# 2IPC0 Programming Methods

## From Small to Large Programs

Loek Cleophas
Eindhoven University of Technology
Department of Mathematics & Computer Science
Software Engineering & Technology Group

http://canvas.tue.nl/courses/473

# Overview

- Iteration Abstraction: Ch. 10.1.5 in Eck
- Iterator Design Pattern: Ch. 3 in Burris
- Testing iterators

# Iteration: Traditional Approach

Iterate over a collection: visit each item exactly once
(possibly in a prescribed order)

Ad hoc approach uses standard for-loop:

```
String[] a;  // collection represented by an array

for (int i = 0; i != a.length; ++ i) { ... a[i] ... }


ArrayList<String> list;  // collection represented by ArrayList

for (int i = 0; i != list.size(); ++ i) { ... list.get(i) ... }
```

---

How to iterate over a `Set`?

# Iteration: Decomposition

| **Operation** | on `String[] a` | on `List<String> list` |
|---|---|---|
| Initialize the iteration | `int i = 0` | `int i = 0` |
| Check whether iteration is done | `i != a.length` | `i != list.size()` |
| Retrieve next item to be visited | `a[i]` | `list.get(i)` |
| Step to next item | `++ i` | `++ i` |

# Iteration Abstraction

Iteration Abstraction: Facility for iteration that abstracts from

- type of collection
- type of its items
- implementation details of how to accomplish iteration
  e.g. choosing an order

Iteration abstraction provides a uniform* solution.

*Unfortunately, details of iteration abstraction are programming-language specific.

## Enhanced `for` Statement, or the 'for-each' Loop

New in Java 5.0:

**`for`** ( *Type Identifier* **`:`** *Expression* ) *Statement*

*Identifier* is a local variable, whose values are of the declared *Type*, iterating over the collection defined by *Expression*.

Read as 'for each . . . in . . . '.

*Expression* can be

- an array, e.g. obtained through method `values()` from an **`enum`**
- a type that implements interface `Iterable` (a subtype of `Iterable`)

## 'for-each' Loop Examples

- Iterate over an array:

```
float[] a;

for (float f : a) { ... f ... }
```

- Iterate over the values in an **enum**:

```
private enum PrimColor { RED, GREEN, BLUE }

for (PrimColor c : PrimColor.values()) { ... c ... }
```

- Iterate over a type that implements `Iterable`

```
HashSet<Card> cards;

for (Card card : cards) { ... card ... }
```

## Semantics of enhanced `for` Statement: `for ( T v : E ) S`

- if `E` is an array:

```
T[] r = E;  // fresh identifier r; E evaluated only once
    // r is a reference, and not a copy
for (int i = 0; i != r.length; ++ i) { // fresh identifier i
    T v = r[i];
    S  // operates on v  (N.B. i is invisible/inaccessible)
}
```

- if `E` is subtype of `Iterable<T>` (i.e., `E` provides a default iterator):

```
for (Iterator<T> iter = E.iterator(); iter.hasNext(); ) {
    T v = iter.next();
    S  // operates on v
}
```

# Iteration: Decomposition with Iterator

| **Operation** | on `Iterable<T> E` |
|---|---|
| Initialize the iteration | `Iterator<T> iter = E.iterator()` |
| Check whether iteration is done | `iter.hasNext()` |
| Retrieve next item to be visited | `T v = iter.next()` |
| Step to next item | side-effect of `iter.next()` |

# Interfaces `Iterable` and `Iterator`

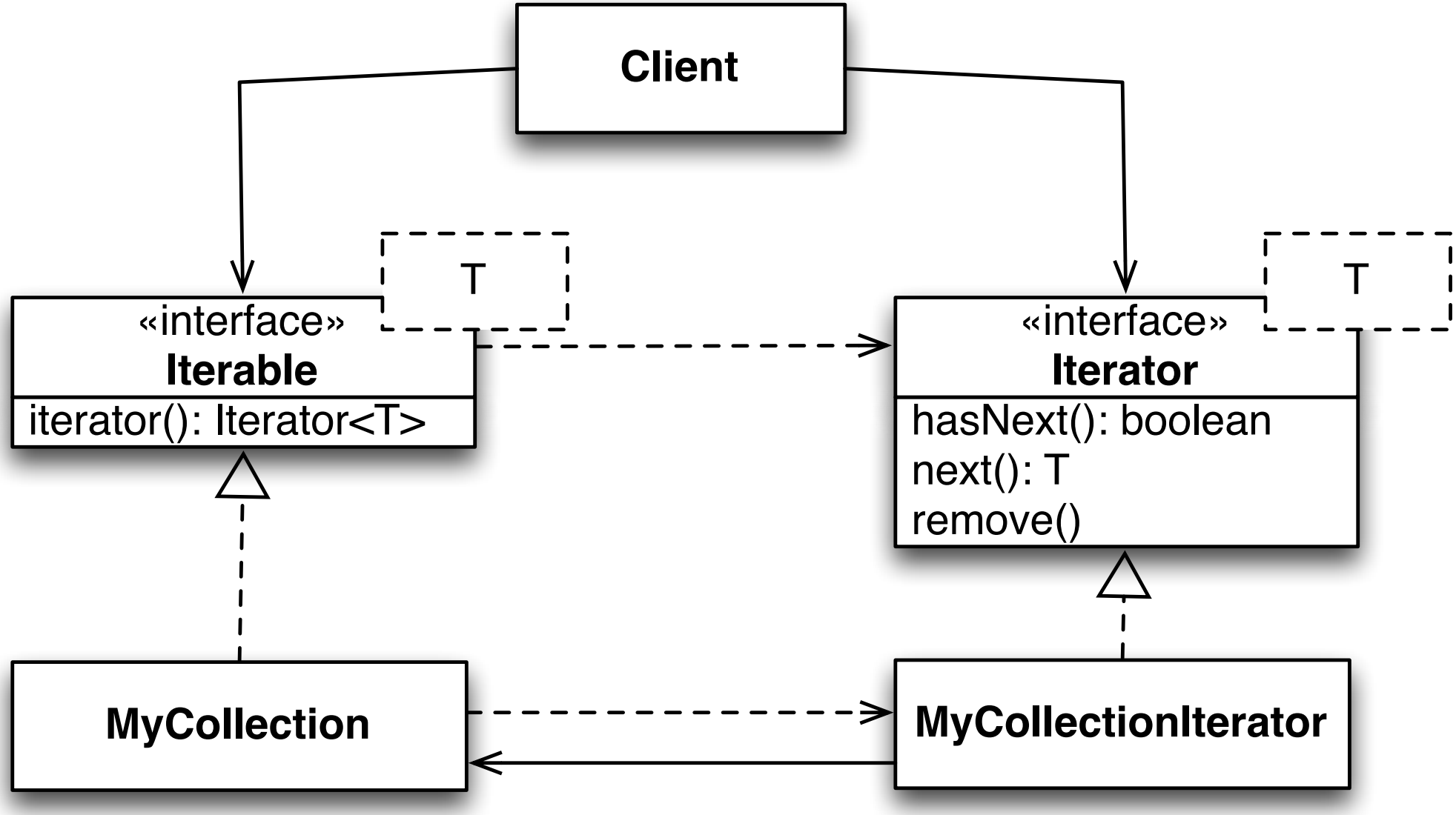
```
public interface Iterable<T> {
    /** Returns new iterator for collection over type T. (Factory Method) */
    Iterator<T> iterator();
}

public interface Iterator<T> {
     /** Returns whether a next item is available. */
    boolean hasNext();

    /** Returns the next item in the iteration.
     *  @pre hasNext()
     *  @throws NoSuchElementException if not available
     */
    T next();

    /** Removes current element from collection. */
    void remove();  // optional (not treated here; see book)
}
```

# Iterator Design Pattern: Class Diagram



## Iterator Design Pattern: Context and Intent

- Given a class (ADT) that manages a collection,
- provide a way to traverse the elements of the collection,
- such that code for iterating is separate from and loosely coupled
- both to client code and to the collection.

## Iterator<E> versus Iterable<E>

- `Iterator<E>`:
  - holds state of one specific iteration over `E`
  - cannot be reused after the iteration terminates
- `Iterable<E>`
  - represents a collection over `E` that
    can (only) be iterated over with **`for`**-each statement
  - can be reused for multiple iterations
  - works via its `iterator()` that returns an `Iterator<E>`

# Iterator and `iterator` method that creates an iterator

```
 1 import java.util.Iterator;
 2
 3 /** C is collection over type T (similar to ArrayList<T>) */
 4 public class C implements Iterable<T> {
 5     private ... r;  // data representation of the collection
 6
 7     public Iterator<T> iterator() { return new ForIterator(...); }
 8
 9     private class ForIterator implements Iterator<T> {
10         private ... s; // private state of iterator
11
12         ForIterator(...) { ... s ... } // constructs initial state
13
14         public boolean hasNext() { ... r, s ... }
15
16         public T next() throws NoSuchElementException { ... r, s ... }
17     }
18 }
```

## Iterator Pattern, Iteration Abstraction, Language Support

- Java language provides further abstraction than Iterator pattern

  **`for`**-each statement hides local variable for the iterator object and how it is manipulated (anonymous iterator)

  Cannot invoke `remove()` in for-each loop.

## Iterators: standard, and in general

- Standard iterator to be used in for-each loop:
  - Collection class must implement interface `Iterable<T>` and define method `iterator()` returning an `Iterator<T>`.
  - See example `Range.java`
- Iterator in general (not usable in for-each loop):
  - Collection class must define one or more methods returning an `Iterator<T>`.
  - The name of the iterator constructor is not prescribed.
  - See example `DownRange.java`
  - Alternative: return `Iterable<T>` (`DualRange.java`)

## Iterators: Loose Ends

- Each iteration uses a new iterator object, which cannot be 'reused'.

  Can return an `Iterable<T>` (an iterable view) instead of `Iterator<T>` to allow multiple iterations over a collection. See `DualRange.java`

- Multiple iterators over the same collection can be active at the same time

- E.g. iterations over the same collection can be nested.

  This involves multiple iterator objects.

- The iterator object in a for-each statement is hidden.

- `for (float f : a) { f = 1; }` does not initialize array `a` to `1`s

## Design Issues

- Data types that store a collection of items typically offer one or more iterators.
- Mutable collections require that the loop using an iterator does *not* change the collection 'outside' the iterator.
  Doing so will result in a `ConcurrentModificationException`.
- The `Iterator` interface offers one safe modification operation: `remove()`; not always implemented; not usable in for-each loop.
- For use in the for-each statement, the collection must implement `Iterable<...>` and provide the iterator constructor `iterator()`.
- Other iterators must be used through standard loops using `hasNext()` and `next()`.

## Implementation Issues

- Java iterator has operation `next()` that is command and query.
  - Command: steps iterator to next item
  - Query: return next item

  Cannot retrieve current item more than once!
- `hasNext()` need not be called before `next()`
- `hasNext()` can be called more than once before `next()`
- Administrate as little as possible to do iteration

  Avoid copying data from the collection

# Testing Issues

?

## Testing Issues

- Check that every element is returned at least once (no misses).
- Check that every element is returned at most once (no duplicates).
- Check that it works without calls to `hasNext()`.
- Check that it works with multiple calls to `hasNext()`.

## Iterator Anti-Patterns

- Give access to internal data representation, to let client iterate.

  In **class** `IntRelationListOfSets`, include query

```
public Set<Integer> getRelated(int a) {
    return relation.get(a);
}
```

  This leaks the representation: client can break rep invariant...

  Safe but inefficient: `return new HashSet(relation.get(a));`

- Include operations for iteration in collection class itself.

```
public int getFirstRelated(int a) { ... }
public int getNextRelated(int a) { ... }
```

  Then you cannot have multiple iterations active concurrently.

## Homework

- Read in Eck: 5.3, 5.7-5.8, 10.1.5, 10.2.1 and 10.2.3
- Read in Burris: Chapter 3 (Iterator)
- Graded assignment #1 (two parts!)
- Next up will be Exercises 18 and 20 of *Test Driven Development*: *Iterable IntRelation* and *Generic Relation*

## Summary

- An *iterator object* provides *iteration abstraction*:

  visit each element of a collection exactly once,

  without worrying about the underlying traversal mechanism.
- An `Iterator<T>` object provides methods `hasNext()` and `next()`.
- The for-each loop uses `iterator()` from `Iterable<T>`.