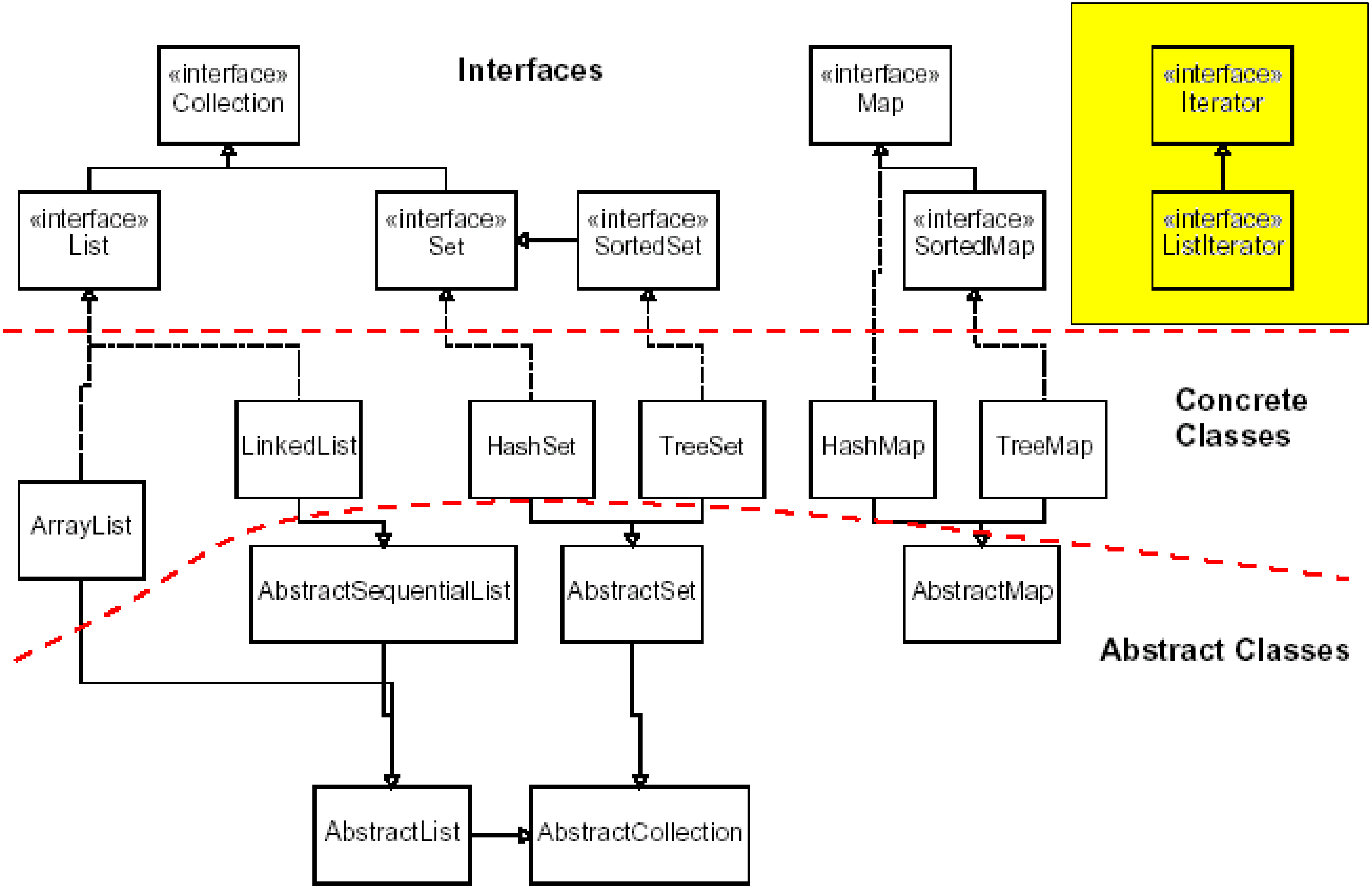
Iterators

Maria Zontak

Java collections framework



Iterator interface

`hasNext()`	returns `true` if there are more elements to examine
`next()`	returns the next element from the collection (throws a `NoSuchElementException` if there are none left to examine)
`remove()` optional	removes from the collection the last value returned by `next()` (throws `IllegalStateException` if `next()` has NOT been called yet)

Iterator

- Remembers a position within a collection, and allows to:
  - get the element at that position
  - advance to the next position
  - (optionally) remove the element at that position
- Allows to traverse the elements of a collection, regardless of its implementation → promotes abstraction

Why do we need Iterators?- The "for each" loop

```
for (type name : collection) {
    statements;
}
```

→ A clean syntax for looping over the elements of a `Set`, `List`, array, or **other collection**

```java
List<Integer> grades = new ArrayList<>(14);
…
for (int grade : grades) {
    System.out.println("Student's grade: " + grade);
}
```

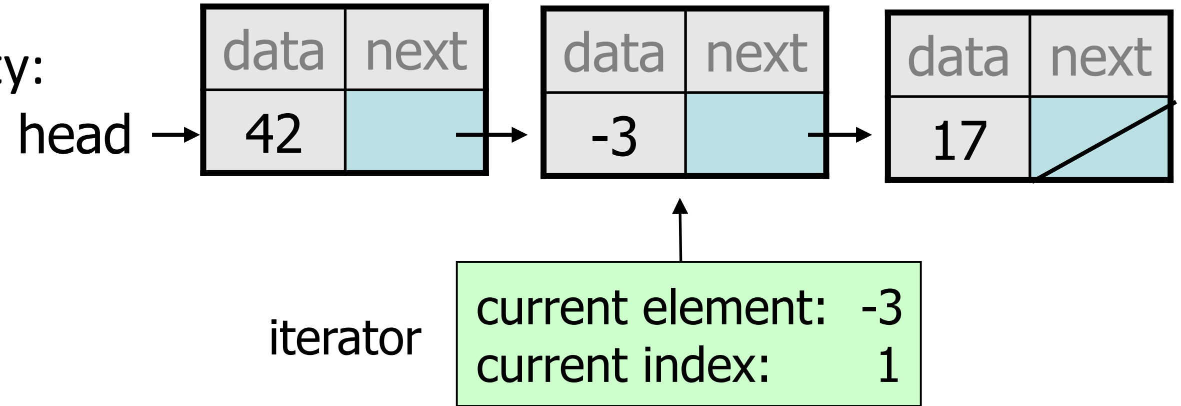
Why do we need Iterators? – Improve runtime complexity (in some cases)

The following code has two problems:

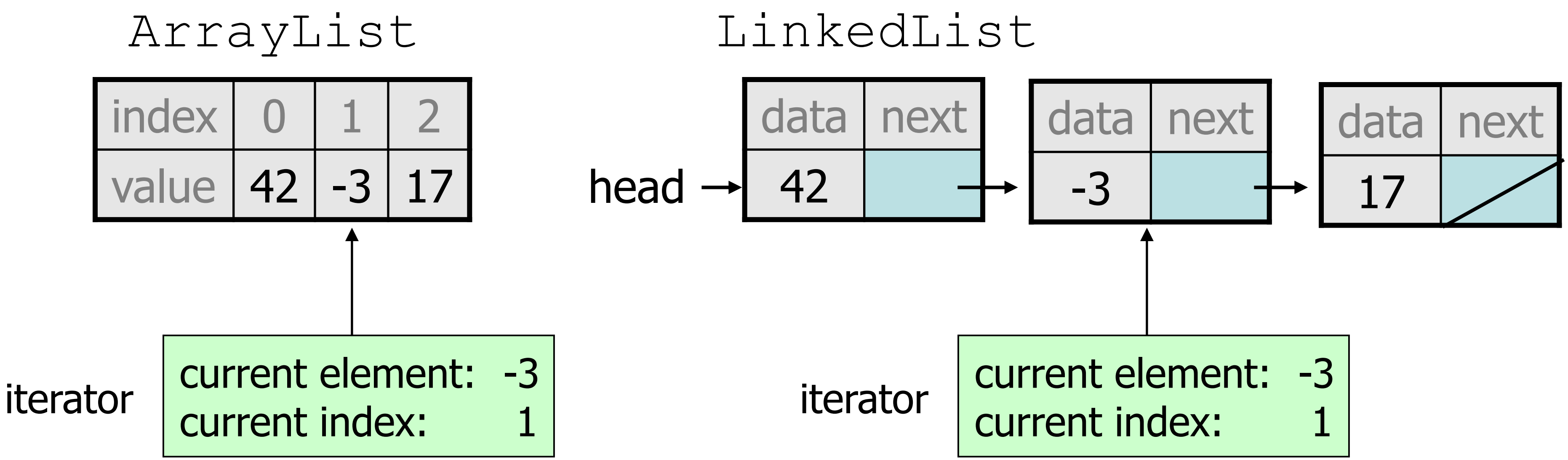
- Has a bug (where?)
- Particularly slow on **linked lists** (why?) $O(n^2)$

```
List<Integer> list = new LinkedList<>();
...//set values here
for (int i = 0; i < list.size(); i++) {
    int value = list.get(i);
    if (value % 2 == 1) {
        list.remove(i);
    }
}
```

To improve the complexity:



List Iterator



Iterator

- is **not** the same as the list (collection) that it is pointing to
- provides a **view** of the collection

Interface Iterable<E>

```
interface List<E> extends Iterable<E> {
…
}


public abstract class AList<E> implements List<E> {
…
}


public class List<E> extends AList<E> {
…
}
```

Interface `Iterable<E>`

`iterator()`	Returns an iterator `Iterator<E>` over a set of elements of type `E`.

GenericListIterator Implementation

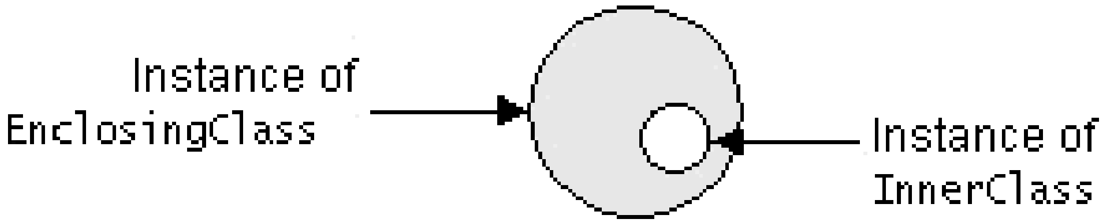
```
interface List<E> extends Iterable<E> {
…
}

public abstract class AList<E> implements List<E> {
…
    public Iterator<E> iterator() {
          return new GenericListIterator(this);
    }

}

public class List<E> extends AList<E> {
…
}
```

Let’s look at the Iterator implementation for functional List in Theo’s notes

Inner classes

- **inner class**: A class defined inside of another class.
  - can be created as `static` or non-`static` (nested)

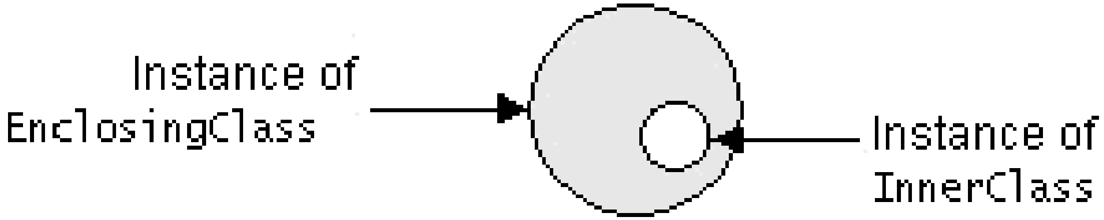
- usefulness:
  - inner classes are hidden (if `private`) from other classes (**encapsulated**)
  - inner objects can access/modify the fields of the outer object



Inner class syntax

```
// outer (enclosing) class
public class name {
    ...
    // inner (nested) class
    private class name {
        ...
    }
}
```

- Only this file can see the inner class or make objects of it.
- Each inner object is associated with the outer object that created it, so it can access/modify that outer object's methods/fields.
- If necessary, can refer to outer object as **OuterClassName**.`this`



Generics and inner classes

```
public class Foo<E> {
    private class Inner<E> {}    // incorrect

    private class Inner {}       // correct
}
```

- If an outer class declares a type parameter, inner classes can also use that type parameter.
- Inner class should NOT redeclare the type parameter. (If you do, it will create a second type parameter with the same name.)

MyLinkedListIterator Implementation

```
interface List<E> extends Iterable<E> {
…
}
public abstract class AbstractList<E> implements List<E> {
…

}

public class MyLinkedList<E> extends AbsractList<E> {
…
   public Iterator<E> iterator() {
          return new MyLinkedListIterator();
   }

   private class MyLinkedListIterator implements Iterator<E>{
    …
     }


}
```

MyLinkedListIterator Implementation

```
public class MyLinkedList<E> extends AbstractList<E> {
    ...

    private class MyLinkedListIterator implements Iterator<E> {
        private Cell current;    // current position in list

        public MyLinkedListIterator() {
            current = head;
        }

        public boolean hasNext() {
            return current != null;
        }

        public E next() {
            if (!hasNext()) throw new NoSuchElementException();
            E result = current.getVal();
            current = current.getNext();
            return result;
        }

        public void remove() {        // not implemented for now
            throw new UnsupportedOperationException("not
                               perfect; doesn't support remove");
        }
    }
}
```

Why do we need Iterators?- The "for each" loop

```
for (type name : collection) {
        statements;
}
```

→ A clean syntax for looping over the elements of a `Set`, `List`, array, or **other collection**

```
List<Integer> grades = new ArrayList<>(14);
…
for (int grade : grades) {
    System.out.println("Student's grade: " + grade);
}
```

- Is equivalent to `Iterator`

```
Iterator<Integer> intItr = grades.iterator();
while(intItr.hasNext())  {
   System.out.println(" Student's grade: " + intItr.next());
}
```

Why do we need Iterators? – Improve runtime complexity (in some cases)

The following code has two problems:

- Has a bug (where?)
- Particularly slow on **linked lists** (why?) $O(n^2)$

```
List<Integer> list = new LinkedList<>();
...//set values here
for (int i = 0; i < list.size(); i++) {
    int value = list.get(i);
    if (value % 2 == 1) {
        list.remove(i);
    }
}
```

Why do we need Iterators? –
Improve runtime complexity (in some cases)

One possible solution to fix the bug:

```
List<Integer> list = new LinkedList<>();
...//set values here
for (int i = 0; i < list.size();) {
    int value = list.get(i);
    if (value % 2 == 1) {
        list.remove(i);
    }
   else {
        i++;
   }
}
```

Fixed here

But this does NOT solve the complexity

Why do we need Iterators? – Improve runtime complexity (in some cases)

Another possible correct solution:

```
List<Integer> list = new LinkedList<>();
...//set values here
Iterator<Integer> itr = list.iterator();
while (itr.hasNext()) {
    int value = itr.next();
    if (value % 2 == 1) {
        itr.remove(); //implemented in Java
    }
}
```

Complexity now is O(n)

Note that...

- We can iterate only in one direction (unless you use `ListIterator`)
- Iteration can be done only once, till the end of the series

  → to iterate again, get a new Iterator
- Iterator returned by iterator() is **fail-fast**: if the list is structurally modified at any time after the iterator is created, in any way except through the iterator's own remove methods, the iterator will throw a `ConcurrentModificationException`.

Beyond traversing Collections

- **Iterators are not just for lists!**
- Let's look at the **Fibonacci implementation in Theo's notes**