Collections

Adam Krechowicz

Generic classes

```
1  public class Simple {
2      Object o;
3
4      public void setO(Object o){
5          this.o = o;
6      }
7
8      public Object getO(){
9          return this.o;
10     }
11 }
```

Universal classes

```
1   public  static  void main(String[] args){
2           Simple s = new Simple();
3           s.set0("Hello");
4           String str = (String)s.get0();
5           System.out.println(str);
6
7           s.set0(123);
8           int i = (int)s.get0();
9           System.out.println(i);
10
11          String abc = (String)s.get0();
12          System.out.println(abc);
13      }
```

Generic classes

```
1  public class Generics<T> {
2      T t;
3      public void setT(T t){
4          this.t = t;
5      }
6      public T getT(){
7          return this.t;
8      }
9      public static void main(String[] args){
10         Generics<Integer> g = new Generics<>();
11         g.setT(3);
12         int i = g.getT();
13         System.out.println(i);
14
15         Generics<String> h = new Generics<>();
16         h.setT("Hello");
17         String s = h.getT();
18         System.out.println(s);
19     }
20 }
```

Generic methods

```
1  public class Generics {
2      public <T> boolean test(T t){
3          return this.t.equals(t);
4      }
5
6      public static void main(String[] args){
7          Generics<String> h = new Generics<>();
8          h.setT("Hello");
9          String s = h.getT();
10         System.out.println(s);
11
12         System.out.println(h.<Float>test(3.5f));
13         System.out.println(h.<String>test("Hello"));
14     }
15
16 }
```

Constraints

```
1  public class Generics{
2      public <T extends Number> void numerki(T t){
3          System.out.println(t);
4      }
5
6      public static void main(String[] args){
7          Generics<String> h = new Generics<>();
8
9          h.<Integer>numerki(123);
10         h.<Float>numerki(3.5f);
11         //h.<String>numerki("Hello");
12     }
13
14 }
```

Properties of generic elements

- We can not use primitive types only classes
- We can not create object based on generic parameter
- We can not create static field based on generic type
- We can not cast or test instance of based on generic type
- We can not create array of generic types

Wildcard

```
1  public class Wildcard {
2
3      public boolean check(Generics<?> g){
4          return (g.getT() != null);
5
6      }
7
8      public static void main(String[] args){
9          Wildcard w = new Wildcard();
10         Generics<Integer> g = new Generics<>();
11         System.out.println(w.check(g));
12     }
13
14 }
```

Collections

Collections

Collections allow to easily store and manage set of objects

Interfaces:

- Collection
- Set
- List
- Queue
- Deque
- Map

Collection

Collection

All collections implements this interface

Metody:

- add() – add new element to collection
- remove() – delete element from colection
- isEmpty() – test if collection contains some elements
- clear() – clear the content of collection
- size() – number of elements inside collection
- contains() – test if collection contains specified element
- toArray() – creates array from collection

Set

Set

Set is a collection that allows to store distinct objects.

Implementation:

- HashSet – it use hash function to store elements, order of elements is not preserved
- TreeSet – it use binary tree to store elements, elements are sorted
- LinkedHashSet – it use double link list, preserve the order of inserting

Set

```
1   Set<Integer> set = new HashSet<>();
2   set.add(3);
3   set.add(5);
4   set.add(7);
5   System.out.println(set.toString());
6
7   set.add(3);
8   System.out.println(set.toString());
9
10  set.remove(5);
11  System.out.println(set.toString());
12
13  System.out.println(set.size());
14
15  System.out.println(set.contains(7));
16
17  set.clear();
18  System.out.println(set.toString());
19  System.out.println(set.isEmpty());
```

List

List

List groups elements and preserve the order created during object inserting. May contain duplicates.

Implementation:

- ArrayList – Use array to store elements
- LinkedList – Use linked list to store elements
- Vector – similar to ArrayList, use different optimisation
- Stack – Stack implemented on Vector class

List

```
1         List<Integer> list = new ArrayList<>();
2         list.add(4);
3         list.add(5);
4         list.add(6);
5         System.out.println(list.toString());
6         list.add(4);
7         System.out.println(list.toString());
8
9         System.out.println(list.indexOf(4));
10        System.out.println(list.lastIndexOf(4));
11        System.out.println(list.isEmpty());
12        list.remove(0);
13        System.out.println(list.toString());
```

Queue

Queue

Queue preserve the order of elements. Elements are inserted at one end and are acquired on other.

Implementations:

- ArrayQueue – use array to store elements
- LinkedList – may also be used as queue
- PriorityQueue – inserts elements based on priority

Queue

```
1         Queue<Integer> queue = new PriorityQueue<>();
2         queue.add(5);
3         queue.add(4);
4         queue.add(7);
5         System.out.println(queue.peek());
6         System.out.println(queue.poll());
7         System.out.println(queue.remove());
8         System.out.println(queue.remove());
9         System.out.println(queue.poll());
10        System.out.println(queue.remove());
```

Double ended queue

Similar to queue but allows to insert and remove from both ends

Implementations:

- ArrayDeque – use array to store elements
- LinkedList – may also be used as double ended queue

Double ended queue

```
1        Deque<String> deque = new ArrayDeque<>();
2        deque.addFirst("World");
3        deque.addFirst("Hello ");
4        deque.addLast("!");
5        System.out.println(deque.getFirst());
6        System.out.println(deque.getLast());
7        System.out.println(deque.pollFirst());
8        System.out.println(deque.pollFirst());
9        System.out.println(deque.pollFirst());
```

Map

Map

Maps allows to store elements in a form key-value

Implementations:

- HashMap – use hash function to store elements
- TreeMap – use binary tree to store elements
- LinkedHashMap – use double ended queue to store elements

Map

```
1        Map<Integer, String> map = new HashMap<>();
2        map.put(1, "jeden");
3        map.put(2, "dwa");
4        map.put(3, "trzy");
5        System.out.println(map.get(3));
6        System.out.println(map.containsKey(4));
```

Double linked lists

- preserve the order of elements
- elements can easily be added in any place
- searching needs to be performed by traversing
- to acquire the element we need to have link from front or from back
- we can easily remove element O(1)

Arrays

- ArrayList, ArrayQueue, ArrayDeque
- preserve the order of elements
- operations are aggregated (usually fast but from time to time may be very slow)
- we can easily access any element
- removing element may be very slow (depending on which element is removed)
- searching array is faster than searching list
- we can define the expected size be using initialCapacity parameter

Hashing

- stores elements in buckets
- do not preserve the order elements
- uses hashCode() and equals() methods
- we can define the expected size be using initialCapacity parameter

hashCode():

- we need to define implementation of hashCode in such a way that element are evenly distributed on buckets
- method needs to perform fast
- do not need to return unique values
- should take into consideration all valuable fields

Hashing

```
1  class HashElement{
2      int i;
3      public HashElement(int i){
4          this.i = i;
5      }
6      @Override
7      public boolean equals(Object o){
8          if (o instanceof HashElement){
9              System.out.println("Equals "+this.i+" and "+((HashElement)o).i);
10             return this.i == ((HashElement)o).i;
11         } else return false;
12     }
13     @Override
14     public int hashCode(){
15         System.out.println("HashCode for "+ this.i);
16         return this.i % 5;
17     }
18 }
```

Hashing

```
1     public static void main(String[] args){
2         HashSet<HashElement> hs = new HashSet<>();
3         System.out.println("Adding 5");
4         hs.add(new HashElement(5));
5         System.out.println("Adding 6");
6         hs.add(new HashElement(6));
7         System.out.println("Adding 10");
8         hs.add(new HashElement(10));
9         System.out.println("Adding 15");
10        hs.add(new HashElement(15));
11    }
```

Hashing

```
1  Adding 5
2  HashCode for 5
3  Adding 6
4  HashCode for 6
5  Adding 10
6  HashCode for 10
7  Equals 10 and 5
8  Adding 15
9  HashCode for 15
10 Equals 15 and 5
11 Equals 15 and 10
```

Binary tree

- stores elements in form of binary tree
- do not preserve order of elements
- objects inside this collections needs to be comparable
- objects needs to implements Comparable interface

Binary tree

```
1  class SimpleElement{
2
3  }
4
5  public class TreeTest {
6
7      public static void main(String[] args){
8          TreeSet<SimpleElement> ts = new TreeSet<>();
9          ts.add(new SimpleElement());
10     }
11
12 }
```

Binary tree

```
1  class SimpleElement{
2
3  }
4
5  public class TreeTest {
6
7      public static void main(String[] args){
8          TreeSet<SimpleElement> ts = new TreeSet<>();
9          ts.add(new SimpleElement());
10     }
11
12 }
```

Exception in thread "main"java.lang.ClassCastException: com.wyklad.collections.SimpleElement cannot be cast to java.lang.Comparable

Binary tree

```
1  class ComparableElement implements Comparable<ComparableElement>{
2
3      @Override
4      public int compareTo(ComparableElement o) {
5          return 1;
6      }
7
8  }
9
10 public class TreeTest {
11
12     public static void main(String[] args){
13         TreeSet<ComparableElement> ts = new TreeSet<>();
14         ts.add(new ComparableElement());
15     }
16
17 }
```

Binary tree

```
1  class ComparableElement implements Comparable<ComparableElement>{
2
3      int i;
4
5      ComparableElement(int i){
6          this.i = i;
7      }
8
9      @Override
10     public int compareTo(ComparableElement o) {
11         System.out.println("Compare "+this.i+" and "+o.i);
12         return this.i - o.i;
13     }
14
15 }
```

Binary tree

```
1   public static void main(String[] args){
2       TreeSet<ComparableElement> ts = new TreeSet<>();
3       System.out.println("Adding 5");
4       ts.add(new ComparableElement(5));
5       System.out.println("Adding 6");
6       ts.add(new ComparableElement(6));
7       System.out.println("Adding 7");
8       ts.add(new ComparableElement(7));
9       System.out.println("Adding 8");
10      ts.add(new ComparableElement(8));
11      System.out.println("Adding 4");
12      ts.add(new ComparableElement(4));
13  }
```

Binary tree

```
1  Adding 5
2  Compare 5 and 5
3  Adding 6
4  Compare 6 and 5
5  Adding 7
6  Compare 7 and 5
7  Compare 7 and 6
8  Adding 8
9  Compare 8 and 6
10 Compare 8 and 7
11 Adding 4
12 Compare 4 and 6
13 Compare 4 and 5
```

Iterators

Iterators

Iterators allows to traverse the content of collections no matter how it was implemented.

```
1        Integer[] tablica = {1,2,3,4,5,6};
2        List<Integer> list = Arrays.asList(tablica);
3        Set<Integer> set = new HashSet(Arrays.asList(tablica));
4        Queue<Integer> queue = new LinkedList(Arrays.asList(tablica));
5        Deque<Integer> deque = new LinkedList(Arrays.asList(tablica));
```

Iterators

```
1         Iterator<Integer> i;
2         i = list.iterator();
3         while (i.hasNext()){
4             System.out.println(i.next());
5         }
6
7         for (Iterator<Integer> iter = set.iterator(); iter.hasNext();){
8             System.out.println(iter.next());
9         }
10
11        for(Integer j: queue){
12            System.out.println(j);
13        }
14
15        deque.stream().forEach((j) -> {
16            System.out.println(j);
17        });
```

Risk of using interators

```
1  public class RemovingAndIterators {
2
3      public static void main(String[] args){
4          Integer[] tablica = {1,2,3,4,5,6};
5          List<Integer> list = new ArrayList(Arrays.asList(tablica));
6          //UnsupportedOperationException
7          Iterator<Integer> i = list.iterator();
8          while(i.hasNext()){
9              int j = i.next();
10             System.out.println(j);
11             // list .remove(j);
12             i.remove();
13         }
14     }
15 }
```

Risk of using iterators

```
1  public class Risk {
2      public static void main(String[] args){
3          Integer[] tab = {1,2,3,4,5};
4          List<Integer> list = new LinkedList(Arrays.asList(tab));
5          Iterator<Integer> i = list.iterator();
6
7          i.next();
8          list.add(55);
9          i.next();
10     }
11 }
```

Operations on collections

```
1  class Element implements Comparable<Element>{
2
3      int i;
4
5      public Element(int i){
6          this.i = i;
7      }
8
9      @Override
10     public int compareTo(Element o) {
11         return this.i - o.i;
12     }
13
14     @Override
15     public String toString(){
16         return ""+i;
17     }
18
19 }
```

Operations on collections

```
1         Element e1 = new Element(5);
2         Element e2 = new Element(3);
3         Element e3 = new Element(8);
4         Element e4 = new Element(2);
5         List<Element> list = new LinkedList<>();
6         list.add(e1); list.add(e2);
7         list.add(e3); list.add(e4);
8
9         System.out.println(list);
```

Operations on collections

```
1         System.out.println(Collections.max(list));
2         System.out.println(Collections.min(list));
3
4         Element e = Collections.max(list, new Comparator<Element>() {
5
6             @Override
7             public int compare(Element o1, Element o2) {
8                 return o2.i - o1.i;
9             }
10        });
11        System.out.println(e);
```

Operations on collections

```
1        Collections.addAll(list, new Element(4), new Element(6), new Element(0), ↩
       null);
2        System.out.println(list);
3
4        System.out.println(Collections.frequency(list, e1));
5        System.out.println(Collections.frequency(list, null));
6
7        Collections.reverse(list);
8        System.out.println(list);
```

Operations on collections

```
1         Collections.shuffle(list);
2         System.out.println(list);
3
4         Collections.rotate(list, 1);
5         System.out.println(list);
6
7         Collections.swap(list, 0, 3);
8         System.out.println(list);
9
10        Collections.fill(list, new Element(0));
11        System.out.println(list);
```

THE END!

Additional reading:

- https://docs.oracle.com/javase/tutorial/java/generics/index.html
- https://docs.oracle.com/javase/tutorial/collections/index.htmll
- Thinking in Java chapters by Bruce Eckel:
  - Generics
  - Containers in depth