Collections – Part III

Nico Ludwig (@ersatzteilchen)

TOC

- Collections – Part III
  - Sequential Collections – Linked Lists
  - Structural Pattern Matching in functional Programming Languages
  - Ordered vs. unordered Collections
  - Iterators
  - An Interface-based Approach
  - The Dependency Inversion Principle (DIP) explained with Collection Frameworks

Initial Words

Yes, my slides are heavy.

I do so, because I want people to go through the slides at their own pace w/o having to watch an accompanying video.

On each slide you'll find the crucial information. In the notes to each slide you'll find more details and related information, which would be part of the talk I gave.

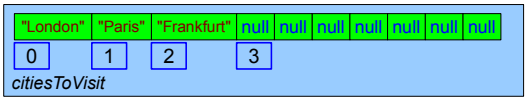
Have fun!

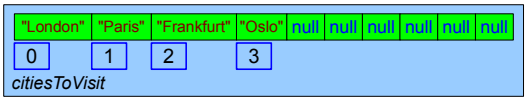
When indexed Collections reach their Limits

- Often we need to append elements at the end or prepend elements to the front of an indexed collection (e.g. a list):

```
// Java
// The specified cities need to be visited in the given order:
ArrayList citiesToVisit = new ArrayList(List.of("London", "Paris", "Frankfurt"));
// But then we've been informed that we need to visit "Oslo" after we visited "Frankfurt". So let's append "Oslo" to the list:
citiesToVisit.add("Oslo");
```

- Appending elements to a list works like so:
  - (Internally *citiesToVisit* is backed by an array.)
  - (1) Find the free slot after the last element.
  - (2) If there is no free slot, resize the backing array to create space.
  - (3) Insert the new value "Oslo" into the free slot.





- Of course this works, but it is somewhat inefficient:
  - In the worst case (in step (2)) the backing array's capacity is too small to hold another element, e.g. if we add a tenth element!
    - Then a completely new array with a larger capacity needs to be created!
  - This "enlarging" problem is present, when elements are appended, prepended or otherwise inserted or removed.
    - Therefor capacity control (initial specification or ad hoc adjustment of the capacity) of an indexed collection is possible and wise.
  - But there is another way to address the resizing problem with another type of collections: sequential collections, e.g. linked lists.

Sequential Collections – Linked Lists

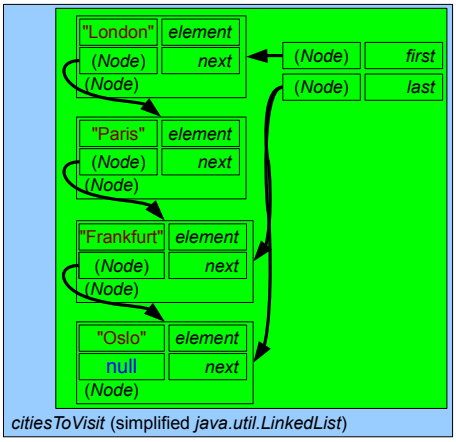
- The internal representation of linked lists (here we'll discuss Java's *java.util.LinkedList*) is no longer array-based!
  - Its elements are not stored in a contiguous block of memory, but as a bunch of linked nodes.

```
// Java
// The specified cities need to be visited in the given order:
LinkedList citiesToVisit = new LinkedList(List.of("London", "Paris", "Frankfurt"));
```

- Appending of a new element usually works like so:
  - (1) Get the last node.
  - (2) Create a new node with the value "Oslo".
  - (3) Link the new node with the last node.
  - (4) Designate the new node as last node.

```
// Append "Oslo":
citiesToVisit.add("Oslo");
```

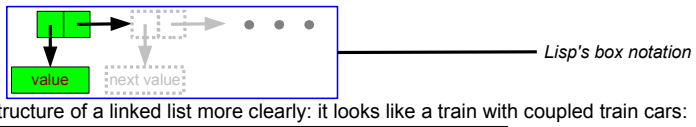
- The structure of *LinkedList*:
  - *LinkedList* does directly refer the first (also called "head") and the last node.
  - Each node refers its successor.
  - A node referring to null as its successor is the last node.
  - That's basically it!



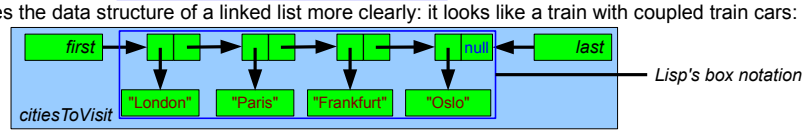
- In the programming language Lisp the concept of linked lists is basically the only idiom used to write programs. – Lisp programs themselves are linked lists.

Linked List – Adding and Removing Elements

- For further discussions, we use the box notation from the Lisp programming language:



  - The box notation unleashes the data structure of a linked list more clearly: it looks like a train with coupled train cars:

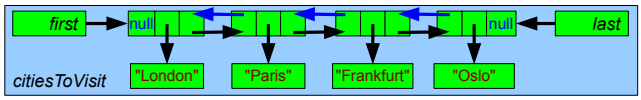


  - Esp. we can add train cars between any other two train cars, which is easy at the start and the end of the train.

- Linked lists have O(1) complexity, when elements are added (prepend/append) or removed at the first or last node.
  - This is true because:
    - (1) no continuous block of memory needs administration in form of reallocations, instead, references (i.e. shortcuts) must be moved around.
    - (2) We can directly access the first and last node.
  - The performance is independent of the count of the linked list's elements.
    - (However, some effort is required to create new node objects.)

Linked List – Doubly Linked Lists

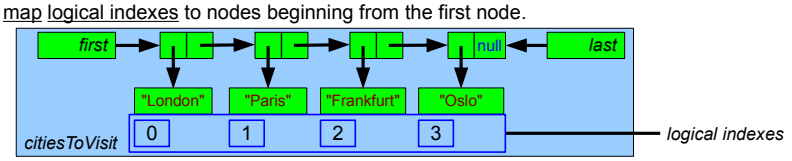
- Linked lists that have nodes referring to successor and predecessor nodes are called doubly linked lists.
  - (Java's and .NET's *LinkedList*s are in fact doubly linked lists. C++' *std::list* is usually implemented as doubly linked list.)



- Another derivate of linked lists are so called circular lists.

Linked List – Accessing Elements

- More facts on linked lists:
  - Virtually, linked lists don't have a notion of physical indexes, because no "indexable" memory builds up its base.
  - However, it is possible to map logical indexes to nodes beginning from the first node.



  - Accessing an arbitrary element by an index *k* is a relatively expensive operation, because all *k* nodes need to be visited sequentially!

```
Node currentNode = citiesToVisit.first;
int k = 2;
int currentIndex = 0;
while (null != currentNode && currentIndex != k) {
      currentNode = currentNode.next;
      ++currentIndex;
}
System.out.printf("element at index %d: '%s'.%n", k, null != currentNode ? currentNode.element : "");
```

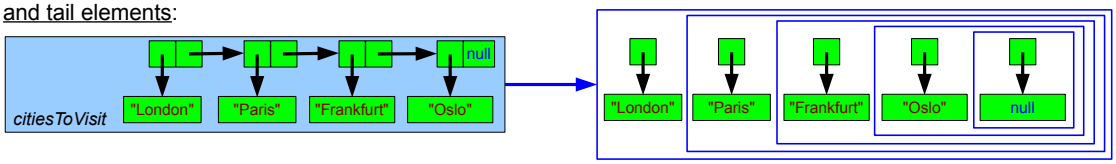
Node
+ item : Object + next : Node + prev : Node

    - The performance of index-accessing elements is dependent of the index that is to be accessed!
    - Index-accessing elements are O(n) operations on a linked list, because individual elements are not directly accessible. Loops are required here!
    - Or: linked list don't allow random access to elements. – Well, they are no indexed collections and we only have sequential access to elements!

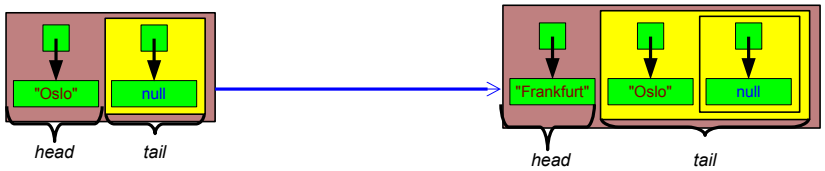
Linked List – Functional Programming – Part I

- Esp. in functional programming (fp) languages, lists do often play an important role; therefor I'll lose some words about it.
  - As a matter of fact immutable (linked) lists are often first class citizens in fp languages, often they have direct syntactical support.
    - (In opposite to imperative languages, where (mutable) arrays are often first class citizens with direct syntactical support.)

- Before discussing syntactical support, we've to clarify how linked lists can be understood as a chain of cascaded head and tail elements:



- So a single list element consists of two parts: the head and the tail:



  - Cascading allows the tail of the element to be yet another element having a head and a tail recursively.

Linked List – Functional Programming – Part II

- The idea of the mentioned syntactical support is to break lists into head and tail with a special operator.
  - E.g. lets sum the numbers contained in a (linked) list, by chopping its head and tail recursively:

```
// F#
let rec sumList list =
    match list with
    | head :: tail -> head + sumList tail
    | [] -> 0

printfn "%A" (sumList [1; 2; 3; 4; 5; 6; 7; 8; 9])
// >45
```

```
// Scala
def sumList(list: List[Int]) : Int =
    list match {
        case head :: tail => head + sumList(tail)
        case Nil => 0
    }

sumList(List(1, 2, 3, 4, 5, 6, 7, 8, 9))
// =45
```

```
-- Haskell
import System.Environment

sumList (x:xs)  = x + sumList xs
sumList []      = 0

main = print(sumList [1, 2, 3, 4, 5, 6, 7, 8, 9])
-- >45
```

  - In each snippet the passed list is matched against a given pattern:
    - If the list is not empty, it'll be broken into its head and tail. The head will be added to the result of a recursive call with the tail.
    - If the list is empty the result is 0 and the recursion ends.
  - This idiom of F#, Haskell and Scala is called (structural) pattern matching. Pattern matching is a kind of control structure.

- In Lisp we don't have syntactical support like pattern matching, but there are functions to get the head and the tail of a list:
  - *car* (Contents of the Address part of Register number) and *cdr* (Contents of the Decrement part of Register number).

```
; Lisp
(defun sum-list(l)
  (if (not (null l))
      (+ (car l)(sum-list (cdr l)))
      0))

(sum-list '(1 2 3 4 5 6 7 8 9))
; =45
```

10

- *car* and *cdr* have the aliases *first* and *rest* meanwhile :).
- There exist more functions to address the contents of a list apart from the operators ::/: and *car*/*cdr*, e.g.: *nth* or handy combinations of *car* and *cdr* like *caar* and *cadr*.
- It should be mentioned that memory management of fp languages is very sophisticated. Esp. during processing lists, many new lists are generated, because lists are often immutable; all the remaining superfluous lists need to be freed from memory. Therefor fp languages/runtimes were the first systems introducing garbage collection. Mind that the general ways of processing lists and also the memory considerations are equivalent to those of processing strings in Java and .NET, where strings are immutable (lists of characters).

List vs Linked List

- Terminology alert: often the terms "list" and "linked list" are used interchangeably! – If in doubt consult respective manuals!

- This is a comparison esp. pondering the aspects of fast random access vs. sequential access.

- When to use list/indexed collections (e.g. Java's *ArrayList*)?
  - Generally spoken: it should be your default choice! Usually the collection you'll use is filled once and/or iterated and read lot.
    - Or, if it is filled at different places, but the effective count of elements to be added is known. Mind, that this can be handled via capacity-control.
  - Use list, if the collection you have to use must be accessed often (random access is at O(1)) and iterated often.
  - They are "CPU-cache-friendly": fetching of data happens in adjacent/rectangular memory regions, which easily fit into the cache.
    - Linked lists instead force jumping in memory following references of nodes and this is hard to predict for the CPU and cannot be fit into the CPU cache.
  - Iterating of lists is faster, because accessing nodes of a linked list needs more references to follow.

- When to use linked list/sequential collections (e.g. *LinkedList*)?
  - Use linked list, if the collection you have to use must be manipulated often (add, remove can be faster than with list).
  - Use linked list, if the length should not be constrained by the length of a native array.
  - Classically: add/remove are O(1), but insertAt, removeAt, find and getAt are O(n)-operations, because access is sequential. [11]

Ordered and unordered Collections

- Up to now we discussed so called ordered collections.
  - (The opposite are unordered collections.)
- Basically in ordered collections the order, in which elements are added is the same order, in which they'll be iterated.
- Notice that "being ordered" has nothing to do with being sorted!
- Arrays, lists and linked lists are ordered collections.
- Now its time to understand what "iteration of collections" means.

Iterability – The least common Denominator of Collections

- Each collection allows accessing its elements iteratively. This can be done with loops, e.g. with for loops:

```
// Java
ArrayList numbers = new ArrayList(List.of(1, 2, 3, 4, 5));
for (int i = 0; i < numbers.length; ++i) {
        System.out.println(numbers.get(i));
}
```

  - Not all collections have indexes, on which iteraton can be done. But nevertheless generalized iteration is possible in most languages.
    - Such languages introduce the concept of an iterator that abstracts iteration from an underlying collection.
    - If collections provide an iterator, iterative access to these collections can be done with the same pattern/syntax. "Iterator" defines a basic design pattern.

```
// Java
ArrayList numbers = new ArrayList(List.of(1, 2, 3, 4, 5));
for (int element : numbers) {
        System.out.println(element);
}
```

- Iterative access: elements of collections can be retrieved in a loop in a defined manner and usually in a stable order.
  - Defined manner: the algorithm with which the elements are retrieved is strictly defined. Usually it is a forward-iteration.
  - Stable order: the order of iterated elements won't change in between full iterations as long as the underlying collection wasn't modified .
  - In solid words: iterability allows programming concise forward loops with an unknown end condition.
    - The implicit end condition is "no more elements to iterate".

13

- Now let's understand how iterability can help us as a feature of collections.

- An iterator defines, how to "walk" through a collection of objects.
- The iterator pattern does not mandate a stable order.

Iterability – The Iterator Concept

- "Iterator" is one of the easiest (behavioral) design patterns. The concept works like this:
  - Iterator provides a way to support iteration w/o unleashing the real structure of a collection.
    - E.g. no indexed collection is required and no length of the collection must be specified or known (= unknown end condition).
  - The aim of iterator is to separate the collection from element access.
  - Usually the elements visited during iteration are called items. Hence we'll use both terms (elements and items) interchangeably.
  - Not all items of a collection are fetched at once, but only the current item.

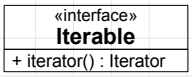
- How does this concept help us?
  - One benefit is that we have a guarantee that all collections can be iterated, e.g. via for each loops (e.g. for(:) in Java).
  - The concept is like a contract: all collections support iterability.

- Good! But how do collections support iterability? How do collections "publish" the feature of "being iterable"?
  - We have to define the concept of being "iterable", in other words we've to define iterable behavior...

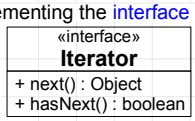
- Now we'll talk about the way Java's and .NET's modern oo-technologies handle this "abstraction of behaviors".
  - => Let's discuss/revisit Java and C#/.NET's interfaces.

Interface-based Iterability Support in Java – Part I

- Iterability can be abstracted: collections providing iterability implement a certain interface.
  - An interface is a type that only declares a set of unimplemented methods, furthermore it doesn't provide any data members (fields).
  - An interface only defines a behavior that implementors of that interface have to adopt.
- In Java iterable collections implement the interface *Iterable (java.lang.Iterable)*:

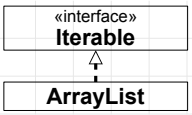


  - And *Iterable.iterator()* returns another object implementing the interface *Iterator (java.lang.Iterator)*:



  - And to drive iterability home: the type *ArrayList* does implement *Iterable*, because it is an iterable collection:

```
// (simplified)
public class ArrayList implements Iterable {
        public Iterator iterator() { /* pass */ }
}
```



- Now its time to see *Iterable* in action...

15

- Basically interfaces provide no method implementations (exception: Java's default/defender methods and Java's/.NET's static methods) and no data.
- The names of .NET's iterator-related types *IEnumerable*/*IEnumerator* carry "enumerate" in their names. This is a hint to how the iterator concept is closely related to the mathematical set **N**, which relies on the fact, that each number (read: item) has a successor.
- The segregation of the iterator concept into two interfaces allows an object to separate the pure ability to be iterated (implementing *Iterable*/*IEnumerable*) from the iterator object itself (something implementing *Iterator*/*IEnumerator*).
  - From the .NET Framework Design Guidelines: Do not implement both *IEnumerable*/*IEnumerator* on the same type!

Interface-based Iterability Support in Java – Part II

- Following Java for(:) loop ...

```
// We have following list of ints:
ArrayList numbers = new ArrayList(List.of(1, 2, 3, 4, 5));
```

```
// Iterate over an ArrayList with for each:
for (Object item : numbers) {
        System.out.println(item);
}
```

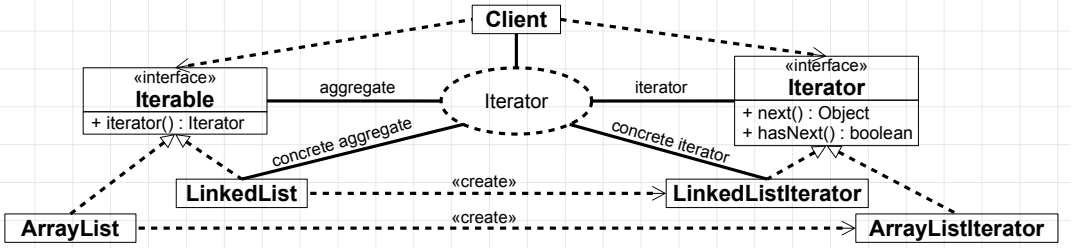
- … will really be transformed into code working like this:

```
// Virtually following iteration is executed for for(:) via the iterator (simplified):
Iterator items = numbers.iterator();
while (items.hasNext()) {
        System.out.println(items.next());
}
```

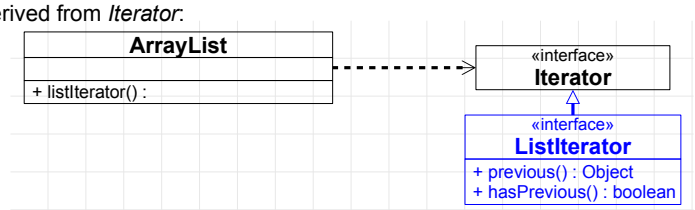
- To drive this point home:
  - In Java iterable collections, which are basically all collections in Java, implement the interface *Iterable*.
    - .NET's collections implement the interface *IEnumerable*, Cocoa's collections implement the @protocol *NSFastEnumeration*.
    - .NET's *IEnumerable* (and *IEnumerable<T>*) is the basis for LINQ.
  - Modern collection APIs express/publish their behavioral aspects via interfaces (or contracts or @protocols).
  - We'll revisit this aspect when we discuss further collection-relevant topics.

The Iterator Pattern

- Iterator is also one of the famous oo design patterns: Iterator abstracts element-traversal from the element's aggregate.
  - Traversal means the "moving" through the collection or data-structure.
  - The class-diagram looks complicated, but we already know, that iterator in Java is all but complicated:



- Java's *ListIterator* is derived from *Iterator*:



17

- Actually, the interface *ListIterator* provides more methods: such to add, remove and set elements during iteration.

Iterability – With Arrays

- Java's arrays are *Object*s, and they can be iteratered using for(:), but arrays do not implement *Iterable*.
  - Instead Java simply allows the idiomatic usage of for(:) with arrays without any sophisicated oo-pattern behind:

```
int[] numbers = {1, 2, 3, 4, 5};
for (int item : numbers) {
        System.out.println(item);
}
```

- Allthough arrays do not implement *Iterable*, they implement the iterator-pattern idomatically (i.e. only for for(:)).

- Since Java 1.8 we can create a *Stream* from an array, which can be understood as more common oo-approach.
  - A *Stream*, no matter from "where" the *Stream* was created, allows a common form of iteration via the method *Stream.forEach()*:

```
Arrays
        .stream(numbers)
        .forEach( item ->
                System.out.println(item)
        );
```

Iterability – The Iterator Concept in other Frameworks – Part I

- General benefits of iterators:
  - We can use them, if we don't know the length of the collection we're iterating.
  - Iterators are good to avoid off-by-one errors, because there is no index.
- .NET/C# and Java's implementations of iterator are similar:
  - E.g.: Java: *java.lang.Iterable<T>* needs to be implemented by collections to support Java's for(:) loop.
    - (This is different for Java's arrays.)
  - During iteration, the collection as well as the current item are generally mutation guarded:

```
// C#: Fail fast.
ArrayList numbers = new ArrayList{1, 2, 3, 4, 5, 6, 7, 8, 9};
foreach (int item in numbers) {
    numbers.Add(10);    // Will throw InvalidOperationException,
                        // because numbers was modified
                        // concurrently during the iteration.
}
```

```
// Java: Fail fast.
ArrayList numbers = new ArrayList(List.of(1, 2, 3, 4, 5, 6, 7, 8, 9));
for (Object item : numbers) {
    numbers.add(10);    // Will throw ConcurrentModificationException,
                        // because numbers was modified concurrently
                        // during the iteration.
}
```

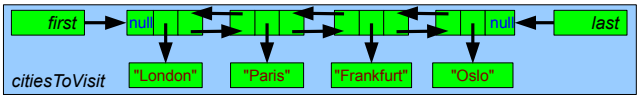
    - Because access to collections is checked every time in foreach/for(:) loops, foreach/for(:) iteration is slightly slower, than using an ordinary for loop.
    - Also because foreach/for(:) use interface-based/virtual method calls on the iterator objects, performance is slower than using for with [].
- Java and C++ support bidirectional iterators that can move to next as well as previous positions (e.g. Java's *ListIterator*).
  - The *ListIterator* is special in Java, as it can also modify the collection being iterated.
  - Let's understand why Java's *ListIterator* is useful when used together with *LinkedLists*...

19

- In Java, mutation guards are usually implemented using modification counters. The state of the counter is hold in the iterator, when it is created. During each structural change in the iterated collection, the modification counter in the collection is updated and compared to the counter hold in the iterator. However, which operations are "structural changing" operations is not clearly defined. – These operations are those operations, which directly or indirectly update/increment the protected field *modCount* of the iterated collection.
  - Structural modifications are operations like adding/removing elements, but not two iterators writing the same element at the same time.
- The collections in the package *java.util.concurrent* having the name prefix "*Concurrent*" (e.g. *ConcurrentHashMap* and *ConcurrentSkipList*) do not throw *ConcurrentModificationExceptions*, i.e. they do not fail fast!
  - Those collections provide fail-safe-iterators. Such iterators work on a copy of data, which is detached from the original collection and does not reflect any modifications done to the original collection after the fail-safe-iterator was created. The downside of these iterators lies in the overhead needed when creating the respective copy of the data.

Iterability – The Iterator Concept in other Frameworks – Part II

- The iterability of linked list is a special topic:
  - Due to the linked structure, forward iteration is a very efficient operation. (And also backward iteration for doubly linked lists.)




```
Node<String> currentNode = citiesToVisit.first;
while (null != node) {
      // do something with currentNode.item
      currentNode = currentNode.next;
}
```

```
for (String item : citiesToVisit) {
      // do something with currentNode.item
}
```

  - We can of course use an iterator to hide us from the details of the linked structure.

- Java supports bidirectional iterators that can move to next as well as previous items with the *ListIterator (java.util.ListIterator)*:

```
ListIterator listIterator = citiesToVisit.listIterator(); // Get a bidirectional iterator, a ListIterator from the ArrayList.
Object value1 = listIterator.next();          // value1 = "London"
Object value2 = listIterator.next();          // value2 = "Paris"
Object value3 = listIterator.previous();      // value3 = "Paris"; a switch of the direction will return the last item as previous/next item.
Object value4 = listIterator.previous();      // value4 = "London"
```

  - Inserting/removing items from a linked list are expensive operations in general (e.g. from an index), but they're cheap during iteration:

```
// Clear the whole LinkedList via a ListIterator:
while (listIterator.hasNext()) {
      listIterator.next();          // advances the iterator
      listIterator.remove();        // removes the item
}
```

20

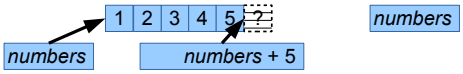
- Mind that we could have used pattern matching in an fp language for iteration (which is implemented as recursion).

- In the example above using the *ListIterator*, the object *citiesToVisit* is a *LinkedList*. Here, the segregation of the iterator concept into *Iterable* and *Iterator* shines: *LinkedList* itself implements *Iterable*, but the *Iterator* it provides can only do forward iterating. However, when we need more *List*-specific operations, we can can get a *ListIterator* by explicitly calling *List.listIterator()*. With the returned *ListIterator*, we can handle the linked list as proper sequence and iterate forward and backwards and we can even remove elements from the *List*. – This elegant and clear using of a *LinkedList* is possible, because *Iterable* and *Iterator* are nicely segregated.

Iterability – The Iterator Concept in other Frameworks – Part III

- In C/C++ iteration of arrays is possible via pointer arithmetics.
  - Arrays have a strict feature in C/C++: elements reside on a contigouos block of memory. Therefor pointer arithmetics work nicely.
  - We can iterate from a pointer to the first element of the array to the array's past-the-end-pointer (it doesn't belong to the array).

```
int numbers[] = {1, 2, 3, 4, 5}; // five elements
for (int* iter = numbers; iter != numbers + 5; ++iter) {
        std::cout<<*iter<<std::endl;
}
```



- C++ adopts the operators that have been used for pointer arithmetics to implement iterability over C++ collections.
  - In C++ the collection APIs are part of the standard template library (STL). The STL has another name for collections: containers.
  - The most important indexed collection in the STL is the *std::vector* (<vector>).
  - *std::vector* is a generic/templated collection (in opposite to an object-based collection). That means we have to specify the element type.

```
std::vector<int> numbers; // numbers is a vector containing int-elements
numbers.push_back(1); numbers.push_back(2); numbers.push_back(3); // Add five ints.
numbers.push_back(4); numbers.push_back(5);
for (std::vector<int>::iterator iter = numbers.begin(); iter != numbers.end(); ++iter) {
        std::cout<<*iter<<std::endl;
}
```

```
C++11
// In C++11 it is much simpler to iterate containers:
// Initialize numbers with an initializer list:
std::vector<int> numbers{1, 2, 3, 4, 5};
// Iterate over numbers with a range-based for loop:
for (const int item : numbers) {
        std::cout<<item<<std::endl;
}
```

  - The STL introduces the UDT *std::iterator* that abstracts iterability of STL collections.
  - *std::iterator* overloads several operators to use instances like pointers to implement pointer arithmetics (++, */->, ==/!=).
    - Some STL types support other iterators with special features: forward, input, output, bi-directional, random access and const iterators.

21

- In this example we are using a forward iterator.
- The idea of C++ iterators: allow iteration of containers via for loops and allow the operators that were used for array arithmetics to drive the iteration.
- C++ guarantees, that the past-the-end-pointer of an array will exist. It can be read, but neither dereferenced (for arrays this includes using []) nor written.
  - There are no guarantees for a "before-the-beginning-pointer" of an array. – This is esp. useful for platforms, because then they have the freedom to place arrays at any address-limit.
- Strange C++ iterators: we can, e.g., have five items but six possible iterator positions.

Iterability – The Iterator Concept in other Frameworks – Part IV

- "Old" JavaScript is a special: It offers a loop looking like an iterator loop ("foreach"), but isn't one! There are two problems:
  - (1) The current item in a loop is the index of that item in the iterated list, not the item itself:

```
// Often wrong:
var names = ["Mona", "Lisa"];
for (var item in names) {
        console.log("item: "+item);
}
// >item: 0 // Often unexpected!
// >item: 1
```

```
// Correct:
for (var i in names) { // Use the "item" as index:
        console.log("item: "+names[i]);
}
// >item: Mona // Aha!
// >item: Lisa
```

  - (2) The order of items/indexes yielded by the iteration is undefined! (The iterator pattern doesn't mandate order, but most coders assume it.)

- => Avoid using JavaScript's for-in loop! Better use good old for loops!
  - for loops are the most general way to iterate in JavaScript.
  - (JavaScript's for-in allows to enumerate the values of all the properties of an object, or to get the count of an object's properties.)

ECMAScript

```
// Instead of for, we can use the better for of-loop,
// It iterates items and does so in an ordered way.
for (let item of names) {
        console.log("item: "+item);
}
```

- Maybe the best way with expectable result is using the "foreach-class" of JavaScript functions:
  - However there are downsides:
    - Those may take longer time to complete!
    - Can't use flow control via break, continue or return.

```
// Also correct:
names.forEach(function(item) {
        console.log("item: "+item);
});
// >item: Mona
// >item: Lisa
```

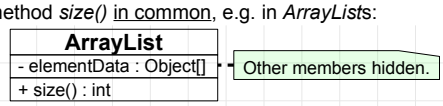
```
// Also correct:
jQuery.each(names, function(index, item) {
        console.log("item: "+item);
});
// >item: Mona
// >item: Lisa
```

22

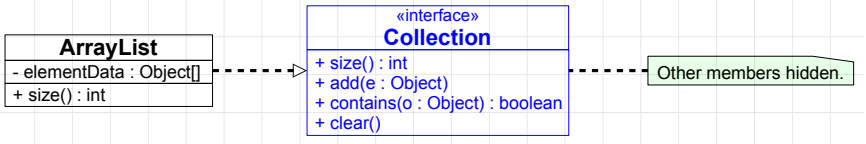
- JavaScript's order of iteration with for-in is also undefined for arrays! The order of iteration for an object's properties is usually irrelevant.

Interface-based Design of modern Collection APIs – Part I

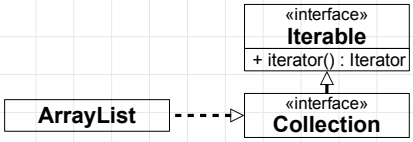
- Apart from iterability we can abstract more aspects, e.g. that a Java type is a collection per se.
  - All Java collections seem to have the method *size()* in common, e.g. in *ArrayLists*:



  - From this consideration we can derive Java's common interface that all collections have to implement: *Collection (java.util.Collection)*.
  - Java assumes collections to have a lot more in common, these commonalities are put into the interface *Collection* as methods:
    - E.g. adding elements, checking if a *Collection* contains an element, removing all elements from a *Collection* and a few more:



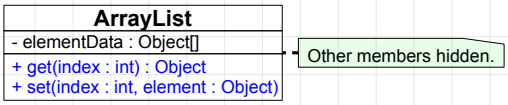
- And Java holds iterability for important enough to make *Iterable* a super interface of *Collection*:



  - The take away is good news for Java programmers: each type implementing *Collection* is *Iterable*, e.g. can be iterated with for(:).

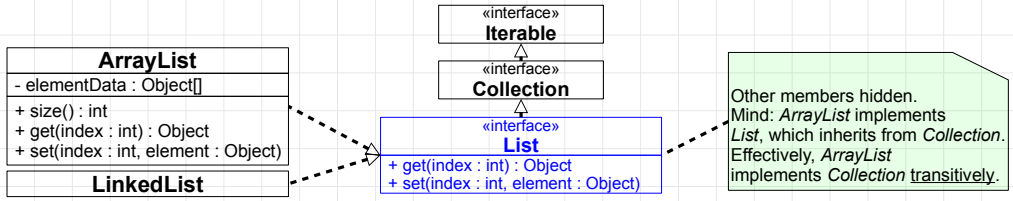
Interface-based Design of modern Collection APIs – Part II

- Apart from bare "being a collection" and iterability we can also abstract indexability of indexed collections.
  - E.g. for *ArrayList*s the index-access can be extracted (among other methods) as an important feature of indexed collections:



  - From this consideration we derive Java's common interface that indexable collections have to implement: *List (java.util.List)*.
    - Common operations are cast into methods to get and set elements at a certain index (i.e. index-access), find the index of an element and a few more.

  - And in the end *List* implements *Collection*, which implements *Iterable*: Java's indexable collections are *Collection*s and thus *Iterable*:



24

- In fact Java's *List* is relatively fat interface containing a large amount of methods. These interfaces do somewhat break the interface segregation principle (ISP).

Interface-based Design of modern Collection APIs – Part III

- That's all well and good, but what have we gained having interfaces?

- The answer is that interfaces allow abstracting usage from implementation. Let's discuss following perspectives:
  - Collection types (e.g. classes) that share the same behavior implement the same interface, but have different implementations.
  - A collection can expose different behaviors at the the same time, i.e. it can implement multiple interfaces.
  - Here an example:

```
Collection aCollection;
// Create two collections: an ArrayList and a LinkedList:
ArrayList listOfNumbers = new ArrayList(List.of(20, 13, 45, 60);
LinkedList linkedListOfNumbers = new LinkedList(List.of(0, 1, 2, 3, 4, 5, 6, 7, 8, 9));
// ArrayList implements Collection, implicit conversion is allowed:
aCollection = listOfNumbers;
System.out.println(aCollection.size());
// linkedList also implement Collection, implicit conversion is allowed:
aCollection = linkedListOfNumbers; // Implicit conversion!
System.out.println(aCollection.size());
```

- The next example shows...

```
public static void outputLength(Collection anyCollection) {
        System.out.println(anyCollection.size());
}
// Both calls work, because ArrayList and LinkedList implement Collection.
outputLength(listOfNumbers); // Implicit conversion!
outputLength(linkedListOfNumbers);
```

  - … that *ArrayList* and *LinkedList* are *Collection*s

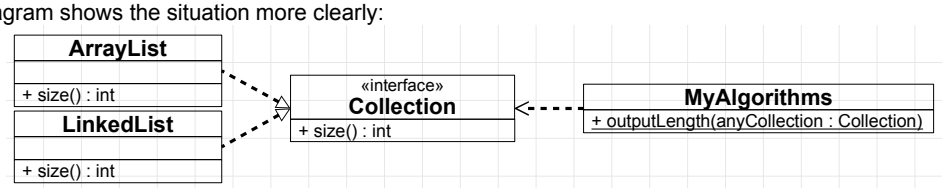
25

Interface-based Design of modern Collection APIs – Part IV

```
public static void outputLength(Collection anyCollection) {
      System.out.println(anyCollection.size());
}
outputLength(listOfNumbers);
outputLength(linkedListOfNumbers);
```

- The method *outputLength()* unleashes following views:
  - Callers: "I can pass any type that implements *Collection* to *outputLength()*!"
  - Callee (*outputLength()*): "Just give me an *Collection* object, the concrete type doesn't matter to me!"
  - The conclusion to be drawn from both views: interfaces separate interfaces of types from their implementations.

- A class diagram shows the situation more clearly:



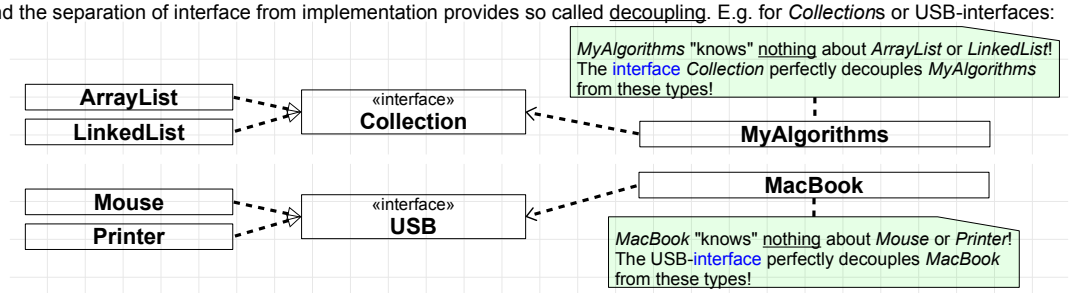
  - *outputLength()* only depends on the interface *Collection*, but not on any of its concrete implementations (*ArrayList* or *LinkedList*)!
  - This fact makes *outputLength()* a very valuable method, because it can handle any present and future *Collection* types.
  - When concrete types only depend on abstracter types, this can be a basis of the dependency inversion principle (DIP).

26

- The DIP assumes that abstractions (e.g. represented by interfaces) are stable and concrete types are not. There are, however, some concrete types, that are concrete and we can/should/must use them directly, e.g. *String*, but those are stable.
- Unimplemented methods of fat interfaces can be implemented to throw *UnsupportedOperationException* in Java. – This is called the optional feature pattern.

The Dependency Inversion Principle (DIP) – Part I

- (1) Decoupling: The DIP yields the benefit of separation of interface from implementation.
  - And the separation of interface from implementation provides so called decoupling. E.g. for *Collections* or USB-interfaces:



- (2) Static and dynamic type: For example collections can implement the same interface, but have different implementations.
  - An interface defines a concept and a class implements a concept.
  - This idea is another exploitation of the static vs. dynamic type of objects.
  - This idea is also another incarnation of the substitution principle of objects.

```
public static void outputLength(Collection anyCollection) {
      // The static type of anyCollection is Collection, the
      // dynamic type could be LinkedList or ArrayList or
      // any other implementation of Collection.
}
// Call with static type ArrayList:
outputLength(new ArrayList(List.of(20, 13, 45, 60));
// Call with static type LinkedList:
outputLength(new LinkedList(List.of(0, 1, 2, 3, 4));
```

- The word "inversion" stems from the perspective that the initial hierarchy was wrong: more abstract layers depended on more concrete layers and the inversion of this association was the solution. So the solution is that the more abstract layer defines and uses an interface that has to be implemented by the more concrete layer.

The Dependency Inversion Principle (DIP) – Part II

```
public class NoDependencyInversion {
        private ArrayList _elements = new ArrayList();

        public static void outputLength(ArrayList anyList) {
                System.out.println(anyList.size());
        }

        public ArrayList getElements() {
                return _elements;
        }
}
```

```
public class WithDependencyInversion {
        private Collection _elements = new ArrayList();

        public static void outputLength(Collection anyCollection) {
                System.out.println(anyCollection.size());
        }

        public Collection getElements() {
                return _elements;
        }
}
```

- DIP isn't only applicable for collections, in fact it is a basis of many design patterns to create maintainable oo architectures.
  - The reusability of code increases, because all objects of more concrete types can be used (*LinkedList*, *ArrayList* etc.).
  - We only need to know the methods of the abstract type's interface basically (only those of *Collection*).

- The application of DIP can be reduced to three rules:
  - Only expose abstract types in public interfaces (supertypes or interfaces).
  - Only use abstract types when objects are declared (supertypes or interfaces).
  - Concrete types are only used to create instances with the new operator and using a ctor.

The Dependency Inversion Principle (DIP) – Part III

- Let's catch a glimpse of why DIP is important for, e.g., Java's collection framework: we have downright very good reusability.
  - E.g. *ArrayList* provides a ctor and the method *addAll()* accepting *Collection*s.
  - And were an *Collection* is accepted any type implementing *Collection* can be used, so for these methods:

```
public class ArrayList { // (members hidden)

        public ArrayList(Collection c) {
                // pass
        }

        public void addAll(Collection c) {
                // pass
        }
}
```

```
// Using the ArrayList:
LinkedList numbers = new LinkedList(List.of(1, 2, 3, 4, 5));

// Construct an ArrayList with another Collection (LinkedList):
ArrayList allNumbers = new ArrayList(numbers);

ArrayList moreNumbers = new ArrayList(List.of(6, 7, 8, 9, 10));

// Add yet another Collection (ArrayList) to mergedNumbers:
allNumbers.addAll(moreNumbers);
```

- We're going to encounter DIP in many more places on our route through modern collection frameworks.

- In future lectures and examples we'll strive to using DIP wherever applicable.
  - This goes hand in hand with learning the collection frameworks along with its abstraction hierarchies.

Thank you!