Java training

Arrays, collections

Session overview

- Arrays, collections
- `Iterator, Comparator`
- Hands-on - using arrays & collections

Arrays

- **Array** - container object that contains a *fixed* number of elements
- Size - initialized when the array is created
  - After its creation, **the length is fixed**
- 0 based index
- Example:

```
int[] array = new int[5];   // initializes the array and set the length to 5
array[2] = 23;              // sets the 3rd element to 23
```

- Accessing the array for a bigger size → `ArrayOutOfBoundException`

Array values initialization

- Initialized

```
int[] array = {1, 2, 3, 4, 5};// the number of values set the length
```

- Not initialized

```
int[] array = new int[5];  // the length is initialized, but the array is empty
```

Arrays operations

- Arrays operations - copy, sort and search
- JDK classes for array operations
  - `java.util.Arrays` - comprehensive toolset
  - `System`
- Example:

```
int[] elements = {1, 2, 3, 4, 5};
int[] newArray = Arrays.copyOfRange(elements, 2, 3);
    // copies 2 elements, starting from the 3rd position
```

Collections

- **Collection** - container that groups multiple elements into a single unit (object)
- Main collection **types**:
  - **`List`** - keeps items regardless of equality (can contain the same object many times)
  - **`Set`** - keeps only distinct objects, based on their equality (further detailed)
  - **`Map`** - maps *keys* to *values*; both the key and the value *must be* an object
- `Set` & `List` extend the `Collection` interface
  - `isEmpty(), size()`
  - `add(Object o), addAll(Collection c)`
  - `remove(Object o), removeAll(Collection c)`

First example + hands-on

```
List<String> months = new ArrayList<>();        // creating an empty list
months.add(“January”);

months.addAll(Arrays.asList(“February", "March”));    // adding a coll
String month = month.get(1);              // which month will be retrieved?

String removed = months.remove(2);      // which month will be removed?
boolean isRemoved = months.remove(“January”);
                    // can remove a specific object, based on equality
```

`equals()` and `hashCode()`

- Methods used in the processing of:
  - Objects equality
  - Ordering objects in collections
  - Sorting objects in collections
- Each POJO (business properties container) class should override the native implementations
  - POJO - Plain Old Java Object → contains just properties and getters + setters
  - Native - platform specific
- Generally advised to override both methods in a class, to maintain the contract of ‘`hashCode`’

`public boolean equals(Object other)`

- Returns the equality state of two objects
- Two objects are equal if:
  - By **default**: they have the same object reference
  - **Overridden**: their internal properties have the same values
- Example:

```
public boolean equals(Phone another) {
    return this.producer.equals(another.getProducer()) &&
        this.model.equals(another.getModel());
}                        // the main type classes have equals already implemented
```

Equality properties

For any non-null reference values x, y and z:

- **Reflexive**: `x.equals(x)` returns true
- **Symmetric**: if `x.equals(y)`, then `y.equals(x)`
- **Transitive**: if `x.equals(y)` and `y.equals(z)` → `x.equals(z)`
- **Consistent**: multiple invocations of `x.equals(y)` consistently return ‘`true`’ or consistently return ‘false’, provided no information used in ‘equals’ comparisons on the objects is modified
- **Null equality**: `x.equals(null)` should return ‘`false`’

`public int hashCode()`

- Computes the object's **hash code** - a consistent integer signature / identifier
- Mainly used in **HashMap** and **HashTable** (key / value pair collections)
- General **contract**:
    - Invoking on the same object >1 times, during a program execution, the '`hashCode`' method *must consistently return the same integer*, provided no information used in the '`equals`' comparisons on the object is modified
    - The value *need not* remain consistent from one execution of a program to another execution of the same program

A lot of text!

- If two objects are equal according to the *equals* method → calling the *hashCode* method on each of the two objects *must* produce the same integer value

Live demo

A simple `Product` class

- Using `equals` & `hashCode`, when the `equals` and `hashCode` are not implemented
  - `equals` and `hashCode` on two different objects
  - `equals` on the same object reference
- Implement `hashCode` & equals in the `Product` class, re-run the previous examples

List

- Main implementations:
  - `ArrayList` - unordered elements, non-synchronized operations
  - `Vector` - unordered elements, synchronized operations (semi-deprecated)
  - `LinkedList`
    - Elements order - **as they are added**
    - Implemented internally as a double linked list
- Can contain ‘null’ elements
- Not sorted / ordered, by default
  - Can be ordered using the `.sort()` method → further discussed

`ArrayList` vs `LinkedList` → when to use which one?

Set

- Contains **distinct** elements (based on their equality)
- Main implementations:
    - `HashSet` - *unordered* elements
    - `TreeSet` - ordered elements, according to their comparison with the other objects
    - `LinkedHashSet`:
        - Implemented as a double linked list
        - The elements are *ordered as they are added*
- Can contain at most one ‘null’ element

Map

- Maps **keys to values**
- Main implementations:
    - `HashMap` - unordered key / value pairs
    - `TreeMap` - ordered key / value pairs, **according to their key** comparison
    - `LinkedHashMap`:
        - Key / value pairs - ordered as they are added, **according to the key**
        - Implemented as a double linked list
- Can contain at most one 'null' key
- Has methods for returning the:
    - Keys - `public Set<K> keySet()`
    - Values - `public Collection<V> values()`

‘`Comparable<Type>`’ interface

- Contains a ‘`int compareTo(Type type)`’ method → used internally by:
  - The ordered collection classes to add the elements in their ‘natural’ order
  - The sorting methods, for sorting the elements from a collection
  - According to the ‘compared to’ object, returns a value:
    - < 0 - is less than
    - == 0 - is equal to
    - > 0 - is greater than

Collections sorting

- Using a Tree* collection + implementing the `Comparable` interface
- Using the `Collections.sort()` method

Hands-on example

`Iterator` interface

- Traversing / iterating over a collection
- Allow modifying the elements from that collection [during the iteration]
- Usage:

```
List<String> listOfStrings = … ;
Iterator<String> iterator = listOfStrings.iterator();
while (iterator.hasNext()) {
    String element = iterator.next(); // get next element
    iterator.remove();       // delete the current element
}
```

Hands-on →

Queue, Deque

- **`Queue`** - base interface for queue type containers
- Types:
  - LIFO (last in, first out) - stack data structure
  - FIFO (first in, first out) - queue data structure
- **`Deque`** - insert or remove elements from both ends (head and tail)
  - The short name for ‘double ended queue’
- **Operations**:
  - `add, offer` - add elements to the queue / deque
  - `remove, poll` - remove elements
  - `element, peek` - get (without removing) the head of the queue

Best practices / advices

- Use `Arrays.asList()` to quickly build lists
- Use Google's Guava library for many Collections related utilities

Live demo + hands-on: using `Arrays.asList()`

Q&A session

1. You ask, I answer
2. I ask, you answer

Hands-on

- Exercise with several arrays (String, integers)
    - Create arrays and add items to them
    - Display the items from the created arrays
- Exercise with several `Lists`, `Sets` and `Maps`
    - Create a few:
        - `List`
        - `Set`
        - `Map`
    - Retrieve and remove the items via index, display them
    - Iterate over a collection with an iterator; add and remove from it