Final Project Presentation

Modeling software libraries in OPM
Java Collection Framework

Antonovsky Yulia
Mizrachi Assaf
Lepa Sahar
Ziv Meytal
• Model the Java collections API in OPM.

• Create an easy to understand and interpret model, such that it can be used by software system modelers as a built-in OPM library.

• Create a model that will provide the same functionality provided by the Java Collections API.
An **application programming interface (API)** is a particular set of rules and specifications that software programs can follow to communicate with each other.

In object oriented languages, an **API** usually includes a description of a set of class definitions, with a set of behaviors associated with those classes.

The **Collections Framework** is a unified architecture for representing and manipulating collections, allowing them to be manipulated independently of the details of their representation.
Keep it **simple**, while providing full functionality of Java Collections API.

Make the model **easy to read** and understand, but full enough to use when it is needed.

Every used object, that is not an instance, **defined earlier** in “structure model”, so no one should guess “what is it?”

All functions are separated to **packages**, according to collection interface in which they were defined.

All objects are **informatical**, as modeled system is a software system.
Final model has **52 views**, that describes main data structures and behaviors of Java Collections API.

There are 14 views, that present data structure of collections’ interfaces and some other structures, that were used to model the behaviors.

All other views - model full functionality of Java Collections.

*Let’s see it...*
The most basic interface is **Collection**. List, Set and Queue extend Collection. Set is extended by SortedSet and Queue by BlockingQueue. Each extension has its own additional functionality (except Set) and some of them overwrite the functionality, that was defined earlier.
Collection is the root interface in the *collection hierarchy*. A collection represents a group of objects, known as its *elements*. Collection Interface represents also set of behaviors, that called “Collection Functionality” in the model.

Collection exhibits Type Abstract.
Collection consists of many Elements and Collection Functionality.
Element is instance of a Type Abstract.
CollectionImplementationClass is a Collection.
Collection interface presents 14 basic methods, that are part of all Collections’ interfaces and implemented by all collections’ implementation classes.
To make it easy to understand, we added short description to each function model. Also, when there were more than one possible result, we added condition to each result link in all the models.
Set - collection that contains no duplicate elements. Set interface, has no additional functionality for those defined in collection.
SortedSet is a set that further guarantees that its iterator will traverse the set in ascending element order. Note that the ordering maintained by a sorted set. SortedSet defines some new additional functionality.
SortedSet Functionality un-folded
**SortedSet Functionalities views**

SubSet returns a view of the portion of this sorted set whose elements range from ’FromElement’, inclusive, to ’ToElement’, exclusive. (If ’FromElement’ and ’ToElement’ are equal, the returned sorted set is empty).
All elements inserted into a sorted set must implement the Comparable interface (or be accepted by the specified Comparator). The method returns the comparator associated with this sorted set, or null if it uses its elements' natural ordering.
List - an ordered collection (also known as a sequence). The user of this interface has precise control over where in the list each element is inserted. The user can access elements by their integer index (position in the list), and search for elements in the list.
In addition to functionality, that defined by Collection, List defines some new methods and overwrite some functions defined earlier.
The List interface provides a special iterator, called a `ListIterator`. A method is provided to obtain a list iterator that starts at a specified position in the list.
The List interface provides two methods to efficiently insert and remove elements at an arbitrary point in the list. For example, `Remove` - Removes the element at the specified position in this list.

![Diagram showing the functionalities of the List interface](image-url)
The List interface provides two methods to **search for a specified object**. One of them is `LastIndexOf`, that returns the index in this list of the last occurrence of the specified element.
Queue - collection designed for holding elements prior to processing. Queues typically, but do not necessarily, order elements in a FIFO (first-in-first-out) manner.
Besides basic Collection operations, queues provide additional insertion, extraction, and inspection operations.

Let’s see all of them, as there models are quite different from the previous.
The **Element** and **Peek** methods return, but don’t remove, the head of the queue.
The **Offer** method inserts an element if possible, otherwise returns `false`. This differs from the Collection. **Add** method, can fail to add an element only by throwing an unchecked exception.
The **Remove** and **Poll** methods remove and return the head of the queue. Exactly which element is removed from the queue is a function of the queue's ordering policy, which differs from implementation to implementation. The **Remove** (defined by Collection) and **poll()** methods differ only in their behavior when the queue is empty: the **remove()** method throws an exception, while the **poll()** method returns null.
BlockingQueue un-folded

BlockingQueue

- ArrayBlockingQueue
- DelayQueue
- LinkedBlockingQueue
- PriorityBlockingQueue
- SynchronousQueue

Fullness
- empty
- non-empty

RemainingCapacity: unsigned integer
- MAX_INTEGER
- 1

BlockingQueueImplementationClass

BlockingQueue Functionality
BlockingQueue is a Queue that additionally supports operations that wait for the queue to become non-empty when retrieving an element, and wait for space to become available in the queue when storing an element.

This is the most difficult, but also the most interesting Collection to model!
RemainingCapacity returns the number of elements that this queue can ideally (in the absence of memory or resource constraints) accept without blocking.
**DrainTo** removes all available elements from this queue and adds them into the given collection.
**Put** adds the specified element to this queue, waiting if necessary for space to become available.
Take retrieves and removes the head of this queue, waiting if no elements are present on this queue.
Poll retrieves and removes the head of this queue, waiting if necessary up to the specified wait time if no elements are present on this queue.
Thanks for your attention...

Questions?