Introduction to Object Oriented Programming

Roy Schwartz, The Hebrew University (67125)

Lecture 6:

Generics and Collections

Last Week

- Reuse Mechanisms
- Casting
- Intro to Design Patterns
- Façade Design Pattern

Lecture 6a: Overview

- What is a collection
- Intro to Generics
- Collection Interfaces
- Collection Implementations
- Hash Tables
- Iterators

What is a Collection?

- A **collection** is an object that groups multiple elements into a single unit
  - A **data structure**
  - Sometimes called a **container**
- Collections are used to **store**, **retrieve**, **manipulate**, and **communicate** aggregate data
- Often represent a **natural group**
  - A poker hand (a collection of **cards**), a mail folder (a collection of **emails**), an address book (a mapping of **names** to **phone numbers**)

The Collections Framework

- A **collections framework** is an architecture for representing and manipulating collections
- All collections frameworks contain the following:
  - **Interfaces**
  - **Implementations**
  - **Algorithms**

In java:
import java.util.*

The Collections Framework

Parts

- **<u>Interfaces</u>:** Abstract collections
  - Allow manipulation **independently** of the implementation

Map, Set, List, …

- **<u>Implementations</u>:** Concrete interface implementations
  - ***Reusable data structures***

TreeMap, HashSet, LinkedList, …

- **<u>Algorithms</u>:** Perform useful computations on collection objects
  - Are **polymorphic**: same method works with many different implementations. ***Reusable functionality***

binarySearch(), sort(), shuffle(), …

HISTORY

V1.0(‘96): Vector, Dictionary, Hashtable, Stack, Enumeration

V1.2(‘98): Collection, Iterator, List, Set, Map, ArrayList, HashSet, …

V1.4(‘02): RandomAccess, IdentityHashMap, LinkedHashMap,

V1.5(‘04): Queue, java.util.concurrent, ...

V1.6(‘06): Deque, ConcurrentSkipListSet/Map, ...

V1.7(’11): TransferQueue, LinkedTransferQueue

V1.8(’14): Many enhancements to the collections framework

Benefits of the Java Collections Framework

- **Reduces programming & design effort:**
  - Programmer is free to concentrate on her concrete program
  - No low-level "plumbing” required
  - No need to reinvent the wheel each time
- **Increases program speed and quality**:
  - High-performance, high-quality implementations
  - Many bugs are prevented, or at worst, are more easily discovered
  - The various interface implementations are interchangeable. Programs can be easily tuned by switching implementations

Benefits of the Java Collections Framework (cont’d)

- **Allows interoperability among unrelated APIs**:
  - A way for different APIs to pass collections back and forth
  - A common language for all programs
- **Reduces effort to learn and to use new APIs:**
  - Many APIs naturally take collections as input/output

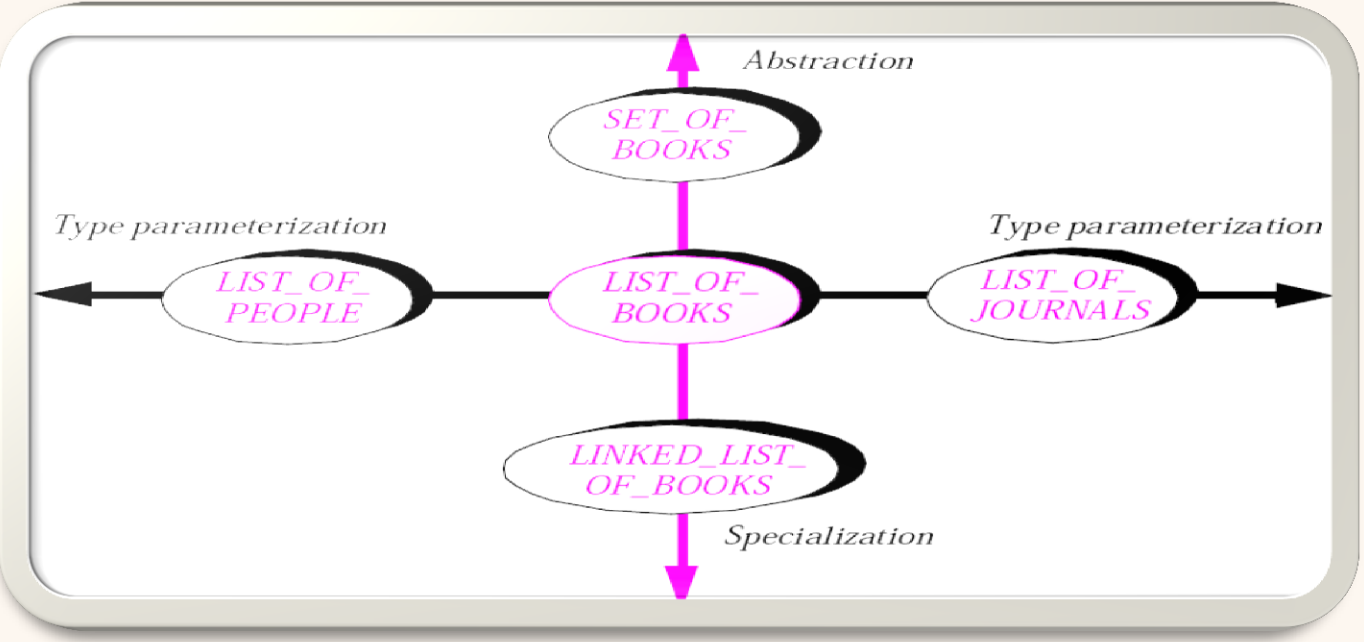
Arrays

- Arrays are a common type of data structure
  - Supported by the java language
- Arrays are hardly enough for what we need from a data structure
  - Non resizable
  - Impossible to modify their behavior (prohibit duplicates, force sorting)
  - …
- The Collections framework introduces many data structures that provide answers to these problems

Lecture 6b: Overview

- What is a collection
- Intro to Generics
- Collection Interfaces
- Collection Implementations
- Hash Tables
- Iterators

Introduction to Genericity



Generic API

- A generic class defines one or more type parameters
  - List<**E**>
  - Map<**K**,**V**>
- APIs of generic classes can make use of these parameters
  - **E** List.get(**int** index)
  - **V** Map.put(**K** key, **V** value)
- Generic parameters can be replaced by any (non-primitive) java type to create a supposedly new class
  - A list of Strings, a list of Integers, a list of lists, …

Instances of Generic Classes

- When we create an object of a generic class, we set **concrete** parameters
    - LinkedList<**String**> myList = **new** LinkedList<**String**>();
    - HashMap<**String**,**Double**> map = **new** HashMap<**String**,**Double**>();
- Consequently, when using these objects, we replace the parameters with their concrete values
    - **String** s = myList.get(0);
    - map.put(“hello”, **new** **Double**(5.7));

Examples

```
// A list of strings
LinkedList<String> list = new LinkedList<String>();
list.add("hello");
String s = list.get(0);
list.add(new Double(3.14));          // Compilation error – list is a list of strings
Double d = list.get(0);               // Compilation error

// A list of doubles
LinkedList<Double> list2 = new LinkedList<Double>();
list2.add(new Double(2.71));
Double d = list2.get(0);
list2.add("hello");                   // Compilation error
```

Generics and Collections

- All the core collection components are **generic**
  - I.e., they all have a generic type parameter
  - For example, this is the declaration of the *LinkedList* class:

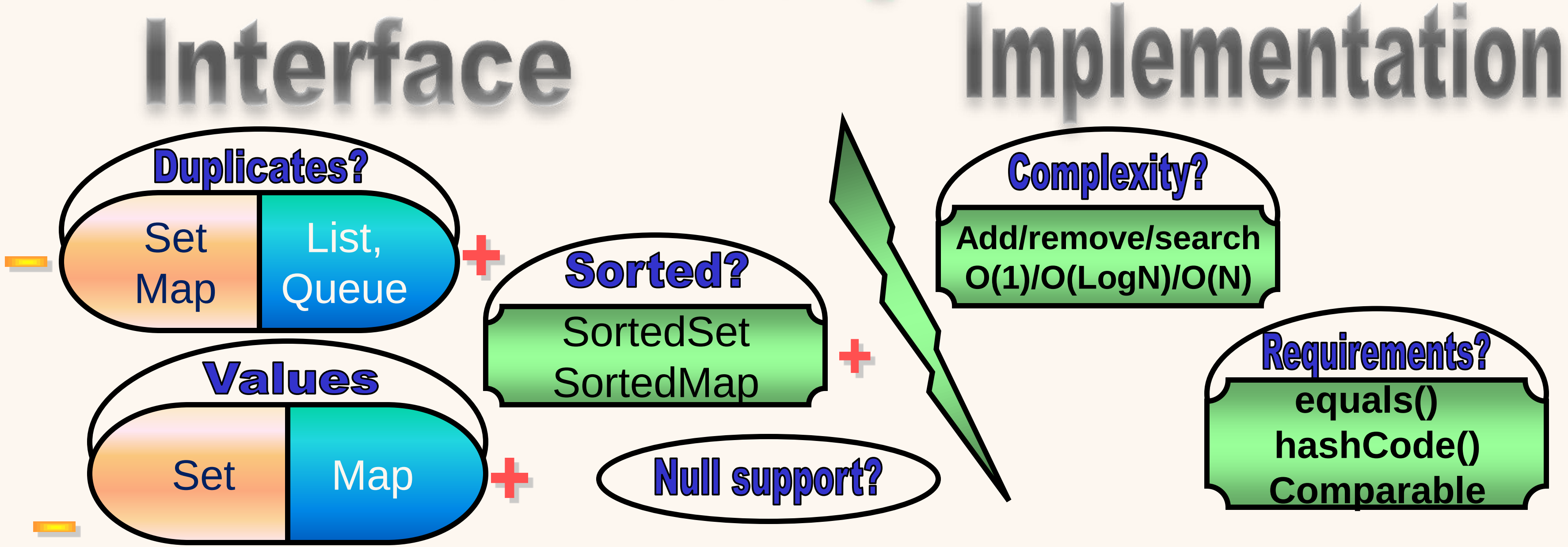
```
public class LinkedList<E>
```

Lecture 6c: Overview

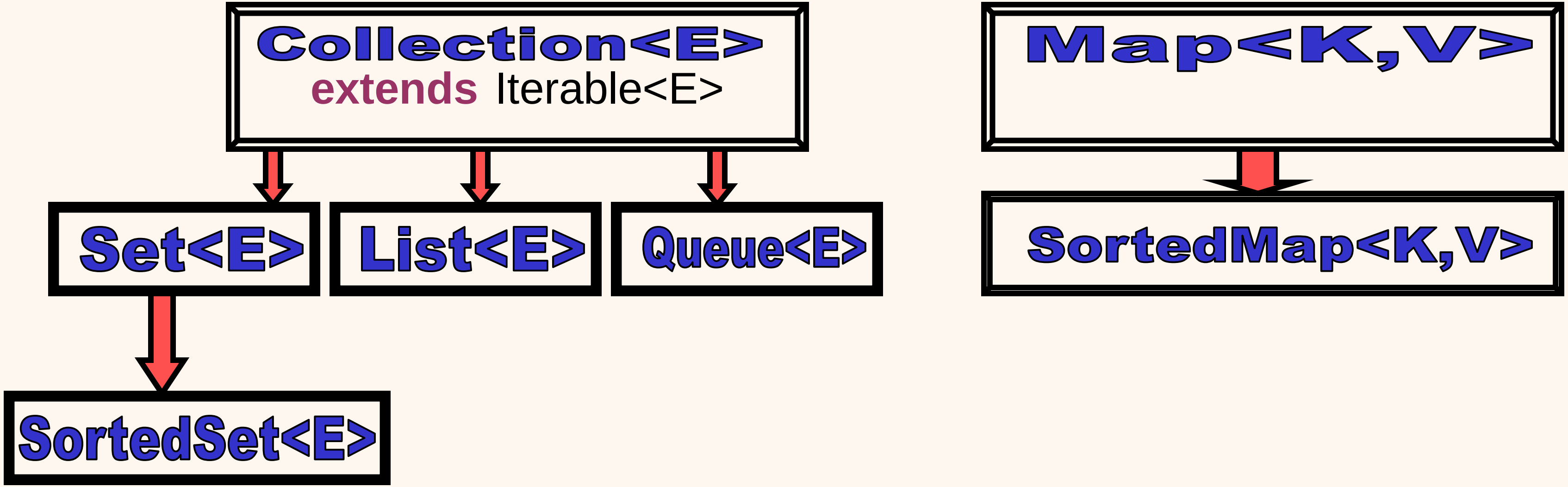
- What is a collection
- Intro to Generics
- Collection Interfaces
- Collection Implementations
- Hash Tables
- Iterators

Interface vs. Implementation

How to choose the right collection?



7 Basic Collection Interfaces



Collection<E> Interface:

Basic, general, flexible operations to retrieve/add/remove members

```
public interface Collection<E> extends
            Iterable<E> {
    // Basic operations
    int size();
    boolean isEmpty();
    boolean contains(Object element);
    boolean add(E element); //optional
    boolean remove(Object element); //optional
    Iterator<E> iterator();

    // Bulk operations
    boolean containsAll(Collection<?> c);
    //optional Bulk operations
    boolean addAll(Collection<? extends E> c);
    boolean removeAll(Collection<?> c);
    boolean retainAll(Collection<?> c);
    void clear();

    // Array operations
    Object[] toArray();
    <T> T[] toArray(T[] a);
}
```

Core Collection Interfaces

List<E>

- **List** — an ordered collection (sometimes called a *sequence*)
  - Insert / access elements by their **index**
    - **get**() / **set**() / **indexOf**() methods
  - Lists are not (necessarily) **sorted**
  - Lists can contain duplicate elements

Core Collection Interfaces

Queue<E>

- **Queue** – an ordered collection that allows access only to one of its elements (the **head** of the queue)

– Queues provide
  - *insertion* (**push**())
  - *inspection* of the head element in the queue (**peek**())
  - *extraction* of the head element (**pop**())

Core Collection Interfaces

Queue<E>

- The ***head*** is determined by some criterion
  - Queues are typically, **but do not necessarily, FIFO** (first-in-first-out)
- Other kinds of queues may use different placement rules
  - For example, **priority queues,** which order elements according to a supplied comparator or the elements' natural ordering
- Whatever the ordering used, **peek**() and **pop**() return the ***head*** of the queue

Core Collection Interfaces

Set<E>

- **Set** — a collection that cannot contain duplicate elements
  - Models the mathematical set abstraction
  - No general order of elements
  - Useful in representing real life sets, such as a deck of cards, a list of courses and the processes running on a machine
- **SortedSet** — a sorted version of the **Set interface**
  - Several additional operations are provided
  - Used for naturally ordered sets (set of words, set of candidates)

Core Collection Interfaces

Map<K,V>

- **Map** — an object that maps keys to values
  - Used for collections of key/value pairs
    - student id ➔ student name
  - Cannot contain duplicate **keys**
  - **Can** contain duplicate values
  - Maps are analogous to Sets (a Map is a Set of with a value associated with each key)
  - No general order for map keys (or values)

Core Collection Interfaces

SortedMap<K,V>

- **SortedMap** — a map where the **keys** are ordered
  - Map analog of SortedSet
  - Used for naturally sorted collections of key/value pairs
    - Dictionaries, telephone directories

Lecture 6d: Overview

- What is a collection
- Intro to Generics
- Collection Interfaces
- Collection Implementations
- Hash Tables
- Iterators

Collection Conventions

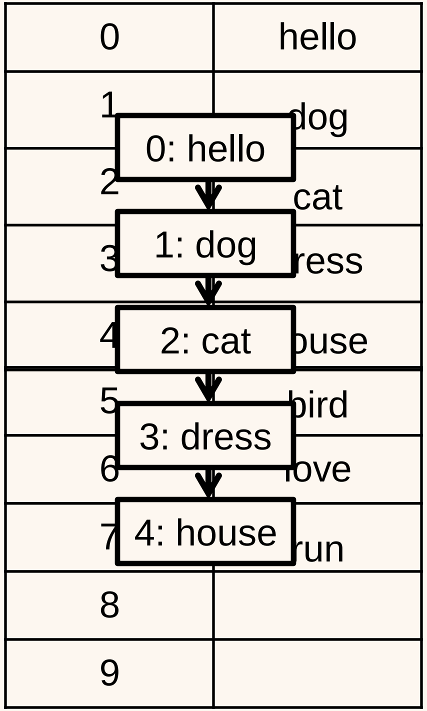
Constructor

- All implementations should provide two "standard" constructors:
  - A void (no arguments) constructor – creates an empty collection
  - A constructor with a single argument of type *Collection* – creates a new collection with the same elements as its argument
    - Allows the user to copy any collection, producing an equivalent collection of the desired implementation type (***Copy Constructor***)
- There is no way to enforce this convention
  - Interfaces cannot contain constructors
  - Nevertheless, all java implementations comply to this convention

Collection Implementations

List<E>

- **ArrayList<E>** – Resizable-array implementation
  - get(), set() – constant time
  - contains(), indexOf(), remove() – O(n)
  - add() – **amortized** constant time
    - adding *n* additional elements requires O(*n*) time
- **LinkedList<E>** – Linked list implementation
  - add() – constant time
  - get(), set(), contains(), indexOf(), remove() – O(n)
  - Generally requires less memory than ArrayList



Collection Implementations

Set<E>

- **TreeSet<E>** – a tree based implementation
  - Elements are ordered
  - add(), remove(), contains() – O(log(n)) time
- **HashSet<E>** – a hash table java implementation
  - No guarantees as to the iteration order of the set
    - In particular, no guarantee that this order remains the same over time
  - add(), remove(), contains() – *average* constant time

Collection Implementations

Map<K,V>

- HashSet<E> ➔ HashMap<K,V>

- TreeSet<E> ➔ TreeMap<K,V>

Computational Complexity

	Add	Remove	Get by index	Contains	Iteration
ArrayList	O(1)*	O(N)	O(1)	O(N)	O(N)
LinkedList	O(1)	O(N)	O(N)	O(N)	O(N)
HashSet	O(1) avg	O(1) avg	–	O(1) avg	**O(T)****
***TreeSet**	O(logN)	O(logN)	–	O(logN)	O(N)

* *amortized constant time*, that is, adding *n* elements requires O(*n*) time

** see later

Working with Collections

Static library with many useful algorithms

Searching…

```
int pos = Collections.binarySearch(list, key);
```

Counting…

```
int frequency = Collections.frequency(myColl,item) ;
```

shuffling, sorting, reversing, performing set operations and much more…

Collection algorithms:
- min / max
- frequency
- disjoint

List algorithms:
- sort
- binarySearch
- reverse
- shuffle
- swap
- fill
- copy
- replaceAll
- indexOfSubList
- lastIndexOfSubList

Collection factories:
- EMPTY_SET
- EMPTY_LIST
- EMPTY_MAP
- emptySet
- emptyList
- emptyMap
- singleton
- singletonList
- singletonMap
- nCopies
- list(Enumeration)

Collection Wrappers:
- unmodifiableCollection
- unmodifiableSet
- unmodifiableSortedSet
-…
- synchronizedCollection
- synchronizedSet
- synchronizedSortedSet
-…
- checkedCollection
- checkedSet
- checkedSortedSet
- …

Lecture 6e: Overview

- What is a collection
- Intro to Generics
- Collection Interfaces
- Collection Implementations
- Hash Tables
- Iterators

Hash Table - Introduction

- Motivation: Hash tables are good for performing quick search and delete operations
- Example:
  - consider a large set of data *s* (1000 elements), and a specific element *e* that we wish to find
  - How can we find if *e* is in *s*, *efficiently*?
  - A trivial solution is to search through all of *s*'s elements (O(n))
  - Using hash tables, this can be done in O(1)!

Solution: Hash Function

- A hash function gets an object, and returns an index in an array
  - For example, for an array of size 1000 the hash function should return a number in the range 0…999 (a valid index)
- A hash function must be **efficient** (constant time)

Hash Table Operations

- Hash Tables are implemented using an array
- Using a hash function the access to an element position can be achieved in a constant time:
  - Given an element → compute hash function → get index position
  - Used for the operations of *add(), remove()* and *contains()*

Hash Table Operations

Example

- A hash set of **Strings** is implemented using an array of Strings of length *N*, where each key String is mapped to a number in [0, N-1]
  - set.add("hello")
    - Map "hello" to an int *j* in the range [0,N-1]
    - Put "hello" in the *j*'th cell
  - set.contains("goodbye")
    - Map "goodbye" to an int *k* in the range [0,N-1]
    - Check if the *k*'th cell is "goodbye"

	car			**goodbye**		**hello**		**dog**	
0	1	2	…	k	…	j	...	N - 2	N - 1

Collisions

- A “hash collision” occurs when two or more elements are mapped to the same bucket
- There are different strategies to handle collisions

Iterating Hash Table

- Iterating a hash table generally requires traversing the cell array
- In general, if iteration complexity is important, consider using other set implementations (e.g., TreeSet or LinkedHashSet)

Lecture 6f: Overview

- What is a collection
- Intro to Generics
- Collection Interfaces
- Collection Implementations
- Hash Tables
- Iterators

Iterators

- An object which can “walk” through a collection
- Defines two major operations:
  - *hasNext()* – returns **true** iff there are more elements in the collection
  - *next()* – advances the iterator to the next element

```
List <String> myList = …;
Iterator <String> myIterator = myList.iterator();
while ( myIterator.hasNext() ) {
    String next = myIterator.next();
    System.out.println(next);
}
```

Iterators are also generic (Parameter must match the collection parameter)

Reasons for using Iterators

- Decouple data representation from data traversing
  - Information hiding
    - User need not be aware of the internal representation in order to traverse data structures
  - Implementation independent
    - Same iterator can work with various data structure
- Using iterators allows collections to define different traversing orders
  - Random, reverse order, …

Reasons for using Iterators (2)

- When working with collections, a natural order is not always defined
  - For example, HashSets do not define an order of elements
  - Iterator is the natural (and sometimes only) way to iterate such collections
- Iterators can also be more efficient than index-best iteration
  - E.g., iterating LinkedList

So Far…

- What is collection?
- Generics
- Collections framework:
  - Interfaces
    - **Collection** → List, Queue, Set → Sorted Set
    - **Map** → SortedMap
  - Implementations, Algorithms
- Iterators

Next Week

- Exceptions
- Packages
- Nested Classes