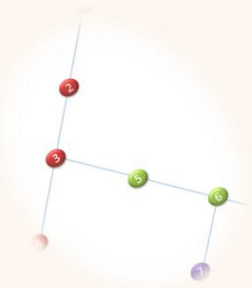
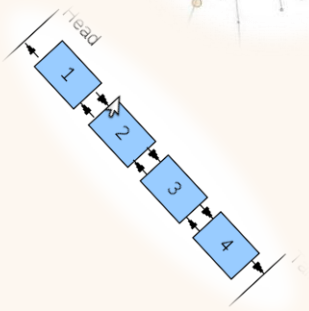
Introduction to Object Oriented Programming

(Hebrew University, CS 67125 / Spring 2014)

Lecture 6

Java Collections





What is a Collection?

- A **collection** is an object that groups multiple elements into a single unit
  - A **data structure**
  - Sometimes called a **container** or a **sequence**

- Collections are used to **store**, **retrieve**, **manipulate**, and **communicate** aggregate data

- Typically represent a **natural group**
  - A poker hand (a collection of **cards**), a mail folder (a collection of **letters**), or a telephone directory (a mapping of **phone numbers** to **names**)

The Collections Framework

- A **collections framework** is a unified architecture for representing and manipulating collections

- All collections frameworks contain the following:
  - **Interfaces**
  - **Implementations**
  - **Algorithms**

Java:
import java.util.*

The Collections Framework

Parts

- **Interfaces:** Abstract collections
  - Allow manipulation **independently** of the implementation

Map, Set, List, …

- **Implementations:** Concrete interface implementations
  - ***Reusable data structures***

TreeMap, HashSet, LinkedList, …

- **Algorithms:** Perform useful computations such as *searching* and *sorting*, on collection objects
  - Are **polymorphic**: same method works with many different implementations. ***Reusable functionality***

Collections.binarySearch, Collections.shuffle, …

HISTORY

JDK 1.0(‘96): Vector, Dictionary, Hashtable, Stack, Enumeration
JDK 1.2(‘98): Collection, Iterator, List, Set, Map, ArrayList, HashSet, TreeSet, HashMap, WeakHashMap
JDK 1.4(‘02): RandomAccess, IdentityHashMap, LinkedHashMap, LinkedHashSet
JDK 1.5(‘04): Queue, java.util.concurrent, ...
JDK 1.6(‘06): Deque, ConcurrentSkipListSet/Map, ...
JDK 1.7(’11): TransferQueue, LinkedTransferQueue
JDK 1.8(’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 of useful data structures and algorithms
  - 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 answer to these problems

Generics and Collections

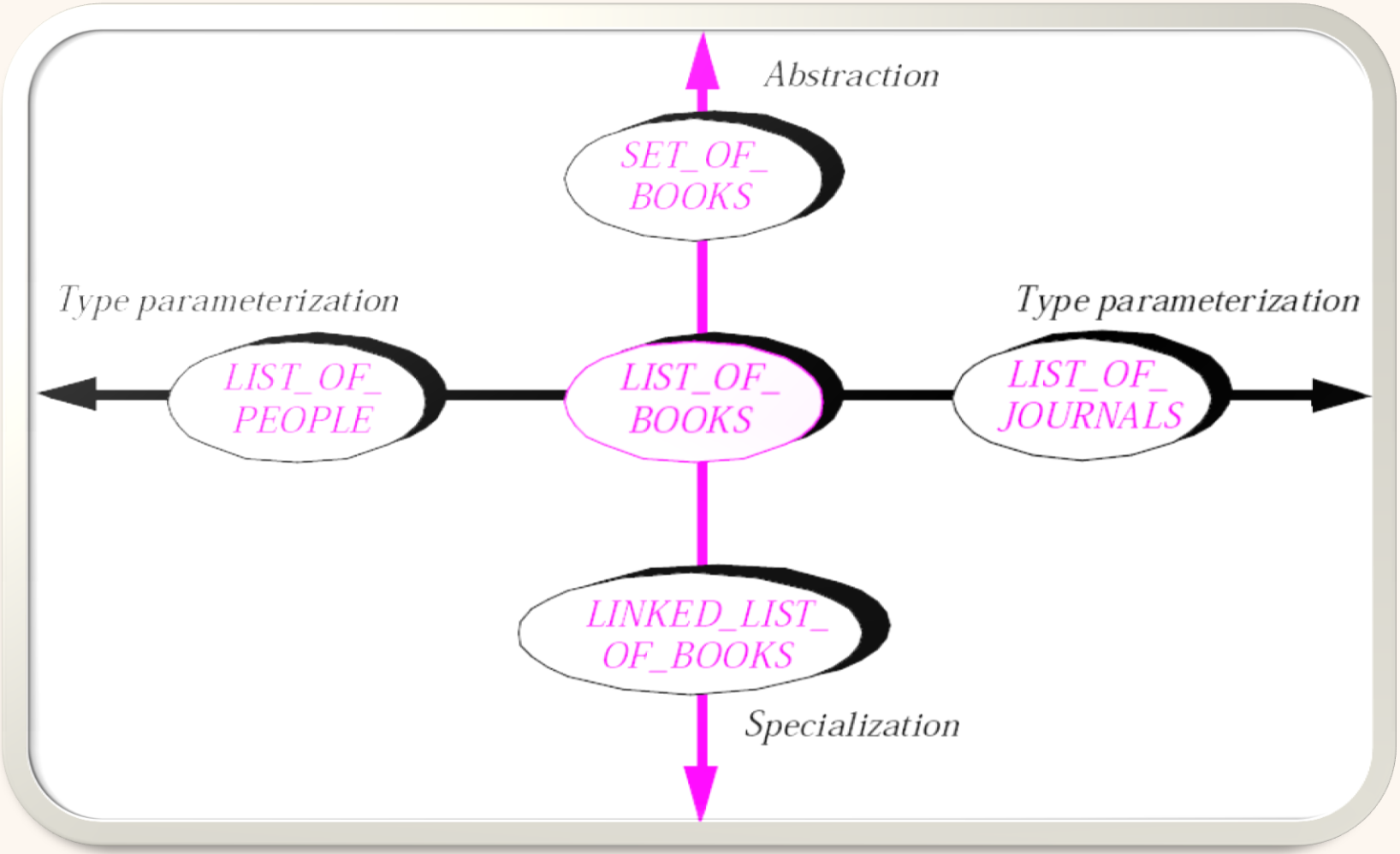
- All the core collection components are **generic**
  - For example, this is the declaration of the Collection interface:

public interface *Collection<E>*

What are Generics?

- Generics are **abstraction** over **non-primitive types**
  - Classes or arrays (including primitive arrays)
- *Classes*, *interfaces* and *methods* can be **parameterized** by types
- Generics provide increased **readability** and **type safety**
  - More to come

Introduction to Genericity



Generic API

- A generic class defines one or more parameters
  - List<**E**>
  - Map<**K**,**V**>

Generic API

- A generic class defines one or more 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 API

- When we create an object of a generic class, we set **concrete** parameters
  - Reference parameter and concrete parameter must agree
  - Up-casting is allowed, as long as the same parameters are used
  - List<**String**> myList = **new** LinkedList<**String**>();
  - Map<**String**,**Double**> map = **new** HashMap<**String**,**Double**>();

Generic API

- When we create an object of a generic class, we set **concrete** parameters
  - Reference parameter and concrete parameter must agree
  - Up-casting is allowed, as long as the same parameters are used
  - List<**String**> myList = **new** LinkedList<**String**>();
  - Map<**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);
  - **Double** d = map.put("hello", 5.7);

Generic API

- When we create an object of a generic class, we set **concrete** parameters
  - Reference parameter and concrete parameter must agree
  - Up-casting is allowed, as long as the same parameters are used
  - List<**String**> myList = **new** LinkedList<**String**>();
  - Map<**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);
  - **Double** d = map.put("hello", 5.7);

Using a Generic Class

```
// A list of strings
List<String> list = new LinkedList<String>();
list.add("hello");
String s = list.get(0);
```

Using a Generic Class

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

Using a Generic Class

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

Using a Generic Class

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

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

Using a Generic Class

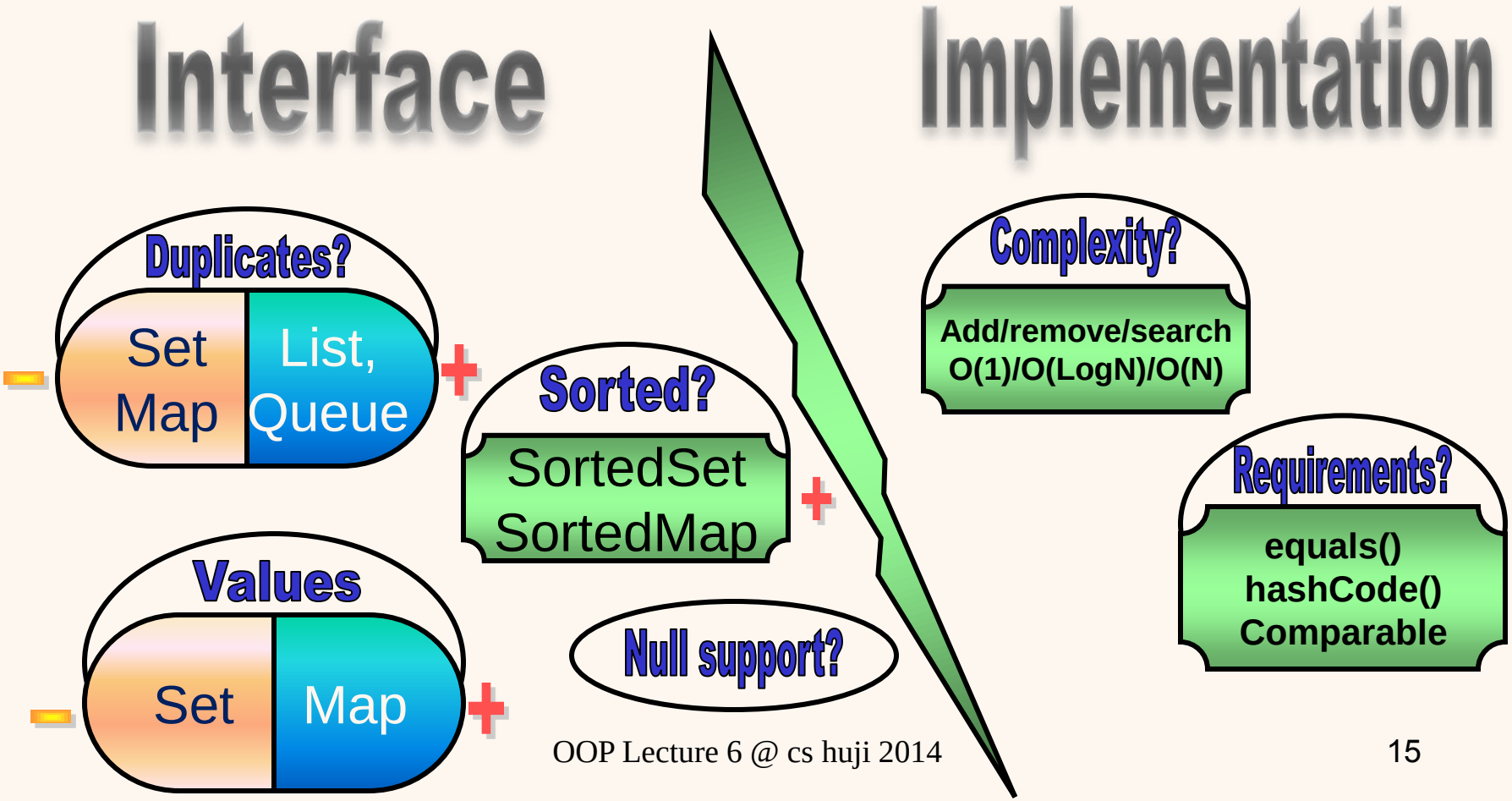
```
// A list of strings
List<String> list = new LinkedList<String>();
list.add("hello");
String s = list.get(0);
list.add(new Double(3.14));    // Compilation error
Double d = list.get(0);        // Compilation error

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

Using a Generic Class

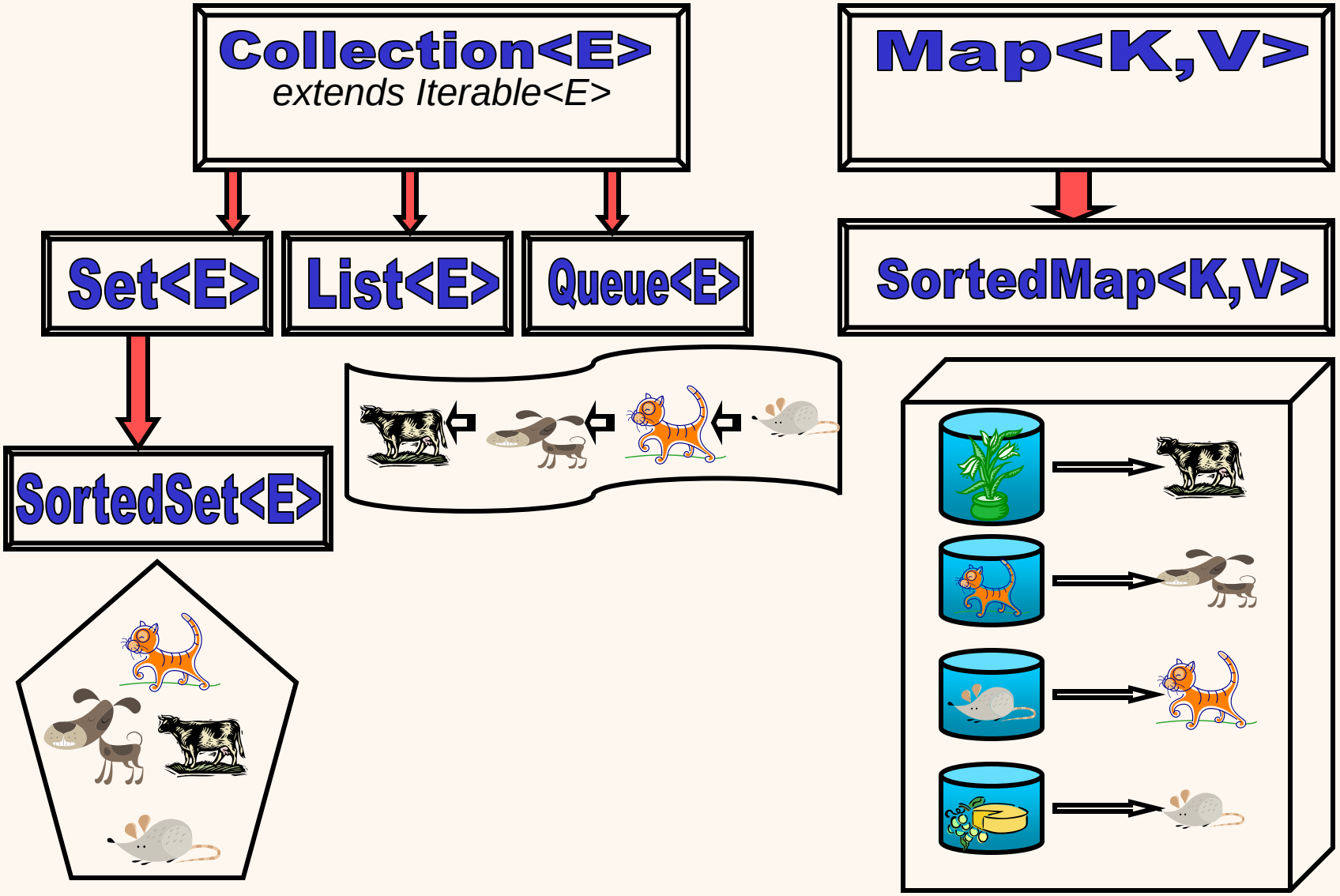
```
// A list of strings
List<String> list = new LinkedList<String>();
list.add("hello");
String s = list.get(0);
list.add(new Double(3.14));    // Compilation error
Double d = list.get(0);        // Compilation error

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

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

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

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

Core Collection Interfaces
List<E>

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

Core Collection Interfaces
Queue<E>

- **Queue** — a collection used to hold multiple elements prior to processing
  - Queues provide
    - *insertion* (**push**())
    - *extraction* (**pop**())
    - *inspection* of the top element in the queue (**peek**())
  - Typically, **but do not necessarily, FIFO** (first-in-first-out)
    - In FIFO queue, new elements are inserted at the **tail** of the queue
    - Other kinds of queues may use different placement rules
    - Counter example: **priority queues,** which order elements according to a supplied comparator or the elements' natural ordering
    - Whatever the ordering used, **peek**() or **pop**() returns the head of the queue

Core Collection Interfaces

Set<E>

- **Set** — a collection that cannot contain duplicate elements
  - Models the mathematical set abstraction
  - Represent Sets
    - A deck of cards
    - A list of courses
    - The processes running on a machine

- **SortedSet** — a sorted version of the **Set interface**
  - Several additional operations are provided
  - Used for naturally ordered sets
    - Word lists
    - Lists of candidates for a position

Core Collection Interfaces

Map<K,V>

- **Map** — an object that maps keys to values
  - Cannot contain duplicate **keys**:
    - Each key can map to at most one value
  - **Can** contain duplicate values
  - Used for collections of key/value pairs
    - Student id ➔ login
  - Maps are implemented similarly to Sets

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

Map Views

- The Collection view methods allow a Map to be viewed as a Collection in three different ways:
  - **keySet -** the **Set** of keys contained in the Map
  - **values -** The **Collection** of values contained in the Map
    - This is not a *Set*, because multiple keys can map to the same value
  - **entrySet -** the Set of key-value pairs contained in the Map
    - Map provides a small static nested interface called *Map.Entry*, the type of the elements in this Set
- This is how we iterate maps

Map Views: Example

- The idiom for iterating over values is analogous

```
for (ValType value : m.values()) {
    System.out.println(value);
}
```

- Here is how to iterate over key-value pairs:

```
for (Map.Entry<KeyType, ValType> e : m.entrySet()) {
    System.out.println(e.getKey() + ": " + e.getValue());
}
```

Collection Implementations
List<E>

- **ArrayList<E>** – Resizable-array implementation
  - get(), set() – constant time
  - add() – **amortized** constant time
    - adding *n* elements requires O(*n*) time
  - contains(), indexOf(), remove() – O(n)

- **LinkedList<E>** – Linked list implementation
  - add() – constant time
  - get(), set(), containts(), 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	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(1) avg	**???**
TreeSet	O(logN)	O(logN)	O(logN)	O(logN)	O(N)

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

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
- unmodifiableList
- unmodifiableMap
- unmodifiableSortedMap
- synchronizedCollection
- synchronizedSet
- synchronizedSortedSet
- synchronizedList
- synchronizedMap
- synchronizedSortedMap
- checkedCollection
- checkedSet
- checkedSortedSet
- checkedList
- checkedMap
- checkedSortedMap

Writing New **Classes** for **Use** with *Collections*

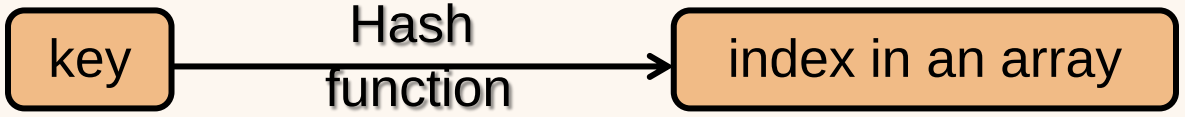
- The **equals** method should be overridden for every new class we write
  - Object.*equals*() returns **true** iff this is exactly **the same object**
    - String a = “hello”;
    - String b = “hello”;
    - a.equals(b)?
  - We should override equals() to ensure the answer is **true**

- When implementing this method, it should return **false** if the specified object is **null** or of **an inappropriate type**

Hash Code

- HashSet<E> is the java implementations of a hash table
  - See DaSt lecture

- Each key is mapped to an array of cells using a hash function



Hash Code (2)

- Each key is mapped to an array cell using the *hashCode*() method

public int *hashCode()*

- This method is implemented by the *Object* class
  - Thus, it is part of the API of every java class

- However, Object.*hashCode*() is generally different for any two objects
  - Even if they are essentially the same
  - String a = “hello”; String b = “hello”;
  - a.hashCode() == b.hashCode()?

Hash Code and Equals

- *hashCode*() should be overridden if the class's objects are expected to be used with any hash-based collection

- This is essential for any class that redefines the equals method (**equal objects must have equal hash codes**)
  - **If** *a.equals(b)* **then** *a.hashCode() == b.hashCode()*

- This doesn't work the other way around!
  - Equal hash code **does not** mean equal objects!
  - If *a.hashCode() == b.hashCode()* **then not necessarily** *a.equals(b)*

Implementing Hash Code

- A complicated task
- Generally speaking, the code should be:
  - As random as possible
  - Efficient
  - Consistent with the *equals*() method
    - A change in the object that effects the *equals*() method should result in a different *hashCode* value
  - A simple solution is to recursively use the *hashCode*() values of all the data members used by *equals*()

Implementing Hash Code
Suggestion

- A simple suggestion:
  - If your class has *n* relevant data members ($m_1,\ldots,m_n$)
  - choose *n* different prime numbers ($p_1,\ldots,p_n$)

```
public int hashCode() {
        int result = 0;
        for (int i = 1 ; i <=  n ;  ++i) {
           result += p_i*m_i.hashCode();
        }

        return result;
}
```

Immutable Keys

- In order to maintain the stability of hash-based classes, key objects (**elements** in sets or **keys** in maps) must be **immutable**
  - For example, *String* is an immutable class
  - These collections will **break** if you modify their elements or keys while they're in the collection

toString()

- Another method to override is **toString()**
  - Should print the object's string representation in a **human-readable form**

- Important for objects that are going to get be put inside collections
  - The various collection types' **toString** methods depend on the toString methods of their elements, keys, and values

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

- 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 aggregate data structure
  - Implementation independent
    - Same iterator can work with various data structure
- Define different traversing orders
  - Random
  - Reverse order
  - …

Reasons for using Iterators (2)

- When working with collections, a natural order is not always defined
  - It is not necessarily possible to use a **for** (**int** i = 0 ;…) loop
- Iterator is the natural (and sometimes only) way to iterate such collections

Iterators and foreach loop

- Remember the foreach loop?

```
String[ ] strings = …;
for (String s: strings) {
        …
}
```

- This mechanism is actually implemented using iterators

So Far…

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