Recap: Iterator and Iterable

- Before: Iterating via `for`/`while` loop
  - ‘Looping logic’ is in the code that executes the loop
- Iterator: An interface that represents an iteration
  - Easy to be used in conjunction with `while` loops
  - Two (central) methods: `boolean hasNext()` and `T next()`
  - Allows encapsulating looping conditions in an object
- Iterable: Signifies that one can iterate over an instance of the class
  - `Iterator<T> iterator()` returns an iterator

UNIVERSITÄT
ZU KÖLN

Session 5: Generics and Collections, part 1

Fortgeschrittene Programmierung (Java 2)

Nils Reiter
nils.reiter@uni-koeln.de

May 15, 2024

INSTITUT FÜR
DIGITAL HUMANITIES
UNIVERSITÄT ZU KÖLN

Section 1

Generics

Generics

Motivation

- Duplicating code is bad
  - Errors fixed in one copy are not fixed in the other
  - Disk space
- Many things we do are similar, but for different types
  - E.g., collecting things, iterating over them, ...

Iterator<Accounts

Generics

Motivation

- Duplicating code is bad
  - Errors fixed in one copy are not fixed in the other
  - Disk space
- Many things we do are similar, but for different types
  - E.g., collecting things, iterating over them, …

Generics

- Method to write ‘template classes’
- Instantiated for different types
- Syntax: `Iterator<T>`, `MyClass<E extends Car>`, …
  - `T`, `E` are variable names for class names
- Type is filled in at compile time
  - I.e., while we implement a generic class, we don’t know what type it is used for

demo

StringList, GenericList

Section 2

Collections

Java Collections Framework

A collection is an object that represents a group of objects (such as the classic Vector class). A collections framework is a unified architecture for representing and manipulating collections, enabling collections to be manipulated independently of implementation details.

Javadoc

Java Collections Framework

A collection is an object that represents a group of objects (such as the classic Vector class). A collections framework is a unified architecture for representing and manipulating collections, enabling collections to be manipulated independently of implementation details.

Javadoc

Benefits

- Reduces programming effort
- Increases performance
- Fosters software reuse

Interfaces

`java.util.Collection`

- `java.util.List` 🡐 today!
- `java.util.Set`
- `java.util.Queue`

`java.util.Map`

- `java.util.SortedMap`

List

- Finite number of ordered elements, allowing duplicates
  - Size is not fixed
- Access via index values
- java.util.List
  - `add`, `addAll`, `set`, `replaceAll`
  - `contains`, `containsAll`, `isEmpty`, `size`
  - `remove`, `removeAll`, `clear`
  - `subList`, `iterator`, `listiterator`
  - `sort`

List

- Finite number of ordered elements, allowing duplicates
  - Size is not fixed
- Access via index values
- java.util.List
  - `add`, `addAll`, `set`, `replaceAll`
  - `contains`, `containsAll`, `isEmpty`, `size`
  - `remove`, `removeAll`, `clear`
  - `subList`, `iterator`, `listiterator`
  - `sort`
- Implementations
  - `java.util.ArrayList`: Uses an array internally
  - `java.util.LinkedList`: Uses a linked list internally

ArrayList vs. LinkedList

```java
// ArrayList
List<Student> arr = new ArrayList<Student>();
// ...
arr.add(new Student("Maria"));
arr.set(203, new Student("Hans"));
// ...
arr.get(203).doSomething();

// LinkedList
List<Student> ll = new LinkedList<Student>();
// ...
ll.add(new Student("Maria"));
ll.set(203, new Student("Hans"));
// ...
ll.get(203).doSomething();
```

Speed Differences

- Many library functions hide complexity
- ⚠ This does not mean that the complexity is gone

Speed Differences

- Many library functions hide complexity

⚠ This does not mean that the complexity is gone

Arrays / ArrayList

- ‘constant access’: Accessing the 5th or the 9000th elements takes the same time
- Enlarging an array after creation is costly (because the entire array needs to be copied elsewhere)

Speed Differences

- Many library functions hide complexity
- ⚠ This does not mean that the complexity is gone

Arrays / ArrayList

- ‘constant access’: Accessing the 5th or the 9000th elements takes the same time
- Enlarging an array after creation is costly (because the entire array needs to be copied elsewhere)

LinkedList

- The longer the list the longer it takes to access an element
- Enlarging is constant, removal in the middle as well

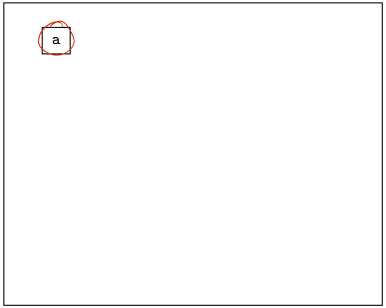
demo

Section 3

How do Linked Lists Work?

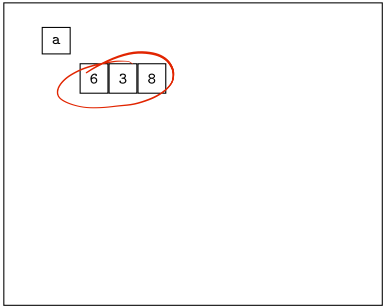
Memory Handling (Reminder)

```
// primitive types
char ch = 'a';
```



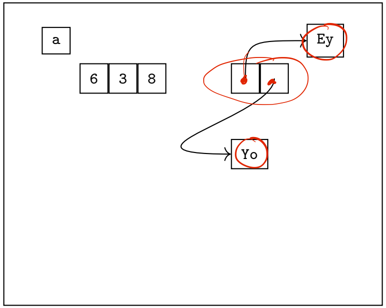
Memory Handling (Reminder)

```
// primitive types
char ch = 'a';
int[] iArr = new int[] {6,3,8};
```



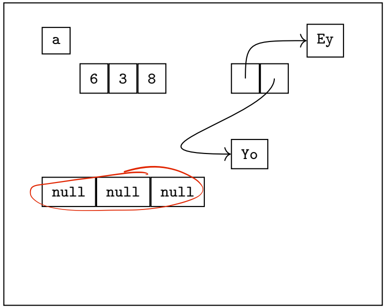
Memory Handling (Reminder)

```
// primitive types
char ch = 'a';
int[] iArr = new int[] {6,3,8};
// reference types
String[] sArr = new String[] {"Ey", "Yo"};
```



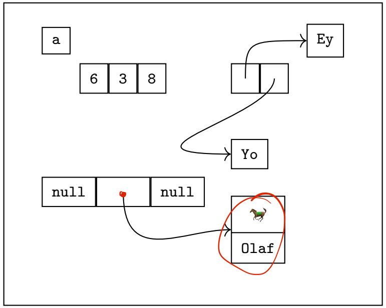
Memory Handling (Reminder)

```
// primitive types
char ch = 'a';
int[] iArr = new int[] {6,3,8};
// reference types
String[] sArr = new String[] {"Ey", "Yo"};
Horse[] hArr = new Horse[3];
```



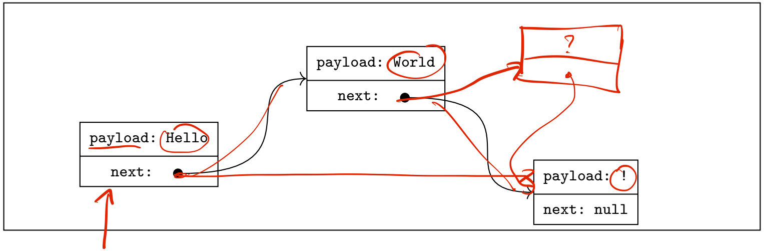
Memory Handling (Reminder)

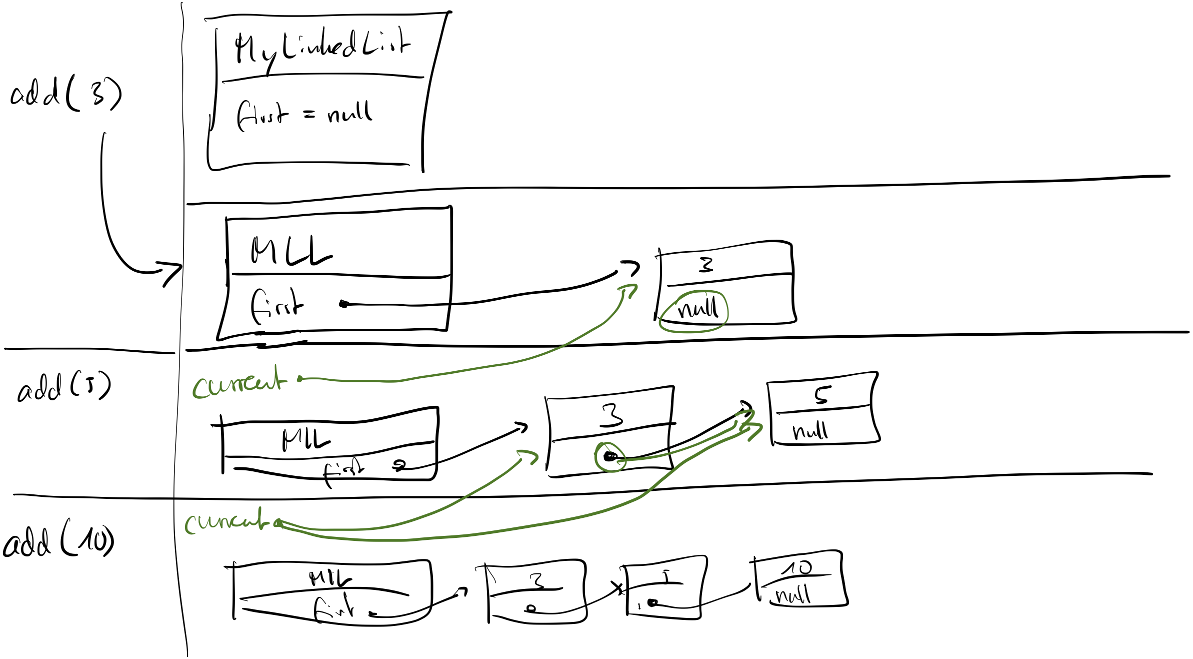
```
// primitive types
char ch = 'a';
int[] iArr = new int[] {6,3,8};
// reference types
String[] sArr = new String[] {"Ey", "Yo"};
Horse[] hArr = new Horse[3];
hArr[1] = new Horse("Olaf");
```



Linked Lists

- Well-established idea on how to store elements in a sequence
- Each element is an object that stores
  - the actual element (called `payload`)
  - the next element (called `next`)
- For the last element, `next == null`




add(3)
MyLinkedList
first = null
MLL
first
3
null
add(5)
current
MLL
first
3
5
null
add(10)
current
MLL
first
3
5
10
null

demo

Implementation of a linked list

Exercise

https://github.com/idh-cologne-java-2-summer-2024/exercise-05