Recap: Generics and Lists

Generics

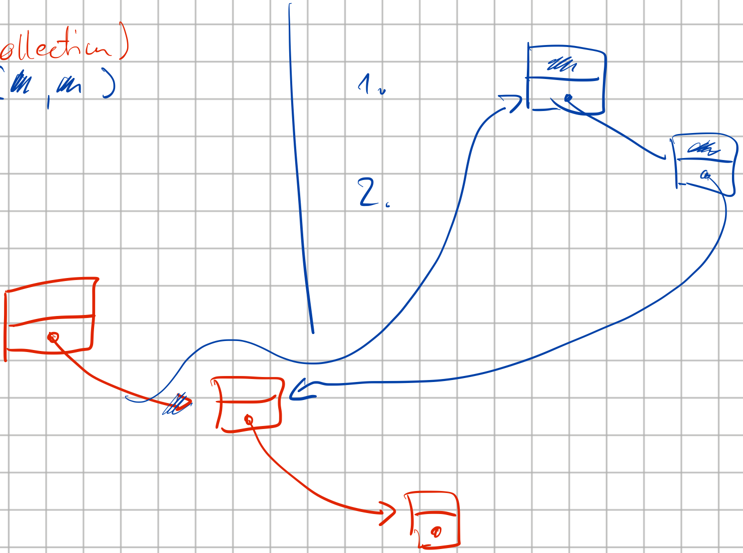
- Template classes usable for multiple classes
  - E.g., collections
- Syntactically denoted by < >

Lists

- Unidimensional, ordered collection
- Two implementations available
  - ArrayList: Uses an array internally
    - Array: Elements are stored in a continuous block of memory
  - LinkedList: Uses a linked list internally
    - List: Elements are distributed all over the place, but linked

addAll(index, Collection)

1 , (m, m)



Exercise 5: MyLinkedList

- Using the iterator and `getElement()` solves most of the problems
- Handling the first element sometimes requires extra care (`add()`, `addAll()`, `remove()`)
- Most complex method `addAll(int index, Collection c)` can be solved in two steps:
  - Put the elements of `c` into a linked list
  - Insert it at the right position
- Testing all methods is tedious – automatic testing to the rescue

➔ Later in the semester

UNIVERSITÄT
ZU KÖLN

Session 6: Collections, Part 2 (Queues and Sets)

Fortgeschrittene Programmierung (Java 2)

Nils Reiter
`nils.reiter@uni-koeln.de`

May 29, 2024

INSTITUT FÜR
DIGITAL HUMANITIES
UNIVERSITÄT ZU KÖLN

Interfaces

`java.util.Collection`

- `java.util.List` 🡐 last week
- `java.util.Queue` 🡐 today
- `java.util.Set` 🡐 today

`java.util.Map` 🡐 next week

Section 1

Queue and Stack

SCOTLANDS WORK AND WORTH
1964-65 SPONS' ARCHITECTS' AND BUILDERS' PRICE BOOK
SHELLEY
A WONDER BOOK AND TANGLEWOOD TALES
LITTLE DORRIT
BYRON'S WORKS
MASTERPIECE LIBRARY OF SHORT STORIES
Northanger Abbey
JANE AUSTEN
HARRAP
Dostoevsky
The Possessed
DON QUIXOTE
WALDEN
Pilgrims Progress 1775
The Portrait of a Lady
D.H. LAWRENCE

Queue and Stack

- Ordered collection, changeable by adding/removing only from one end
- Last In, First Out (LIFO, Stack)
  - Same end for adding and removing elements
- First In, First Out (FIFO, Queue)
  - Different end for adding and removing
- No random access (i.e., no access to elements in the middle)

Examples (from Real-Life)

Queue in Java

- Interface java.util.Queue<E>
  - Special case: capacity-restricted Queue (i.e., one with a limited size)
- Defines several methods:

	Throws exception	Returns special value
Insert	`add(e)`	`offer(e)`
Remove	`remove()`	`poll()`
Examine	`element()`	`peek()`

Table: Queue Methods

Queue in Java

Sub Interfaces

- java.util.Deque<E>
  - "Deque": double ended queue
  - Access on both ends (but not in the middle)
- java.util.BlockingQueue<E> / java.util.BlockingDeque<E>
  - Wait for the queue to become non-empty when retrieving
  - Wait for space to become available in the queue when adding

Implementations

- Based on an array: java.util.ArrayDeque<E>
- Based on linked list: java.util.LinkedList<E>

demo

Queues in Action

Section 2

Set

Sets

$$\{1,2,3\} \cap \{3,4,5\} = \{3\}$$

$$\{1,2,3\} \cup \{3,4,5\} = \{1,2,3,4,5\}$$

$$|\{3,5\}| = 2$$

$$\{3,2,1\} = \{1,2,3\}$$

Sets

W Set_theory

- Mathematical concept
- No order: $S = \{1,2,3\} = \{3,1,2\} = \{2,3,1\}$
- Cannot contain the same element twice: $\{1,2,3\} = \underline{\{1,1,2,3\}}$
- Special symbol for empty set: $\emptyset = \{\}$
- Operations
  - Union / Vereinigungsmenge: $\{1,2\} \cup \{2,3\} = \{1,2,3\}$
  - Intersection / Schnittmenge: $\{1,2\} \cap \{2,3\} = \{2\}$

Sets in Java

- java.util.Set<E>
- `add(e)` returns `false` if `e` is already in the set
- No random access to specific elements: Because there is no order, there cannot be an index value
- Access only via iterators

Implementation: java.util.HashSet

This class implements the Set interface, backed by a hash table (actually a HashMap instance). It makes no guarantees as to the iteration order of the set; in particular, it does not guarantee that the order will remain constant over time. This class permits the null element.

java.util.HashSet<E>

When are two Objects ‘the Same’?

- This is something we can control (and usually should)
- Every object inherits from java.lang.Object
- Two important methods: `hashCode()` and `equals(Object o)`

When are two Objects ‘the Same’?

- This is something we can control (and usually should)
- Every object inherits from java.lang.Object
- Two important methods: `hashCode()` and `equals(Object o)`

```
boolean equals(Object o)
```

- Reflexive, symmetric, transitive, consistent
- `x.equals(null)` is `false` for any object x

x.equals(x) == true

x.equals(y) == y.equals(x)

x.equals(y) $\wedge$ y.equals(z)

$\Rightarrow$ x.equals(z)

When are two Objects ‘the Same’?

- This is something we can control (and usually should)
- Every object inherits from java.lang.Object
- Two important methods: `hashCode()` and `equals(Object o)`

```
boolean equals(Object o)
```

- Reflexive, symmetric, transitive, consistent
- `x.equals(null)` is `false` for any object x

```
int hashCode()
```

- If `x.equals(y)` returns `true`, `x.hashCode() == y.hashCode()`
- Used extensively in collections

`equals()` vs. `==`

- `==` compares if the objects are the same
  - I.e.: If they refer to the same unit in memory
- `equals()` lets the objects decide their equality
  - By overwriting the method in a class
  - By default (java.lang.Object.equals()): `x.equals(y)` is `true` iff `x==y`

demo

Sets in Action, implementing equals() and hashCode()

Exercise

`https://github.com/idh-cologne-java-2-summer-2024/exercise-06`