

Објектно орјентисано програмирање



Владимир Филиповић

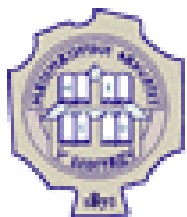
vladaf@matf.bg.ac.rs

Колекције



Владимир Филиповић

vladaf@matf.bg.ac.rs

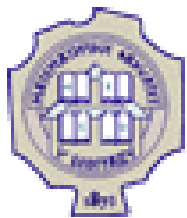


Колекције

Ова презентација описује како Јава библиотека може помоћи у традиционалном структурисању података који су неопходни за озбиљно програмирање.

Јава је на почетку испоручивана са малим скупом класа за најкорисније структуре података: класе `Vector`, `Stack`, `Hashtable`, `BitSet` и интерфејс `Enumeration` су обезбеђивали рад са структурама података.

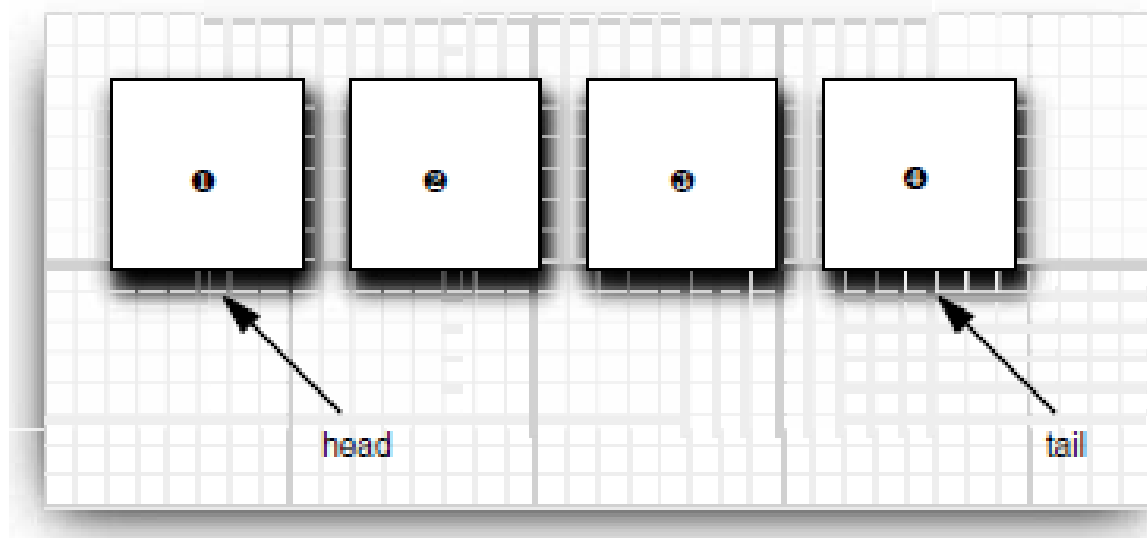
У даљем развоју Јаве је требало помирити супротстављене захтеве: библиотека за рад са колекцијама треба да буде мала и да се лако може научити, да не буде сложена као што је `STL` код `of C++`, али да омогући рад са генеричким алгоритмима на начин сличан оном који је уведен код `STL`-а. Надаље, било је потребно да се старе, већ испоручене класе природно уклопе у нови оквир.



Интерфејс и имплементација

Као код свих модерних библиотека за структуре података, и овде је интерфејс одвојен од имплементације.

Начин одвајања ће бити детаљније приказан на структури података ред, која се користи када елементе треба обрађивати по редоследу приспећа (енг. first in, first out - FIFO).



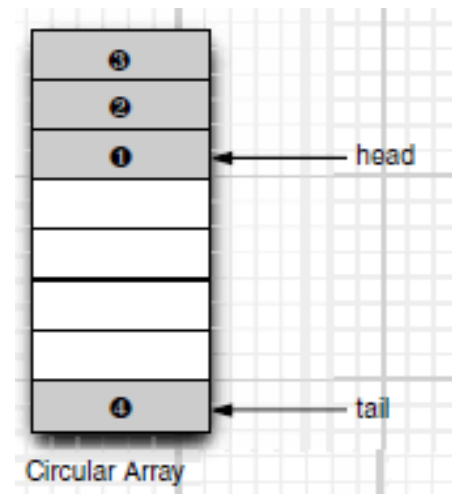
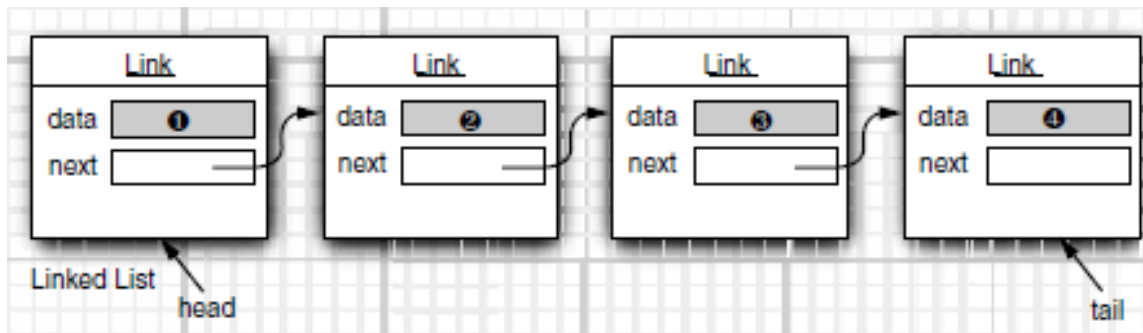


Интерфејс и имплементација (2)

Минимална форма интерфејса за ред може да има следећи облик:

```
interface Queue<E> // a simplified form of the interface in the standard library
{
    void add(E element);
    E remove();
    int size();
}
```

Интерфејс не говори ништа о томе на који ће начин ред бити имплементиран (као кружни низ, као повезана листа или на неки трећи начин).





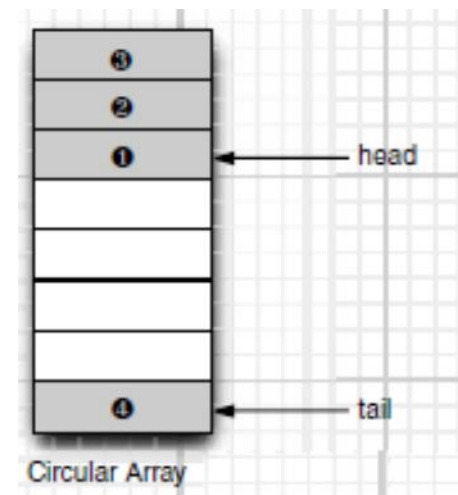
Интерфејс и имплементација (3)

Свака од имплементација је одређена класом која имплементира интерфејс Queue.

```
class CircularArrayQueue<E> implements Queue<E> // not an actual library class
{
    CircularArrayQueue(int capacity) { . . . }
    public void add(E element) { . . . }
    public E remove() { . . . }
    public int size() { . . . }

    private E[] elements;

    private int head;
    private int tail;
}
```

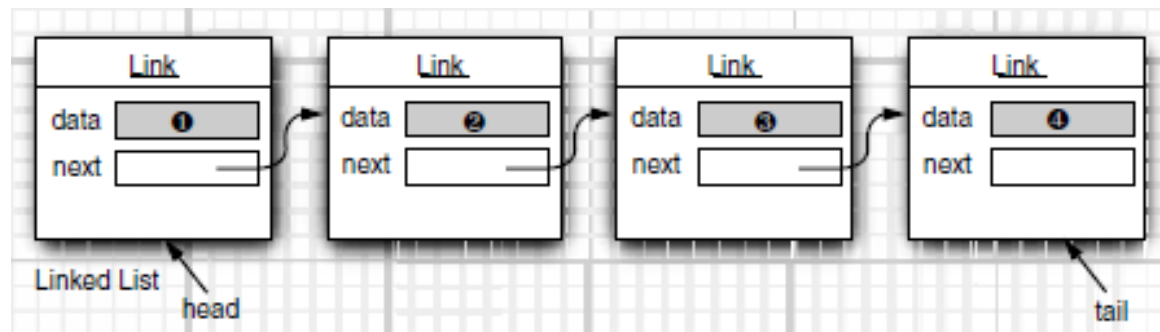




Интерфејс и имплементација (4)

```
class LinkedListQueue<E> implements Queue<E> // not an actual library class
{
    LinkedListQueue() { . . . }
    public void add(E element) { . . . }
    public E remove() { . . . }
    public int size() { . . . }

    private Link head;
    private Link tail;
}
```





Интерфејс и имплементација (5)

Када програм користи колекцију, он (када се колекција направи) не мора знати која је имплементација колекције стварно коришћена. Стога, има смисла да се конкретна класа користи само при креирању објекта, а да се за чување референце на објекат користи тип интерфејса.

```
Queue<Customer> expressLane = new CircularArrayQueue<Customer>(100);  
expressLane.add(new Customer("Harry"));
```

На овај начин се, чак иако дође до предомисљања, лако може користити и другачија имплементација. Тада програм треба променити само на једном месту, тамо где се формира нови објекат.

На пример, ако се донесе одлука да је `LinkedListQueue` ипак бољи избор, тада код постаје:

```
Queue<Customer> expressLane = new LinkedListQueue<Customer>();  
expressLane.add(new Customer("Harry"));
```




Интерфејс и имплементација (6)

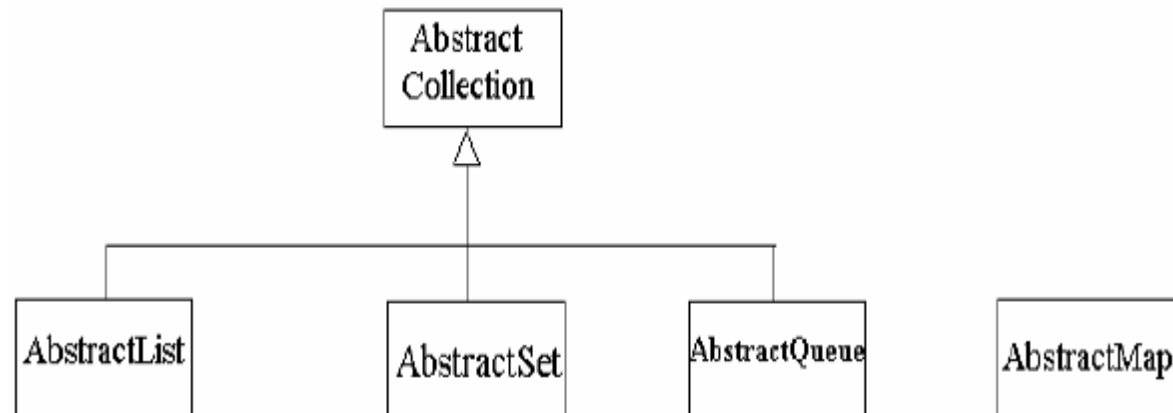
Зашто би се давала предност једној имплементацији у односу на другу? Интерфејс ништа не казује о ефикасности имплементације. Кружни низ је нешто ефикаснији од повезане листе. Међутим, како је то уобичајено, за његово коришћење треба платити додатну цену.

Кружни низ је ограничена колекција и она има коначан капацитет. Ако није унапред позната горња граница броја објеката који ће бити у колекцији, тада је боље користити мање ефикасну имплементацију која је заснована на повезаној листи.



Интерфејс и имплементација (7)

Када се прегледа API документација, уочава се да постоји још један скуп класа, чије име почиње са речи **Abstract**, као што је класа **AbstractQueue**. Ове класе треба да користе програмери који имплементирају библиотеке класа. У оним (вероватно јако ретким) ситуацијама када програмер сам треба да имплементира своју класу за ред, то ће лакше реализовати уколико наследи класу **AbstractQueue** него ако одлучи да имплементира све методе интерфејса **Queue**.





Колекције и итератори

Основни интерфејс за колекцијске класе у Јави је интерфејс **Collection**. Овај интерфејс садржи два најважнија метода:

```
public interface Collection<E>
{
    boolean add(E element);
    Iterator<E> iterator();
    ...
}
```

Поред њих, постоје и додатни методи у оквиру овог интерфејса и они ће бити размотрени касније.

Метод **add** додаје елеменат у колекцију. Овај метод враће **true** ако је додавање метода заиста променило колекцију, а враће **false** ако је колекција непромењена. На примере, ако се додаје у скуп елеменат који се већ налази у том скупу, тада захтев за додавање нема ефекта јер скуп одбија дупликате, па метод **add** враће **false**.



Колекције и итератори (2)

Метод `iterator` враће објекат који имплементира интерфејс `Iterator`. Тако добијени објекат-итератор се може користити да се редом, један по један, обиђу елементи у колекцији.

Интерфејс `Iterator` садржи три метода:

```
public interface Iterator<E>
{
    E next();
    boolean hasNext();
    void remove();
}
```

Поновљеним позивима метода `next`, могу се један за другим посетити сви елементи у колекцији. Ипак, као се стигне до краја колекције и тада позове метод `next`, биће избачен изузетак типа `NoSuchElementException`.



Колекције и итератори (3)

Стога је потребно да се метод `hasNext` позива пре позива метода `next`. Метод `hasNext` враће `true` ако објекат-итератор има још елемената које треба да посети.

Ако треба испитати све елементе у колекцији, тада програмер креира итератор над колекцијом и потом у циклусу понавља позивање метода `next` све док метод `hasNext` враће `true`.

Пример.

Илуструје обраду елемената колекције коришћењем итератора.

```
Collection<String> c = . . . ;  
Iterator<String> iter = c.iterator();  
while (iter.hasNext())  
{  
    String element = iter.next();  
    do something with element  
}
```



Колекције и итератори (4)

Почев од Јава 5.0, претходни циклус се може краће и елегантније записати коришћењем “for each” циклуса:

```
for (String element : c)
{
    do something with element
}
```

Преводац једноставно преводи “for each” циклус у циклус са итератором. Циклус “for each” добро функционише са сваким објектом који имплементира интерфејс `Iterable`, који садржи само један метод:

```
public interface Iterable<E>
{
    Iterator<E> iterator();
}
```



Колекције и итератори (5)

Како интерфејс `Collection` проширује интерфејс `Iterable`, то се циклус “for each” може користити са ма којом колекцијом из стандардне библиотеке.

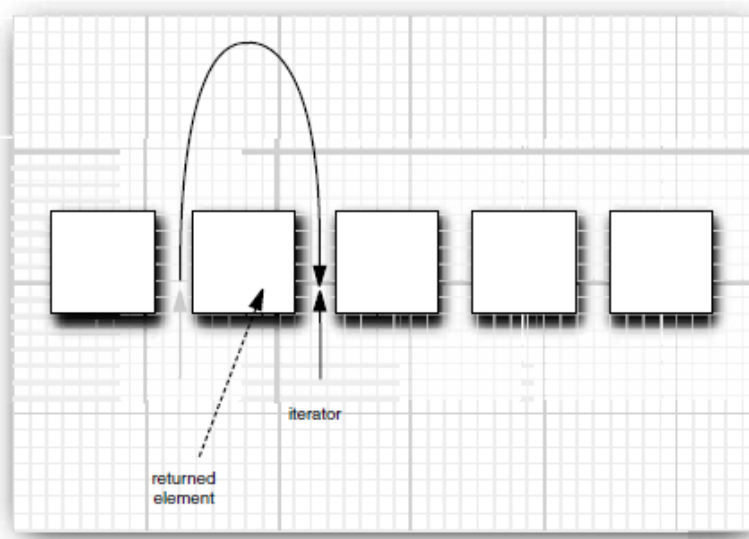
Ред посећивања елемената колекције зависи од типа колекције. Ако се итерира кроз колекцију `ArrayList`, тада итератор почиње од индекса 0 и у сваком кораку увећава индекс.

Међутим, ако се посећују елементи у колекцији типа `HashSet`, они ће бити набрајани у суштински случајном редоследу. Програмер може бити сигуран да ће тоом итерирања бити побројани сви елементи колекције, али не сме да се ослања на редослед. Ово најчешће и није ограничење, јер редослед набрајања елемената није битан приликом израчунавања суме или пребројавања поклапања.



Колекције и итератори (6)

Код Јава итератора је дохватање елемента чврсто спрегнуто са променом позиције – итератор приликом позива `next` прелази преко елемента тј. напредује за једну позицију.



Може се посматрати као да су Јава итератори на позицијама између елемената. При позиву метода `next`, итератор прескаче преко тог следећег елемента, и враће референцу на елемент преко ког је управо прешао.



Колекције и итератори (7)

Метод `remove` интерфејса `Iterator` уклања елемент враћен последњим позивом метода `next`. У многим ситуацијама ово има смисла - треба да се види елемент пре него што се одлучи да ли га треба уклонити.

Међутим, уколико треба да се уклони елемент са задате позиције, ипак треба да се он прође итератором.

Пример. Уклањање првог елемент из колекције ниски `c`:

```
Iterator<String> it = c.iterator();  
it.next(); // preskoci se prvi element  
it.remove(); // sada se ukloni
```

Још је битније да постоји веза између позива метода `next` и `remove`. Није исправно позвати метод `remove` ако му није претходио позив метода `next`. Уколико се то ипак уради, долази до избацивања изузетка `IllegalStateException`.



Колекције и итератори (8)

Пример. Ако треба да се уклоне два суседна елемента, не може се просто позвати:

```
it.remove();  
it.remove(); // Greska!
```

Уместо тога, прво се мора позвати метод `next` како би итератор прешао преко елемента који треба уклонити:

```
it.remove();  
it.next();  
it.remove(); // OK
```



Масовне операције са колекцијама

Пример. Тражи се пресек два скупа, тј. њихове заједничке елементе.

Прво се направи нови скуп који ће представљати резултат, при чему се користи чињеница да свака колекција има конструктор чији је параметар друга колекција.

```
Set<String> rezultat = new HashSet<String>(a);
```

Потом се користи метод `retainAll` који задржава све елементе који се појављују и у `b`:

```
result.retainAll(b);
```

Пресек је формиран без програмирања петље.



Масовне операције са колекцијама (2)

Повремено су потребна превођења између традиционалних низова и модернијих колекција. Ако је потребно да се низ претвори у колекцију, користи се `Arrays.asList`.

Пример. У овом случају се низ ниски претвара у хеш скуп.

```
String[] vrednosti = ...;
```

```
HashSet<String> osoblje = new HashSet<String>(Arrays.asList(vrednosti));
```

Добијање низа од колекције је нешто компликованије.

Пример. Ако се користи метод `toArray` на следећи начин:

```
Object[] vrednosti = osoblje.toArray();
```

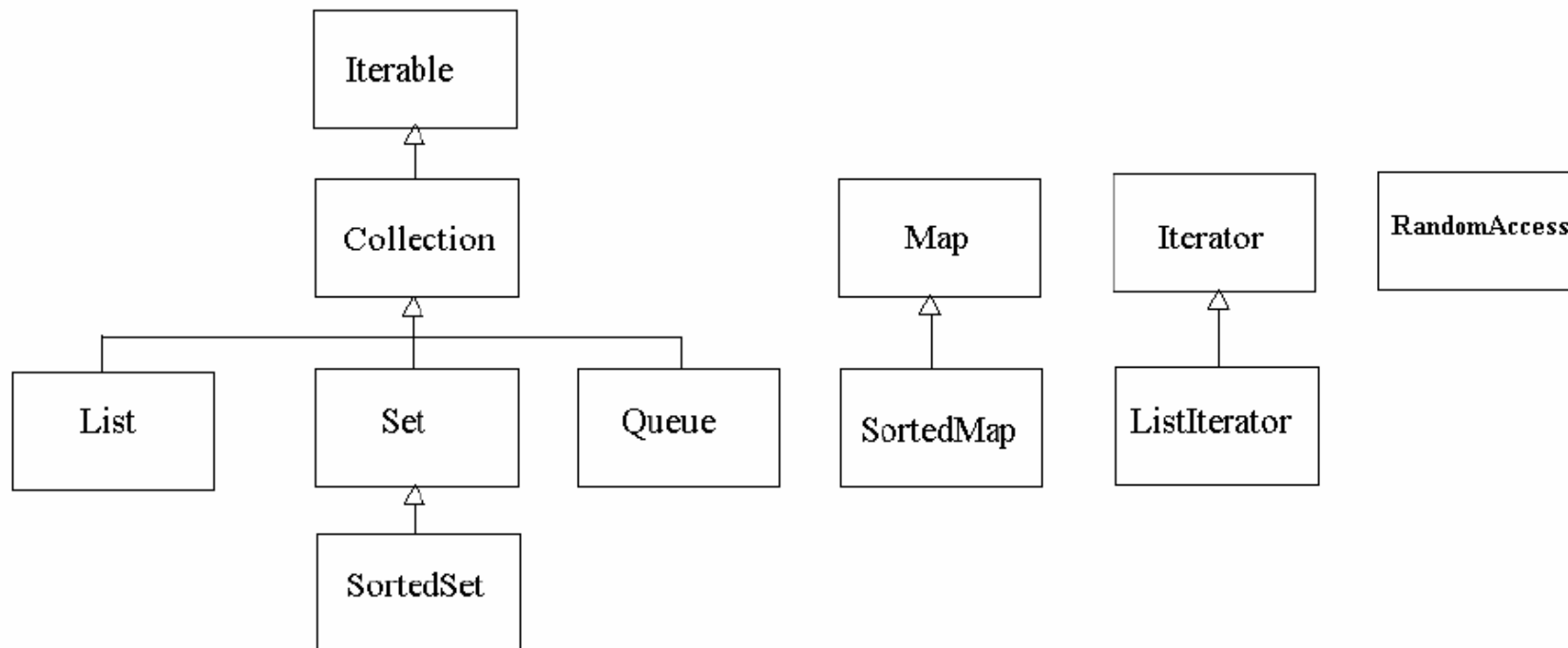
Добија се низ објеката, а не низ ниски. Наиме, низ који враћа `toArray` креиран је као `Object[]` и није могуће променити његов тип. Уместо тога, може се користити друга верзија метода `toArray` којој се проследи низ дужине 0 жељеног типа компоненте:

```
String[] vrednosti = osoblje.toArray(new String[0]);
```



Колекцијски интерфејси

Поред претходно побројаних, у стандардној библиотеци постоје и следећи интерфејси који се односе на колекције:





Колекцијски интерфејси (2)

java.util.Collection<E>

- **Iterator<E> iterator()**
враће итератор који се користи за приступ елементима колекције.
- **int size()**
враће број елемената у колекцији.
- **boolean isEmpty()**
враће true ако је колекција празна.
- **boolean contains(Object obj)**
враће true ако колекција садржи објекат једнак објекту obj.
- **boolean containsAll(Collection<? Extends E> other)**
враће true ако колекција садржи све елементе колекције other.
- **boolean add(Object element)**
додаје element у колекцију, враће true ако је успело додавање.
- **boolean addAll(Collection<? Extends E> other)**
додаје све елементе колекције other, враће true ако је успело додавање.



Колекцијски интерфејси (3)

java.util.Collection <E>

- `boolean remove(Object obj)`
брише `obj` из колекције. Враће `true` ако је успело брисање.
- `boolean removeAll(Collection<? Extends E> other)`
брише из ове колекције све елементе `other` колекције. Враћа `true` ако је успело брисање.
- `void clear()`
брише цео садржај колекције.
- `boolean retainAll(Collection<? Extends E> other)`
брише све елементе из колекције који не припадају колекцији `other`.
- `Object[] toArray()`
враћа све елементе колекције као низ објеката

java.util.Iterator<E>

- `boolean hasNext()`
враћа `true` ако има још елемената које треба посетити
- `next()`
враћа следећи објекат. Ако нема више елемената тј. ако је на крају избацује изузетак



Колекцијски интерфејси (4)

java.util.List<E>

- `ListIterator<E> listIterator()`
враће итератор за приступ елементима листе
- `ListIterator<E> listIterator(int index)`
враће итератор за приступ елементима листе, код које ће први позив `next()` вратити елемент са који је на позицији `index`
- `void add(int i, E element)`
додаје елемент на одређену позицију `i`
- `void addAll(int i, Collection<? Extends E> elements)`
додаје елементе из колекције од одређене позиције
- `E remove(int i)`
брише и враћа елемент на датој позицији
- `E set(int i, E element)`
заменаје елемент на датој позицији са новим елементом и враћа стари
- `int indexOf(Object element)`
враћа прву позицију на којој је пронађен дати елемент или `-1`, уколико исти није откривен



Колекцијски интерфејси (5)

java.util.List<E>

- `int lastIndexOf(Object element)`
враћа последњу позицију на којој је пронађен дати елемент или -1, уколико га нема

java.util.ListIterator<E>

- `boolean remove(Object obj)`
брише `obj` из колекције. Враће `true` ако је успело брисање.
- `boolean removeAll(Collection<? Extends E> other)`
брише из ове колекције све елементе `other` колекције. Враћа `true` ако је успело брисање.
- `void clear()`
брише цео садржај колекције.
- `boolean retainAll(Collection<? Extends E> other)`
брише све елементе из колекције који не припадају колекцији `other`.
- `Object[] toArray()`
враћа све елементе колекције као низ објеката
- `void add(E new Element)`
додаје елемент пре текуће позиције.



Колекцијски интерфејси (5)

java.util.ListIterator<E>

- void set(E new Element)
заменава последњи елемент посећен са next или previous са новим елементом.
- boolean hasPrevious()
враћа true ако постоји елемент ком се може приступити при итерирању уназад
- E previous()
враћа претходни објекат. Ако смо на почетку листе избацује се изузетак NoSuchElementException
- int nextIndex()
враће индекс елемента који ће вратити наредни позив метода next
- int previousIndex()
враће индекс елемента који ће вратити наредни позив метода previous



Колекцијски интерфејси (6)

java.util.Set<E>

- `Iterator<E> iterator()`
враће итератор који се користи за приступ елементима скупа.
- `int size()`
враће број елемената у скупу.
- `boolean isEmpty()`
враће `true` ако је скуп празан.
- `boolean contains(Object obj)`
враће `true` ако скуп садржи објекат једнак објекту `obj`.
- `boolean containsAll(Collection<? Extends E> other)`
враће `true` ако скуп садржи све елементе колекције `other`.
- `boolean add(Object element)`
додаје `element` у скуп, враће `true` ако је успело додавање.
- `boolean addAll(Collection<? Extends E> other)`
додаје све елементе колекције `other`, враће `true` ако је успело додавање.



Колекцијски интерфејси (7)

java.util.Set<E>

- `boolean remove(Object obj)`
уклања `obj` из скупа. Враће `true` ако је успело уклањање.
- `boolean removeAll(Collection<? Extends E> other)`
брише из скупа све елементе колекције `other`. Враће `true` ако је успело брисање.
- `void clear()`
брише цео скуп.
- `boolean retainAll(Collection<? Extends E> other)`
брише све елементе из скупа који не припадају колекцији `other`.
- `Object[] toArray()`
враћа све елементе скупа као низ објеката



Уређење у колекцији

Како се одређује начин на који ће се сортирати елементи у колекцији? Обично се претпоставља да се у колекцију додају елементи који имплементирају интерфејс `Comparable`.

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

Позив `a.compareTo(b)` враће 0 ако су `a` и `b` једнаки, негативан цео број ако `a` треба да се налази испред `b` у сортираном поретку, или позитиван цео број ако `a` треба да се налази иза `b`.

Када се у колекцију убацују елементи неког новог типа, начин сортирања елемената колекције се дефинише тако што тај нови тип имплементира интерфејс `Comparable`.



Уређење у колекцији (2)

Дата класа може имплементирати интерфејс само једном. Али шта ако треба да се сортира колекција елемената по различитим критеријумима? Надаље, шта да се учини ако су потребни сортирани примерци класе чији дизајнер није имплементирао интерфејс `Comparable`?

Може се обезбедити да колекција користи различите методе поређења, тако што се конструктору колекције прослеђује објекат који имплементира интерфејс `Comparator`.

Интерфејс `Comparator` декларише метод `compare`, који прихвата два параметра:

```
public interface Comparable<T>
{
    int compareTo(T a, T b);
}
```



Уређење у колекцији (3)

Баш као и `compareTo` метод интерфејса `Comparable`, метод `compare` интерфејса `Comparator` враће 0 ако су `a` и `b` једнаки, негативан цео број ако `a` треба да се налази испред `b` у сортираном поретку, или позитиван цео број ако `a` треба да се налази иза `b`.

На пример, за сортирање елемената по њиховом опису, једноставно се дефинише класа која имплементира `Comparator` интерфејс:

```
class ItemComparator implements Comparator<Item>
{
    public int compare (Item a , Item b)
    {   String descrA = a.getDescription();
        String descrB = b.getDescription();
        return descrA.compareTo(descrB);
    }
}
```



Уређење у колекцији (4)

Примећује се да овако направљен компаратор нема променљивих. Он је само власник метода за поређење.

Такав објекат се обично назива функцијски објекат.

Функцијски објекти су обично заједнички дефинисани при креирању, као примерци анонимних унутрашњих класа:

```
SortedSet<Item> sortByDescription = new TreeSet<Item>( new
    Comparator<Item>()
    {
        public int compare(Item a, Item b)
        {   String descrA = a.getDescription();
            String descrB = b.getDescription();
            return descrA.compareTo(descrB);
        }
    });
```




Уређење у колекцији (5)

java.lang.Comparable<T>

- `int compareTo(T other)`

Упоређује два објекта и враће негативну вредност ако текући објекат треба да буде испред објекта `other`, нулу ако се сматрају идентичним у сортираном поретку, или позитивну вредност ако текући објекат треба да буде после `other-a`.

java.lang.Comparator<T>

- `int compare(T a, T b)`

Упоређује два објекта и враћа негативну вредност ако је `a` испред `b`, нулу ако се сматрају идентичним у сортираном поретку, или позитивну вредност ако је `a` иза `b`.

java.util.SortedSet<E>

- `Comparator<? Super E> comparator()`

Враће компаратор коришћен за сортирање елемената, или `null` ако су елементи упоређивани методом `compareTo` интерфејса `Comparable`.

- `E first()`
- `E last()`

Враће први или последњи елемент у сортираној колекцији.



Колекције у оквиру ЈДК-а

Најчешће коришћене, већ развијене, класе доступне у Јава библиотеци :

LinkedList - повезана секвенца која дозвољава уметање и брисање са било ког места

ArrayList - индексирана секвенца која се смањује и расте динамички

HashSet - неповезана колекција која не прихвата дупликате (хеш скуп)

TreeSet - сортирани скуп (дрвоидни скуп)

PriorityQueue - колекција која дозвољава уклањање елемента са почетка (ред са приоритетима)

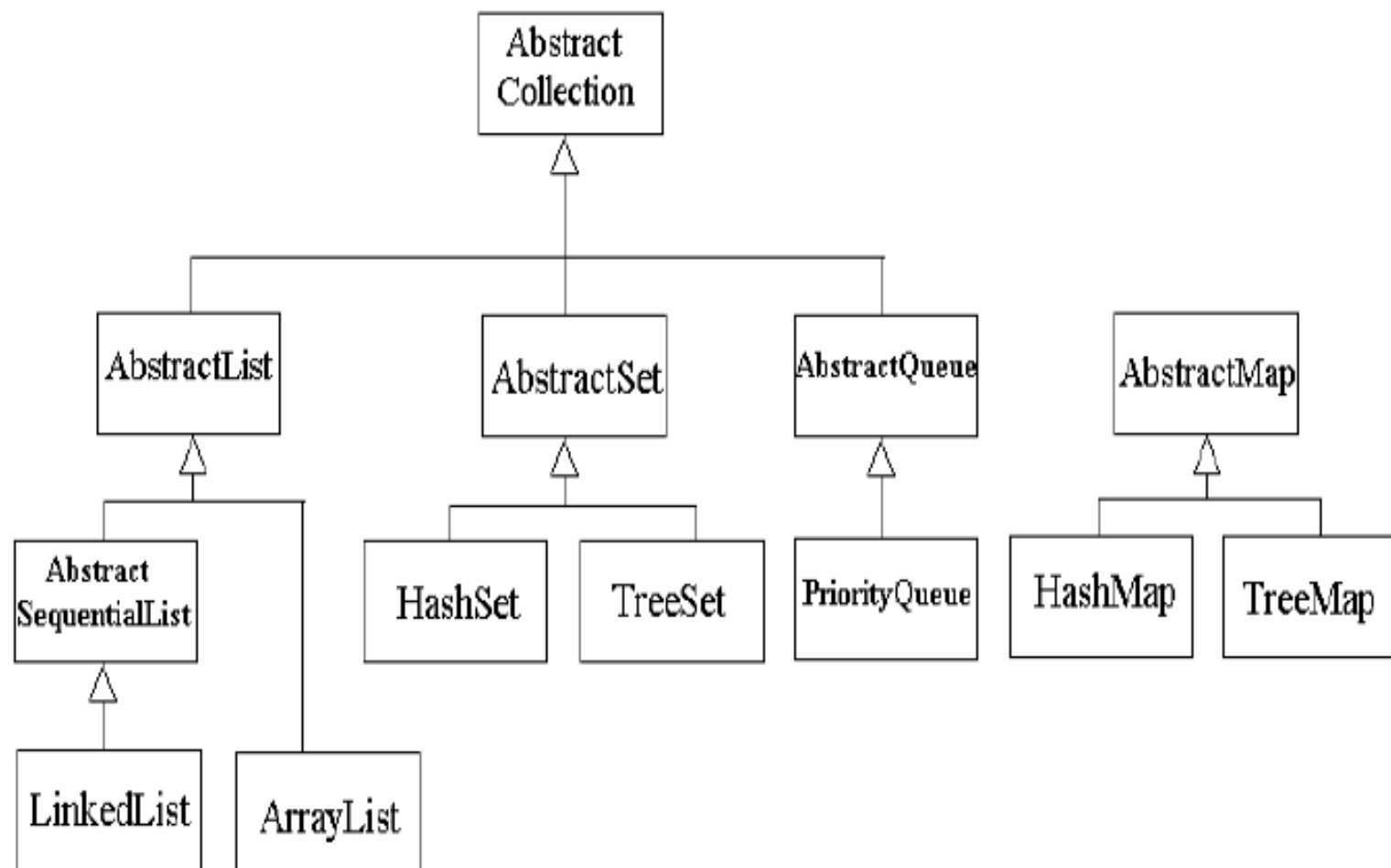
HashMap — каталог, тј. колекција структура са паровима кључ/вредност

TreeMap - каталог са сортираним кључевима



Колекције у оквиру ЈДК-а (2)

Хијерархија апстрактних и конкретних класа у ЈДК-у:





Повезана листа

Постоји битна разлика између повезаних листа и обичних колекција. Код повезане листе је важна позиција објекта.

У пракси је често потребно додати елементе у средину листе. Верзија метода `add` који зависи од позиције је зависна од итератора, јер итератор описује позицију у колекцији.

Коришћење итератора за додавање елемената има смисла једино код колекције са природним поретком. Тако, на пример, за разлику од листе, каталог података тј. мапа не захтева никакав редослед елемената.

Према томе, нема метода `add` у интерфејсу `Iterator`, али постоји подинтерфејс `ListIterator`, који садржи метод `add`:

```
interface ListIterator<E> extends Iterator<E>{  
    void add(E element);  
    ...  
}
```



Повезана листа (2)

Метод **add** додаје нови елемент испред позиције итератора.

Ако се метод **add** позове више пута, елементи се просто додају тим редом, један за другим, испред текуће позиције итератора.

Када се операција **add** користи за новокреирани итератор који показује на почетак повезане листе, додати елемент постаје глава листе.

Када је итератор иза последњег елемента листе (тј. **hasNext** је вратио **false**), додати елемент постаје нови реп листе.

Непосредно након позива **next**, метод **remove** уклања елемент лево од итератора. Међутим, ако је био позван метод **previous**, уклања се елемент десно. Није могуће звати **remove** два пута узастопно.

Метод **set** замењује последњи елемент враћен позивом метода **next** или **previous** новим елементом.



Повезана листа (3)

За разлику од метода `add` интерфејса `Collection`, овај метод не враћа `boolean` - претпоставља се да операција `add` увек мења листу.

Надаље, интерфејс `ListIterator` поседује два метода која се могу користити за обилажење листе уназад:

`E previous()`

`boolean hasPrevious()`

Попут метода `next`, метод `previous` враћа објекат преко кога је итератор управо прешао.

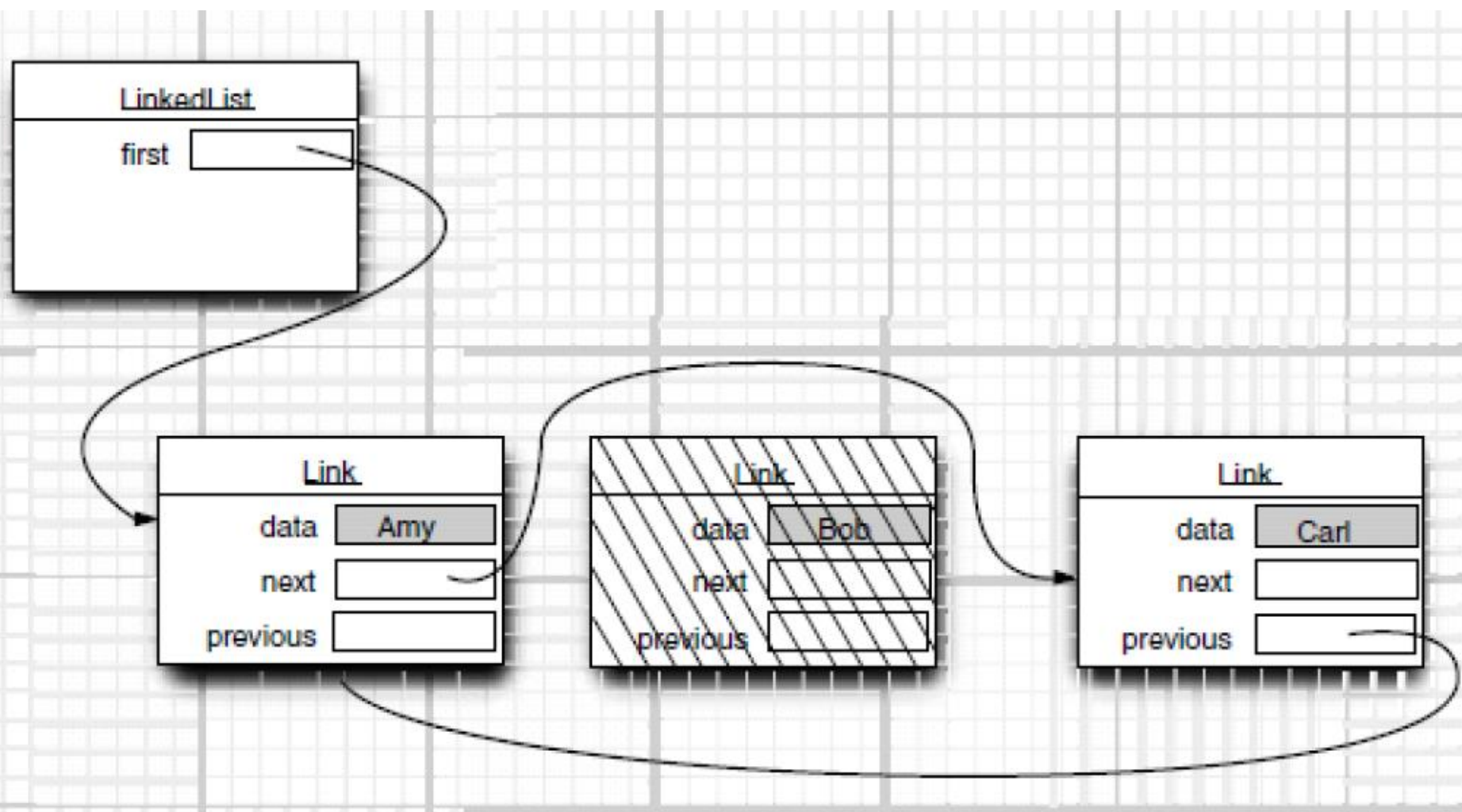
Класа `LinkedList` имплементира интерфејс `ListIterator` који садржи методе за додавање и уклањање елемената као и за пролазак кроз листу према напред и према назад.

Метод `listIterator` класе `LinkedList` враћа итератор, објекат класе која имплементира интерфејс `ListIterator`.

```
ListIterator<String> iter = osoblje.listIterator();
```



Повезана листа (4)





Повезана листа (5)

Многи корисни методи за оперисање повезаним листама декларисани су у интерфејсу `Collection`. Они су, већим делом, имплементирани у суперкласи `AbstractCollection` класе `LinkedList`. Метод `toString` позива `toString` за елементе и производи једну дугу ниску облика `[A, B, C]`. То је згодно приликом исправљања грешака.

Метод `contains` користи се за утврђивање да ли је елемент присутан у повезаној листи.

Пример. Позив `osoblje.contains("Pera")` враћа `true` ако повезана листа већ садржи стринг једнак стрингу `"Pera"`.

Повезана листа не подржава брз случајан приступ. Ако желимо да видимо n -ти елемент, морамо да пођемо од почетка и најпре пређемо преко првих $n-1$ елемената. Не постоји пречица.



Повезана листа (6)

Дакле, програмери обично не користе повезане листе у ситуацијама у којима је потребно приступати елементима коришћењем целобројног индекса.

Без обзира на то, класа `LinkedList` поседује метод `get` који омогућује приступ одређеном елементу:

```
LinkedList<String> lista = ... ;  
String obj = lista.get(n);
```

Интерфејс `ListIterator` такође поседује два метода који враћају индекс текуће позиције: `nextIndex`, који враћа целобројни индекс елемента који ће бити враћен наредним позивом метода `next`, и `previousIndex`, који враћа индекс елемента који ће бити враћен наредним позивом метода `previous`.



Повезана листа (7)

java.util.LinkedList<E>

- **LinkedList()**
прави празну повезану листу
- **LinkedList(Collection<? extends E> elements)**
прави листу и додаје све елементе из колекције
- **void addFirst(E element)**
додаје елемент на почетак листе
- **void addLast(E element)**
додаје елемент на крај листе
- **E getFirst()**
враће елемент са почетка листе
- **E getLast()**
враће елемент са краја листе
- **E removeFirst()**
брише и враће елемент са почетка листе
- **E removeLast()**
брише и враћа елемент са краја листе



Низовна листа

Интерфејс листе представља уређену колекцију у којој је позиција у листи елемента битна.

Постоје два правила за приступ елементима: преко итератора, као и метода **get** и **set** за директан приступ.

Други метод није препоручив за повезану листу али има смисла за низове.

Колекцијска библиотека класом **ArrayList** подржава низовну листу. Ова класа, исто као **LinkedList**, такође имплементира интерфејс **List**. Низовна листа енкапсулира низ динамички алоцираних објеката.



Низовна листа (2)

Методи `size`, `isEmpty`, `get`, `set`, `iterator`, и `listIterator` се извршавају за конкретно време. Операција `add` се извршава за амортизовано константо време, што значи да додавање n елемената захтева време $O(n)$. Додавање елемента дуже траје него што је то случај са `LinkedList` имоментацијом, али је зато бржи приступ елементу преко његовог индекса.

Сваки примерак класе `ArrayList` има свој капацитет. Капацитет је величина низа који служи за смештај елемената листе. Он аутоматски расте током додавања елемената у `ArrayList` објекат.

Коришћењем метода `ensureCapacity` апликација може увећати капацитет примерка `ArrayList` класе пре него што се дода велики број елемената. На овај начин се смањује број узастопних реалокација, чиме се убрзава рад програма.



Низовна листа (3)

Класе `ArrayList` и `Vector` имплементирају интерфејс `RandomAccess`.

Јава програмери који су ветерани користе класу `Vector` кад год им је потребан динамички низ. Зашто би требало користити `ArrayList` уместо `Vector`?

Из једноставног разлога. Сви методи класе `Vector` су синхронизовани што обезбеђује конзистентност података у вишеничном програмирању, али успорава извршење метода.

Насупрот томе, методи класе `ArrayList` нису синхронизовани, па их треба користити кад год није потребна синхронизација.



Хеш скуп

Повезане листе и низови дозвољавају убацивање елемената произвољно и притом воде рачуна о њиховом редоследу.

Међутим, ако треба наћи елеменат а коме није запамћена позиција, потребно је претражити (у најгорем случају) све елементе колекције.

Ово може одузети много времена уколико колекција има велики број елемената.

Уколико је редослед елемената небитан, може се користити колекција која обезбеђује много бржи приступ елементима. Једино је незгодно што се тада не зна ништа о редоследу елемената, већ их та структура организује по сопственом редоследу.



Хеш скуп (2)

Добро позната структура за брзо проналажење елемената је хеш табела. Ту се рачуна са целим бројевима, који се називају хеш-код и који се одређују за сваки објекат.

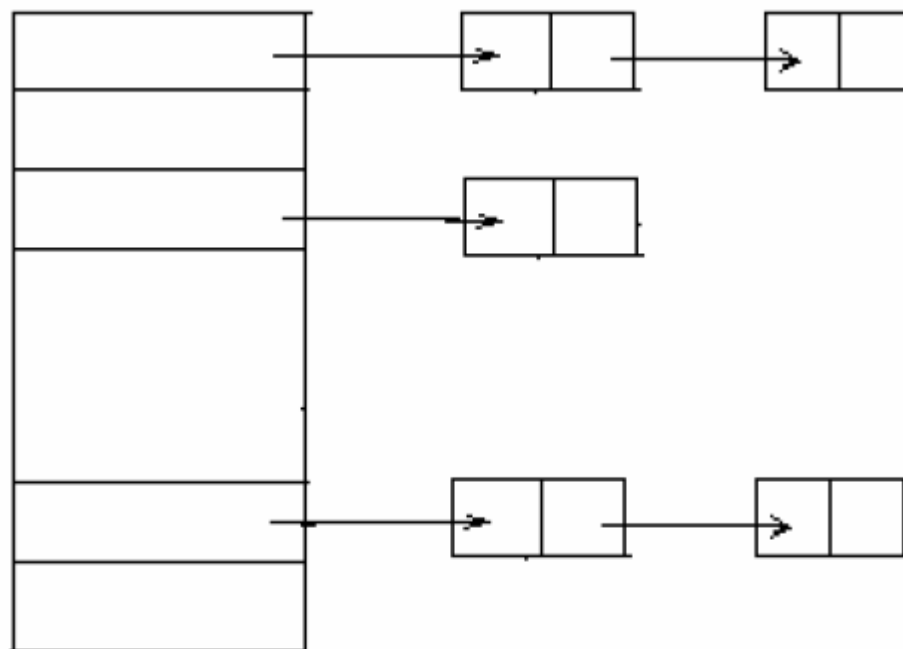
Уколико програмер дефинише сопствену класу, он је одговоран за имплементацију сопственог `hashCode` метода. Имплементација мора бити сагласна са методом `equals`. Ако важи `a.equals(b)`, онда `a` и `b` морају имати исти хеш-код.

Битно је да хеш код буде брзо израчунат и да то израчунавање зависи од објекта ког треба хеширати, а не од осталих објеката у табели.



Хеш скуп (3)

Отворена хеш табела је обично реализована као низ повезаних листи. За проналажење места објекта у хеш табели, израчунава се хеш-код и дели се по модулу са димензијом низа. Резултат је индекс члана у низу тј. индекс повезане листе која садржи дати елементат.





Хеш скуп (4)

Као што видимо, хеш табела је имплементирана у Јави као низ једноструко повезаних листи. Да би се одредила позиција објекта у табели, израчунава се његов хеш-код и нађе остатак при дељењу са укупним бројем листи. Тако добијени број је индекс листе који садржи дати елемент.

Понекад нема других елемената у тој листи и елемент се једноставно смести на своје место.

Наравно, неизбежно је да се понекад деси да има елемената и тада долази до тзв. колизије. У том случају, нови објекат се пореди са свим објектима из листе како би се видело да ли је већ присутан. Ако хеш-кодови имају разумну случајну дистрибуцију и број листи довољно велик, требало би да буде потребно свега неколико поређења.



Хеш скуп (5)

За већу контролу над перформансама хеш табеле, може са задати иницијални број листи које се користе за прикупљање објеката са идентичним хеш-кодом.

Ако се превише објеката убаци у хеш-табелу, број колизија расте, а перформансе опадају.

Обично се број листи се поставља на нешто између 75% и 150% очекиваног броја елемената. Стандардна библиотека за број листи користи степене двојке, подразумевано 16. и свака вредност која се зада за број листи аутоматски бива заокружена на следећи степен двојке.

Ако се хеш-табела препуни, неопходно је да буде рехеширана. Да би се табела рехеширала, неопходно је да се креира табела са већим бројем листи, сви елементи убаце у нову табелу и стара табела одбаци. Тзв. „load factor“ одређује када се табела рехешира.



Хеш скуп (6)

Хеш табела се користи за имплементацију неколико важних структура података. Најједноставнија међу њима је скуп. То је колекција елемената без понављања. ЈДК садржи класу `HashSet` која имплементира скуп базиран на хеш табели. `HashSet` се користи једино кад редослед елемената у колекцији није битан. Метод `add` увек прво проверава да ли се елемент налази у скупуви тек ако га нема, убацује га.

Метод `contains` је предефинисан да брзо проверава да ли се елемент већ налази у скупу - проверавају се елементи у једној повезаној листи хеш табеле, а не сви елементи скупа.

Хеш итератор редом приступа свим повезаним листама у хеш табели. Пошто хеширање раштрка елементе по табели, приступа им се привидно случајним редом.



Хеш скуп (7)

java.util.HashSet<E>

- **HashSet()**
креира празан каталог.
- **HashSet(Collection<? extends E> elements)**
креира каталог и у њега додаје све елементе из колекције.
- **HashSet(int initialCapacity)**
креира празан каталог одређеног капацитета.
- **HashSet(int initialCapacity, float loadFactor)**
креира празан каталог одређеног капацитета и датог фактора пуњења (број између 0.0 1.0 који одредјује проценат при поновном хеширању).

java.lang.Object

- **int hashCode()**
враће хеш-код за објекат `this`. Хеш-код може бити цео, позитиван и негативан. Дефиниција `equals` и `hashCode` морају бити сагласни: ако `x.equals(y)` враће `true`, онда `x.hashCode()` мора бити исти као `y.hashCode()`.



Дрвоидни скуп

Дрвоидни скупови су слични хеш скуповима, али уз једну битну разлику. Дрвоидни скуп је сортирана колекција, што значи да се могу додавати елементе у колекцију произвољним редоследом а да се при пролазу кроз колекцију елементи аутоматски обилазе у сортираном поретку.

Пример. Претпоставимо да се додају нике у дрвоидни скуп и потом се приказују елементи који су додати у колекцију.

```
SortedSet<String> sorter = new TreeSet<String>();  
sorter.add("Bob");  
sorter.add("Amy");  
sorter.add("Carl");  
for (String s : sorter)  
    System.println(s);
```

Као што се може очекивати, приказани елементи су сортирани:
Amy Bob Carl.



Дрвоидни скуп (2)

Као што име класе говори, сортирање се извршава по принципу дрволике структуре података. Садашња имплементација користи црвено-црно дрво. Сваки пут када се елемент дода у дрво, он се поставља на одговарајућу позицију дрвета. Стога, итератори увек посећују елементе у сортираном поретку.

Како дрвоидни скуп „зна“ начин на који елементи треба да буду сортирани? Подразумевано, се претпоставља да се убацују елементи класе која имплементира интерфејс `Comparable`, па је метод `compareTo` тог интерфејса задужен да упореди елементе приликом сортирања.

Ако убацујемо сопствене објекте, тада морамо сами дефинисати поредак имплементирањем интерфејса `Comparable`.

Ипак, коришћење интерфејса `Comparable` за дефинисање поретка има очигледна ограничења.



Дрвоидни скуп (3)

Шта се може предузети ако је потребно сортирати артикле по броју артикла у једној колекцији, а по опису у другој? Надаље, шта се може предузети ако је потребно сортирати објекте класе чији креатор није имплементирао интерфејс `Comparable`?

Тада се наложи дрвоидном скупу да користи другачији метод за поређење тако што се конструктору класе `TreeSet` проследи објекат која имплементира интерфејс `Comparator`.

Додавање елемената дрвету је спорије од додавања хеш табели, али је много брже од постављања елемената на право место у низу или повезаној листи. Ако дрво садржи n елемената, тада је у просеку потребно $\log_2 n$ поређења да би се нашла права позиција за нови елемент.

Ако дрво садржи 1000 елемената, додавање новог захтева око 10 поређења - много више него код додавања елемента у хеш скуп.



Дрвоидни скуп (4)

Поставља се питање да ли треба увек да се користи дрво уместо хеш скупа?

Наиме, додавање елемената не изгледа да захтева много времена, а елементи се аутоматски сортирају. Одговор зависи од података који се смештају у колекцију. Ако нису потребни сортирани подаци, нема разлога да се троши време на сувишно сортирање. Много важније, неке податке је веома тешко сортирати.



Дрвоидни скуп (5)

java.lang.Comparable<T>

- `int compareTo(T other)`
поређи текући и објекат `other` и враћа негативну вредност ако текући објекат долази испред `other`, 0 ако су идентични у сортираном поретку, а позитивну вредност иначе.

java.util.Comparator<T>

- `int compare(T a, T b)`
поређи два објекта и враћа негативну вредноста ако `a` долази испред `b`, 0 ако су идентични у сортираном поретку, а позитивну вредност иначе.

java.util.SortedSet<E>

- `Comparator<? super E> comparator()`
враћа компаратор који се користи за сортирање елемената или `null` ако се елементи пореде методом `compareTo()` интерфејса `Comparable`.
- `E first()`
- `E last()`
враћа најмањи или највећи елемент сортираног скупа.



Дрвоидни скуп (6)

java.util.NavigableSet<E>

- `E higher(E value)`
враћа најмањи елемент $> \text{value}$ или највећи елемент $< \text{value}$ или `null` ако такав елемент не постоји.
- `E lower(E value)`
враћа најмањи елемент $> \text{value}$ или највећи елемент $< \text{value}$ или `null` ако такав елемент не постоји.
- `E ceiling(E value)`
- `E floor(E value)`
враћа најмањи елемент $\geq \text{value}$ или највећи елемент $\leq \text{value}$ или `null` ако такав елемент не постоји.
- `E pollFirst()`
- `E pollLast()`
уклања и враћа најмањи или највећи елемент скупа или `null` ако је скуп празан.
- `Iterator<E> descendingIterator()`
- враћа итератор који обилази скуп у опадајућем смеру.



Дрвоидни скуп (7)

java.util.TreeSet<E>

- **TreeSet()**
Конструира дрво за чување Comparable објеката.
- **TreeSet(Comparator<? Super E> c)**
Конструира дрво и користи дати компаратор за сортирање елемената.
- **TreeSet(SortedSet<? extends E> elements)**
Конструира дрво, додаје све елементе из сортиране колекције, и користи исти компаратор као и сортирани скуп прослеђен као аргумент позива метода.



Ред

Редови омогућују ефикасно додавање елемената на крај и уклањање елемената са почетка. Ред са два краја, (енг. deque), омогућује ефикасно додавање и уклањање елемената и са почетка и са краја. Додавање елемената у средину није подржано.

Java SE 6 уводи интерфејс `Deque`. Овај интерфејс имплементирају класе `ArrayDeque` и `LinkedList`, при чему обе обезбеђују колекције чија величина расте по потреби.



Ред (2)

java.util.Queue<E>

- boolean add(E element)
- boolean offer(E element)

додаје дати елемент на крај и враћа true када ред није пун. Ако је ред пун, први метод избацује `IllegalStateException`, а други враћа `false`.

- E remove()
- E poll()

уклања и враћа елемент са почетка реда када ред није празан. Ако је ред празан, први метод избацује `NoSuchElementException`, а други враћа `null`.

- E element()
- E peek()

враћа елемент са почетка реда не уклањајући га када ред није празан. Ако је ред празан, први елемент избацује `NoSuchElementException`, а други враћа `null`.



Ред (3)

`java.util.Deque<E>`

- `void addFirst(E element)`
- `void addLast(E element)`
- `boolean offerFirst(E element)`
- `boolean offerLast(E element)`

додаје дати елемент на почетак или крај када ред није пун. Ако је ред пун, прва два метода избацују `IllegalStateException`, а последња два враћају `false`.

- `E removeFirst()`
- `E removeLast()`
- `E pollFirst()`
- `E pollLast()`

уклања и враћа елемент са почетка или краја када ред није празан. Ако је ред празан, прва два метода избацују `NoSuchElementException`, а последња два враћају `null`.

- `E getFirst()` `E getLast()`
- `E peekFirst()` `E peekLast()`

враћа елемент са почетка или краја реда не уклањајући га када ред није празан. Ако је ред празан, прва два метода избацују `NoSuchElementException`, а последња два враћају `null`.



Ред (4)

java.util.ArrayDeque<E>

- `ArrayDeque()`
- `ArrayDeque(int initialCapacity)`

конструише неограничени deque иницијалног капацитета 16 или задатог иницијалног капацитета.



Ред са приоритетом

Редови са приоритетом враћају елементе у сортираном поретку мада су претходно унесени у произвољном поретку.

Прцизније, кад год се позове `remove` метод, добија се тренутно најмањи елемент у реду са приоритетом.

Ипак, ред са приоритетом не сортира све своје елементе — када се пролази кроз елементе реда са приоритетом, ти елементи нису обавезно сортирани.

Редови са приоритетом користе једну елегантну и ефикасну структуру података, која се зове гомила (енг. `heap`). То је самоорганизовано бинарно дрво, у ком операције `add` и `remove` проузрокују да најмањи елемент гравитира ка корену (корен је увек мањи од потомака), па нема потребе да се троши време на сортирање свих елемената.



Ред са приоритетом (2)

Баш као и код дрвоидних скупова, редови са приоритетом могу да чувају елементе класе која имплементира интерфејс `Comparable` или да у конструктору користе објекат типа `Comparator`.

Типично коришћење за редове са приоритетом је распоређивање послова. Сваки посао има приоритет. Послови се додају у случајном поретку. Било кад када се може започети нови посао, посао са највећим приоритетом (тј. најмањом вредношћу) се уклања из реда.

`java.util.PriorityQueue`

- `PriorityQueue()`
- `PriorityQueue(int initialCapacity )`
Конструише структуру за чување објеката који подржавају `Comparable`.
- `PriorityQueue(int initialCapacity, Comparator<? super E> c )`
Конструише дрвоидну структуру и користи прослеђени компаратор за упоређење елемената при смештају у дрво.



Каталози

Скупови су колекције уз помоћу којих се брзо проналазе постојећи елементи. Ипак, да би се пронашао неки елемент, мора да буде поштуно познат тај елемент који се тражи.

Много је чешћа ситуација у којој се поседује нека кључна информација, и треба да се на основу ње пронађете дотични елемент. Структуре података у облику **каталога** служе за ту сврху.

Каталози чувају парове кључ/вредност. Код њих се лако може пронаћи вредност ако се наведе кључ.

На пример, може се чувати табела службеника, где су кључеви ниске - службенички идентификатори, а вредности објекти типа `Employee`.



Каталози (2)

Јава библиотека подржава две главне имплементације за каталог: `HashMap` и `TreeMap`.

Обе класе имплементирају `Map` интерфејс. Хеш каталог не сортира кључеве, за разлику од дрвоидног каталога који користи тотални поредак кључева како би их организовао у претраживачко дрво.

Да ли користити хеш каталог или дрвоидни каталог? Као и са скуповима, хеширање је нешто брже, и то је бољи избор уколико не треба обезбедити кључеве у сортираном поретку.



Каталози (3)

Пример.

Како се може дефинисати хеш каталог за чување запослених:

```
Map<String,Employee> staff = new HashMap<String,Employee>();
```

```
Employee harry = new Employee("Harry Hacker");
```

```
staff.put("987-98-9996",harry);
```

Кад год се додаје објекат у каталог, мора се добро дефинисати кључ. У претходном примеру, кључ је ниска, а одговарајућа вредност је објекат типа `Employee`. Да би се могао користити објекат из каталога, мора се користити (и стога запамтити) кључ.

Пример.

Добијање објекта који се налази у каталогу на основу кључа се реализује на следећи начин:

```
String s = "987-98-9996";
```

```
e= staff.get(s);
```



Каталози (4)

Ако нема сачуване информације у каталогу за дати кључ, тада метод `get` враћа `null`.

Кључеви морају бити јединствени. Не могу се сачувати две вредности са истим кључем. Ако се позове `set` метод два пута за исти кључ, тада друга вредност замењује прву.

Метод `remove` брише елемент са датим кључем из каталога.

Метод `size` враћа број елемената у каталогу.



Каталози (5)

Каталог није самостална колекција у ЈДК-у. Међутим, могуће је у ЈДК-у добити погледе на каталог, и ти погледи су типа који имплементира интерфејс **Collection** или неки од подинтерфејса тог интерфејса.

Постоје три погледа: скуп кључева, колекција (није скуп) вредности, те скуп парова кључ/вредност.

Кључеви и парови кључ/вредност образују скуп због тога јер може бити само једна копија кључа у колекцији.

Ова три претходно побројана погледа обезбеђују методе:

`Set<K> keySet()`

`Collection<V> values()`

`Set<Map.Entry<K,V>> entrySet()`



Каталози (6)

java.util.Map<K, V>

- **V get(K key)**
враће вредност кључа, тј. враће објекат на који показује кључ, или null ако се кључ не налази у каталогу. Кључ може бити null.
- **V put(K key, V value)**
убацује у каталог пар кључ/вредност. Ако кључ већ постоји, тада нови објекат замењује стари, претходни на који је показивао кључ. Овај метод враће стару вредност за дати кључ, или null ако кључ претходно није био дефинисан. Кључ може бити null, али вредност не сме бити null.
- **void putAll(Map<? extends K, ? extends V> entries)**
убацује све елементе из спецификованог каталога entries у овај каталог.
- **boolean containsKey(Object key)**
враће true ако је кључ већ у каталогу.
- **boolean containsValue(Object value)**
враће true ако је вредност већ у каталогу.
- **Set<Map.Entry<K, V>> entrySet()**
враће скуп погледа објеката типа Map.Entry, парове кључ/вредност из каталога. Могу се избрисати елементи овог скупа и они ће бити избрисани из каталога, али се не могу додавати елементи.



Каталози (6)

java.util.Map<K, V>

- **Set<K> keySet()**
враће скуп свих кључева у каталогу. Могу се избрисати елементи овог скупа и кључеви и вредности на њима биће избрисани из каталога, али се не могу додавати елементи.
- **Collection<V> values()**
враће вредности свих вредности у каталогу. Могу се избрисати елементи овог скупа и кључеви и вредности на њима биће избрисани из каталога, али се не могу додавати елементи.

java.util.Map.Entry<K, V>

- **K getKey()**
- **V getValue()**
враћа кључ или вредност за ову величину.
- **V setValue(V newValue)**
поставља вредност у елементу каталога и враћа стару вредност.



Каталози (7)

java.util.SortedMap<K, V>

- `Comparator<? super K> comparator()`
враћа компаратор коришћен за сортирање кључева, или нулл ако су кључеви упоређени са `compareTo` методом интерфејса `Comparable`.
- `K firstKey()`
- `K lastKey()`
враћа најмањи и највећи кључ у каталогу.

java.util.HashMap<K, V>

- `HashMap()`
- `HashMap(int initialCapacity)`
- `HashMap(int initialCapacity, float loadFactor)`
конструише празну хеш мапу са наведеним капацитетом, и фактором испуњености (број између 0.0 и 1.0 који одређује када ће хеш табела бити поново хеширана у већу). Подразумевани фактор испуњености је 0.75.



Каталози (8)

java.util.TreeMap<K, V>

- `TreeMap(Comparator<? super K> c)`
креира дрвоидни каталог и користи наведени компаратор за сортирање кључева.
- `TreeMap(Map<? extends K, ? extends V> entries)`
конструира дрвоидни каталог и додаје све `entries` из каталога-аргумента.
- `TreeMap(SortedMap<? extends K, ? extends V> entries)`
конструира дрвоидни каталог, додаје све `entries` из наведеног каталога и користи наведени компаратор за сортирање кључева.



Колекције и генерици

```
void printCollection(Collection c) {  
    Iterator i = c.iterator();  
    while (i.hasNext()) {  
        System.out.println(i.next());  
    }  
}
```

Треба
направити
генерички
метод који
приказује све
елементе
колекције.

```
void printCollection(Collection<Object> c) {  
    for (Object e: c){  
        System.out.println(e);  
    }  
}
```

Ово је први,
наивни
покушај...

printCollection(stones);

Не може се превести!



Колекције и генерици (2)

Проблем је решен увођењем генеричког типа који може бити било шта, а који се назива **џокер тип** и означава симболом ?

`Collection<?>`

“колекција непознатих” је колекција чији елементи могу одговарати било ком типу - **џокер тип**

```
void printCollection(Collection<?> c) {  
    for (Object e: c){  
        System.out.println(e);  
    }  
}
```

`printCollection(stones);`

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



Колекције и генерици (3)

Проблем са коришћењем колекције и џокер типа наступа зато што се не зна тип компоненте, па преводилац не може да преведе наредбу за доделу у колекцију елемената непознатог типа.

```
String myString;  
Object myObject;  
List<?> c = new ArrayList<String>();
```

```
// c.add("hello world");
```

```
// compile error
```

```
// c.add(new Object());
```

```
// compile error
```

```
((List<String>) c).add("hello world");
```

```
((List<Object>) c).add(new Object());
```

```
// no compile error!
```

```
// String myString = c.get(0);
```

```
// compile error
```

```
myString = (String) c.get(0);
```

```
myObject = c.get(0);
```

```
myString = (String) c.get(1);
```

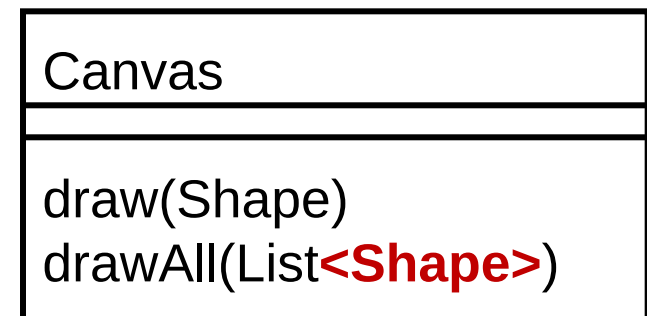
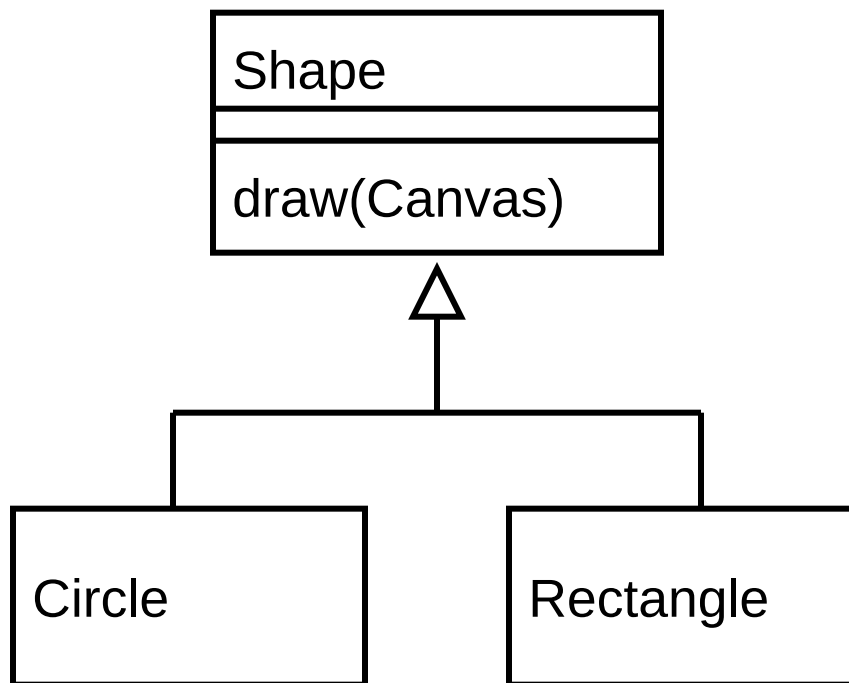
```
// run-time error!
```



Колекције и генерици (4)

Понекад је потребно направити ограничење над цокер-типом.

Пример. Треба направити апликацију која исцртава различите облике (кругове, правоугаонике итд.). Класе у апликацији би биле:



Ограничено
на `List<Shape>`
и зато није добро



Колекције и генерици (5)

Пример. Потребно је креирати метод који као аргумент прихвата колекцију са ма којом врстом облика, тј. са ма којом подкласом класе Shape.

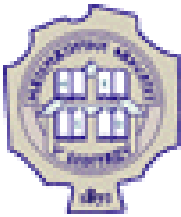
Истовремено, треба инструкисати преводилац да спречи да елементи колекције буду примерци класе која није подкласа Shape.

То се постиже на следећи начин:

```
public void drawAll(List<? extends Shape>)  
{  
    ...  
}
```

ограничени цокер

У овом случају, класа Shape је **горње ограничење** за цокер



Колекције и генерици (6)

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

```
...
```

```
public void pushAll(Collection<? extends E> collection) {  
    for (E element : collection) {  
        this.push(element);  
    }  
}
```

Сви елементи морају бити **бар** типа E (тј. типа подкласе класе E)

```
public List<E> sort(Comparator<? super E> comp) {  
    List<E> list = this.asList();  
    Collections.sort(list, comp);  
    return list;  
}
```

Метод за поређење мора захтевати да се извршава над типовима **највише** E (тј. над типовима неке од наткласа класе E)



Колекције и генерици (7)

Генерички колекцијски интерфејси имају огромну предност - потребно је имплементирати своје алгоритме само једанпут.

Пример. Размотримо једноставан алгоритам за израчунавање максималног елемента колекције. За највећи елемент низа:

```
if (a.length == 0)
    throw new NoSuchElementException();
T largest = a[0];
for (int i = 1; i < a.length; i++)
    if (largest.compareTo(a[i]) < 0)
        largest = a[i];
```

Налажење максимума низовне листе је незнатно другачије:

```
if (v.size() == 0)
    throw new NoSuchElementException();
T largest = v.get(0);
for (int i = 1; i < v.size(); i++)
    if (largest.compareTo(v.get(i)) < 0)
        largest = v.get(i);
```



Колекције и генерици (8)

Пример (наставак). У повезаној листи немамо ефикасан случајан приступ, али можемо користити итератор:

```
if (l.isEmpty())  
    throw new NoSuchElementException();  
Iterator<T> iter = l.iterator();  
T largest = iter.next();  
while(iter.hasNext())  
{  
    T next = iter.next();  
    if(largest.compareTo(next) < 0)  
        largest = next;  
}
```

Ове петље су напорне за писање и омогућују грешке. Пожељно је да се избегне понављање тестирања и имплементација мноштва метода попут:

```
static <T extends Comparable> T max(T[] a)  
static <T extends Comparable> T max(ArrayList<T> v)  
static <T extends Comparable> T max(LinkedList<T> l)
```



Колекције и генерици (9)

Пример (наставак). Ту на сцену ступају колекцијски интерфејси. Треба осмислити минимални колекцијски интерфејс који је потребан за ефикасан алгоритам.

Израчунавање максимума може се урадити просто итерирањем кроз елементе. Имплементација метода `max` тако да прихвата објекат `ма` које класе која имплементира интерфејс `Collection`:

```
public static <T extends Comparable> T max(Collection<T> c){  
    if (c.isEmpty())  
        throw new NoSuchElementException();  
    Iterator<T> iter = c.iterator();  
    T largest = iter.next();  
    while (iter.hasNext()) {  
        T next = iter.next();  
        if (largest.compareTo(next) < 0)  
            largest = next;  
    }  
    return largest;  
}
```



ЈДК колекције и генерици

Генерици су моћан концепт, који се у Јавиној библиотеци користи у алгоритмима сортирања, бинарне претраге и још неким корисним алгоритмима.

Можемо се запитати како метод `sort` сортира листу? Алгоритми за сортирање у књигама су презентовани за низове и користе директан приступ елементима. Међутим, листа је колекција, а случајан приступ у листи може бити неефикасан.

Имплементација у Јави просто прекопира све елементе у низ, сортира га користећи варијанту `merge sort` алгоритма, а затим копира сортирану секвенцу натраг у листу.

`Merge sort` алгоритам коришћен у библиотеци је за нијансу спорији од `quick sort`-а, али има једну предност: стабилан је, тј. не врши размену једнаких елемената.

Зашто нам је битан поредак једнаких елемената?



ЈДК колекције и генерици (2)

Пример. Следи уобичајени сценарио. Претпоставимо да имамо листу запослених која је већ сортирана по имену. Сада сортирамо по заради. Шта се дешава са запосленима који имају исту зараду? Када је сортирање стабилно, а алгоритам сортирања је merge sort, поредак по имену бива очуван. Другим речима, излаз је листа сортирана најпре по заради, а потом по имену.

Класа `Collections` поседује метод `shuffle` који случајно пермутује редослед елемената у листи.

Пример. Мешање карата

```
ArrayList<Karta> karte = ...;  
Collections.shuffle(karte);
```

Ако се проследи листу која не имплементира интерфејс `RandomAccess`, метод `shuffle` копира елементе у низ, промеша тај низ и ископира промешане елементе натраг у листу.



ЈДК колекције и генерици (3)

Метод `binarySearch` класе `Collections` имплементира бинарну претрагу. Наравно, колекција мора претходно бити сортирана или ће алгоритам вратити погрешан резултат.

Да би се пронашао елемент, прослеђује се колекција (која мора имплементирати интерфејс `List`) и елемент који се тражи. Ако колекција није сортирана коришћењем метода `compareTo` интерфејса `Comparable`, мора се проследити и одговарајући компаратор.

```
i = Collections.binarySearch(c, element);
```

```
i = Collections.binarySearch(c, element, comparator);
```

Повратна вредност ≥ 0 означава индекс пронађеног елемента у складу са коришћеним поређењем. Ако је повратна вредност негативна, не постоји тражени елемент у колекцији.



ЈДК колекције и генерици (4)

Међутим, повратна вредност се може користити за израчунавање позиције на коју треба уметнути `element` у колекцију како би она остала сортирана. Та позиција је:

```
insertionPoint = -i - 1;
```

Није просто `-i` јер онда би вредност 0 била двосмислена. Другим речима, операција

```
if ( i < 0)
```

```
    c.add(-i - 1, element);
```

додаје елемент на исправну позицију.

Како би имала смисла, бинарна претрага захтева случајан приступ. Метод `binarySearch` проверава да ли листа која му је дата као аргумент имплементира интерфејс `RandomAccess`, па ако то јесте случај, примењује бинарну претрагу, а у супротном користи линеарну.



ЈДК колекције и генерици (5)

java.util.Collections

- `static <T extends Comparable<? super T>> void sort(List<T> elements)`
- `static <T> void sort(List<T> elements, Comparator<? super T> c)`
сортира елементе листе користећи стабилни алгоритам. Временска сложеност алгоритма је $O(n \log n)$, где је n дужина листе.
- `static void shuffle(List<?> elements)`
- `static void shuffle(List<?> elements, Random r)`
случајно меша елементе листе. Временска сложеност алгоритма је $O(n a(n))$, где је n дужина листе, док је $a(n)$ просечно време приступа елементу.
- `static <T> Comparator<T> reverseOrder()`
враћа компаратор који сортира елементе у обрнутом поретку од поретка одређеног методом `compareTo()` интерфејса `Comparable`.
- `static <T> Comparator<T> reverseOrder(Comparator<T> comp)`
враћа компаратор који сортира елементе у обрнутом поретку од поретка одређеног компаратором `comp`.



ЈДК колекције и генерици (6)

java.util.Collections

- static <T extends Comparable<? super T>>
int binarySearch(List<T> elements, T key)
- static <T> int binarySearch(List<T> elements, T key, Comparator<? super T> c)
тражи кључ у сортираној листи, користећи линеарну претрагу ако elements не имплементира RandomAccess интерфејс, а бинарну у супротном.
Временска сложеност алгоритма је $O(a(n) \log n)$, где је n дужина листе, док је $a(n)$ просечно време приступа елементу. Методи враћају или индекс кључа у листи или негативну вредност i ако кључ није присутан у листи. У том случају, кључ треба убацити у листу на позицију $-i - 1$ како би листа остала сортирана.
- static <T extends Comparable<? super T>> T min(Collection<T> elements)
- static <T extends Comparable<? super T>> T max(Collection<T> elements)
- static <T> min(Collection<T> elements, Comparator<? super T> c)
- static <T> max(Collection<T> elements, Comparator<? super T> c)
враћа најмањи или највећи елемент у колекцији (ограничења за типске параметре су упрошћена због једноставности).



ЈДК колекције и генерици (7)

java.util.Collections

- `static <T> void copy(List<? super T> to, List<T> from)`
копира све елементе из изворне листе на исте позиције у одредишној листи. Одредишна листа мора бити дуга бар колико изворишна.
- `static <T> void fill(List<? super T> l, T value)`
попуњава све позиције у листи истом вредношћу.
- `static <T> boolean addAll(Collection<? super T> c, T... values)`
додаје све вредности у дату колекцију и враћа `true` ако је тиме колекција промењена.
- `static <T> boolean replaceAll(List<T> l, T oldValue, T newValue)`
заменаје све елементе једнаке старој вредности новом вредношћу.
- `static int indexOfSubList(List<?> l, List<?> s)`
- `static int lastIndexOfSubList(List<?> l, List<?> s)`
враћа индекс прве или последње подлисте од `l` једнаке са `s` или `-1` ако нема такве подлисте у `l`.
- `static void swap(List<?> l, int i, int j)`
разменаје елементе на датим позицијама.



ЈДК колекције и генерици (8)

java.util.Collections

- `static void reverse(List<?> l)`
обрће редослед елемената у листи. Временска сложеност алгоритма је $O(n)$, где је n дужина листе.
- `static void rotate(List<?> l, int d)`
ротира елементе листе, померајући елемент са индексом i на позицију $(i + d) \% l.size()$. Временска сложеност алгоритма је $O(n)$, где је n дужина листе.
- `static int frequency(Collection<?> c, Object o)`
враћа број елемената листе једнаких датом објекту.
- `boolean disjoint(Collection<?> c1, Collection<?> c2)`
враћа `true` ако колекције немају заједничких елемената.



Захвалница

Велики део материјала који је укључен у ову презентацију је преузет из презентације коју је раније (у време када је он држао курс Објектно орјентисано програмирање) направио проф. др Душан Тошић.

Хвала проф. Тошићу што се сагласио са укључивањем тог материјала у садашњу презентацију, као и на помоћи коју ми је пружио током конципирања и реализације курса.

Надаље, један део материјала је преузет од колегинице Марије Милановић.

Хвала Марији Милановић на помоћи у реализацији ове презентације.