 2
        String first = states.get(2);
        System.out.println(first);
        // получим весь набор ключей
        Set<Integer> keys = states.keySet();
        // получить набор всех значений
        Collection<String> values = states.values();
```

```
// получить набор всех значений
Collection<String> values = states.values();
//заменить элемент
states.replace(1, "Бельгия");
// удаление элемента по ключу 2
states.remove(2);
// перебор элементов
for(Map.Entry<Integer, String> item : states.entrySet()){

    System.out.printf("Ключ: %d  Значение: %s \n", item.getKey(), item.getValue());
}

Map<String, Person> people = new HashMap<String, Person>();
people.put("1240i54", new Person("Tom"));
people.put("1564i55", new Person("Bill"));
people.put("4540i56", new Person("Nick"));
```

```
        Map<String, Person> people = new HashMap<String, Person>();
        people.put("1240i54", new Person("Tom"));
        people.put("1564i55", new Person("Bill"));
        people.put("4540i56", new Person("Nick"));

        for(Map.Entry<String, Person> item : people.entrySet()){

            System.out.printf("Ключ: %s  Значение: %s \n", item.getKey(), item.getValue().getName());
        }
    }
}
class Person{

    private String name;
    public Person(String value){

        name=value;
    }
    String getName(){return name;}
}
```

Чтобы добавить или заменить элемент, используется метод put, либо replace, а чтобы получить его значение по ключу - метод get. С помощью других методов интерфейса Map также производятся другие манипуляции над элементами: перебор, получение ключей, значений, удаление.

Проход по каждой паре ключ-значение — самая базовая, основная процедура прохода по Map. В Java каждая пара хранится в поле Map, называемом **Map.Entry**. **Map**.*entrySet()* возвращает набор ключ-значений, потому самым эффективным способом пройтись по всем значениям Map будет:

```
for(Entry entry: Map.entrySet()) {
  //получить ключ
  K key = entry.getKey();
  //получить значение
  V value = entry.getValue();
}
```

Также мы можем использовать Iterator, особенно в версиях младше JDK 1.5

```
Iterator itr = Map.entrySet().iterator();
while(itr.hasNext()) {
   Entry entry = itr.next();
   //получить ключ
   K key = entry.getKey();
   //получить значение
   V value = entry.getValue();
}
```

# Проход по элементам коллекции, шаблон проектирования Iterator

Идея паттерна Iterator заключается в том, что к коллекции привязывается объект, который поможет обойти в некотором порядке все элементы коллекции. Рассмотрим интерфейс Iterator реализующий данный подход:

- next() — возвращает следующий элемент коллекции, при этом итератор переходит на следующий элемент
- hasNext() — возвращает существует ли следующий элемент
- remove() — удаляет текущий элемент

Интерфейс Collection имеет метод iterator(), который возвращает итератор текущей коллекции. Поэтому обход произвольной коллекции можно реализовать следующим образом:

```
Collection<Integer> c;
Iterator<Integer> iterator = c.iterator();
while (iterator.hasNext()) {
    System.out.print(iterator.next());
}
```

Более интересный итератор применяется в случае, если мы имеем дело с упорядоченной коллекцией, т.е коллекцией, которая реализует интерфейс List. Тогда в качестве итератора мы можем использовать ListIterator, который содержит следующие методы:

- previous() — возвращает предыдущий элемент из коллекции, при этом итератор переходит на предыдущий элемент
- hasPrevious() — возвращает существует ли предыдущий элемент
- add(T e) — добавляет новый элемент перед текущим элементом
- set(T e) — заменяет текущий элемент
- nextIndex() — возвращает индекс следующего элемента
- previousIndex() — возвращает индекс предыдущего элемента

Для того, чтобы получить ListIterator, необходимо вызвать listIterator() у объекта, который реализует интерфейс List.

# Как избежать ConcurrentModificationException при использовании коллекций

Название ConcurrentModificationException многих вводит в заблуждение. При слове Concurrent первое, что приходит на ум — многотредность. Однако данное исключение относится вовсе не к многотредности. Исключение может происходить при работе с коллекциями при обычной однотредной работе.

Для «прохода» по коллекции используются структуры данных, реализующие паттерн «View». Структуры данных эти называются итераторами и могут использоваться явно и не явно. Неявно итераторы используются при использовании конструкции foreach.

ConcurrentModificationException возникает, когда коллекция модифицируется «одновременно» с проходом по коллекции итератором любыми средствами, кроме самого итератора.

Например, ниже при удалении элемента из map произойдет ConcurrentModificationException, поскольку в цикле for неявно формируется итератор, и из map удаляется элемент, в процессе прохода по map.

```java
@Test(expected = ConcurrentModificationException.class)
public void testForEachFail() {
    Map<String,String> map = new HashMap<String,String>();
    map.put("a", "a");
    map.put("b", "b");

    for(String key:map.keySet()) {
        map.remove(key);
    }
}
```

Заблуждения добавляет тот факт, что в случае, если в map в приведенном примере будет лишь один элемент, то ConcurrentModificationException не возникнет (ниже тот же код, но добавляется лишь один элемент в map):

```java
@Test
public void testForEachOneElement() {
    Map<String,String> map = new HashMap<String,String>();
    map.put("a", "a");

    for(String key:map.keySet()) {
        map.remove(key);
    }
}
```

Тест выше проходит, несмотря на то, что неверен… Еще один пример неверного кода, уже со StackOverflow:

```
Iterator it = map.entrySet().iterator();
while (it.hasNext())
{
    Entry item = it.next();
    map.remove(item.getKey());
}
```

В вышеприведенном примере, ConcurrentModificationException возникает при явном использовании итератора для прохода по элементам. Но в то же самое время элемент удаляется по ключу: map.remove(item.getKey()).

Как же сделать удаление правильно? Нужно использовать метод итератора для удаления элемента:

```
Iterator it = map.entrySet().iterator();
while (it.hasNext())
{
    Entry item = it.next();
    it.remove(item);
}
```

ConcurrentModificationException не всегда является следствием неверной синхронизации работы с коллекциями. ConcurrentModificationException возникает при изменении коллекции любыми средствами, отличными от итератора, при проходе по коллекции с помощью итератора. Изменение такое, как продемонстрировано выше, может происходить и в одно-тредной среде. Можно также добавить, что исключение может возникать не только при удалении, но и при добавлении элемента, а также что модификация контейнера, может возникать и в многотредной среде. Так что исключение не обязательно тривиальное.

# Сравнение и Сортировка элементов коллекции

Интерфейс Comparator описывает два метода сравнения:

int compare (Object obji, object obj2 ) — возвращает отрицательное число, если objl в каком-то смысле меньше obj2 ; нуль, если они считаются равными; положительное число, если objl больше obj2 .

boolean equals (Object obj) — сравнивает данный объект с объектом obj , возвращая true , если объекты совпадают в каком-либо смысле, заданном этим методом.

Для каждой коллекции можно реализовать эти два метода, задав конкретный способ сравнения элементов. Элементы коллекции будут автоматически отсортированы в заданном порядке.

Ниже показан один из возможных способов упорядочения комплексных чисел — объектов класса Complex. Здесь описывается класс ComplexCompare , реализующий интерфейс Comparator ,  он применяется для упорядоченного хранения множества комплексных чисел.

## Сравнение комплексных чисел

```
import java.util.*;

class ComplexCompare implements Comparator{

public int compare(Object objl, Object obj2){

Complex zl = (Complex)objl, z2 = (Complex)obj2;

double rel = zl.getReO, iml = zl.getlm();

double re2 = z2.getRe(), im2 = z2.getlm();

if (rel != re2) return (int)(rel - re2);

else if (iml != im2) return (int)(iml — im2);

else return 0;

}

public boolean equals(Object z) {

return compare(this, z) == 0;

 }

}
```

# Интерфейсы Comparable и Comporator. Сортировка

Так как TreeSet реализует интерфейс NavigableSet, а через него и SortedSet, то мы можем применить к структуре дерева различные методы:

```
import java.util.*;

public class CollectionApp {

    public static void main(String[] args) {

        TreeSet<String> states = new TreeSet<String>();

        states.add("Германия");
        states.add("Франция");
        states.add("Италия");
        states.add("Великобритания");
        System.out.println(states.first()); // получим первый - самый меньший элемент
        System.out.println(states.last()); // получим последний - самый больший элемент
        // получим поднабор от одного элемента до другого
        SortedSet<String> set = states.subSet("Германия", "Франция");
        System.out.println(set);
        // элемент из набора, который больше текущего
        String greater = states.higher("Германия");
        // элемент из набора, который больше текущего
        String lower = states.lower("Германия");
```

```
        // возвращаем набор в обратном порядке
        NavigableSet<String> navSet = states.descendingSet();
        // возвращаем набор в котором все элементы меньше текущего
        SortedSet<String> setLower=states.headSet("Германия");
        // возвращаем набор в котором все элементы больше текущего
        SortedSet<String> setGreater=states.tailSet("Германия");
    }
}
```

Это работа коллекции TreeSet, типизированной объектами String. При добавлении новых элементов объект TreeSet автоматически проводит сортировку, помещая новый объект на правильное для него место. Однако со строками все понятно. А что если бы мы использовали не строки, а свои классы, например, следующий класс Person:

```
class Person{

    private String name;
    public Person(String value){

        name=value;
    }
    String getName(){return name;}
}
```

Объект TreeSet мы не сможем типизировать данным классом, поскольку в случае добавления объектов TreeSet не будет знать, как их сравнивать, и следующий кусок кода не будет работать:

```
TreeSet<Person> people = new TreeSet<Person>();
people.add(new Person("Tom"));
```

Для того, чтобы объекты Person можно было сравнить и сортировать, они должны применять интерфейс Comparable<E>. При применении интерфейса он типизируется текущим классом. Применим его к классу Person:

```java
class Person implements Comparable<Person>{

    private String name;
    public Person(String value){

        name=value;
    }
    String getName(){return name;}

    public int compareTo(Person p){

        return name.compareTo(p.getName());
    }
}
```

Интерфейс Comparable содержит один единственный метод int compareTo(E item), который сравнивает текущий объект с объектом, переданным в качестве параметра. Если этот метод возвращает отрицательное число, то текущий объект будет располагаться перед тем, который передается через параметр. Если метод вернет положительное число, то, наоборот, после второго объекта. Если метод возвратит ноль, значит, оба объекта равны.

В данном случае мы не возвращаем явным образом никакое число, а полагаемся на встроенный механизм сравнения, который есть у класса String. Но мы также можем определить и свою логику, например, сравнивать по длине имени:

```
public int compareTo(Person p){

    return name.length()-p.getName().length();
}
```

Теперь мы можем типизировать TreeSet типом Person и добавлять в дерево соответствующие объекты:

```
TreeSet<Person> people = new TreeSet<Person>();
people.add(new Person("Tom"));
```

Однако перед нами может возникнуть проблема, что если разработчик не реализовал в своем классе, который мы хотим использовать, интерфейс Comparable, либо реализовал, но нас не устраивает его функциональность, и мы хотим ее переопределить? На этот случай есть еще более гибкий способ, предполагающий применение интерфейса Comparator<E>.

```
1  public interface Comparator<E> {
2
3      int compare(T a, T b);
4  }
```

Метод compare также возвращает числовое значение - если оно отрицательное, то объект a предшествует объекту b, иначе - наоборот. А если метод возвращает ноль, то объекты равны. Для применения интерфейса нам сначала надо создать класс компаратора, который реализует этот интерфейс:

```
class PersonComparator implements Comparator<Person>{

    public int compare(Person a, Person b){

        return a.getName().compareTo(b.getName());
    }
}
```

Здесь опять же проводим сравнение по строкам. Теперь используем класс компаратора для создания объекта TreeSet:

```
1  PersonComparator pcomp = new PersonComparator();
2  TreeSet<Person> people = new TreeSet<Person>(pcomp);
3  people.add(new Person("Tom"));
4  people.add(new Person("Nick"));
5  people.add(new Person("Alice"));
6  people.add(new Person("Bill"));
7  for(Person  p : people){
8
9      System.out.println(p.getName());
10 }
```

Для создания TreeSet здесь используется одна из версий конструктора, которая в качестве параметра принимает компаратор. Теперь вне зависимости от того, реализован ли в классе Person интерфейс Comparable, логика сравнения и сортировки будет использоваться та, которая определена в классе компаратора.