Programming Languages and Techniques (CIS120)

Lecture 28

April 2, 2014

Generics and Collections

Announcements

- Midterm 2 is Friday
  - Towne 100 last names A - L
  - DRLB A1 last names M - Z

- Review sessions:
  - Tonight! 7:00 – 9:00, Towne 100
  - Lab this week is review (bring questions!)

- Reading: Chapters 26 & 27
- Looking for TAs for Fall 2014! See piazza post for details.

Java Generics

Example: Queues in Java

Mutable Queue ML Interface

```
module type QUEUE =
sig
  (* type of the data structure *)
  type 'a queue

  (* Make a new, empty queue *)
  val create : unit -> 'a queue

  (* Add a value to the end of the queue *)
  val enq : 'a -> 'a queue -> unit

  (* Remove the front value and return it (if any) *)
  val deq : 'a queue -> ‘a

  (* Determine if the queue is empty *)
  val is_empty : ‘a queue -> bool

end
```

Mutable Queues in Java

```
module type QUEUE =
sig
  type 'a queue

  val create : unit -> 'a queue
  val is_empty :
    'a queue -> bool

  val enq :
    'a -> 'a queue -> unit

  val deq : 'a queue -> 'a
end
```

```
public interface Queue<E> {




   public boolean isEmpty ();

   public void enq (E elt);

   public E deq ();

}
```

Implementing Queues: QNodes

```
type 'a qnode = {
    v: 'a;
    mutable next : 'a qnode option
}

type 'a queue = {
    mutable head : 'a qnode option;
    mutable tail : 'a qnode option
}
```

```
public class QNode<E> {
  public final E v;
  public QNode<E> next;
  public QNode(E v0, QNode<E> next0) {
     this.v = v0;
     this.next = next0;
   }
}
```

- The QNode class will only be used internally by our Queue implementation.
- Can make instance variables public because we will *encapsulate* this data structure: no clients of the queue can ever access a queue node.
- We make the element final because the queue will never need to change it.

Mutable Queues in Java

```
public class QueueImpl<E>
      implements Queue<E>{

  private QNode<E> head;
  private QNode<E> tail;

  public QueueImpl () {
    head=null; tail=null;
  }

  public boolean isEmpty() {
    return (head == null);
  }

  public void enq(E elt) { … }

  public E deq() { … }
}
```

```
public interface Queue<E> {

   public boolean isEmpty ();

   public void enq (E elt);

   public E deq ();

}
```

Why Generics?

Subtype Polymorphism

```
public interface ObjQueue {
  public void enq(Object o);
  public Object deq();
  public boolean isEmpty();

  …
}
```

```
ObjQueue q = …;

q.enq(" CIS 120 ");
___A___  x = q.deq();
```

What type for A?

1. String
2. Object
3. ObjQueue
4. None of the above

Subtype Polymorphism

```
public interface ObjQueue {
  public void enq(Object o);
  public Object deq();
  public boolean isEmpty();

  …
}
```

```
ObjQueue q = …;

q.enq(" CIS 120 ");
Object  x = q.deq();
System.out.println(x.trim());
```

← Does this line type check

1. Yes
2. No
3. It depends

Subtype Polymorphism

```
public interface ObjQueue {
  public void enq(Object o);
  public Object deq();
  public boolean isEmpty();

  …
}
```

```
ObjQueue q = …;

q.enq(" CIS 120 ");
Object  x = q.deq();
//System.out.println(x.trim());
q.enq(new Point(0.0,0.0));
___B___  y = q.deq();
```

What type for B?

1. String
2. Object
3. ObjQueue
4. None of the above

Generics (Parametric Polymorphism)

```
public interface Queue<E> {
  public void enq(E o);
  public E deq();
  public boolean isEmpty();

  …
}
```

```
Queue<String> q = …;

q.enq(" CIS 120 ");
___A___  x = q.deq();
System.out.println(x.trim());
q.enq(new Point(0.0,0.0));
___B___  y = q.deq();
```

```
// What type for A?  String
// Is this valid?    Yes!
// Is this valid?    No!
```

The Java Collections Library

A case study in subtyping and generics

(Also very useful!)

Java Packages

- Java code can be organized into *packages* that provide namespace management.
  - Somewhat like OCaml's modules
  - Packages contain groups of related classes and interfaces.
  - Packages are organized hierarchically in a way that mimics the file system's directory structure.
- A .java file can *import* (parts of) packages that it needs access to:

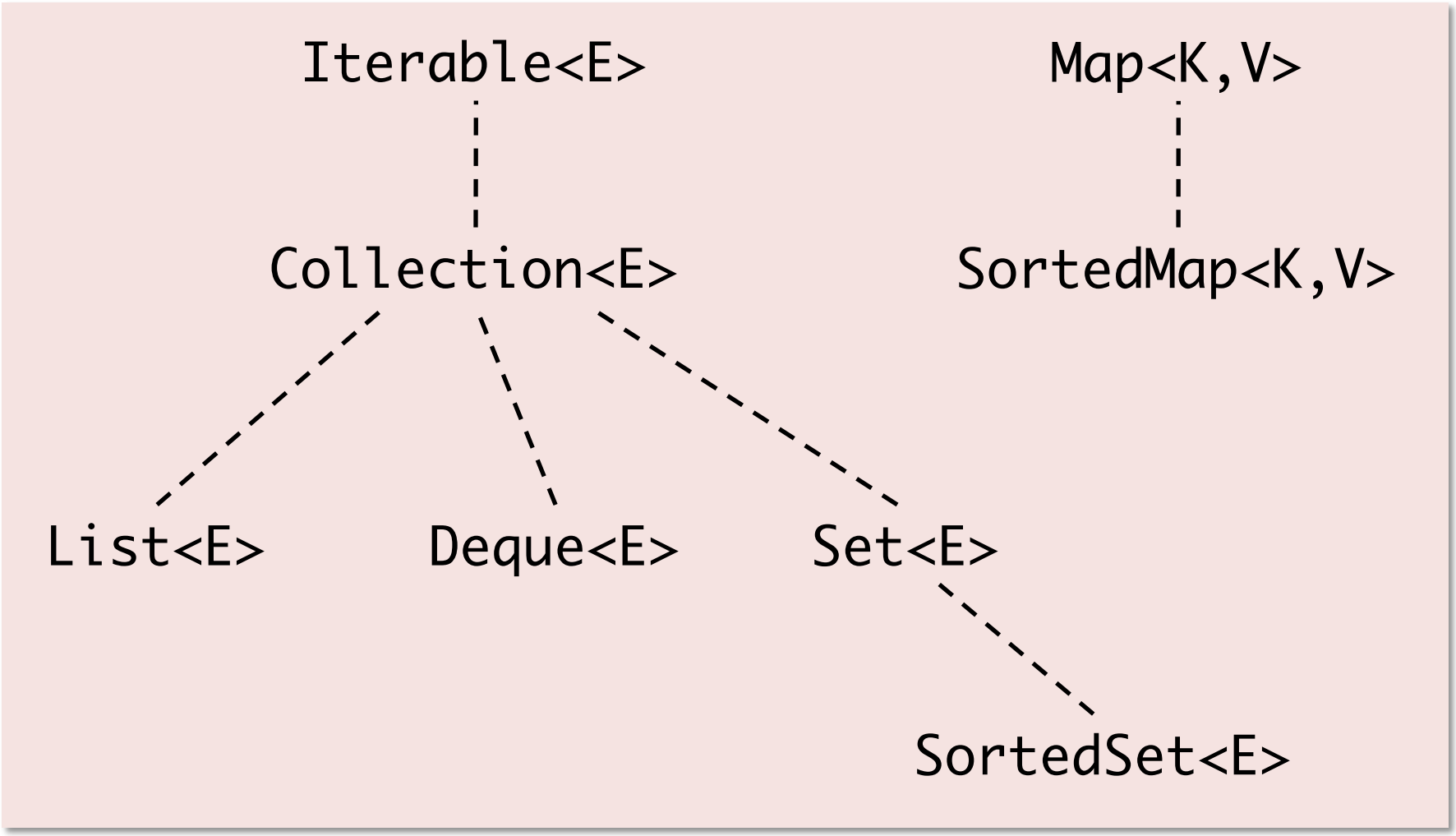
```
import org.junit.Test;     // just the JUnit Test class
import java.util.*;        // everything in java.util
```

- Important packages:
  - java.lang, java.io, java.util, java.math, org.junit
- See documentation at: http://download.oracle.com/javase/7/docs/api/index.html

Reading Java Docs

http://docs.oracle.com/javase/7/docs/api/java/util/package-summary.html

Interfaces* of the Collections Library



*not all of them!

Collection<E> Interface (Excerpt)

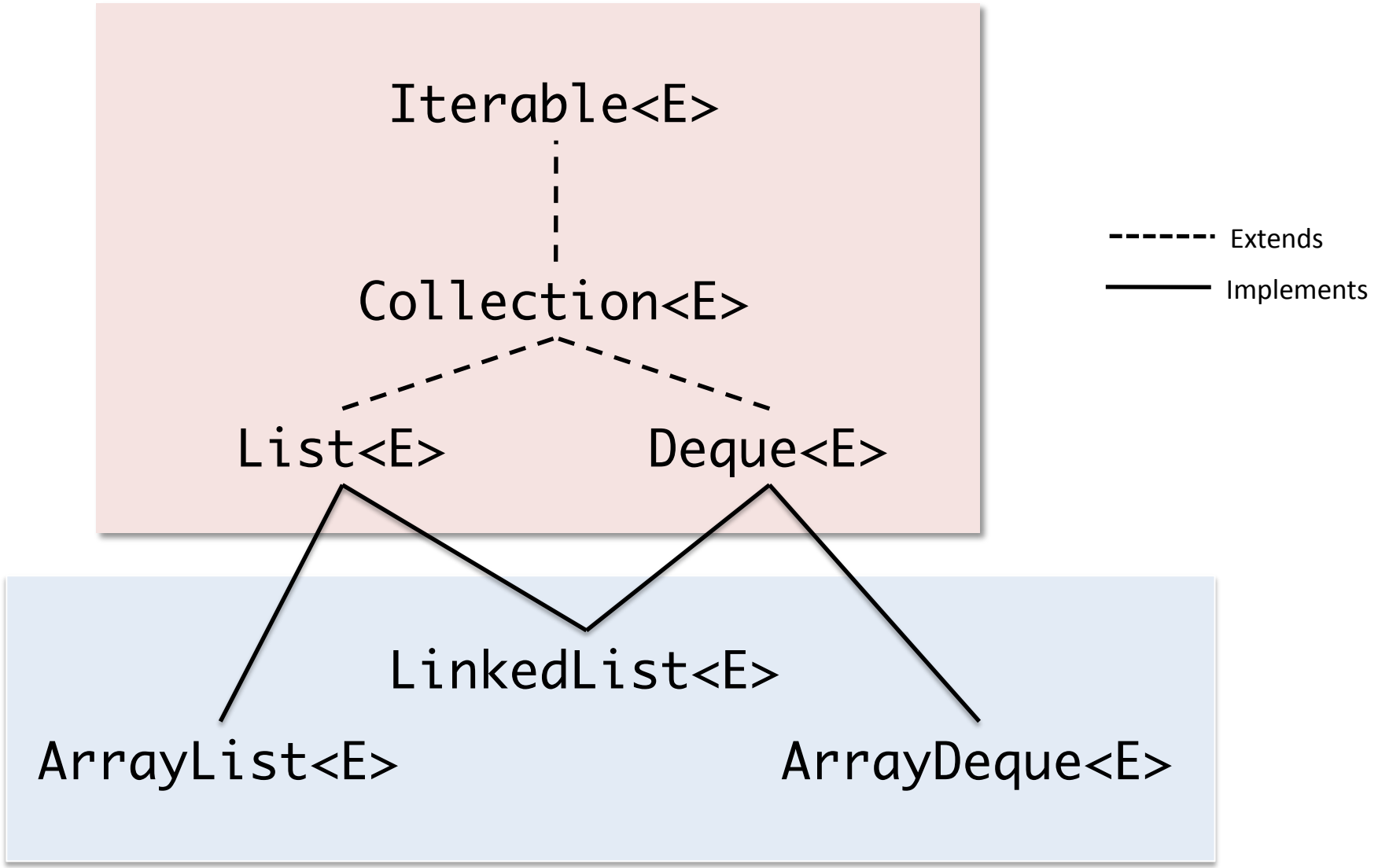
```
public interface Collection<E> extends Iterable<E> {
  // basic operations
  int size();
  boolean isEmpty();
  boolean add(E o);
  boolean remove(Object o);     // why not E?*
  boolean contains(Object o);

  // bulk operations
  …
}
```

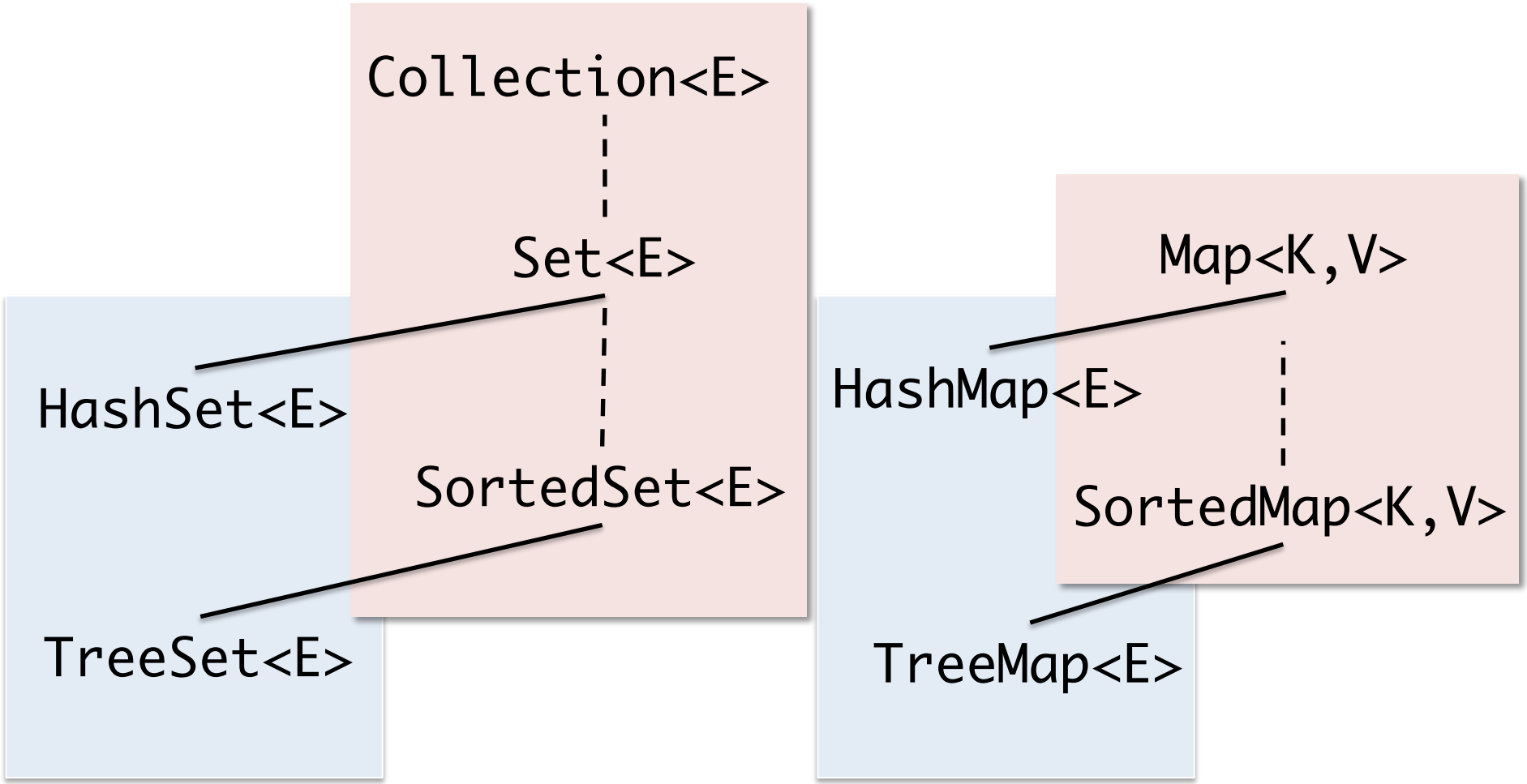
- We've already seen this interface in the OCaml part of the course.
- Most collections are designed to be *mutable* (like queues)

* Why not E? Internally, collections use the `equals` method to check for equality – membership is determined by o.equals, which does not have to be false for objects of different types. Most applications only store and remove one type of element in a collection, in which case this subtlety never becomes an issue.

Sequences


Iterable<E>
Collection<E>
List<E>
Deque<E>
LinkedList<E>
ArrayList<E>
ArrayDeque<E>
Extends
Implements

Sets and Maps*



*Read javadocs before instantiating these classes! There are some important details to be aware of to use them correctly.

Iterating over collections

iterators, while, for, for-each loops

Iterator and Iterable

```
interface Iterator<E> {
    public boolean hasNext();
    public E next();
    public void delete();   // optional
}
```

```
interface Iterable<E> {
    public Iterator<E> iterator();
}
```

Challenge: given a List<Book> how would you add each book's data to a catalogue using an iterator?

While Loops

syntax:

```
// repeat body until condition becomes false
while (condition) {
   body
}
```

boolean *guard* expression

statement

example:

```
List<Book> shelf = …  // create a list of Books

// iterate through the elements on the shelf
Iterator<Book> iter = shelf.iterator();
while (iter.hasNext()) {
    Book book = iter.next();
    catalogue.addInfo(book);
    numBooks = numbooks+1;
}
```

For-each Loops

syntax:

```
// repeat body for each element in collection
for (type var : coll) {
   body
}
```

element type

array or instance of Iterable<E>

example:

```
List<Book> shelf = …  // create a list of books

// iterate through the elements on a shelf
for (Book book : shelf) {
    catalogue.addInfo(book);
    numBooks = numbooks+1;
}
```

For-each Loops (Cont'd)

Another example:

```
int[] arr = …  // create an array of ints

// count the non-null elements of an array
for (int elt : arr) {
    if (elt != 0) { cnt = cnt+1; }
}
```

For-each can be used to iterate over arrays or any class that implements the `Iterable<E>` interface (notably `Collection<E>` and its subinterfaces).