Object Oriented Programming in Java

9: Collections

Licence

Arrays

- In previous examples we have stored data in arrays
  - Arrays of *Items*, *Perishables*, arrays of some other types, …
  - We can also have arrays of a parametrized type although we need some workaround to create such arrays
- Some of the drawbacks
  - Arrays are not resizable
  - We must specify arrays size during allocation
    - No too small, not to big …
- How to solve the problem when array is full
  - Create a new, larger one, copy current elements to new array and change the reference (old arrays would be removed by garbage collector)
    - Copy one by one or use built-in method *Array.copy* (copies faster because it copies whole blocks of memory at once)

ArrayList

- Java already contains implementation for “resizable” array that does exactly what we wants – *ArrayList*
  - We can set initial capacity, but if the number of elements raises beyond capacity, grow is ensured by creating new larger array
    - encapsulated inside implementation

09_Collections/swu/oopj/collections/ArrayListMain.java

```
package swu.oopj.collections;
import java.util.ArrayList;
public class ArrayListMain {
    public static void main(String[] args) {
        ArrayList<Integer> arr = new ArrayList<>(10); //init.capacity
        System.out.println("Size: " + arr.size()); // 0
        for(int i=0; i<1000; i++)
                arr.add(2*i);
        System.out.println("Size: " + arr.size());  //1000
        System.out.println("ELement at pos. 750: " + arr.get(750));
    }
}
```

Useful methods in *ArrayList*

- Variable is declared as *ArrayList<E>* and not as *E[]*, thus square brackets cannot be used to get element at specific position
  - Instead get method is used: *E get(int index)*
  - E is parameter type (Integer was argument type in previous example)
- Other useful methods
  - *add(E element)*
    - Appends the specified element to the end
  - *add(int index, E element)*
    - Inserts the specified element at the specified position
  - *E set(int index, E element)*
    - Replaces the element at the specified position with the specified element and return the old element
  - E remove(int index)
    - Removes the element at the specified position (and return it)

Linked lists

- Arrays (and *ArrayList*) have some obvious benefits, but performance my suffer if we
  - must insert or remove an element
    - all elements after insert or remove position must be shifted
  - capacity must be "increased"
- Linked lists consist of nodes where each node contain data and reference (pointer) to the next element
  - It may also contain reference to previous element (double linked list)
- Linked lists contains reference to the first (and the last) element in the list
  - Insert and delete is easy

```
class LinkedList<E> {
   Node<E> first;
   Node<E> last;
   ...
}
class Node<E> {
   E item;
   Node<E> next;
   Node<E> prev;
   ...
}
```

Do not reinvent the wheel – class *LinkedList*

- Java already contains *LinkedList* and its code is optimized

```
public E get(int index) {
       checkElementIndex(index);
       return node(index).item;
}
Node<E> node(int index) {
    if (index < (size >> 1)) {
        Node<E> x = first;
        for (int i = 0; i < index; i++)
            x = x.next;
        return x;
    } else {
        Node<E> x = last;
        for (int i = size - 1; i > index; i--)
            x = x.prev;
        return x;
    }
}
```

LinkedList

- *LinkedList* has many methods common with *ArrayList*
  - They implement the same interface: *List* (more details later)

```
package swu.oopj.collections;
import java.util.LinkedList;

public class LinkedListMain {
    public static void main(String[] args) {
        LinkedList<Integer> list = new LinkedList<>();
        System.out.println("Size: " + list.size());
        for(int i=0; i<1000; i++)
               list.add(2*i);
        System.out.println("Size: " + list.size());
        System.out.println("ELement at pos. 750: " + list.get(750));
    }
}
```

09_Collections/swu/oopj/collections/LinkedListMain.java

Iterating through a *LinkedList* or *ArrayList*

- In both case we can use for-loop and *get* method, however getting the i-th element of linked list is slow (especially if compared to the fact that we goes sequentially through the list)
- Like for arrays, we can use for-each variant of the for loop
  - When and why is this possible is discussed later

09_Collections/swu/oopj/collections/ListIterate.java

```
package swu.oopj.collections;
import java.util.LinkedList;
import java.util.List;
public class ListIterate {
	public static void main(String[] args) {
		List<Integer> list = new LinkedList<>();
		for(int i=0; i<10; i++)
			list.add(2*i);
		for(Integer i : list)
			System.out.println(i);
...
```

Fixed size lists and unmodifiable lists

- Java supports creation of fixed size lists using *Arrays.asList* or unmodifiable lists using *List.of* and *List.copyOf*
  - The first two are methods with variable number of elements

09_Collections/swu/oopj/collections/UnmodifiableList.java

```
import java.util.Arrays;
import java.util.List;
public class UnmodifiableList {
      public static void main(String[] args) {
            List<Integer> list = List.of(1,  2,  3);
            //list.add(4); //throws an Exception
            //list.set(0, 5); //throws and Exception
            System.out.println(list);

            list = Arrays.asList(1, 2, 3);
            //list.add(4); //throws an Exception
            list.set(0, 5);
            System.out.println(list);
...
```

Java Collections Framework (1)

- *List*, *ArrayList*, and *LinkedList* are all part od *Java Collection Framework*

- Collection (container)
  - Object that groups multiple elements into a single unit.
  - Collections are used to store, retrieve, manipulate, and communicate aggregate data.

- Collections framework
  - Unified architecture for representing and manipulating collections

Java Collections Framework (2)

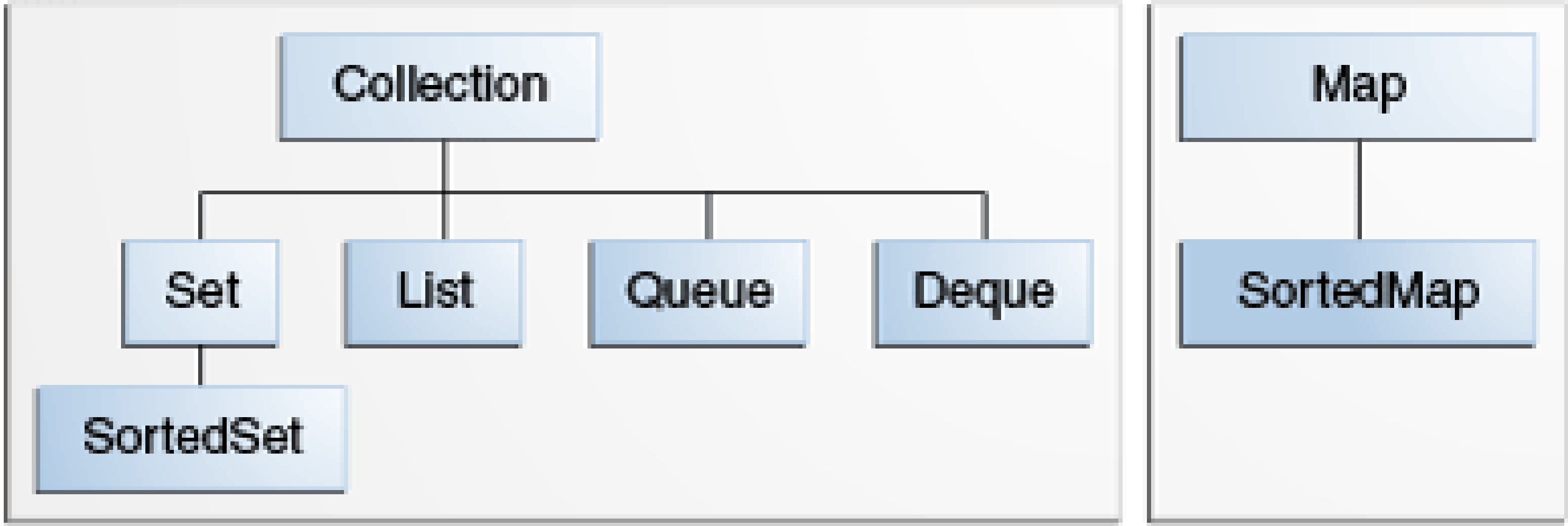
- Interfaces:
  - abstract data types that represent collections
  - allows collections to be manipulated independently of the details of their representation
- Implementations
  - concrete implementations of the collection interfaces
- Algorithms:
  - methods that perform useful computations (e.g. searching, sorting) on objects that implement collection interfaces.
  - algorithms are polymorphic: the same method can be used on many different implementations of the appropriate collection interface

Advantages of Java Collection Framework

- Do not reinvent the wheel and foster (good) software reuse
  - Reduces programming effort
  - Algorithms depends on interfaces - allows easy switch of collection implementations.
  - Increases program speed and quality
    - see e.g. *get* method in *LinkedList*
- Using standard collections reduces effort to learn, use or design new APIs
  - Allows interoperability among unrelated APIs
  - New data structures and algorithms that conform to the standard collection interfaces are by nature reusable.

Core Collection interfaces

- As shown in https://docs.oracle.com/javase/tutorial/collections/interfaces/index.html there are two core collection hierarchies
  - one derived from interface *Collection*
  - another from interface *Map*
- Note, figure does not show complete hierarchy, just main interfaces



Collection interface

- Models collections of objects using maximum generality
  - Does not specifies anything about order, duplicates, null elements
    - it is left to other interfaces that extends this interface (i.e. List for ordered collections)
- Interfaces cannot enact the existence of specific constructors but is a common practice that concrete implementations in Java Collection Framework should have at least:
  - constructor without arguments
    - creates an empty collection
  - constructor that takes a Collection argument (*conversion constructor*)
    - initializes the new collection to contain all elements of the specified collection
    - allows conversion of collection's type

Optional and *default* methods

- Java Collection Framework contains many useful *default* methods
  - Used to extend interfaces without breaking compatibility with an old code
  - Provides (in most cases satisfying) default code for the methods
- Some implementations creates e.g. immutable, fixed-size, … collections that does not support all operations, but this methods must be implemented
  - If an unsupported operation is invoked, a collection throws an *UnsupportedOperationException*.
  - In documentation these methods are marked as optional
  - Implementations are responsible for documenting which of the optional operations they support.
  - Note: optional ≠ default !

Methods defined by *Collection* interface

- List od methods that could be useful or it would be used later

```
public interface Collection<E> extends Iterable<E> {
       int size();
       boolean isEmpty();
       boolean contains(Object element);
       boolean add(E element); //optional
       boolean remove(Object element); //optional
       Iterator<E> iterator();
       boolean containsAll(Collection<?> c);
       boolean addAll(Collection<? extends E> c); //optional
       boolean removeAll(Collection<?> c); //optional
       boolean retainAll(Collection<?> c); //optional
       void clear(); //optional
       Object[] toArray();
       <T> T[] toArray(T[] a);
       default boolean removeIf(Predicate<? super E> filter) {...}
       default Stream<E> stream() {...}
```

A note about some methods in *Collection*

- A question could arise why some methods have unusual signatures like:

  boolean remove(Object element) instead of
  boolean remove(E element)

  - Implementations use static method Objects.equals to do equality check (and uses equals from *Object*)

  boolean addAll(Collection<? extends E> c) instead of
  boolean addAll(Collection<E> c);

  - Suppose that we have a class *Food* that extends *Item*. This allows us to add all elements from Collection<Food> to Collection<Item>

- Some other examples would be described later (i.e. super and Predicate in *removeIf* method)

Interface *Iterable* and iterators

- Interface *Collection* extends interface *Iterable* that enables use of for-each construct for traversing through a collection

```
for(Type item : collection)
        do something with item (but do not change collection!)
```

- It is a simplified way to use general concept of iterator (an object that enables you to traverse through a collection and to remove elements from the collection selectively)
  - Interface *Iterable<T>* defines method *Iterator<T> iterator()*
  - Interface *Iterator* defines three methods: *hasNext, next*, and (optional) *remove*

```
Iterator<SomeType> it = collection.iterator();
while(it.hasNext()) {
        Type item = it.next();
        do something with item
}
```

Interface *List* extends *Collection* interface

- *List* “is a” *Collection*
  - Interface *List* extends interface *Collection* with methods for ordering elements in collection

```
public interface List<E> extends Collection<E> {
   E get(int index);
   E set(int index, E element);    //optional
   boolean add(E element);         //optional
   void add(int index, E element); //optional
   E remove(int index);            //optional
   boolean addAll(int index, Collection<? extends E> c);  //optional
   int indexOf(Object o);
   int lastIndexOf(Object o);
   ListIterator<E> listIterator();
   ListIterator<E> listIterator(int index);
   List<E> subList(int from, int to);
```

Access based on element position

Searching

Enables iterating in both directions

Example #1

09_Collections/swu/oopj/collections/example1/*.java

- Add integers from the standard input to a list, until negative number appears. Remove elements that are below average value and sort the list.
- Solution is split in several parts/classes
  - Custom class *Loader* that loads non negative numbers using *Scanner*
    - It would be set to *System.in* in the main program
  - Calculation average in the list
  - Custom class *BelowThreshold* (implements interface *Predicate*)
    - Predicate (in general): boolean value function, i.e. statement that may be true or false depending on the values of its variables.
  - Remove elements using predicate and default *List* method *removeIf*
  - Sort elements using class *Collections* that contains only static methods (many useful methods like sort, reverse, shuffle, …)
    - <u>not to be confused with interface *Collection*</u>

Example #1 – note on a Predicate and *super*

09_Collections/swu/oopj/collections/example1/*.java

- We have a list of integers : List<Integer>
- Default method *removeIf* expects a predicate that can test whether some *Integer* is good or not
  - *removeIf* has the following signature

    default boolean removeIf(Predicate<? super E> filter)
  - This means that valid argument could be Predicate<Integer>, but also Predicate<Number> , i.e. *Predicate<? super Integer>*
  - *removeIf* passes an *Integer* to *test* method. We can use any predicate that can accept an *Integer*
    - *Integer extends Number*

```
public class BelowThreshold implements Predicate<Number> {
	public boolean test(Number d) {
		...
	}
}
```

Interface *Set*

- A ***Set*** is a *Collection* that **cannot contain duplicate elements**.
  - The Set interface contains only methods inherited from Collection and adds the restriction that duplicate elements are prohibited.
  - This restriction is semantic (interface cannot enforce such constraint syntactically) and implemented by concrete Set implementation
- Java has three general-purpose Set implementations:
  - *HashSet*, *TreeSet*, *LinkedHashSet*
  - Which to choose? It depends on what we need and do, and the answer depends on these questions.
    - Is iterating order or efficiency important?
    - How many reads and writes we have?

Set implementations

- *HashSet*
  - stores elements in buckets
  - the best-performing implementation; constant time performance for the basic operations assuming elements are dispersed properly among the buckets
  - makes no guarantees concerning the order of iteration.
- *TreeSet*
  - stores elements in a red-black tree (kind of kind of self-balancing binary search tree) and orders its elements based on their values;
- *LinkedHashSet*
  - Similar to HashSet with additional LinkedList to maintain order of insertion (used when iterating through)

An example: Using set to display unique program arguments (1/2)

- Custom method *addToSet* fill the set and return reference to it
- Custom method *print* iterates through anything that is Iterable

```
public static void main(String[] args) {
       System.out.println("Using HashSet:");
       print(addToSet(new HashSet<String>(), args));

       System.out.println("Using TreeSet:");
       print (addToSet(new TreeSet<String>(), args));

       System.out.println("Using LinkedHashSet:");
       print (addToSet(new LinkedHashSet<String>(), args));
}
private static Set<String> addToSet(Set<String> set, String[] arr) {
       for (String element : arr)
              set.add(element);
       return set;
}
```

09_Collections/swu/oopj/collections/UniqueArguments.java

An example: Using set to display unique program arguments (2/2)

Program arguments:

23 76 55 23 12 99 76 11 10

```
private static void print(Iterable<String> col) {
        for (String element : col)
                System.out.println(element);
        System.out.println();

//if using iterator instead of for-each
//      Iterator<String> iterator = col.iterator();
//      while(iterator.hasNext())
//              System.out.println(iterator.next());
//      System.out.println();
}
```

```
Using HashSet:
55
99
11
23
12
76
10

Using TreeSet:
10
11
12
23
55
76
99

Using LinkedHashSet:
23
76
55
12
99
11
10
```

09_Collections/swu/oopj/collections/UniqueArguments.java

- Note: the elements were Strings
  - What happens if there is an additional argument with value 150?

Complexity of common methods in *Set* and *List* implementations

- (Time) complexity – computation complexity that describes the amount of time it takes to do some tasks
  - Instead in time units, expressed as a function of the size of the input
    - Order of number of instructions, steps, …
- What is the complexity of:
  - *contains(Object e)?*
  - *remove(Object e)?*
  - *add(E e)?*
    for HashSet and TreeSet and similar operation in ArrayList and LinkedList?

Example #2

- Write a function that has array of names as arguments and prints each name only once in reverse order
  - 2a: without use of any additional data structures
  - 2b: using list to store unique names and set to do the fast lookup for duplicates
  - 2c: using only set(s)
- Discuss (time) complexity of the solutions

09_Collections/swu/oopj/collections/example2/*.java

Map interface

- A ***Map*** is an object that maps keys to values
  - Collection of ordered pairs (key, value) : modeled using Map.Entry
    - models the mathematical function abstraction
  - Each key can map to at most one value.
    - Key cannot be changed
      - only removed from the map
  - Map cannot contain duplicate keys
    - But multiple key can have the same value

```
interface Entry<K,V> {
  K getKey();
  V getValue();
  V setValue(V value);
}
```

- Some examples of mapping
  - Person → Phone number (or vice versa)
  - Course → Set of enrolled students
  - Name → number of occurrences
- Also known as dictionary (C#) or associate array (JavaScript, PHP)

Map Interface

- Notice that *Map* have separate hierarchy and it is not *Iterable*

```
public interface Map<K,V> {
       int size();
       boolean isEmpty();
       boolean containsKey(Object key);
       boolean containsValue(Object value);
       V get(Object key);
       V put(K key, V value); //optional
       V remove(Object key); //optional
       void putAll(Map<? extends K, ? extends V> m); //opt.
       void clear(); //optional
       Set<K> keySet();
       Collection<V> values();
       Set<Map.Entry<K, V>> entrySet();

       boolean equals(Object o);
       int hashCode();
```

Basic operations

Bulk operations

Collection views that are is Iterable

Map Interface – default methods

- Many useful, but advanced methods
  - discussed in some other presentations

```
default V getOrDefault(Object key, V defaultValue)
default void forEach(BiConsumer<? super K, ? super V> action)
default void replaceAll(BiFunction<? super K, ? super V, ? extends V>
                                                                function)
default V putIfAbsent(K key, V value)
default boolean remove(Object key, Object value)
default boolean replace(K key, V oldValue, V newValue)
default V replace(K key, V value)
default V computeIfAbsent(K key,
            Function<? super K, ? extends V> mappingFunction)
default V computeIfPresent(K key,
   BiFunction<? super K, ? super V, ? extends V> remappingFunction)
default V compute(K key,
     BiFunction<? super K, ? super V, ? extends V> remappingFunction)
default V merge(K key, V value,
   BiFunction<? super V, ? super V, ? extends V> remappingFunction)
```

Map implementations

- Behavior and performance analogous to *Set* implementations
- *HashMap*
  - stores elements in buckets
  - the best-performing implementation; constant time performance for the basic operations assuming elements are dispersed properly among the buckets
  - makes no guarantees concerning the order of iteration.
- *TreeMap*
  - stores elements in a red-black tree (kind of kind of self-balancing binary search tree) and orders its elements based on key values;
- *LinkedHashMap*
  - Similar to HashMap with additional LinkedList to maintain order of insertion (used when iterating through)

A *Map* example (1/2)

- Count how many time some name has occurred
  - Run the program with different Map implementations and see the difference

09_Collections/swu/oopj/collections/MapExample.java

```
public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        Map<String, Integer> names = new HashMap<>();
        // Map<String, Integer> names = new TreeMap<>();
        // Map<String, Integer> names = new LinkedHashMap<>();
        System.out.println("Enter names (quit for end):");
        String name;
        while (!(name = scanner.next()).equals("quit")) {
                Integer val = names.get(name);
                names.put(name, val == null ? 1 : val + 1);
        }
        for (Map.Entry<String, Integer> entry : names.entrySet())
                System.out.format("%s occured %d time(s)%n",
                        entry.getKey(), entry.getValue());
}
```

A *Map* example (2/2)

- Reminder: Map does not extend neither Collection nor *Iterable*
  - Also it does not have a method that returns iterator.
- Iterating can be done by using one of three possible collection views
  - keySet : set od keys
  - values: collection of values
    - Discuss why this is not a set?
  - entries: set of pairs (key, value)
  - allows change of values while iterating (but not change of the map)

```
public interface Map<K,V> {
  Set<K> keySet();
  Collection<V> values();
  Set<Map.Entry<K, V>> entrySet();
  ...
```

```
public static void main(String[] args) {
       Map<String, Integer> names = new ...
       ...
       for (Map.Entry<String, Integer> entry : names.entrySet())
              System.out.format("%s occured %d time(s)%n",
                     entry.getKey(), entry.getValue());
}
```

09_Collections/swu/oopj/collections/MapExample.java

Other Java Collection Framework interfaces

- Java Collection Framework contains many other useful data structures, e.g.
  - *Queue* — typically for processing element in FIFO (first-in, first-out) manner
  - *PriorityQueue* – for priority heap
  - *Deque* — double ended queue, can also be used for LIFO (last-in, first out) manner (i.e. stack)
  - …

- Note: Java also contains some legacy classes like Stack and Vector
  - Some of them are not recommended to use (e.g. Deque should be used instead of Stack), and some of them should be used only in multithreading environment if thread-safe implementation is needed (e.g. Vector)