

КОЛЕКЦИИ

ЛЕКЦИОНЕН КУРС “ООП(JAVA)”



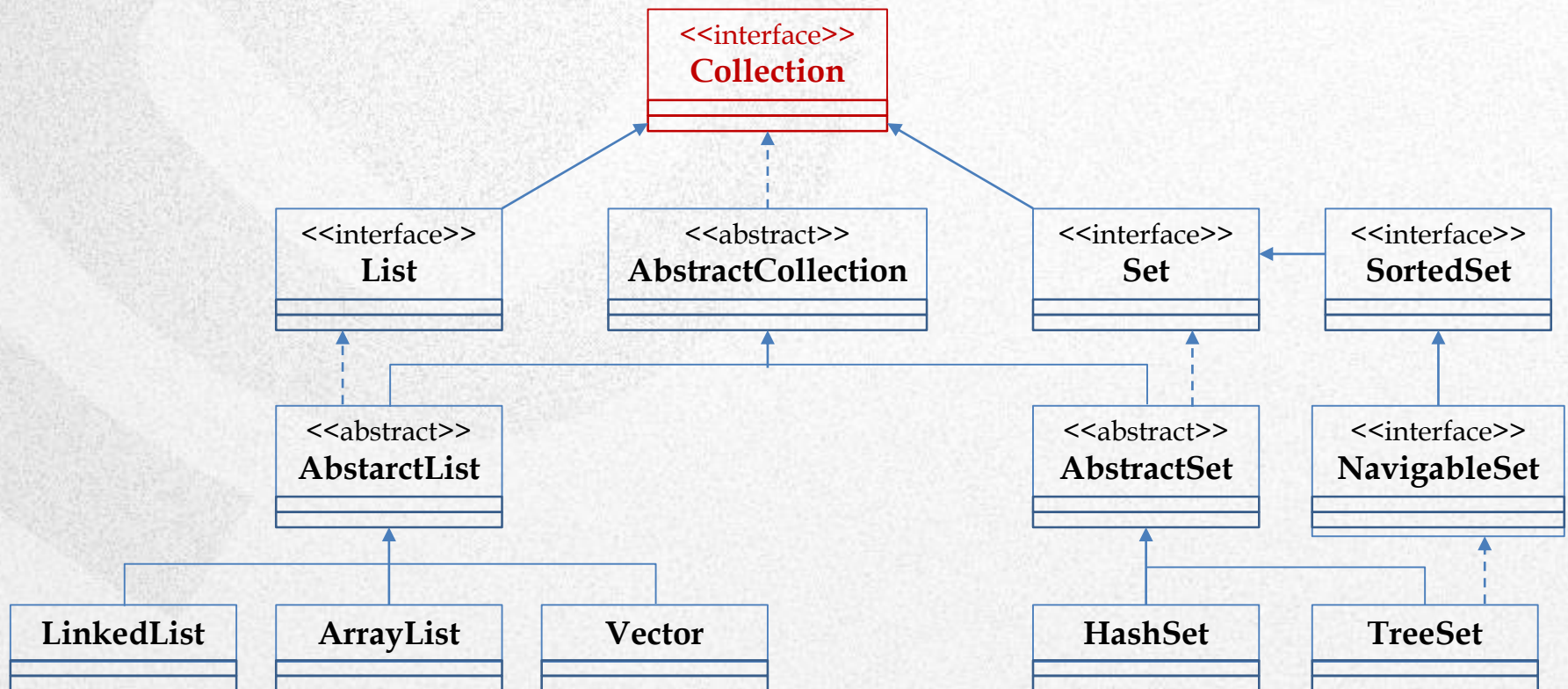
КОНТЕЙНЕРНИ КЛАСОВЕ

- 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>`

КЛАСИФИКАЦИЯ

- Две йерархии
 - Интерфейс $\text{Collection}\langle E \rangle$
 - Интерфейс $\text{Map}\langle K, V \rangle$
- 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



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

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

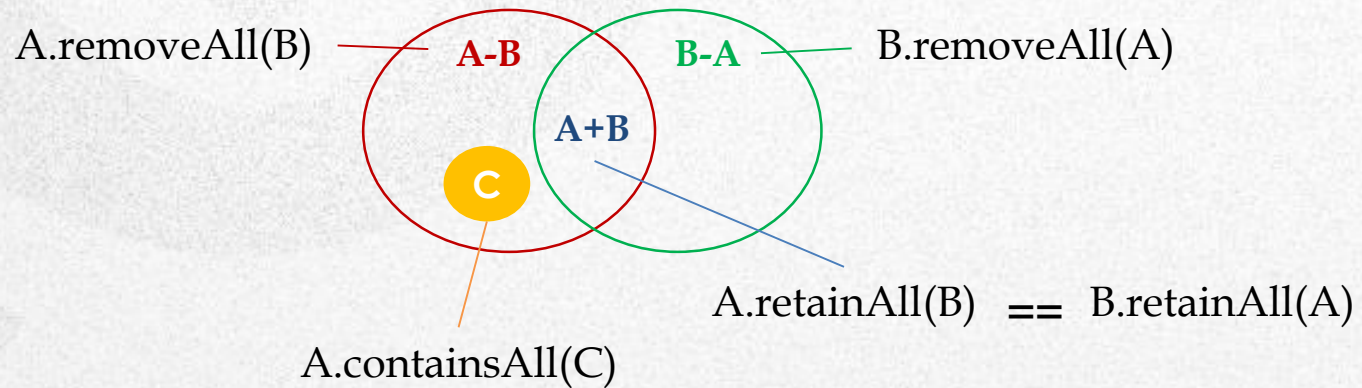
МЕТОДИ НА ИНТЕРФЕЙСА

<code>boolean</code>	<code>add(E e)</code> Ensures that this collection contains the specified element (optional operation).
<code>boolean</code>	<code>addAll(Collection<? extends E> c)</code> Adds all of the elements in the specified collection to this collection (optional operation).
<code>void</code>	<code>clear()</code> Removes all of the elements from this collection (optional operation).
<code>boolean</code>	<code>contains(Object o)</code> Returns <code>true</code> if this collection contains the specified element.
<code>boolean</code>	<code>containsAll(Collection<?> c)</code> Returns <code>true</code> if this collection contains all of the elements in the specified collection.
<code>boolean</code>	<code>equals(Object o)</code> Compares the specified object with this collection for equality.
<code>int</code>	<code>hashCode()</code> Returns the hash code value for this collection.
<code>boolean</code>	<code>isEmpty()</code> Returns <code>true</code> if this collection contains no elements.
<code>Iterator<E></code>	<code>iterator()</code> Returns an iterator over the elements in this collection.

МЕТОДИ НА ИНТЕРФЕЙСА

<code>Iterator<E></code>	<code>iterator()</code> Returns an iterator over the elements in this collection.
<code>default Stream<E></code>	<code>parallelStream()</code> Returns a possibly parallel Stream with this collection as its source.
<code>boolean</code>	<code>remove(Object o)</code> Removes a single instance of the specified element from this collection, if it is present (optional operation).
<code>boolean</code>	<code>removeAll(Collection<?> c)</code> Removes all of this collection's elements that are also contained in the specified collection (optional operation).
<code>default boolean</code>	<code>removeIf(Predicate<? super E> filter)</code> Removes all of the elements of this collection that satisfy the given predicate.
<code>boolean</code>	<code>retainAll(Collection<?> c)</code> Retains only the elements in this collection that are contained in the specified collection (optional operation).
<code>int</code>	<code>size()</code> Returns the number of elements in this collection.
<code>default Spliterator<E></code>	<code>spliterator()</code> Creates a Spliterator over the elements in this collection.
<code>default Stream<E></code>	<code>stream()</code> Returns a sequential Stream with this collection as its source.
<code>Object[]</code>	<code>toArray()</code> Returns an array containing all of the elements in this collection.
<code><T> T[]</code>	<code>toArray(T[] a)</code> Returns an array containing all of the elements in this collection; the runtime type of the returned array is that of the specified array.

ОПЕРАЦИИ С МНОЖЕСТВА



ИТЕРАТОР

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

МЕТОДИ НА ИТЕРАТОР

default void	<code>forEachRemaining(Consumer<? super E> action)</code> Performs the given action for each remaining element until all elements have been processed or the action throws an exception.
boolean	<code>hasNext()</code> Returns <code>true</code> if the iteration has more elements.
E	<code>next()</code> Returns the next element in the iteration.
default void	<code>remove()</code> Removes from the underlying collection the last element returned by this iterator (optional operation).

МЕТОД REMOVE()

- Съществува също метод `remove()`
 - Премахва елемент, доставен преди това с метода `next()`
- Този метод може да се извика само веднъж на извикване `next()`
- Поведението на итератора е неопределено, ако основната колекция се променя, докато итерацията е в ход по друг начин, различен от този, който извиква този метод
- Изглежда излишно дублирането на този метод
- Съществува метод `remove(Object)` в интерфейса `Collection`
- Защо се налага дублиране на метода?
 - Ще демонстрираме с пример

ПРИМЕР - ИТЕРАТОР

Траверсиране на колекции с интерфейса Iterator

```
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", "итератор" };

        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(", ");
        }
    }
}
```

ПРИМЕР - ИТЕРАТОР

Траверсиране на колекции с интерфейса Iterator

```
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", "итератор" };

        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

ПРИМЕР - ИТЕРАТОР

Траверсиране на колекции с интерфейса Iterator

```
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", "итератор" };

        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(", ");
        }
    }
}
```

?

Резултат

Траверс, с, итератор

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);
            }
        }
    }
    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);
    }
}
```

Проверява дали низът започва със
специфицирания префикс

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);
            }
        }
    }
}
```

? **Резултат**

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)

```
namesList.addAll(Arrays.asList(names));

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

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 изчислява стойността на брояча 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
            }
        }
    }
    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);
    }
}
```


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
            }
        }
    }
}

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);
}
```

?

Резултат

[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

boolean	add(E e) Appends the specified element to the end of this list (optional operation).
void	add(int index, E element) Inserts the specified element at the specified position in this list (optional operation).
boolean	addAll(Collection<? extends E> c) Appends all of the elements in the specified collection to the end of this list, in the order that they are returned by the specified collection's iterator (optional operation).
boolean	addAll(int index, Collection<? extends E> c) Inserts all of the elements in the specified collection into this list at the specified position (optional operation).
void	clear() Removes all of the elements from this list (optional operation).
boolean	contains(Object o) Returns true if this list contains the specified element.
boolean	containsAll(Collection<?> c) Returns true if this list contains all of the elements of the specified collection.
boolean	equals(Object o) Compares the specified object with this list for equality.
E	get(int index) Returns the element at the specified position in this list.
int	hashCode() Returns the hash code value for this list.

МЕТОДИ НА ИНТЕРФЕЙСА

<code>int</code>	<code>indexOf(Object o)</code> Returns the index of the first occurrence of the specified element in this list, or -1 if this list does not contain the element.
<code>boolean</code>	<code>isEmpty()</code> Returns <code>true</code> if this list contains no elements.
<code>Iterator<E></code>	<code>iterator()</code> Returns an iterator over the elements in this list in proper sequence.
<code>int</code>	<code>lastIndexOf(Object o)</code> Returns the index of the last occurrence of the specified element in this list, or -1 if this list does not contain the element.
<code>ListIterator<E></code>	<code>listIterator()</code> Returns a list iterator over the elements in this list (in proper sequence).
<code>ListIterator<E></code>	<code>listIterator(int index)</code> Returns a list iterator over the elements in this list (in proper sequence), starting at the specified position in the list.
<code>E</code>	<code>remove(int index)</code> Removes the element at the specified position in this list (optional operation).
<code>boolean</code>	<code>remove(Object o)</code> Removes the first occurrence of the specified element from this list, if it is present (optional operation).
<code>boolean</code>	<code>removeAll(Collection<?> c)</code> Removes from this list all of its elements that are contained in the specified collection (optional operation).
default void	<code>replaceAll(UnaryOperator<E> operator)</code> Replaces each element of this list with the result of applying the operator to that element.
<code>boolean</code>	<code>retainAll(Collection<?> c)</code> Retains only the elements in this list that are contained in the specified collection (optional operation).

МЕТОДИ НА ИНТЕРФЕЙСА

<code>E</code>	<code>set(int index, E element)</code> Replaces the element at the specified position in this list with the specified element (optional operation).
<code>int</code>	<code>size()</code> Returns the number of elements in this list.
default void	<code>sort(Comparator<? super E> c)</code> Sorts this list according to the order induced by the specified Comparator .
default <code>Splitterator<E></code>	<code>spliterator()</code> Creates a Spliterator over the elements in this list.
<code>List<E></code>	<code>subList(int fromIndex, int toIndex)</code> Returns a view of the portion of this list between the specified <code>fromIndex</code> , inclusive, and <code>toIndex</code> , exclusive.
<code>Object[]</code>	<code>toArray()</code> Returns an array containing all of the elements in this list in proper sequence (from first to last element).
<code><T> T[]</code>	<code>toArray(T[] a)</code> Returns an array containing all of the elements in this list in proper sequence (from first to last element); the runtime type of the returned array is that of the specified array.

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);
```

Въвеждане елементи (в края на списъка)

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

Достъп посредством индекс

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

Изтриване на елементи

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

```
    }  
}
```

ПРИМЕР - СПИСЪК

? Резултат

```
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);
    }
}
```

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

ПРИМЕР - ПОДСПИСЪК

```
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();
        }
    }
}
```

ПРИМЕР - ПОДСПИСЪК

Комбинация от конструктор и помощен метод

- `asList()`: Връща немодифицируем списък с фиксирани размери, инициализиран от посочения масив

```
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" ));
```

Чрез конструктора е
трансформиран в променлив
СПИСЪК

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


ПРИМЕР - ПОДСПИСЪК

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

```
public final class SubListExample {
    public static void main(final String[] args) {
```

```
        final List<String> listToTruncate = new ArrayList<>();
        truncateListToMaxSize(listToTruncate, 5);
        System.out.println(listToTruncate);
    }
    • Връща част от списъка между първия (включен) и втория (изключен) параметър
    • Резултатният списък се поддържа от оригиналния - промени в резултатния списък се отразяват в оригиналния списък и обратно
```

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



ПРИМЕР - ПОДСПИСЪК

```
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
- clear() има също ефект върху оригиналния списък

ПРИМЕР - ПОДСПИСЪК

? Резултат

```
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();
        }
    }
}
```

[Error1, Error2, Error3]

ИНТЕРФЕЙС LISTITERATOR

- Всички структури данни, които изпълняват итератора `List<E>` предлагат достъп до специален итератор от тип `ListIterator<E>`
 - Адаптира се към особеностите на индексния достъп чрез метода `listIterator()`
- Възможно също траверс в обратна посока

МЕТОДИ НА ИНТЕРФЕЙСА

void	add(E e) Inserts the specified element into the list (optional operation).
boolean	hasNext() Returns true if this list iterator has more elements when traversing the list in the forward direction.
boolean	hasPrevious() Returns true if this list iterator has more elements when traversing the list in the reverse direction.
E	next() Returns the next element in the list and advances the cursor position.
int	nextIndex() Returns the index of the element that would be returned by a subsequent call to next() .
E	previous() Returns the previous element in the list and moves the cursor position backwards.
int	previousIndex() Returns the index of the element that would be returned by a subsequent call to previous() .
void	remove() Removes from the list the last element that was returned by next() or previous() (optional operation).
void	set(E e) Replaces the last element returned by next() or previous() with the specified element (optional operation).

КЛАСОВЕ ARRAYLIST И VECTOR

- Използват съхраняване на данни като масив, като допълнително го разширяват с контейнерни възможности
 - При нарастване броя на елементите, автоматично се осигурява достатъчно памет
 - Старите елементи се копират автоматично в новата по-голяма област
- Двата класа се различават само по един детайл
 - В класа `Vector<E>` методите са дефинирани като `synchronized`

КЛАС LINKEDLIST

- Използва за съхраняване на елементи, като свързани помежду си малки градивни блокове, наречени възли
 - Всеки възел съхранява референции освен към данните, също така към предшественика и наследника
 - Така е възможна навигация в двете посоки – напред и назад

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
- Два класа:
 - `HashSet<E>`
 - `TreeSet<E>`

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

?

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");
```

[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](#))

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

ПРИЧИНИ

- Различните колекции използват различни методи
 - `hashCode()`, `equals(Object)`, `compareTo(T)`, `compare(T,T)`
- За да се осигури еднозначност е необходима синхронизация между тези методи

HASHSET<E>

- HashSet<E> - съхранява елементите не подредено в хеш-контейнер
 - Чрез това – минимално време за операции като напр. add(E), remove(Object), contains(Object)

ПРИМЕР 1

```
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

```
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() {
    }
}
```

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

Initial: [1, 2, 3]

ПРИМЕР 2

```
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]

TREESet<E>

- 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 1

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

public class TreeSetStorageExample {
    public static void main(final 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
    }

    private TreeSetStorageExample() { }
}
```


ПРИМЕР 1

? Результат

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

public class TreeSetStorageExample {
    public static void main(final 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
    }

    private TreeSetStorageExample() { }
}
```

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

ПРИМЕР 2

```
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 2

```
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
    }
}
```

Методи на класа `TreeSet<E>`:

first(), **last()**: доставя първия, съответно 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)

ПРИМЕР 2

```
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
    }
}
```

?

Резултат

```
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
 - Времето за добавяне, изтриване и достъп по принцип е независимо от броя на съхраняваните елементи
 - Изискват допълнителни разходи
 - Изчисляват се специални хеш-стойности за постигане на ефективност
 - По-трудни за разбиране

АНАЛОГИЯ

- Шкаф с номерирани чекмеджета
- Във всяко чекмедже има място за достатъчен брой неща
- Ако трябва да се постави един обект в шкафа, първо му се определя номера на чекмеджето, който зависи от свойствата (атрибути) на обекта
- Когато се търси един обект се използва номера на чекмеджето
- За организацията:
 1. Ако се използва едно и също чекмедже – трудно за намиране на обектите
 2. Ако обектите са равномерно разпределени между чекмеджетата – намирането става с малки еднакви разходи (предполагаме познаване на номера на чекмеджето)
 3. Ако нямаме целево търсене – в екстремн случай се претърсват всички чекмеджета



ОРГАНИЗАЦИЯ

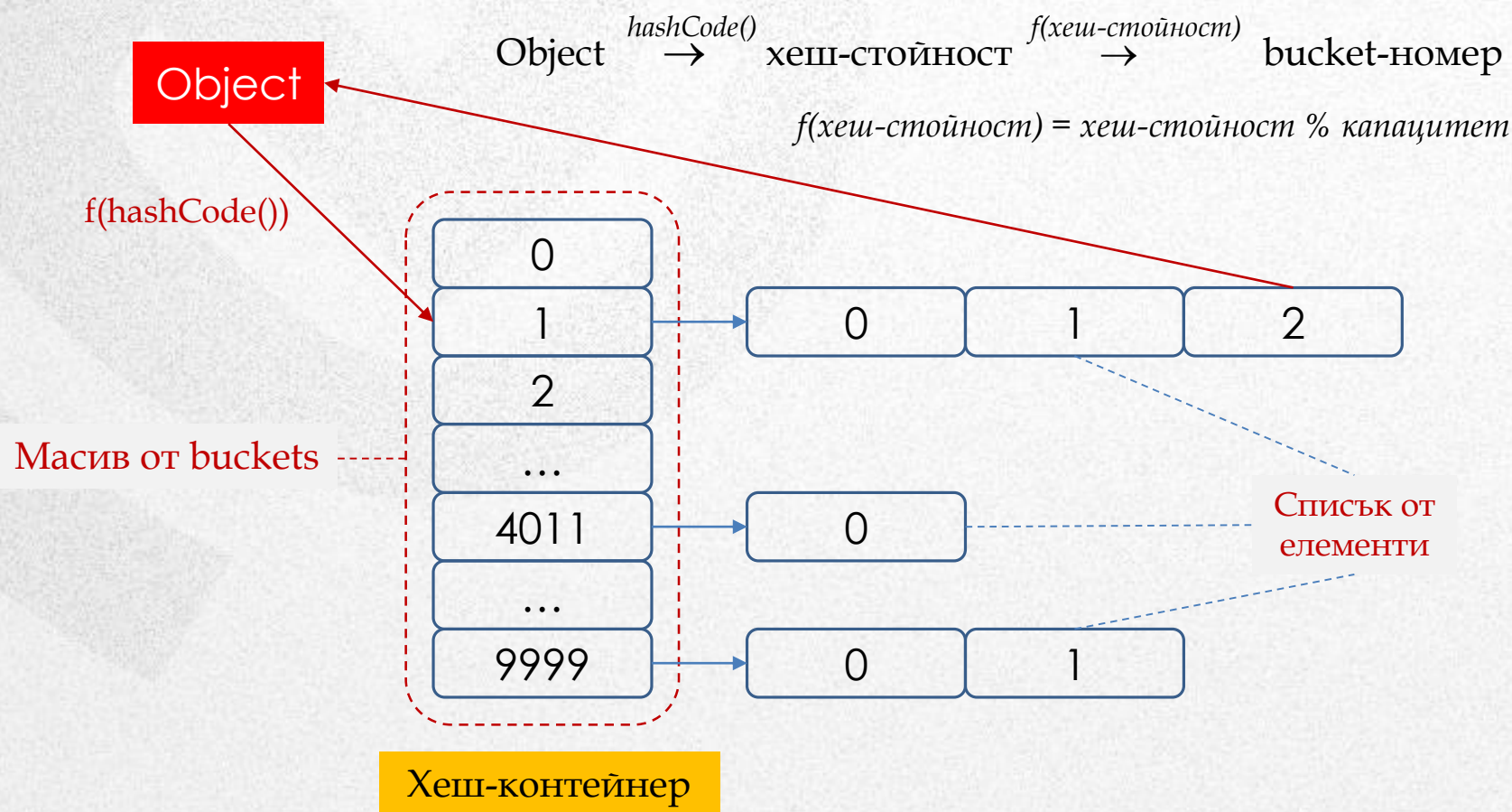
hashCode():

- 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 едномерен 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



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

? Резултат

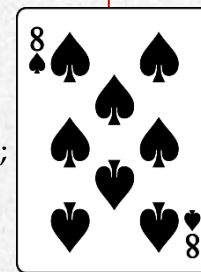
```
@Override
public int hashCode() {
    int hash = 17;
    hash = HashUtils.calcHashCode(hash, color);
    hash = HashUtils.calcHashCode(hash, value);
    return hash;
}

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

АВТОМАТИЧНО СОРТИРАЩИ КОНТЕЙНЕРИ

- За много приложения е удобно ако данните, поддържани в един контейнерен клас, са сортирани
- Съществуват контейнерни класове, които доставят автоматично сортиране в кода
 - `TreeSet<E>`, `TreeMap<K,V>`
- За масивите и списъците няма такава поддръжка
 - Необходимо е сортирането да се направи ръчно
 - Метод `sort()`

АВТОМАТИЧНО СОРТИРАЩИ КОНТЕЙНЕРИ

- Независимо от това, дали сортировката ще бъде автоматична или ръчна, винаги трябва да се определи наредба за изразяване на интерпретацията на сравнителни критерии
 - „по-малко“, „по-голямо“, „еднакво“
- Това може да се направи посредством имплементиране на интерфейсите `Comparable<T>` и `Comparator<T>`

ЕСТЕСТВЕНА НАРЕДБА

- Често стойностите или обектите притежават естествена наредба
 - Напр., числата и низовете
- За комплексните типове данни „по-малко“ съотв. „по-голямо“ не винаги са очевидни
 - Трябва да се дефинират
- За такива случаи интерфейсът `Comparable<T>` позволява типични сравнения
 - Декларира метода `compareTo()`

ЕСТЕСТВЕНА НАРЕДБА

```
public interface Comparable<T> {  
    public int compareTo(T o);  
}
```



= 0

Равенство между актуалния и предадения като параметър обект

< 0

Актуалният обект е по-малък от предадения като параметър обект

> 0

Актуалният обект е по-голям от предадения като параметър обект

Wrapper класовете имплементират интерфейса Comparable<T> и така са автоматично сортируеми

ПРИМЕР

```
public final class Person implements Comparable<Person> {  
    private final String name;  
    private final String city;  
    private final int age;
```

```
    public Person(final String name, final String city, final int age) {  
        this.name = Objects.requireNonNull(name, "name must not be null");  
        this.city = Objects.requireNonNull(city, "city must not be null");  
        this.age = age;  
    }
```

- Имплементиране на compareTo() в собствен клас
- Дефинира естествена наредба за Person върху името

@Override

```
    public int compareTo(final Person otherPerson) {  
        Objects.requireNonNull(otherPerson, "otherPerson must not be null");  
        return getName().compareTo(otherPerson.getName())  
    }
```

java.lang.Class.getName () връща името на обекта като низ

КЛАС OBJECTS

- Съдържа статични помощни методи за използване на обекти или проверка на определени условия преди една операция с тях
- Включват нулево-сигурни (null-safe) или нулево-толерантни (null-tolerant) методи за:
- Изчисляване хеш-кода на обекти
 - Връщане на низ за обект
 - Сравняване на два обекта
 - Проверка дали индексите или стойностите са извън граници

МЕТОДИ НА КЛАСА

static int	<code>checkFromIndexSize(int fromIndex, int size, int length)</code>	Checks if the sub-range from <code>fromIndex</code> (inclusive) to <code>fromIndex + size</code> (exclusive) is within the bounds of range from 0 (inclusive) to <code>length</code> (exclusive).
static int	<code>checkFromToIndex(int fromIndex, int toIndex, int length)</code>	Checks if the sub-range from <code>fromIndex</code> (inclusive) to <code>toIndex</code> (exclusive) is within the bounds of range from 0 (inclusive) to <code>length</code> (exclusive).
static int	<code>checkIndex(int index, int length)</code>	Checks if the <code>index</code> is within the bounds of the range from 0 (inclusive) to <code>length</code> (exclusive).
static <T> int	<code>compare(T a, T b, Comparator<? super T> c)</code>	Returns 0 if the arguments are identical and <code>c.compare(a, b)</code> otherwise.
static boolean	<code>deepEquals(Object a, Object b)</code>	Returns <code>true</code> if the arguments are deeply equal to each other and <code>false</code> otherwise.
static boolean	<code>equals(Object a, Object b)</code>	Returns <code>true</code> if the arguments are equal to each other and <code>false</code> otherwise.
static int	<code>hash(Object... values)</code>	Generates a hash code for a sequence of input values.
static int	<code>hashCode(Object o)</code>	Returns the hash code of a non-null argument and 0 for a null argument.
static boolean	<code>isNull(Object obj)</code>	Returns <code>true</code> if the provided reference is null otherwise returns <code>false</code> .
static boolean	<code>nonNull(Object obj)</code>	Returns <code>true</code> if the provided reference is non-null otherwise returns <code>false</code> .

МЕТОДИ НА КЛАСА



<code>static <T> T</code>	<code>requireNonNull(T obj)</code>	Checks that the specified object reference is not <code>null</code> .
<code>static <T> T</code>	<code>requireNonNull(T obj, String message)</code>	Checks that the specified object reference is not <code>null</code> and throws a customized <code>NullPointerException</code> if it is.
<code>static <T> T</code>	<code>requireNonNull(T obj, Supplier<String> messageSupplier)</code>	Checks that the specified object reference is not <code>null</code> and throws a customized <code>NullPointerException</code> if it is.
<code>static <T> T</code>	<code>requireNonNullElse(T obj, T defaultObj)</code>	Returns the first argument if it is non- <code>null</code> and otherwise returns the non- <code>null</code> second argument.
<code>static <T> T</code>	<code>requireNonNullElseGet(T obj, Supplier<? extends T> supplier)</code>	Returns the first argument if it is non- <code>null</code> and otherwise returns the non- <code>null</code> value of <code>supplier.get()</code> .
<code>static String</code>	<code>toString(Object o)</code>	Returns the result of calling <code>toString</code> for a non- <code>null</code> argument and " <code>null</code> " for a <code>null</code> argument.
<code>static String</code>	<code>toString(Object o, String nullDefault)</code>	Returns the result of calling <code>toString</code> on the first argument if the first argument is not <code>null</code> and returns the second argument otherwise.

ПРИМЕР



Резултат от сравнение

```
public static void main(String[] args) {  
    final List<Person> customers = new ArrayList<Person>();  
    ...  
    customers.add(new Person("Иван", "Пловдив", 27));  
    customers.add(new Person("Мария", "София", 37));  
    customers.add(new Person("Иван", "Бургас", 47));  
    ...  
}
```



Не представлява естествена наредба за Person

ПРИМЕР

?

Какво подобрение: equals()

```
@Override
public boolean equals(final Object obj)    {
    if (this == obj)
        return true;
    if (obj == null)
        return false;
    if (getClass() != obj.getClass())
        return false;
    final Person other = (Person) obj;
    return name.equals(other.name) && city.equals(other.city) && age == other.age;
}
```



В сравнението включваме също city и age

ИМПЛЕМЕНТИРАНЕ НА COMPARETO()

- Най-общо, при имплементиране на `compareTo()` можем да се ориентираме към съществуваща реализация на метода `equals(Object)` за даден клас
 - Всички сравнявани там атрибути са съществени за естествената наредба
- Тези атрибути се сравняват както следва:
 - Референтните типове, имплементиращи `Comparable<T>`, използват `compareTo(T)` метода
 - За примитивните типове се използват оператори за сравнения („<“, „=“, „>“)

ПРИМЕР

```
int result = 0;  
if (this.getAge() < otherPerson.getAge()) {  
    return -1;  
}  
  
if (this.getAge() < otherPerson.getAge()) {  
    return 1;  
}
```

Сравнение за примитивни типове

```
int result = Integer.compare(this.getAge(), otherPerson.getAge());
```

ПРИМЕР – ФИНАЛНА ВЕРСИЯ

```
import java.util.ArrayList;
import java.util.Collections;
import java.util.List;

public final class Person implements Comparable<Person> {
    private final String name;
    private final String city;
    private final int age;
    public Person(final String name, final String city, final int age) {
        this.name = name;
        this.city = city;
        this.age = age;
    }
    public final String getName() {
        return name;
    }
    public final String getCity() {
        return city;
    }
    public final int getAge() {
        return age;
    }
}
```

ПРИМЕР – ФИНАЛНА ВЕРСИЯ

```
public String toString() {  
    final StringBuffer buf = new StringBuffer();  
    buf.append("Person: ");  
    buf.append("Name=");  
    buf.append(getName());  
    buf.append(" ");  
    buf.append("City=");  
    buf.append(getCity());  
    buf.append(" ");  
    buf.append("Age=");  
    buf.append(getAge());  
    buf.append("");  
    return buf.toString();  
}  
  
@Override  
public int hashCode() {  
    return name.hashCode();  
}
```


ПРИМЕР – ФИНАЛНА ВЕРСИЯ

```
@Override
public boolean equals(final Object obj)    {
    if (this == obj)
        return true;
    if (obj == null)
        return false;
    if (getClass() != obj.getClass())
        return false;
    final Person other = (Person) obj;
    return name.equals(other.name) && city.equals(other.city) && age == other.age;
}
```

ПРИМЕР – ФИНАЛНА ВЕРСИЯ

```
@Override
public int compareTo(final Person otherPerson) {
    int ret = getName().compareTo(otherPerson.getName());
    if (ret == 0) {
        ret = getCity().compareTo(otherPerson.getCity());
    }
    if (ret == 0) {
        if (getAge() < otherPerson.getAge()) {
            ret = -1;
        }
        if (getAge() > otherPerson.getAge()) {
            ret = 1;
        }
    }
    return ret;
}
```

ПРИМЕР – ФИНАЛНА ВЕРСИЯ

```
public static void main(String[] args) {  
    final List<Person> customers = new ArrayList<Person>();
```

```
    customers.add(new Person("Иван", "Пловдив", 27));  
    customers.add(new Person("Мария", "София", 37));  
    customers.add(new Person("Георги", "Бургас", 47));
```

```
    Collections.sort(customers);  
    for (final Person currentPerson : customers)  
        System.out.println(currentPerson);  
}
```

`java.util.Collections.sort ()` се използва за
сортиране на елементите от посочената
колекция във възходящ ред

ПРИМЕР – ФИНАЛНА ВЕРСИЯ



Резултат

```
public static void main(String[] args) {  
    final List<Person> customers = new ArrayList<Person>();  
  
    customers.add(new Person("Иван", "Пловдив", 27));  
    customers.add(new Person("Мария", "София", 37));  
    customers.add(new Person("Иван", "Бургас", 47));  
  
    Collections.sort(customers);  
    for (final Person currentPerson : customers)  
        System.out.println(currentPerson);  
}
```

```
Person: Name='Иван' City='Бургас' Age='47'  
Person: Name='Иван' City='Пловдив' Age='27'  
Person: Name='Мария' City='София' Age='37'
```

СОРТИРАНЕ С ИНТЕРФЕЙС COMPARATOR

- В много приложения освен естествената наредба са необходими допълнителни сортировки
 - За целта трябва да се изпълни интерфейсът `Comparator<T>` и да се реализира желаната сортировка
- Интерфейсът описва градивна единица за сравняване на обекти от тип `T`.
 - Метод `int compare(T,T)`

ЕСТЕСТВЕНА ПОДРЕДБА

```
public interface Comparator<T> {  
    int compare(T o1, T o2);  
}
```



= 0

Равенство между двата обекта o1 и o2

< 0

o1 се разглежда като по-малък от o2

> 0

o1 се разглежда като по-голям от o2

ПРИМЕР

```
public final class PersonNameComparator implements Comparator<Person> {  
  
    public int compare(final Person person1, final Person person2) {  
        Objects.requireNonNull(person1, "person1 must not be null");  
        Objects.requireNonNull(person2, "person2 must not be null");  
  
        return person1.getName().compareTo(person2.getName());  
    }  
}
```

Примерна имплементация на compare() върху
имена от Person

КОНСИСТЕНТНОСТ МЕЖДУ МЕТОДИ

- Съществено е консистентно да бъдат имплементирани методите:
 - equals(Object)
 - hashCode()
 - compareTo() – ако съществува трябва да бъде консистентен с първия
- Ако не съблюдаваме това изискване, може да се трудно за възпроизвеждане грешки
 - Изразяват се в странно поведение на програмите

ПРИМЕР – ПЪРВА ВЕРСИЯ

```
import java.util.HashSet;
import java.util.Set;
import java.util.TreeSet;
public final class LawOfBig3Example {
    private static class SimplePerson implements Comparable<SimplePerson> {
        private final String name;
        SimplePerson(final String name) {
            this.name = name;
        }
        @Override
        public int compareTo(final SimplePerson other) { ←
            return name.compareTo(other.name);
        }
    }

    public static void main(String[] args) {
        final Set<SimplePerson> hashSet = new HashSet<SimplePerson>();
        hashSet.add(new SimplePerson("Test"));
        hashSet.add(new SimplePerson("Test"));
        System.out.println("HashSet size = " + hashSet.size()); // Size = 2
        final Set<SimplePerson> treeSet = new TreeSet<SimplePerson>();
        treeSet.add(new SimplePerson("Test"));
        treeSet.add(new SimplePerson("Test"));
        System.out.println("TreeSet size = " + treeSet.size()); // Size = 1
    }
    private LawOfBig3Example() { }
}
```


ПРИМЕР – ПЪРВА ВЕРСИЯ

```
import java.util.HashSet;
import java.util.Set;
import java.util.TreeSet;
public final class LawOfBig3Example {
    private static class SimplePerson implements Comparable<SimplePerson> {
        private final String name;
        SimplePerson(final String name) {
            this.name = name;
        }
        @Override
        public int compareTo(final SimplePerson other) {
            return name.compareTo(other.name);
        }
    }
}
```

?

Резултат

HashSet size = 2
TreeSet size = 1

```
public static void main(String[] args) {
    final Set<SimplePerson> hashSet = new HashSet<SimplePerson>();
    hashSet.add(new SimplePerson("Test"));
    hashSet.add(new SimplePerson("Test"));
    System.out.println("HashSet size = " + hashSet.size()); // Size = 2
    final Set<SimplePerson> treeSet = new TreeSet<SimplePerson>();
    treeSet.add(new SimplePerson("Test"));
    treeSet.add(new SimplePerson("Test"));
    System.out.println("TreeSet size = " + treeSet.size()); // Size = 1
}
private LawOfBig3Example() { }
```

КОМЕНТАР

- Изненадващо, в `HashSet<SimplePerson>` се намират 2 елемента, а в `TreeSet<SimplePerson>` само 1
- Обяснение:
 - Методът `equals(Object)`, който се използва за еквивалентност на елементи вътре в едно „чекмедже“ не е имплементиран в класа `SimplePerson`
 - Така, при сравнение на два обекта се сравняват референциите им
 - За класа `TreeSet<SimplePerson>` при добавяне на елемент се изключват дубликати и така се запазва жялостността на множество, което използва метода `compareTo(SimplePerson)` вместо `equals(Object)`
 - Т.е., проблемът е, че методът `compareTo(SimplePerson)` не е съвместим с `equals(Object)` – липсва съответна имплементация на `equals(Object)` в класа `SimplePerson`

ПРИМЕР – ВТОРА ВЕРСИЯ

```
import ...
public final class LawOfBig3Example {
    private static class SimplePerson implements Comparable<SimplePerson> {
        ...
        public boolean equals(final Object other) { ←
            if (other == null)
                return false;
            if (this == other)
                return true;
            if (this.getClass() != other.getClass())
                return false;
            final SimplePerson otherPerson = (SimplePerson) other;
            return compareTo(otherPerson) == 0;
        }
    }
    @Override
    public int compareTo(final SimplePerson other) {
        return name.compareTo(other.name);
    }
}
```

```
public static void main(String[] args) {
    final Set<SimplePerson> hashSet = new HashSet<SimplePerson>();
    hashSet.add(new SimplePerson("Test"));
    hashSet.add(new SimplePerson("Test"));
    System.out.println("HashSet size = " + hashSet.size()); // Size = 2
    final Set<SimplePerson> treeSet = new TreeSet<SimplePerson>();
    treeSet.add(new SimplePerson("Test"));
    treeSet.add(new SimplePerson("Test"));
    System.out.println("TreeSet size = " + treeSet.size()); // Size = 1
}
private LawOfBig3Example() { }
}
```


ПРИМЕР – ВТОРА ВЕРСИЯ

```
import ...
public final class LawOfBig3Example {
    private static class SimplePerson implements Comparable<SimplePerson> {
        ...
        public boolean equals(final Object other) {
            if (other == null)
                return false;
            if (this == other)
                return true;
            if (this.getClass() != other.getClass())
                return false;
            final SimplePerson otherPerson = (SimplePerson) other;
            return compareTo(otherPerson) == 0;
        }
    }
    @Override
    public int compareTo(final SimplePerson other) {
        return name.compareTo(other.name);
    }
}
```

?

Результат

HashSet size = 2
TreeSet size = 1

```
public static void main(String[] args) {
    final Set<SimplePerson> hashSet = new HashSet<SimplePerson>();
    hashSet.add(new SimplePerson("Test"));
    hashSet.add(new SimplePerson("Test"));
    System.out.println("HashSet size = " + hashSet.size()); // Size = 2
    final Set<SimplePerson> treeSet = new TreeSet<SimplePerson>();
    treeSet.add(new SimplePerson("Test"));
    treeSet.add(new SimplePerson("Test"));
    System.out.println("TreeSet size = " + treeSet.size()); // Size = 1
}
private LawOfBig3Example() { }
```

КОМЕНТАР

- Същият резултат както при първата версия – независимо, че сме имплементирали `equals(Object)`
- Обяснение:
 - Два `SimplePerson` обекта се разглеждат като еквивалентни, когато имат еднакво съдържание
 - Но, до такава сравнение въобще не се стига
 - Понеже няма собствена реализация на метода `hashCode()` двата обекта, разглеждани като еднакви от `equals(Object)`, се намират в различни „чекмеджета“
 - Това противоречи на `hashCode()` контракта и води до това, че един и същ обект се записва повторно в `HashSet<SimplePerson>`

ПРИМЕР – ТРЕТА ВЕРСИЯ

```
import ...
public final class LawOfBig3Example {
    private static class SimplePerson implements Comparable<SimplePerson> {
        ...
        public boolean equals(final Object other) { ←
            @Override
            public int hashCode() { ←
                return name.hashCode();
            }
            @Override
            public int compareTo(final SimplePerson other) { ←
                return name.compareTo(other);
            }
        }
    }
}
```

```
public static void main(String[] args) {
    final Set<SimplePerson> hashSet = new HashSet<SimplePerson>();
    hashSet.add(new SimplePerson("Test"));
    hashSet.add(new SimplePerson("Test"));
    System.out.println("HashSet size = " + hashSet.size()); // Size = 2
    final Set<SimplePerson> treeSet = new TreeSet<SimplePerson>();
    treeSet.add(new SimplePerson("Test"));
    treeSet.add(new SimplePerson("Test"));
    System.out.println("TreeSet size = " + treeSet.size()); // Size = 1
}
private LawOfBig3Example() { }
}
```


ПРИМЕР – ТРЕТА ВЕРСИЯ

```
import ...
public final class LawOfBig3Example {
    private static class SimplePerson implements Comparable<SimplePerson> {
        ...
        public boolean equals(final Object other) { ←
            @Override
            public int hashCode() { ←
                return name.hashCode();
            }

            @Override
            public int compareTo(final SimplePerson other) { ←
                return name.compareTo(other);
            }
        }
    }
}
```

? Результат

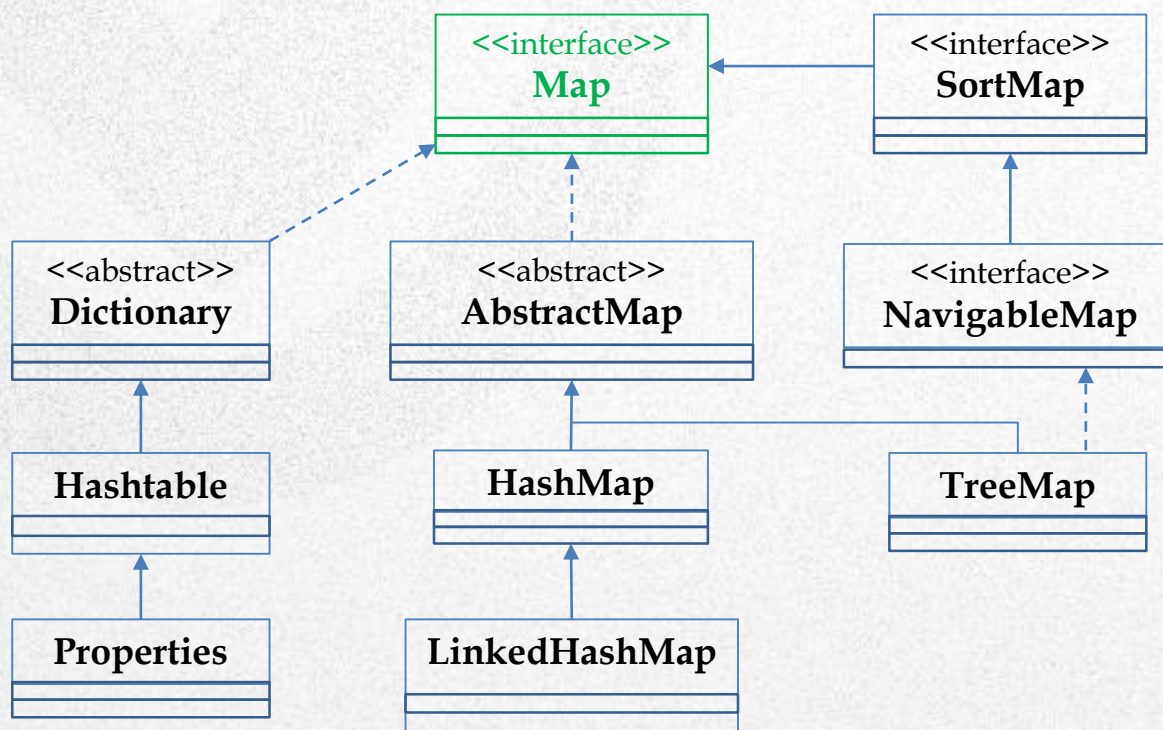
HashSet size = 1
TreeSet size = 1

```
public static void main(String[] args) {
    final Set<SimplePerson> hashSet = new HashSet<SimplePerson>();
    hashSet.add(new SimplePerson("Test"));
    hashSet.add(new SimplePerson("Test"));
    System.out.println("HashSet size = " + hashSet.size()); // Size = 2
    final Set<SimplePerson> treeSet = new TreeSet<SimplePerson>();
    treeSet.add(new SimplePerson("Test"));
    treeSet.add(new SimplePerson("Test"));
    System.out.println("TreeSet size = " + treeSet.size()); // Size = 1
}
private LawOfBig3Example() { }
}
```

КАРТИ

- Ще разгледаме реализации на интерфейса $\text{Map}<K,V>$
- 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



МЕТОДИ НА ИНТЕРФЕЙСА

void	clear() Removes all of the mappings from this map (optional operation).
default V	compute(K key, BiFunction<? super K,? super V,? extends V> remappingFunction) Attempts to compute a mapping for the specified key and its current mapped value (or null if there is no current mapping).
default V	computeIfAbsent(K key, Function<? super K,? extends V> mappingFunction) If the specified key is not already associated with a value (or is mapped to null), attempts to compute its value using the given mapping function and enters it into this map unless null.
default V	computeIfPresent(K key, BiFunction<? super K,? super V,? extends V> remappingFunction) If the value for the specified key is present and non-null, attempts to compute a new mapping given the key and its current mapped value.
boolean	containsKey(Object key) Returns true if this map contains a mapping for the specified key.
boolean	containsValue(Object value) Returns true if this map maps one or more keys to the specified value.
Set<Map.Entry<K,V>>	entrySet() Returns a Set view of the mappings contained in this map.
boolean	equals(Object o) Compares the specified object with this map for equality.
default void	forEach(BiConsumer<? super K,? super V> action) Performs the given action for each entry in this map until all entries have been processed or the action throws an exception.
V	get(Object key) Returns the value to which the specified key is mapped, or null if this map contains no mapping for the key.

МЕТОДИ НА ИНТЕРФЕЙСА

default V	getOrDefault(Object key, V defaultValue) Returns the value to which the specified key is mapped, or <code>defaultValue</code> if this map contains no mapping for the key.
int	hashCode() Returns the hash code value for this map.
boolean	isEmpty() Returns <code>true</code> if this map contains no key-value mappings.
Set<K>	keySet() Returns a <code>Set</code> view of the keys contained in this map.
default V	merge(K key, V value, BiFunction<? super V, ? super V, ? extends V> remappingFunction) If the specified key is not already associated with a value or is associated with null, associates it with the given non-null value.
V	put(K key, V value) Associates the specified value with the specified key in this map (optional operation).
void	putAll(Map<? extends K, ? extends V> m) Copies all of the mappings from the specified map to this map (optional operation).
default V	putIfAbsent(K key, V value) If the specified key is not already associated with a value (or is mapped to null) associates it with the given value and returns null, else returns the current value.
V	remove(Object key) Removes the mapping for a key from this map if it is present (optional operation).
default boolean	remove(Object key, Object value) Removes the entry for the specified key only if it is currently mapped to the specified value.

МЕТОДИ НА ИНТЕРФЕЙСА

default V	replace (K key, V value) Replaces the entry for the specified key only if it is currently mapped to some value.
default boolean	replace (K key, V oldValue, V newValue) Replaces the entry for the specified key only if currently mapped to the specified value.
default void	replaceAll (BiFunction<? super K,? super V,? extends V> function) Replaces each entry's value with the result of invoking the given function on that entry until all entries have been processed or the function throws an exception.
int	size () Returns the number of key-value mappings in this map.
Collection<V>	values () Returns a Collection view of the values contained in this map.

ЧЕСТО ИЗПОЛЗВАНИ 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<? Extends K, ? Extends V> map)	Добавя всички записи от дадената като параметър карта
V remove(Object key)	Изтрива запис
boolean containsKey(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

? 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

? **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

Често срещан в практиката проблем – съпоставяме на множество
ВХОДНИ СТОЙНОСТИ някакво множество ОТ ИЗХОДНИ СТОЙНОСТИ

```
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 + "");  
    }  
}
```

- При малки множества тази структура е приемлива
- При големи – трудна за поддържане и необозрима

ПРИМЕР

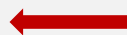
```
import java.awt.Color;
import java.util.*;
public final class HashMapExample {
    private static final Map<String, Color> nameToColor = new HashMap<>();
    static {
        initMapping(nameToColor);
    }
    public static void main(final String[] args) {
        System.out.println(mapToColor("RED"));
        System.out.println(mapToColor("GREEN"));
        System.out.println(mapToColor("BLACK"));
    }
    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);
    }
}
```

Решение:

- Дефинираме таблица на съпоставяме (mapping) като `HashMap<K,V>`
- Съпоставката се реализира чрез достъп със съответен ключ
- Кратка и прегледна реализация

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("BLACK"));
    }
    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);
    }
}
```



Инициализация – като анонимен клас

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("BLACK"));
    }
    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);
    }
}
```

?

Результат

```
java.awt.Color[r=255,g=0,b=0]
java.awt.Color[r=0,g=255,b=0]
java.awt.Color[r=0,g=0,b=0]
```

ПРИМЕР



Результат

```
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 java.awt.Color[r=255,g=0,b=0]
        java.awt.Color[r=0,g=255,b=0]
    }
    private static void initMapping(Map<String, Color> map) {
        map.put("RED", Color.RED);
        map.put("GREEN", Color.GREEN);
    }
}
```

Exception in thread "main" [java.lang.IllegalArgumentException](#): No color for: 'UNKNOWN'
at [HashMapExample.mapToColor\(HashMapExample.java:19\)](#)
at [HashMapExample.main\(HashMapExample.java:13\)](#)

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

```
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

```
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: " + 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"));
    }
}
```

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

ПРИМЕР

? Результат

```
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"));
    }
}
```

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


ФИЛТРИ

- Филтри върху елементи на една колекция по различни критерии
- Основават се върху дефиниране на генетичния интерфейс `Filterable<T>`
 - Съдържа метода `асерт(T)` – решава дали един елемент се акцептира
 - Трябва да бъде приложен за всеки елемент от колекцията

ФИЛТЪР ЗА ЕКВИВАЛЕНТНОСТ

```
public interface IFilter<T> {  
    boolean accept(final T object);  
}
```

```
public class EqualsFilter<T> implements IFilter<T> {  
    private final T acceptedValue;  
  
    public EqualsFilter(final T acceptedValue) { ←  
  
        this.acceptedValue = Objects.requireNonNull(acceptedValue, "acceptedValue must not be null");  
    }  
  
    @Override  
    public boolean accept(final T object) {  
        return acceptedValue.equals(object);  
    }  
}
```

Филтърът проверява за еквивалентност,
използвайки метода equals()

КЛАС FILTERUTILS

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

public final class FilterUtils {
    public static <T> List<T> applyfilter(final List<T> values, final IFilter<T> filter) {
        final List<T> filteredValues = new ArrayList<T>();
        for (final T current : values) {
            if (filter.accept(current))
                filteredValues.add(current);
        }

        return filteredValues;
    }
}
```

- Позволява да реализираме различни филтри
- Списъкът-параметър се обхожда
- Всеки елемент се оценява спрямо условията на филтъра, зададени във втория параметър
- Елементите, удовлетворяващи филтъра се записват в списъка `filteredValues`

ПРИМЕР



Резултат

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

public class SimpleFilterExample {
    public static void main(final String[] args) {
        final List<Integer> intValues = Arrays.asList(1, 2, 3, 4, 5, 6, 7);

        // int-filter on value 2
        final IFilter<Integer> numberFilter = new EqualsFilter<Integer>(2);
        final List<Integer> filteredValues = FilterUtils.applyfilter(intValues, numberFilter);
        System.out.println(filteredValues);
    }
}
```

[2]

СРАВНЕНИЯ

- Често е необходимо да се филтрира не за точно определена стойност, а за условия или области от стойности
 - Напр., искаме да получим всички стойности, които са по-големи или по-малки от определена стойност
- За такива сравнения познаваме вече интерфейса `Comparable<T>`
- За реализиране на типични сравнения използваме генетична дефиниция на филтри, базирана на интерфейса `Comparable<T>`
 - По този начин лесно се реализират различни сравнения

ПО-ГОЛЯМ

```
public class Greater<T extends Object & Comparable<T>> implements IFilter<T> {  
    private final T lowerBound;  
  
    → public Greater(final T lowerBound) {  
  
        this.lowerBound = Objects.requireNonNull(lowerBound, "lowerBound must not be null");  
    }  
  
    @Override  
    public boolean accept(final T object) {  
        return lowerBound.compareTo(object) < 0;  
    }  
}
```

Реализация на филтър „по-голям“

МЕЖДУ

```
public class Between<T extends Object & Comparable<T>> implements IFilter<T> {  
    private final T lowerBound;  
    private final T upperBound;  
  
    → public Between(final T lowerBound, final T higherBound) {  
        this.lowerBound = Objects.requireNonNull(lowerBound, "lowerBound must not be null");  
        this.upperBound = Objects.requireNonNull(upperBound, "upperBound must not be null");  
  
        if (!(lowerBound.compareTo(upperBound) <= 0))  
            throw new IllegalArgumentException("lowerBound " + lowerBound +  
                " must be <= upperBound " + upperBound);  
    }  
  
    @Override  
    public boolean accept(final T object) {  
        return lowerBound.compareTo(object) <= 0 &&  
            higherBound.compareTo(object) >= 0;  
    }  
}
```

Освен сравнения с гранични стойности, в много случаи ни интересуват области от стойности

ЛОГИЧЕСКИ ИЗРАЗИ

- В много приложения е необходимо комбиниране (свързване) на условия
- С помощта на филтри лесно могат да се реализират логическите оператори:
 - AND
 - OR
 - NOT

ИЛИ

```
public class Or<T> implements IFilter<T> {  
    private final IFilter<T> filter1;  
    private final IFilter<T> filter2;  
  
    public Or(final IFilter<T> filter1, final IFilter<T> filter2) {  
        this.filter1 = Objects.requireNonNull(filter1, "filter1 must not be null");  
        this.filter2 = Objects.requireNonNull(filter2, "filter2 must not be null");  
    }  
  
    @Override  
    public boolean accept(final T object) {  
        return (filter1.accept(object) || filter2.accept(object));  
    }  
}
```

Реализация на OR

?

Реализация на AND

И

```
public class And<T> implements IFilter<T> {  
    private final IFilter<T> filter1;  
    private final IFilter<T> filter2;  
  
    public Or(final IFilter<T> filter1, final IFilter<T> filter2) {  
        this.filter1 = Objects.requireNonNull(filter1, "filter1 must not be null");  
        this.filter2 = Objects.requireNonNull(filter2, "filter2 must not be null");  
    }  
  
    @Override  
    public boolean accept(final T object) {  
        return (filter1.accept(object) && filter2.accept(object));  
    }  
}
```

Реализация на AND

HE

```
public class Not<T> implements IFilter<T> {  
    private final IFilter<T> filter;  
  
    public Not(final IFilter<T> filter) {  
        this.filter = Objects.requireNonNull(filter, "filter must not be null");  
    }  
  
    @Override  
    public boolean accept(final T object) {  
        return !(filter.accept(object));  
    }  
}
```

Реализация на NOT

СПЕЦИАЛИЗАЦИЯ ЗА РАЗЛИЧНИ ТИПОВЕ ДАННИ

- Освен общоизвестните филтри за определени типове данни е желателно дефиниране на специални филтри
- Напр., за символни низове
 - Филтър за принадлежност – съдържа ли определен подниз

ПРИНАДЛЕЖНОСТ

```
public class StringContains implements IFilter<String> { ←
    private final String necessarySubstring;

    public StringContains(final String necessarySubstring) {
        this.necessarySubstring = Objects.requireNonNull(necessarySubstring, "necessarySubstring must
                                                                not be null");
    }

    @Override
    public boolean accept(final String value) {
        if (value == null)
            return false;
        return value.contains(necessarySubstring);
    }
}
```

Реализация на филтър за принадлежност

ПРИМЕР

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



Резултат

```
public final class SimpleFilterExample2 {
    public static void main(final String[] args) {
        final List<Integer> intValues = Arrays.asList(1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15);

        // Стойности, които не са в областта 4-11
        final IFilter<Integer> inverseNumberFilter = new Not<Integer>(new Between<Integer>(4, 11));
        final List<Integer> filteredValues1 = FilterUtils.applyfilter(intValues, inverseNumberFilter);
        System.out.println(filteredValues1);

        // Стойности, които са в областта 3-7 или са по-големи от 12
        final IFilter<Integer> range2_7OrGreater12 = new Or<Integer>(new Between<Integer>(3, 7),
                                                                    new Greater<Integer>(12));
        final List<Integer> filteredValues2 = FilterUtils.applyfilter(intValues, range2_7OrGreater12);
        System.out.println(filteredValues2);
    }
}
```

```
[1, 2, 3, 12, 13, 14, 15]
[3, 4, 5, 6, 7, 13, 14, 15]
```

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.add(MALE);
        persons.add(MALE);
        persons.add(FEMALE);
        persons.add(FEMALE);
        persons.add(FEMALE);
        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);
    }
}
```

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 All Persons after shuffle: [Person: Name='Female' City='New York' Age='43', Person: Name='Mister X'
        fi City='Sydney' Age='44', Person: Name='Female' City='New York' Age='43', Person: Name='Male' City='Plovdiv'
        fi Age='42', Person: Name='Male' City='Plovdiv' Age='42', Person: Name='Female' City='New York' Age='43']
        S
        S 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'
        priv City='Sydney' Age='44', Person: Name='Mister X' City='Sydney' Age='44', Person: Name='Female' City='New
        fi York' Age='43']
        fi
        fi Male Persons: 0
        p MisterX Persons: 3
        persons.addAll(femaleList);
        persons.add(MISTER_X);
        return persons;
```

БЛАГОДАРЯ ЗА ВНИМАНИЕТО!

КРАЙ “КОЛЕКЦИИ”

