КОЛЕКЦИИ

ЛЕКЦИОНЕН КУРС "ООП(JAVA)"

Distributed
@learning center

КОНТЕЙНЕРНИ КЛАСОВЕ

- Често използвани структури данни като списъци, множества, карти (maps), …
- Реализацията на такива структури в Java – посредством контейнерни класове
- Наречени така, понеже се използват за управление на други класове
- В Collections-Framework: контейнерните класове са представени чрез интерфейси (пакет java.util), като напр. List<E>, Set<E>, Map<E>

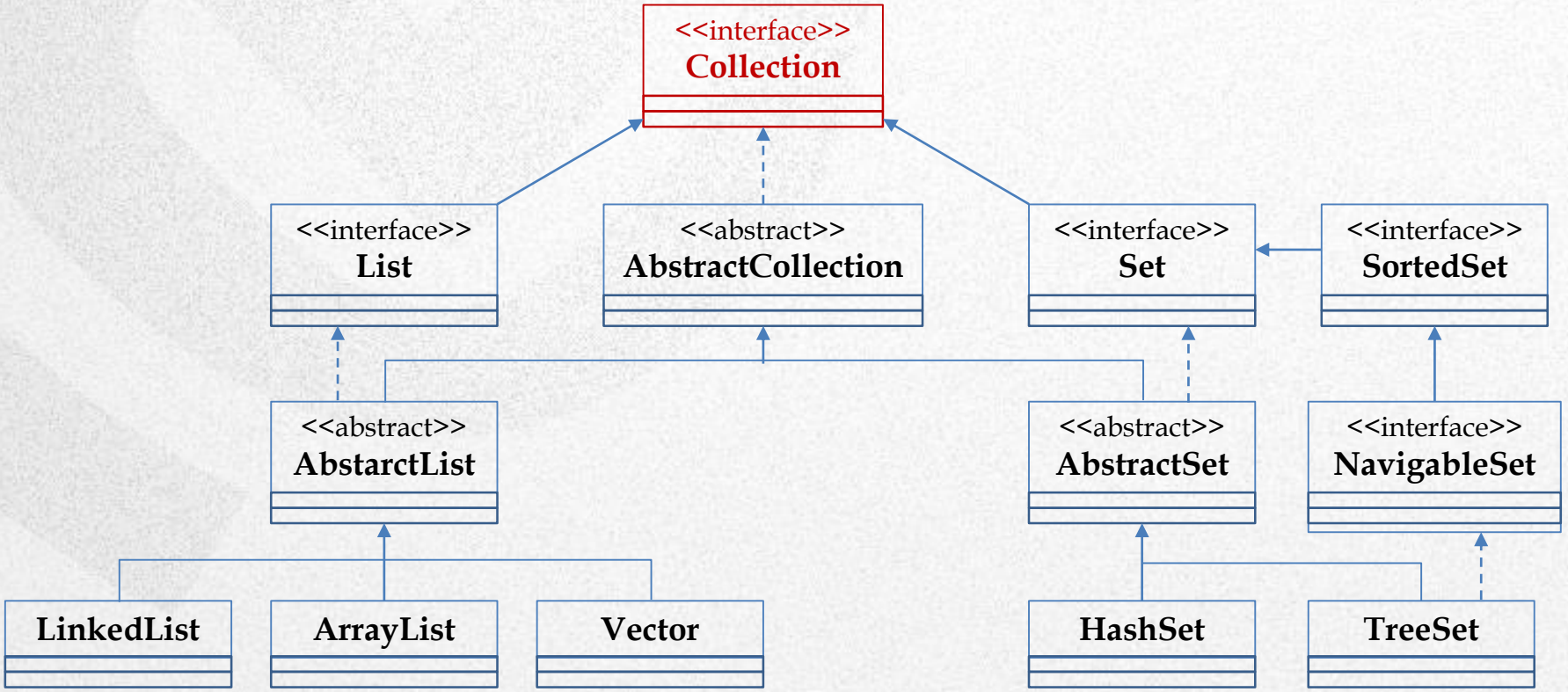
ОСОБЕНОСТ

- Само масивите могат да съхраняват елементи от произволен тип – особено примитивните типове
- Всички контейнерни класове могат да съхраняват референции на обекти
- Управлението на примитивни типове в тях само там възможно, където тези типове могат да се трансформират в Wrapper обекти

КЛАСИФИКАЦИЯ

- Две йерархии
  - Интерфейс Collection<E>
  - Интерфейс Map<K, V>
- Когато се избира подходяща структура данни в зависимост от конкретния проблем трябва да се избере между:
  - Списъци или множества
  - Карти (maps)
- След това да се намери подходяща реализация на избрания интерфейс

КЛАСИФИКАЦИЯ INTERFACE "COLLECTION"



ИНТЕРФЕЙС COLLECTION

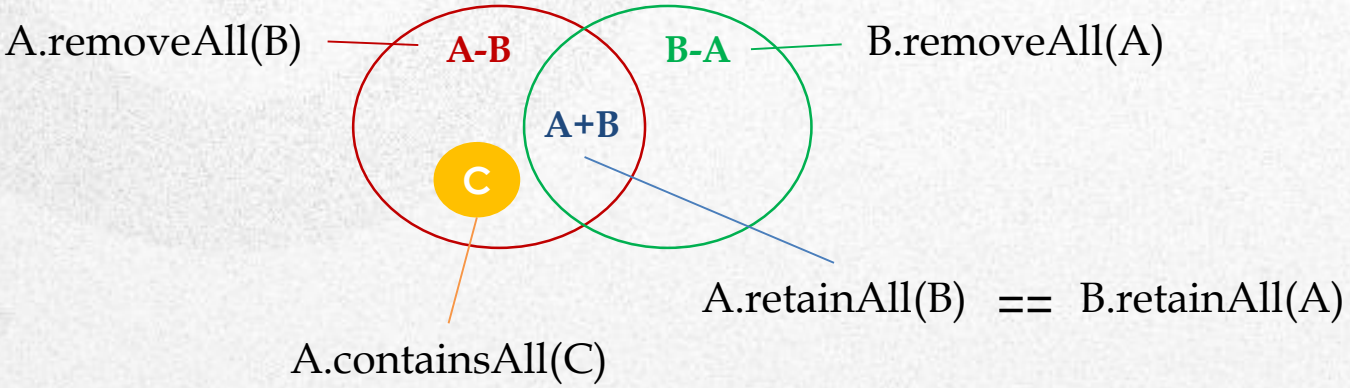
- Дефинира основата за различни контейнерни класове, които удовлетворяват интерфейсите:
  - List<E> - като списъци
  - Set<E> - като множества
- Интерфейсът не предлага индексен достъп
- Предлага група общи за използвани методи

МЕТОДИ НА ИНТЕРФЕЙСА

Повече от 50 метода

Метод	Функция
int size()	Брой елементи в колекцията
boolean isEmpty()	Тест за наличие на елементи в колекцията
boolean add(E element)	Добавяне на елемент в колекцията
boolean addAll(Collection<? Extends E> collection)	Bulk оператори
boolean remove(Object object)	Отстраняване на елемент от колекцията
boolean removeAll(Collection<?> collection)	Отстраняване множество елементи от колекцията
boolean contains(Object object)	Проверява дали елемент се съдържа в колекцията
boolean containsAll(Collection<?> collection)	Проверява дали елементи се съдържат в колекцията
boolean retainAll(Collection<?> collection)	Проверява дали елементи в две колекции

ОПЕРАЦИИ С МНОЖЕСТВА И МЕТОДИТЕ



РАБОТА С КОЛЕКЦИИ

```
import java.util.Collections;
import java.util.LinkedList;
import java.util.List;
public final class AlgorithmsExample {
   private static final Person MALE     = new Person("Male", "Bremen", 42);
   private static final Person FEMALE   = new Person("Female", "New York", 43);
   private static final Person MISTER_X = new Person("Mister X", "Sydney", 44);

   private static List<Person> initPersonList()  {
      final List<Person> maleList = Collections.nCopies(2, MALE);
      final List<Person> femaleList = Collections.nCopies(3, FEMALE);
      final List<Person> persons = new LinkedList<Person>();
      persons.addAll(maleList);
      persons.addAll(femaleList);
      persons.add(MISTER_X);
      return persons;
   }

   public static void main(final String[] args)   {
      final List<Person> persons = initPersonList();
      final int maleCount = Collections.frequency(persons, MALE);
      System.out.println("Male Persons: " + maleCount);
      System.out.println("All Persons: " + persons);
   }
}
```

Връща неизменчив (immutable) списък, състоящ се от n копия на специфицирания обект

Връща броя на елементите в дадената колекция

РАБОТА С КОЛЕКЦИИ

```
import java.util.Collections;
import java.util.LinkedList;
import java.util.List;
public final class AlgorithmsExample {
   private static final Person MALE     = new Person("Male", "Bremen", 42);
   private static final Person FEMALE   = new Person("Female", "New York", 43);
   private static final Person MISTER_X = new Person("Mister X", "Sydney", 44);

   private static List<Person> initPersonList()  {
      final List<Person> maleList = Collections.nCopies(2, MALE);
      final List<Person> femaleList = Collections.nCopies(3, FEMALE);
      final List<Person> persons = new LinkedList<Person>();
      persons.ad
      persons.ad
      persons.ad
      return pers
   }

   public static void main(final String[] args)   {
      final List<Person> persons = initPersonList();
      final int maleCount = Collections.frequency(persons, MALE);
      System.out.println("Male Persons: " + maleCount);
      System.out.println("All Persons: " + persons);
   }
}
```

```
Male Persons: 2
All Persons: [Person: Name='Male' City='Plovdiv' Age='42', Person: Name='Male' City='Plovdiv'
Age='42', Person: Name='Female' City='New York' Age='43', Person: Name='Female' City='New York'
Age='43', Person: Name='Female' City='New York' Age='43', Person: Name='Mister X' City='Sydney'
Age='44']
```

РАБОТА С КОЛЕКЦИИ

```
public final class AlgorithmsExampleMinMax  {
   private static final Person MALE     = new Person("Male", “Plovdiv", 42);
   private static final Person FEMALE   = new Person("Female", "New York", 43);
   private static final Person MISTER_X = new Person("Mister X", "Sydney", 44);
   public static void main(final String[] args)     {
      final List<Person> persons = initPersonList();
      final Person min = Collections.min(persons);
      System.out.println("Min: " + min);
      final Person max = Collections.max(persons);
      System.out.println("Max: " + max);
      final Comparator<Person> cityComparator = new Comparator<Person>() {
         public int compare(final Person person1, final Person person2)   {
            return person1.getCity().compareTo(person2.getCity());
         }
      };
      final Person maxCity = Collections.max(persons, cityComparator);
      System.out.println("Max city: " + maxCity);
   }
   private static List<Person> initPersonList()    {
      final List<Person> maleList = Collections.nCopies(2, MALE);
      final List<Person> femaleList = Collections.nCopies(3, FEMALE);
      final List<Person> persons = new LinkedList<Person>();
      persons.addAll(maleList);
      persons.addAll(femaleList);
      persons.add(MISTER_X);
      return persons;
```

Връща минималния елемент на дадената колекция, съответно наредбата, въведена от специфицирания Comparator

Връща максималния елемент на дадената колекция, съответно наредбата, въведена от специфицирания Comparator

```
Min: Person: Name='Female' City='New York' Age='43'
Max: Person: Name='Mister X' City='Sydney' Age='44'
Max city: Person: Name='Mister X' City='Sydney' Age='44'
```

РАБОТА С КОЛЕКЦИИ

```
import java.util.Collections;
import java.util.LinkedList;
import java.util.List;
public final class AlgorithmsExampleShuffleReplaceAll  {
   private static final Person MALE     = new Person("Male", "Bremen", 42);
   private static final Person FEMALE   = new Person("Female", "New York", 43);
   private static final Person MISTER_X = new Person("Mister X", "Sydney", 44);
   public static void main(final String[] args)    {
      final List<Person> persons = initPersonList();
      Collections.shuffle(persons);
      System.out.println("All Persons after shuffle: " + persons);
      Collections.replaceAll(persons, MALE, MISTER_X);
      System.out.println("All Persons after replace:" + persons);
      final int maleCount = Collections.frequency(persons, MALE);
      final int misterXCount = Collections.frequency(persons, MISTER_X);
      System.out.println("Male Persons: " + maleCount);
      System.out.println("MisterX Persons: " + misterXCount);
   }
   private static List<Person> initPersonList()     {
      final List<Person> maleList = Collections.nCopies(2, MALE);
      final List<Person> femaleList = Collections.nCopies(3, FEMALE);
      final List<Person> persons = new LinkedList<Person>();
      persons.addAll(maleList);
      persons.addAll(femaleList);
      persons.add(MISTER_X);
      return persons;
```

Замества всички появявания на специфицирана стойност с друга в един списък

Случайно разместване на специфицирания списък, използвайки източник по подразбиране за случайност

РАБОТА С КОЛЕКЦИИ

```
import java.util.Collections;
import java.util.LinkedList;
import java.util.List;
public final class AlgorithmsExampleShuffleReplaceAll  {
   private static final Person MALE     = new Person("Male", "Bremen", 42);
   private static final Person FEMALE   = new Person("Female", "New York", 43);
   private static final Person MISTER_X = new Person("Mister X", "Sydney", 44);
   public static void main(final String[] args)    {
      final List<Person> persons = initPersonList();
      Collections.shuffle(persons);
      System.out.println("All Persons after shuffle: " + persons);
      Collections.replaceAll(persons, MALE, MISTER_X);
      S
      fi
      fi
      S
      S
   }
   priv
      fi
      fi
      fi
      p
      persons.addAll(femaleList);
      persons.add(MISTER_X);
      return persons;
```

```
All Persons after shuffle: [Person: Name='Female' City='New York' Age='43', Person: Name='Mister X'
City='Sydney' Age='44', Person: Name='Female' City='New York' Age='43', Person: Name='Male' City='Plovdiv'
Age='42', Person: Name='Male' City='Plovdiv' Age='42', Person: Name='Female' City='New York' Age='43']

All Persons after replace:[Person: Name='Female' City='New York' Age='43', Person: Name='Mister X'
City='Sydney' Age='44', Person: Name='Female' City='New York' Age='43', Person: Name='Mister X'
City='Sydney' Age='44', Person: Name='Mister X' City='Sydney' Age='44', Person: Name='Female' City='New
York' Age='43']

Male Persons: 0
MisterX Persons: 3
```

МАСИВИ

- Само те могат да съхраняват елементи от произволен тип
  - Специално директни примитивни типове като byte, int, double, …
- Другите контейнерни класове могат да съхраняват референции на обекти
  - Управлението на примитивни типове възможно само чрез Wrapper-обекти

ИТЕРАТОРИ

Iterator: абстракция на по-високо ниво

- Обект, чиято задача е да се подпомага преминаването (траверс) през съдържанието на контейнерни структури от данни и да избира всеки обект от този контейнер, без клиент-програмистът да знае или да се интересува от структурата ѝ
- „олекотен“ обект – не изисква много средства за да се създаде

Интерфейс java.util.Iterator<E>: моделира итератор

Употреба:

- Next() – получаваме следващ елемент
- hasNext() – проверка за наличие на елементи

ПРИМЕР ЗА ИТЕРАТОР

```
import java.util.Arrays;
import java.util.Collection;
import java.util.Iterator;
public final class IterationExample
{
   public static void main(final String[] args)   {
      final String[] textArray = { "Траверс", "с", "Iterator" };

      final Collection<String> infoTexts = Arrays.asList(textArray);

      final Iterator<String> it = infoTexts.iterator();

      while (it.hasNext())   {
         System.out.print(it.next());
         if (it.hasNext())  // Съществуват ли още елементи
               System.out.print(", ");
      }
   }
```

Преобразуваме масив в списък

Създаваме итератор

Траверс, с, Iterator

ПРИМЕР ЗА ИЗТРИВАНЕ В ИТЕРАТОР

```
import java.util.ArrayList;
import java.util.Arrays;
import java.util.Iterator;
import java.util.List;
public final class IteratorCollectionRemoveExample  {
   private static void removeEntriesWithPrefix(final List<String> entries, final String prefix)  {
      final Iterator<String> it = entries.iterator();
      while (it.hasNext())    {
         final String name = it.next();
         if (name.startsWith(prefix))     {

                  entries.remove(name);
         }
      }
   }

   public static void main(final String[] args)    {
      final String[] names = { "Иван", "Мария", "Ива", "Петър", "Илия" };

      final List<String> namesList = new ArrayList<String>();
      namesList.addAll(Arrays.asList(names));

      removeEntriesWithPrefix(namesList, „И");
      System.out.println(namesList);
   }
}
```

Интуитивно изглежда коректно

ПРИМЕР: ИЗТРИВАНЕ В ИТЕРАТОР

```
import java.util.ArrayList;
import java.util.Arrays;
import java.util.Iterator;
import java.util.List;
public final class IteratorCollectionRemoveExample  {
   private static void removeEntriesWithPrefix(final List<String> entries, final String prefix)  {
      final Iterator<String> it = entries.iterator();
      while (it.hasNext())    {
         final String name = it.next();
         if (name.startsWith(prefix))     {

                  entries.remove(name);
         }
      }
      namesList.addAll(Arrays.asList(names));

      removeEntriesWithPrefix(namesList, "M");
      System.out.println(namesList);
   }
}
```

```
Exception in thread "main" java.util.ConcurrentModificationException
at java.util.ArrayList$Itr.checkForComodification(Unknown Source)
at java.util.ArrayList$Itr.next(Unknown Source)
at
collections.IteratorCollectionRemoveExample.removeEntriesWithPrefix(IteratorCollectionRemoveExample.java:24)
at collections.IteratorCollectionRemoveExample.main(IteratorCollectionRemoveExample.java:41)
```

ПРИЧИНА ЗА ГРЕШКАТА

- Такова изключение обикновено сочи към проблеми, свързани с промени на структура данни с паралелен достъп посредством повече нишки
  - ✓ В случая имаме обаче, само една нишка
- Изключението е предизвикано от това, че във всяка колекция се поддържа модификационен брояч за защита при конкуриращ се достъп
- Всеки итератор извежда стойността му в началото на итерацията – при всяко извикване на next() сравнява актуалната стойност с началната
  - ✓ Ако има отклонение се предизвиква изключение
- Това поведение на итератора се нарича fail-fast

РЕШЕНИЕ НА ПРОБЛЕМА

- Причина за грешката: извикването на метода remove(Object) върху структурата от данни entries води до промяна на модификационния брояч
- Единствено сигурен начин за изтриване на елементи от една колекция по време на итерация е посредством извикване на метода remove() от Iterator<E>
- По тази причина методът remove() е реализиран в Collection<E> и в Iterator<E>
  - ✓ В Iterator<E> не е необходим параметър, понеже се изтрива получения от next() елемент

ПРИМЕР: ИЗТРИВАНЕ В ИТЕРАТОР

```
import java.util.ArrayList;
import java.util.Arrays;
import java.util.Iterator;
import java.util.List;

public final class IteratorCorrectRemoveExample
{
   private static void removeEntriesWithPrefix(final List<String> entries, final String prefix)
   {
      final Iterator<String> it = entries.iterator();
      while (it.hasNext())
      {
         final String name = it.next();
         if (name.startsWith(prefix))
         {
            // entries.remove(name); → некоректно
            it.remove();// коректно
         }
      }
   }
```

ПРИМЕР: ИЗТРИВАНЕ В ИТЕРАТОР

```
public static void main(final String[] args)
    {
        final String[] names = { "Иван", "Мария", "Ива", "Петър", "Илия" };

        final List<String> namesList = new ArrayList<String>();
        namesList.addAll(Arrays.asList(names));

        removeEntriesWithPrefix(namesList, "И");
        System.out.println(namesList);
    }
}
```

[Мария, Петър]

СПИСЪЦИ

- Списък: наредена последователност от елементи
- За описание на списъци рамката предлага интерфейса List<E>
- Известни реализации на този интерфейс са класовете:
  - ArrayList<E>
  - LinkedList<E>
  - Vector<E>
- Интерфейсът прави възможен индексен достъп, както добавяне и премахване на елементи
- Интерфейсът е основа за всички видове списъци и предлага допълнително (към интерфейса Collection) нови методи

МЕТОДИ НА ИНТЕРФЕЙСА

Метод	Функция
E get (int index)	Доставя елемента от дадената позиция
void add(int index, E element)	Въвежда елемент в посочената позиция
E set(int index, E element)	Замества елемент от посочената позиция
E remove(int index)	Отстранява елемент от посочената позиция
int indexOf(Object object)	Доставя позиция на елемент, започвайки от началото
int lastIndexOf(Object object)	Доставя позиция на елемент, започвайки от края

ПРИМЕР: СПИСЪК

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

public final class FirstListExample  {
   public static void main(final String[] args) {
      final List<String> list = new ArrayList<>();
      list.add("First");
      list.add("Last");
      list.add("Middle");
      System.out.println("List: " + list);

      System.out.println("3rd: " + list.get(2));

      list.remove(0);
      list.remove(list.indexOf("Last"));

      System.out.println("List: " + list);

   }
}
```

Въвеждане елементи (в края на списъка)

Достъп посредством индекс

Изтриване на елементи

```
List: [First, Last, Middle]
3rd: Middle
List: [Middle]
```

ПРИМЕР: ПОДСПИСЪК

[Error1, Error2, Error3]

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

public final class SubListExample  {
   public static void main(final String[] args) {
      final List<String> errors = new ArrayList<>(Arrays.asList( "Error1", "Error2", "Error3",
                                                                 "Critical Error", "Fatal Error" ));
      truncateListToMaxSize(errors, 3);
      System.out.println(errors);
   }

   private static void truncateListToMaxSize(final List<?> listToTruncate, final int maxSize) {
       if (listToTruncate.size() > maxSize) {
         final List<?> entriesAfterMaxSize = listToTruncate.subList(maxSize,  listToTruncate.size());
                          entriesAfterMaxSize.clear();
       }
   }
}
```

Изтрива всички елементи от списъка

Връща част от списъка между първия (включен) и втория (изключен) параметър

МНОЖЕСТВА

- Множествата не съдържат дубликати
- Множествата се описват посредством интерфейса Set<E>
- За разлика от интерфейса List<E> този интерфейс не доставя допълнителни методи към тези на интерфейса Collection
  - Само различно поведение на методите add(…) и addAll(…) – за да се генерира уникалност на елементите

ОСОБЕНОСТИ

- Две различни множества:
  - HashSet<E> - съхранява елементите неподредено в хеш-контейнер
    - Чрез това – минимално време за операции като напр. add(E), remove(Object), contains(Object)
  - TreeSet<E> - имплементира интерфейса SortedSet<E> и съхранява елементите сортирано
    - Сортирането се определя посредством:
      - Интерфейса Comparable<T> или
      - Comparator<T> - явно предаден в конструктора

ПРИМЕР: МНОЖЕСТВА

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

public final class FirstSetExample  {
    public static void main(final String[] args) {
        fillAndExploreHashSet();
        fillAndExploreTreeSet();
   }

   private static void fillAndExploreHashSet() {
      final Set<String> hashSet = new  HashSet<String>();
      addStringDemoData(hashSet);
      System.out.println(hashSet);

      final Set<StringBuilder> hashSetSurprise = new HashSet<StringBuilder>();
      addStringBuilderDemoData(hashSetSurprise);
      System.out.println(hashSetSurprise);
   }
```

StringBuilder: променлива последователност от символи

ПРИМЕР: МНОЖЕСТВА

```
private static void fillAndExploreTreeSet() {
    final Set<String> treeSet = new TreeSet<String>();
    addStringDemoData(treeSet);
    System.out.println(treeSet);

    final Set<StringBuilder> treeSetSurprise = new TreeSet<StringBuilder>();
    addStringBuilderDemoData(treeSetSurprise);
    System.out.println(treeSetSurprise);
}

private static void addStringDemoData(final Set<String>set) {
    set.add("Hello");
    set.add("World");
    set.add("World");
}

private static void addStringBuilderDemoData(final Set<StringBuilder> set) {
    set.add(new StringBuilder("Hello"));
    set.add(new StringBuilder("World"));
    set.add(new StringBuilder("World"));
 }
}
```

ПРИМЕР: МНОЖЕСТВА

1 Какъв резултат?

```
private static void fillAndExploreTree
    final Set<String> treeSet = new Tree
    addStringDemoData(treeSet);
    System.out.println(treeSet);

    final Set<StringBuilder> treeSetSurp
    addStringBuilderDemoData(treeSetSurprise);
    System.out.println(treeSetSurprise);
}

private static void addStringDemoData(final Set<String>set) {
    set.add("Hello");
```

Осигуряване на еднозначност на елементите в множествата:

- За символни низове работи както се очаква
- За StringBuilder дубликатите не се различават
- За TreeSet<StringBuilder> предизвиква изключение

```
[Hello, World]
[World, Hello, World]
[Hello, World]
Exception in thread "main" java.lang.ClassCastException: java.lang.StringBuilder cannot be cast to java.lang.Comparable
at java.util.TreeMap.compare(Unknown Source)
at java.util.TreeMap.put(Unknown Source)
at java.util.TreeSet.add(Unknown Source)
at collections.FirstSetExample.addStringBuilderDemoData(FirstSetExample.java:47)
at collections.FirstSetExample.fillAndExploreTreeSet(FirstSetExample.java:35)
at collections.FirstSetExample.main(FirstSetExample.java:16)
```

Distributed @learning center

ПРИМЕР 1: HASHSET

```
import java.util.Arrays;
import java.util.HashSet;
import java.util.Set;

public final class HashSetIterationExample  {

    public static void main(String[] args)    {
        final Integer[] ints = new Integer[] { 3, 2, 1 };
        final Set<Integer> numberSet = new HashSet<Integer>(Arrays.asList(ints));

        System.out.println("Initial: " + numberSet); // 1, 2, 3
    }

    private HashSetIterationExample()
    {
    }
}
```

ПРИМЕР 1: HASHSET

Стойностите 1-3 в намаляващ ред в една HashSet

- При извеждането (1,2,3) изглежда, че една HashSet създава естествена наредба на въведените стойности
- Това е случаен ефект
- При HashSet се използват неподредени множества
- Вторият пример показва това

```
import java.util.Arrays;
import java.util.HashSet;
import java.util.Set;

public final class HashSetIterationExa

    public static void main(String[] args)  {
        final Integer[] ints = new Integer[] { 3, 2, 1 };
        final Set<Integer> numberSet = new HashSet<Integer>(Arrays.asList(ints));

        System.out.println("Initial: " + numberSet); // 1, 2, 3
    }

    private HashSetIterationExample()
    {
    }
}
```

```
Initial: [1, 2, 3]
```

ПРИМЕР 2: HASHSET

```
import java.util.Arrays;
import java.util.HashSet;
import java.util.Set;
public class HashSetIterationExample2  {
   public static void main(String[] args)     {
      final Integer[] ints = new Integer[] { 3, 2, 1 };
      final Set<Integer> numberSet = new HashSet<Integer>(Arrays.asList(ints));

      System.out.println("Initial: " + numberSet); // 1, 2, 3

      final Integer[] moreInts = new Integer[] { 11, 22, 33 };
      numberSet.addAll(Arrays.asList(moreInts));
      System.out.println("Add: " + numberSet); // 1, 2, 33, 3, 22, 11
   }

   private HashSetIterationExample2()    {
   }
}
```

Initial: [1, 2, 3]
Add: [1, 33, 2, 3, 22, 11]

ПРИМЕР:TREESET

```
import java.util.Arrays;
import java.util.TreeSet;
import java.util.Set;

public class TreeSetStorageExample   {

    public static void main(String[] args) {
        final Integer[] ints = new Integer[] { 3, 2, 1 };
        final Set<Integer> numberSet = new TreeSet<>(Arrays.asList(ints));
        System.out.println("Initial: " + numberSet); // 1, 2, 3

        final Integer[] moreInts = new Integer[] { 33, 11, 22 };
        numberSet.addAll(Arrays.asList(moreInts));
        System.out.println("Add: " + numberSet); // 1, 2, 3, 11, 22, 33
    }
}
```

ПРИМЕР:TREESET

Класът TreeSet: имплементира интерфейса SortedSet<E> - съхранява елементите сортирано

- Сортировката се определя от интерфейса Comparable<T> или явно предаден в конструктора Comparator<T>
- Понеже в конструктора не е предаден Comparator<T>, сортировката се извършва на основата на Comparable<T>, изпълнен в класа Integer

```
import java.util.Arrays;
import java.util.TreeSet;
import java.util.Set;

public class TreeSetStorageExample

    public static void main(String[] ar
        final Integer[] ints = new Integer[] { 3, 2, 1 };
        final Set<Integer> numberSet = new TreeSet<>(Arrays.asList(ints));
        System.out.println("Initial: " + numberSet); // 1, 2, 3

        final Integer[] moreInts = new Integer[] { 33, 11, 22 };
        numberSet.addAll(Arrays.asList(moreInts));
        System.out.println("Add: " + numberSet); // 1, 2, 3, 11, 22, 33
    }
}
```

Initial: [1, 2, 3]
Add: [1, 2, 3, 11, 22, 33]

ПРИМЕР:TREESET

```
import java.util.Arrays;
import java.util.SortedSet;
import java.util.TreeSet;
import java.util.Set;
public class TreeSetStorageExample2   {
    public static void main(String[] args)   {
      final Integer[] ints = new Integer[] { 3, 2, 1, 33, 11, 22 };
      final SortedSet<Integer> numberSet = new TreeSet<>(Arrays.asList(ints));
      System.out.println("Initial: " + numberSet); // 1, 2, 3, 11, 22, 33
      System.out.println("first: " + numberSet.first()); // 1
      System.out.println("last: " + numberSet.last());   // 33

      final SortedSet<Integer> headSet = numberSet.headSet(7);
      System.out.println("headSet: " + headSet); // 1, 2, 3
      System.out.println("tailSet: " + numberSet.tailSet(7)); // 11, 22, 33
      System.out.println("subSet: " + numberSet.subSet(7, 23)); // 11, 22

      headSet.remove(3);
      headSet.add(6);
      System.out.println("headSet: " + headSet); // 1, 2, 6
      System.out.println("numberSet: " + numberSet); // 1, 2, 6, 11, 22, 33
    }
}
```

ПРИМЕР:TREESET

```
import java.util.Arrays;
import java.util.SortedSet;
import java.util.TreeSet;
import java.util.Set;
public class TreeSetStorageExample2   {
    public static void main(String[] args)
        final Integer[] ints = new Integer[] { 3
        final SortedSet<Integer> numberSet
        System.out.println("Initial: " + numbe
        System.out.println("first: " + numberS
        System.out.println("last: " + numberS

        final SortedSet<Integer> headSet = numberSet.headSet(7);
        System.out.println("headSet: " + headSet); // 1, 2, 3
        System.out.println("tailSet: " + numberSet.tailSet(7)); // 11, 22, 33
        System.out.println("subSet: " + numberSet.subSet(7, 23)); // 11, 22

        headSet.remove(3);
        headSet.add(6);
        System.out.println("headSet: " + headSet); // 1, 2, 6
        System.out.println("numberSet: " + numberSet); // 1, 2, 6, 11, 22, 33
    }
}
```

Методи на класа TreeSet<E>:
first(), last(): доставя първия, съотв. последния елемент
headset: доставя подмножество на елементите, които са по-малки на дадения параметър
tailSet: доставя подмножество на елементите, които са по-големи или равни на дадения параметър
subset: доставя подмножество на елементите, започвайки (включен) първия параметър до втория параметър (изключен)

```
Initial: [1, 2, 3, 11, 22, 33]
first: 1
last: 33
headSet: [1, 2, 3]
tailSet: [11, 22, 33]
subSet: [11, 22]
headSet: [1, 2, 6]
numberSet: [1, 2, 6, 11, 22, 33]
```

ХЕШ-БАЗИРАНИ КОНТЕЙНЕРИ

- Масиви и списъци – в определени ситуации неприятен недостатък може да бъде затруднено търсене на съхраняваните елементи
- Хеш-базираните контейнери могат да се използват за ефективно търсене – при тях:
  - Времето за добавяне, изтриване и достъп по принцип е независимо от броя на съхраняваните елементи
  - Изискват допълнителни разходи
  - По-трудни за разбиране

ОРГАНИЗАЦИЯ

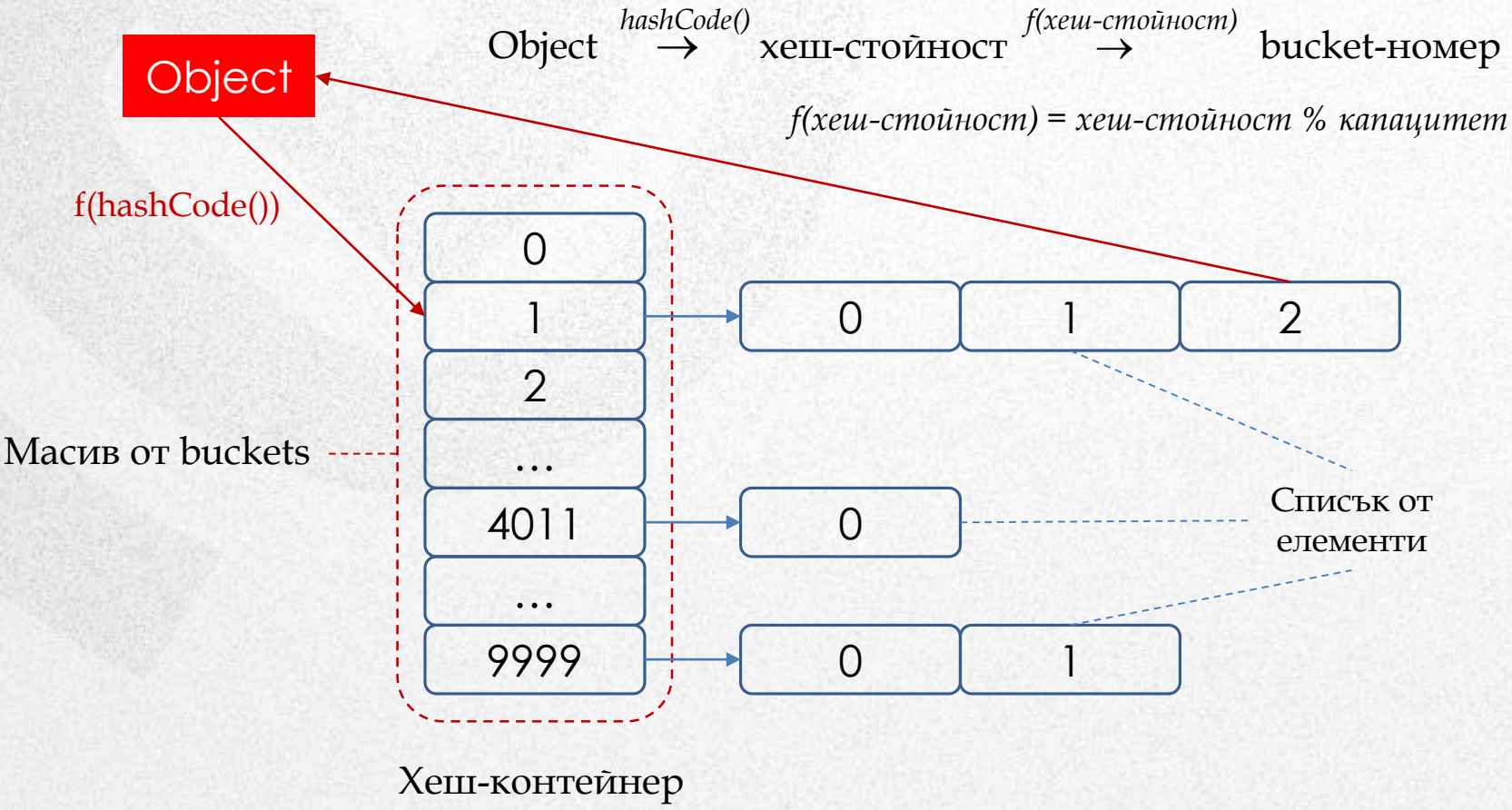
hashCode():

- Изчислява номера на bucket, в който трябва да се съхранява обектът
- По възможност различни стойности за различни обекти

Реализация в Java:

- Броят на buckets е ограничен – изчислената чрез hashCode() int стойност трябва да бъде съпоставена на съществуващите buckets
- Buckets се съхраняват в едноразмерен масив, наречен хеш-таблица
- Броят на наличните buckets се нарича капацитет
- Всеки bucket може да съхранява повече елементи като списък

ОПРЕДЕЛЯНЕ НА BUCKET НОМЕР



ТЪРСЕНЕ

Достъп до елементите в един хеш-контейнер:

- Определяне на bucket – използва се hashCode() и вътрешната функция на контейнера f
- Търси се в списъка на определения bucket с помощта на equals(Object)

- В класа Object се съдържат имплементации по подразбиране на методите hashCode() и equals(Object)
- Те обаче са достатъчни за малко приложения
- По тази причина при използваните приложни класове двата метода трябва да бъдат препокрити целесъобразно

ПРИМЕР: ТЪРСЕНЕ В HASHSET

```
import java.util.Collection;
import java.util.HashSet;
public final class PlayingCardInHashSet  {
   public enum Color { DIAMOND, HEART, SPADE, CLUB  };
   public static final class PlayingCard   {
      private final Color color;
      private final int   value;
      public PlayingCard(final Color color, final int value)  {
         this.color = color;
         this.value = value;
      }

      @Override
      public boolean equals(Object other)   {
         if (other == null) //акцептиране null
            return false;
         if (this == other) // рефлексивност
            return true;
         if (this.getClass() != other.getClass()) // тип-еквивалентност
            return false;

        final PlayingCard card = (PlayingCard) other;
         return this.value == card.value && this.color.equals(card.color);
      }
   }
```

ПРИМЕР: ТЪРСЕНЕ В HASHSET

1 Какъв резултат?

```
    public static void main(final String[] args)    {
        final Collection<PlayingCard> playingcards = new HashSet<PlayingCard>();

        playingcards.add(new PlayingCard(Color.HEART, 7));

        playingcards.add(new PlayingCard(Color.SPADE, 8));
        playingcards.add(new PlayingCard(Color.DIAMOND, 9));

       final boolean found = playingcards.contains(new PlayingCard(Color.SPADE, 8));
        System.out.println("found = " + found);
    }
}
```

found = false

ПРОБЛЕМ

- При достъп към контейнер първо се изчислява bucket посредством hashCode(), в който след това се търсят обектите с equals(Object)
- За класа PlayngCard методът HashCode() не е пренаписан
- Така се използва имплементацията по подразбиране от класа Object, която връща като hashCode() адреса за съхраняване на обекта
- За търсене обаче се използва новосъздаден обект, който представя същата карта, но има различна референция
- Така, за двата обекта се изчисляват различни хеш-стойности и се търси в различни bucket
- Решение: трябва да пренапишем hashCode() в класа PlayingCard

ПРИМЕР: ТЪРСЕНЕ В HASHSET

```
import java.util.Collection;
import java.util.HashSet;
public final class PlayingCardWithEqualsAndHashCode {
   public enum Color { DIAMOND, HEART, SPADE, CLUB  };
   public static final class PlayingCard   {
      private final Color color;
      private final int   value;
      public PlayingCard(final Color color, final int value) {
         this.color = color;
         this.value = value;
      }

      @Override
      public boolean equals(Object other) {
         if (other == null)
            return false;
         if (this == other)
            return true;
         if (this.getClass() != other.getClass())
            return false;

         final PlayingCard card = (PlayingCard) other;
         return this.value == card.value && this.color.equals(card.color);
      }
```

ПРИМЕР: ТЪРСЕНЕ В HASHSET

1 Какъв резултат?

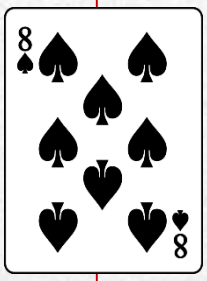
```
@Override
      public int hashCode()
      {
         int hash = 17;
         hash = HashUtils.calcHashCode(hash, color);
         hash = HashUtils.calcHashCode(hash, value);
         return hash;
      }
   }

   public static void main(final String[] args)
   {
      final Collection<PlayingCard> playingcards = new HashSet<PlayingCard>();

      playingcards.add(new PlayingCard(Color.HEART, 7));

      playingcards.add(new PlayingCard(Color.SPADE, 8));
      playingcards.add(new PlayingCard(Color.DIAMOND, 9));

      final boolean found = playingcards.contains(new PlayingCard(Color.SPADE, 8));
      System.out.println("found = " + found);
   }
}
```



found = true

МЕТОД HASHCODE( )

Методът съпоставя на състоянието на един обект едно число

- Необходимо е за да могат обектите да се обработват в хеш-базирани контейнери
- Сигнатура: public int hashCode()

Имплементацията трябва да удовлетворява следните условия:

- Еднозначност – при изпълнение на програмата всяко извикване на метода за един обект по възможност да доставя една и съща стойност
- Съвместимост с equals( ) – когато методът equals(Object) връща стойност true за два обекта, тогава hashCode( ) трябва да доставя една и съща стойност за двата обекта
- Обратното не важи – при еднаква хеш-стойност двата обекта могат да бъдат различни чрез equals( )

МЕТОД HASHCODE( )

Някои насоки за реализиране на метода:

- За изчисляване на хеш-стойност трябва да се използват (по възможност непроменяеми) атрибути, които се използват също при equals( ) за установяване на еквивалентност
- Така автоматично се осигурява съвместимостта
- Обикновено като мултипликатор се използва просто число

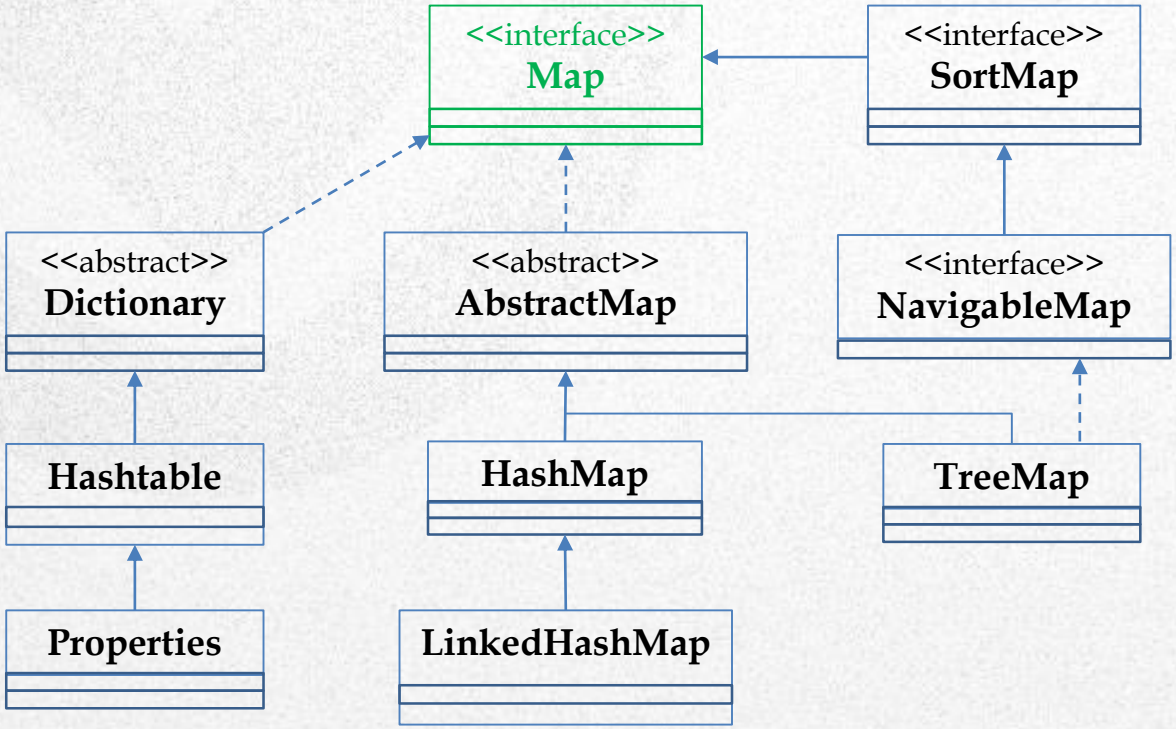
Типични грешки:

- Пренаписан метод equals( ), а hashCode( ) непренаписан
- Използване на променливи атрибути – в определени случаи може да не функционира коректно. В такива случаи преглед на реализация и евентуална смяна на атрибутите, използване за изчисляване на хеш-стойността

КАРТИ

- Ще разгледаме реализации на интерфейса Map<K,V>
- Идея:
  - На всяка съхранявана стойност се съпоставя еднозначен ключ
- Класически пример: телефонен указател
  - Търсене на телефонен номер (стойност) по име (ключ) обикновено е сравнително бързо
  - Намирането на името по телефонен номер е изключително трудно
    - Понеже не съществува обратно съпоставяне – номер на име

КЛАСИФИКАЦИЯ INTERFACE "MAP"



МЕТОДИ НА ИНТЕРФЕЙСА

Метод	Функция
V put(K key, V value)	Добавя нов запис. Ако за ключа има съхранявана стойност, тя се препокрива
void putAll(Map<? Exends K, ? Extends V> map)	Добавя всички записи от дадената като параметър карта
V remove(Object key)	Изтрива запис
boolean containtsKey(Object key)	Проверява за наличието на ключ
V get(Object key)	Доставя асоциирана към ключ стойност
void clear( )	Изтрива всички записи
int size( )	Брой на записите
boolean isEmpty( )	Проверява дали картата е празна
Set<K> keySet( )	Доставя множество с всички ключове
Collection<V> values( )	Доставя стойностите като колекция
Set<Map.Entry<K,V>> entrySet( )	Доставя множеството на всички записи

ПРИМЕР

```
import java.util.List;
import java.util.*;
public final class FirstMapExample  {
   public static void main(final String[] args) {
        final Map<String, Integer> nameToNumber = new TreeMap<>();
        nameToNumber.put("Мария", 4711);
        nameToNumber.put("Стоян", 0714);
        nameToNumber.put("Йордан", 1234);
        nameToNumber.put("Стоян", 1508);
        nameToNumber.put("Росен", 2208);

         System.out.println(nameToNumber);
         System.out.println(nameToNumber.containsKey("Стоян"));
         System.out.println(nameToNumber.get("Йордан"));
         System.out.println(nameToNumber.size());
         System.out.println(nameToNumber.containsKey("Стоян"));
         System.out.println(nameToNumber.keySet());
         System.out.println(nameToNumber.values());
      }
}
```

ПРИМЕР

1 Какъв резултат?

```
import java.util.List;
import java.util.*;
public final class FirstMapExample  {
   public static void main(final String[] args) {
        final Map<String, Integer> nameToNumber = new TreeMap<>();
        nameToNumber.put("Мария", 4711);
        nameToNumber.put("Стоян", 0714);
        nameToNumber.put("Йордан", 1234);
        nameToNumber.put("Стоян", 1508);
        nameToNumber.put("Росен", 2208);

        System.out.println(nameToNumber);
        System.out.println(nameToNumber.containsKey("Стоян"));
        System.out.println(nameToNumber.get("Йордан"));
        System.out.println(nameToNumber.size());
        System.out.println(nameToNumber.containsKey("Стоян"));
        System.out.println(nameToNumber.keySet());
        System.out.println(nameToNumber.values());
    }
}
```

Демонстрация на различните методи на интерфейса

Когато се въвеждат многократно данни за един и същ ключ, се препокриват стойностите

2 Защо така?

```
{Йордан=1234, Мария=4711, Росен=2208, Стоян=1508}
true
1234
4
true
[Йордан, Мария, Росен, Стоян]
[1234, 4711, 2208, 1508]
```

КЛАС HASHMAP<K,V>

Реализация на абстрактния клас AbstractMap<K, V>, който имплементира интерфейса Map<K, V>

- Данните се съхраняват в хеш-таблица
- Последователността на елементите в една итерация изглежда случайна
- В действителност се определя от съответната хеш-стойност, както и от разпределението върху buckets

Пример: за демонстрация на класа HashMap<K, V> ще разгледаме често срещан в практиката случай – множество данни се преобразува в друго

ПРИМЕР

```
private static Color mapToColor(final String colorName) {
   switch (colorName) {
      case "BLACK": return Color.BLACK;
      case "RED": return Color.RED;
      case "GREEN": return Color.GREEN;
      default:
        throw new IllegalArgumentException("No color for: '" + colorName + "'");
   }
}
```

За малко случаи тази структура е приемлива

ПРИМЕР

```
import java.awt.Color;
import java.util.*;
public final class HashMapExample  {
     private static final Map<String, Color> nameToColor = new HashMap<>();
     static {
        initMapping(nameToColor);
     }
public static void main(final String[] args) {
     System.out.println(mapToColor("RED"));
     System.out.println(mapToColor("GREEN"));
     System.out.println(mapToColor("UNKNOWN"));
}
private static Color mapToColor(final String colorName) {
    if (nameToColor.containsKey(colorName)) {
       return nameToColor.get(colorName);
    }
    throw new IllegalArgumentException("No color for: '" + colorName + "'");
}
private static void initMapping(final Map<String, Color> nameToColor) {
    nameToColor.put("BLACK", Color.BLACK);
    nameToColor.put("RED", Color.RED);
    nameToColor.put("GREEN", Color.GREEN);
  }
```

КЛАС TREEMAP<K,V>

Реализация на абстрактния клас AbstractMap<K, V>, който имплементира интерфейсите SortedMap<K, V> и NavigableMap<K, V>

- Поставя автоматично наредба на съхраняваните ключове, като използва за това Comparable<T> или Comparator<T>

ПРИМЕР: MAP

1 Какъв резултат?

```
import java.util.NavigableMap;
import java.util.TreeMap;
public final class TreeMapExample   {
   public static void main(final String[] args)    {

      final NavigableMap<String, Integer> nameToAgeMap = new TreeMap<>();

      nameToAgeMap.put("Max", 47);
      nameToAgeMap.put("Moritz", 39);
      nameToAgeMap.put("Micha", 43);

      System.out.println("floor   Ma: " + nameToAgeMap.floorKey("Ma"));
      System.out.println("higher  Ma: " + nameToAgeMap.higherKey("Ma"));
      System.out.println("lower   Mz: " + nameToAgeMap.lowerKey("Mz"));
      System.out.println("ceiling Mc: " + nameToAgeMap.ceilingEntry("Mc"));
   }

}
```

ПРИМЕР: MAP

1 Какъв резултат?

```
import java.util.NavigableMap;
import java.util.TreeMap;
public final class TreeMapExample   {
   public static void main(final String[] arg

      final NavigableMap<String, Integer>

      nameToAgeMap.put("Max", 47);
      nameToAgeMap.put("Moritz", 39);
      nameToAgeMap.put("Micha", 43);

      System.out.println("floor   Ma: " + nam
      System.out.println("higher  Ma: " + nameToAgeMap.higherKey("Ma"));
      System.out.println("lower   Mz: " + nameToAgeMap.lowerKey("Mz"));
      System.out.println("ceiling Mc: " + nameToAgeMap.ceilingEntry("Mc"));
   }

}
```

Методи на класа TreeSet<E>:
floorKey(): доставя най-големия ключ, който е по-малък и равен на параметъра
lowerKey(): доставя най-големия ключ, който е по-малък от параметъра
higherKey(): доставя най-малкия ключ, който е по-голям от параметъра
ceilingKey(): доставя най-малкия ключ, който е по-голям или равен на параметъра
ceilingEntry(): като ceilingKey(), но връща запис в карта

```
floor   Ma: null
higher  Ma: Max
lower   Mz: Moritz
ceiling Mc: Micha=43
```

БЛАГОДАРЯ ЗА ВНИМАНИЕТО!

КРАЙ “КОЛЕКЦИИ”

Distributed
@learning center