Java Collections Framework

COMS M0103
OOP with Java
Tim Kovacs

2010

Objectives

- To learn what data structures and algorithms the Java library has
- To learn what they can do for you
- To see a few examples of how the classes are meant to be used
  - Sometimes called *idioms*
  - A conventional way of doing something
  - Like a design pattern but at the code level, not the architecture level
- NOT to learn everything you need to know
  - This will let you handle the usual cases
  - For special cases you will need more details

What you should already know

- The concepts of arrays, hash tables and linked lists
- How to use Java's ArrayList and Iterator classes
  - We do not cover basic examples of their use
- A little about the complexity of algorithms
  - that two algorithms with the same functionality can have massively different run-times
  - that run-time efficiency depends a lot on the data
    - So (often) alg. A isn't *always* better than alg. B
    - You need the right tool for the job
- A little about generics (typed collections)
- If you don't know these subjects you can probably still follow this lecture

Contents

* From the collections trail of the Java tutorial for Java 6
http://java.sun.com/docs/books/tutorial/collections/index.html

Collections and the JCF

- A collection is a general-purpose object used to hold and manipulate other objects e.g. ArrayList
- The *Java Collections Framework* (JCF) is Java's collection library
  - Interfaces for different kinds of collections (List, Set...)
  - Implementations of the interfaces
  - Algorithms for manipulating collections (searching, sorting...)
- Implements efficient collections and algorithms so you don't have to
- The JCF is complex and we don't cover everything!
- Don't confuse with the JFC (Java Foundation Classes)
  - A set of GUI packages: AWT, Swing and Java 2D

Some JCF Design Patterns

- Views
  - Viewing (part of) a collection using another collection
    - Possibly a different collection type
  - E.g. getting a sub-list of a List
  - E.g. getting the set of keys in a Map as a Set
  - The view shares its contents with the original
    - More efficient than copying the contents
    - Lets you manipulate contents of the original
- Factory pattern (for iterators)
  - Getting a collection to construct its own iterator
  - Means you don't need to know the correct type of iterator
- Optional interface methods
  - A messy way of not implementing all the functionality specified by an interface
- Strategy pattern
  - Comparator objects used by sorting algorithms

Generic Classes

Generic Classes

- The JCF uses a lot of generic classes
  - Classes with a *type parameter*
  - E.g. ArrayList<String> instead of just ArrayList
    - The two are different types!

```
ArrayList<String> myList = new ArrayList(); // error
ArrayList<String> myList = new ArrayList<String>(); // ok
```

- A type parameter is passed to a class
  - Just like an ordinary parameter is passed to a method
- Why not just use type `Object`?
  - Using more specific types lets the compiler check for errors
- See basic generics tutorial http://java.sun.com/docs/books/tutorial/java/generics/index.html
- There's a much more advanced version on same site

A Generic Class

```
public class Box<T> {        // T stands for "Type"
    private T data;          // define a field of type T, with name “data”

    public void add(T data) {
        this.data = data;
    }

    public T get() {
        return data;
    }
}
```

Now we can create boxes which hold different types

```
Box<Integer> integerBox = new Box<Integer>(); // T = Integer
Box<String> stringBox = new Box<String>();    // T = String
```

Type Parameter Names

- When writing a generic class (or documentation for it) we give the type parameter a name
- Typical names and what they stand for:
  - E - Element in a Collection (used extensively by the JCF)
  - K – Key in a Map
  - N – Number
  - T – Type
  - V – Value in a Map
  - S,U,V etc. - 2nd, 3rd, 4th types
- These are just conventions
  - Like ordinary variables, the name could be (almost) anything
  - Note the convention of 1 letter each
  - Unlike ordinary variables where 1 letter names are considered bad (except on slides!)

More Complex Type Parameters

- Sometimes we have more than one type parameter
  - To put a key/value pair in a map:

```
V put(K key,V value)
```

return type of method `put` is `V`

- The wildcard `?` means 'any type'
- Sometimes we want to restrict types

```
addAll(Collection<? extends E> c)
```

  - `?` can be any subtype of `E`
  - We call this a *bounded type parameter*
  - We can use `super` instead of `extends`
- There are a lot more details to generics
  - e.g. they only exist at compile-time!
    - At run-time `List<String>` is just `List`
  - See generics tutorials for more

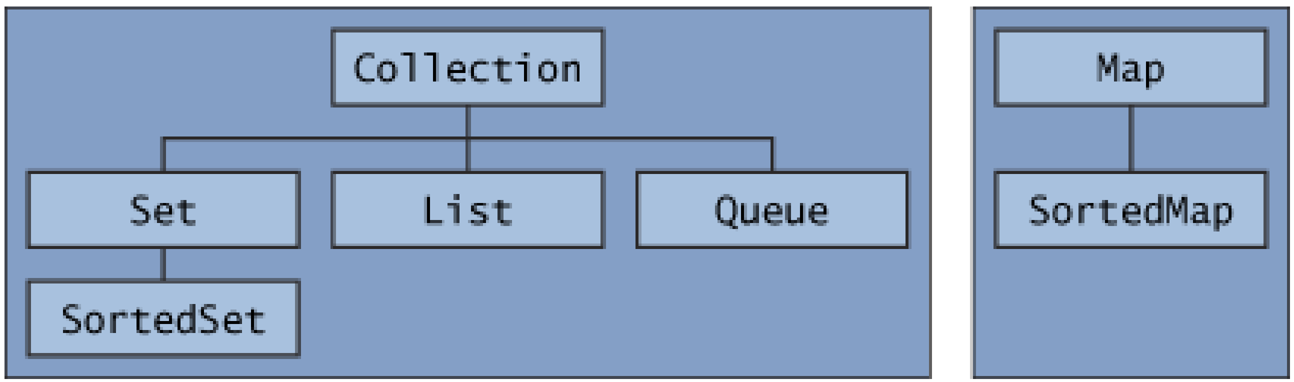
JCF Interfaces

Interfaces as ADTs

- The interface of a module is its set of public methods
- We have seen the utility of hiding objects behind interfaces
  - Polymorphic algorithms: write code using interface type, choose from different implementing types
- The interface for a data structure is sometimes called an Abstract Data Type (ADT)
  - Same principle: write polymorphic code, choose implementing data structure later
  - Choose the right tool (data structure) for the job (data)

Core Collection Interfaces

The most important interfaces in the JCF



- All are generic i.e. collections are typed
  - `Collection<String>`, `Collection<Integer>` ...
  - Since documentation applies to all types we use a variable for the type `Collection<E>`
- We will not look at the Queue interface

Figure is from the Collections Trail

Core Collection Interfaces

- Collection
  - JCF has no implementation; only its children implemented
  - Defines core methods for its children
- Set
  - No duplicate elements: `assert !e1.equals(e2)`
  - Elements unordered
- List
  - Can have duplicates. Elements ordered
- Queue
  - For temporary storage and sorting of dynamically arriving objects
  - E.g. First-In, First-Out (FIFO) protocol
- Map
  - Maps keys to values (like a phone book)
    - A key maps to either 0 or 1 value
  - Cannot have duplicate keys

Typical Uses

- Collection
  - Most cases where you just need to store or iterate through a "bunch of things" and later iterate through them
  - Use List if you care about their order
- Set
  - Remembering which items you've already processed
    - e.g. when doing a web crawl
  - Making other yes-no decisions about an item, e.g.
    - is the item an English word?
    - is the item in the database?
    - is the item in this category?

Based on http://www.javamex.com/tutorials/collections/how_to_choose.shtml

Typical Uses

- Map
  - Used in cases where you need to say "for a given X, what is the Y"?
    - For a given user ID, what is the name?
    - For a given IP address, what is the country code?
    - For a given string, how many instances have I seen?
- Queue
  - For traversing hierarchical structures such as a filing system, or in general where you need to remember "what data to process next", whilst also adding to that list of data

Optional Methods

- Some collection methods are marked “optional” in the documentation
  - But the general-purpose implementations we will see later all implement all the optional methods
- Other implementations may throw an UnsupportedOperationException instead
  - This is because a class must provide code for all methods in an interface; it can't choose a subset
  - Throwing this exception is a hack to allow them to provide functionality for only a subset

Optional Methods

- In general, this is a **bad** design idea
  - The alternative was to have many more interfaces
    - Versions with and without the optional methods
  - The JCF is already very complex so optional methods seemed a better compromise

The Collection Interface

- Very abstract and general since it's the parent of most of the others
- Basic operations:

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

- `add, remove` return true if they modify the collection
  - remove removes only a single instance
- The `E` parameter to `add` is from `Collection<E>`
  - In other words you can only add elements of type `E`

Identity and Equality

- Recall identity and equality:
  - Identity: 2 objects are `==` if they are really the same object
    - I.e. two variables refer to the same object
  - Equality: 2 objects are `equal()` if their state is equivalent

```
Rectangle r1 = new Rectangle(0,0,5,5);
Rectangle r2 = new Rectangle(0,0,5,5);
r1.equals(r2)    // true – same position and size
r1 == r2         // false – different objects
```

- Class `Object` defines `equals()` using `==`
  - So by default `==` and `equals()` are the same
- If we want different rectangle objects to have equality (as in code above) we must override `equals()`

Equality in the JCF

- The contract of many methods uses equals()
  - E.g. `contains(element)` returns true if the collection has some object for which `equals(element)` returns true
- This is the method's contract, not a description of implementation
  - A shortcut may be used instead of calling equals on all elements

When to use the Collection Interface

- Use type `Collection` when you don't care what the subtype is
  - E.g. when you just want to pass collections around
- E.g. all general-purpose collection implementations have a constructor which accepts type `Collection`
- Called a *conversion constructor* because it converts any collection into its own type
- E.g. suppose you have objects of type `List<String>` and `Set<String>` you want to convert to `ArrayList<String>`
- ArrayList only needs one constructor to handle both

```
...new ArrayList<String>(myList);
...new ArrayList<String>(mySet);
```

Traversing Collections

- Two ways:
  - For-each loops
  - Iterator objects

- For-each loop example

```
for (Object current : myCollection)
  System.out.println(current);
```

- Limitations
  - You cannot add or remove elements during traversal
  - You cannot alternate iteration over multiple collections

Iterators

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

- Note: no add method
  - Unlike ListIterator interface
- Example

```
Iterator it = myCollection.iterator();
while (it.hasNext())
   if (noLongerNeeded(it.next()))
      it.remove();
```

Collection's iterator method constructs an iterator for us. An example of the factory pattern.

Traversing Collections

- Use Iterators when you want
  - to remove elements during traversal, or
  - to alternate iteration over multiple collections
- Remember:
  - remove() removes the last element returned by next()
  - remove() can only be used once for each element
  - Using Collection's add() or remove() breaks existing iterators
    - Use Iterator.remove() instead
  - Using Iterator.remove() breaks any other iterators.
- Rules for using iterators
  - If you don't add or remove you can have multiple iterators
  - If you do remove you should have only 1 iterator

Iterators and the Factory Pattern

- Why not call the iterator constructor yourself?
  - To keep the code generic/abstract/polymorphic
    - Each subclass of Collection has its own iterator class
    - You do not want to tie the code to a particular collection type
  - Even if that didn't matter
    - You probably don't know the name of the right iterator class
      - It's hard to find out
    - E.g. what's the iterator class for a HashSet?

```
Collection myCollection = new HashSet();
??? it = new ???();  //HashSetIterator?
// keep it generic; let Java figure it out
Iterator it = myCollection.iterator();
```

The *only* place we specify what type of collection it is

Bulk Operations on Collections

- Bulk operations (except clear) take a collection as parameter

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

- booleans indicate whether collection was modified
- Note bounded type parameter to addAll

Idiom: Removing All Copies of 1 Object

- `remove(Object o)` only removes *one* instance of `o`
- `removeAll(Collection c)` removes *all* instances of all elements of `c`
- What if you want to remove all instances of one object `o`?
  - First put `o` in a collection
  - Then use `removeAll`:

```
c.removeAll(Collections.singleton(e));
```

  - `singleton()` is factory method returning an immutable Set with one element
  - Efficient because the Set is a special lightweight class, not a regular Set class

The Set Interface

- Only has methods inherited from Collection
- Cannot contain duplicate elements
  - Can only contain one `null` value
  - Object duplicates are defined as: `e1.equals(e2)`
  - If you try to add a duplicate it ignores it
- Set A equals() set B if they contain exactly the same elements
- If you change an element so it becomes a duplicate, behaviour of the set is undefined
  - In other words: don't do it!
  - So be careful with sets that contain mutable elements
- A set cannot contain itself

Idiom for Removing Duplicates

- Given collection of strings c1 make a collection c2 which is the same as c1 but without duplicates

```
Collection<String> c2 = new HashSet<String>(c1);
```

HashSet
constructor
removes duplicates

Hide the dynamic type
(HashSet) behind the interface
type (Collection) – a good habit

The List Interface

- Lists are ordered and can contain duplicates
  - Like arrays the first element is at position 0
  - Lists can contain themselves but this can cause problems!
- List inherits from Collection and adds:
  - Random access (also called positional access)
    - Methods like get, set, add, remove which take an index
  - Search methods
    - indexOf and lastIndexOf
  - ListIterator
    - an extended version of Iterator with forward and backward traversal
    - also has an add method so you can add at a position
  - subList(int from, int to)
    - returns a new list which shares objects with the original

Notes on List

- `add` and `addAll` add to the end of the list
  - An iterator can add at other positions
- `remove` removes the first instance
  - Collection's version removes a arbitrary instance
- The only way to get an iterator which points at a particular object is to traverse the list
  - `contains` returns true or false, not a position

Implementations of the List Interface

- The JFC has 2 implementations of List
  - ArrayList and LinkedList
- They have very different run-time behaviour
  - ArrayList has random access
  - LinkedList has sequential access
- Making List fit both of them means
  - List is not very cohesive
  - For each implementation, there are methods in List which are very inefficient
- More in section on implementations

The Map Interface

- Maps keys to values
  - Each key maps to 0 or 1 values
  - Cannot contain duplicate keys
  - Maps are not ordered (but SortedMaps are)
  - “Put” with an existing key replaces the key's last value (and returns it, or `null` if there was no existing value)

```
public interface Map<K,V> {
    V put(K key, V value);
    V get(Object key);
    V remove(Object key);
    boolean containsKey(Object key);
    boolean containsValue(Object value);
    int size();
    boolean isEmpty();
   ...}
```

Map's Collection Views

- Interaction with a Map is limited
  - Basically: put(), get(), containsKey(), containsValue()
  - No (direct) way to iterate through all keys
- Collection view methods return part of a map as another Collection type
  - View shares the contained objects with the map
  - Allows viewing the map's contents
    - But also processing them (not read-only)
  - The only way to iterate through a map

Map's Collection Views

```
public interface Map<K,V> {...
    Set<K> keySet();
    Collection<V> values();
    Set<Map.Entry<K,V>> entrySet();
...}
```

- Each method returns a collection:
  - keySet — the `Set` of keys in the map
  - values — The `Collection` of values in the map
    - Not a `Set` as multiple keys can map to a value
  - entrySet — the `Set` of key/value pairs in the map
    - The pairs have type `Map.Entry`
    - `Map.Entry` is an inner interface of map
- All are public

Iterating through Keys or Values

Printing all keys with a for-each loop

```
for (<Type> key : myMap.keySet())
   System.out.println(key);
```

Using an iterator to filter out all elements with some property

```
Iterator<Type> it = myMap.keySet().iterator();
while (it.hasNext())
   if (it.next().isBogus())
      it.remove();
```

Map.Entry

- An interface defined inside Map
- Key/value pairs are stored with this type

```
public interface Entry {
    K getKey();
    V getValue();
    V setValue(V value);
}
```

Iterating through Key/Value Pairs

```
// print all key/value pairs
for (Map.Entry<KeyType, ValType> e : m.entrySet())
    System.out.println(e.getKey() + ": " +
       e.getValue());
```

Removing, Setting but NOT Adding

- If you remove from a collection view
  - you also remove from its map (since they share their contents)
  - you break any existing iterators of both map and view
    - Except the one that did the removal, if you did it with an iterator
  - See example a few slides ago of filtering out elements
- If you remove a key, you remove its value too
- You can use a view's Entry.setValue() to change the underlying map
- You CANNOT add elements to the map using collection views
  - Just add them directly to the map

Multimaps

- Like a map but each key has multiple values
- Not implemented in the JCF
- Just use a Map and for each value use another collection e.g. a List

Ordering Objects

- A set of objects can be ordered (sorted) in many ways
  - a,b,c
  - b,a,c
  - and so on...
- In a *total ordering* it is possible to compare any pair of elements
  - E.g. the set of integers; any two integers can be compared
  - Elements can be tied, e.g. A and a
- In a *partial ordering* some pairs cannot be compared
  - E.g. ordering letters and other symbols alphabetically
  - How do you compare a and @?
  - Of course you can define an ordering (or many) to do this
  - But that goes beyond alphabetical ordering

Comparable Interface

```
public interface Comparable<Type> {
    public int compareTo(Type other);
}
```

- Used to sort objects
- Compares objects “this” and “other” and returns:
  - negative integer if this < other
  - 0 if they are equal
  - positive integer if this > other
- Comparable's ordering is called the class's *natural order*
- Implemented by all number types, String, Date and some other library classes

Comparator Interface

- What if you want:
  - to choose different orderings?
    - E.g. sort Person objects by name or date of birth?
  - to sort objects that don't implement Comparable?
- Write a class which implements the Compara**tor** interface (not Compara**ble**)

```
public interface Comparator<Type> {
   int compare(Type obj1, Type obj2);
}
```

- Pass an instance of this class to other classes e.g. SortedSet
  - An example of the *strategy* pattern

“Consistent with Equals”

- Comparable & Comparator can only return 0 if the elements are `equal()`
  - We say the comparison must be “consistent with equals”

SortedSet Interface

- A set that keeps its elements in ascending order
  - Uses natural order, or a Comparator passed to constructor
  - Must be a total ordering
    - Furthermore, since it's a set it cannot have duplicates
    - So if compare or compareTo return 0 the second object will not be inserted
- Extends Set interface to add
  - Range-views
    - Return another SortedSet sharing subset of elements with the first
    - The subset is a consecutive range
  - Get methods for the endpoints (first and last element)
  - A get method for the Comparator it uses

SortedSet Interface

```
public interface SortedSet<E> extends Set<E> {
    // Range-views
    SortedSet<E> subSet(E fromElement, E toElement);
    SortedSet<E> headSet(E toElement);
    SortedSet<E> tailSet(E fromElement);

    // Endpoints
    E first();
    E last();

    // Comparator access
    Comparator<? super E> comparator();
}
```

SortedMap Interface

- A map that keeps its elements in ascending order
  - Uses natural order, or a Comparator passed to constructor
  - Sorting is done only on keys, not values
  - Must be a total ordering
- SortedMap is to Map exactly what SortedSet is to Set
  - I.e. extends Map interface to add:
    - Range-views
    - Get methods for the endpoints (first and last element)
    - A get method for the Comparator it uses

```
public interface SortedMap<K, V> extends Map<K, V>{
    SortedMap<K, V> subMap(K fromKey, K toKey);
    SortedMap<K, V> headMap(K toKey);
    SortedMap<K, V> tailMap(K fromKey);
    K firstKey();
    K lastKey();
    Comparator<? super K> comparator();
}
```

Adding to Sorted Collections

- SortedSet and SortedMap impose extra conditions on the elements they accept
  - Elements must either:
    - implement Comparable, or
    - be accepted by the Comparator object provided
  - Furthermore, all objects must be *mutually comparable*
    - If a collection contains only Rectangles, it only needs to compare a Rectangle and another Rectangle
    - But if it contains Rectangles and Triangles, it must also know how to compare a Rectangle and Triangle
    - You may need to extend Comparable or Comparator for this

JCF Implementations

About Implementations

- We have seen the JCF interfaces
- Each can be implemented by multiple JFC classes
  - or by user-supplied classes
- Implementations can have very different run-time characteristics
  - They may also sort elements in different ways

Kinds of JCF Implementations

- General-purpose implementations
  - For everyday use
- Special-purpose imp.
  - for special purposes (!)
- Concurrent imp.
  - for use with threads
- Wrapper imp.
  - to add or remove functionality from others (Decorator pattern)
- Convenience imp.
  - simple implementations for special purposes
- Abstract imp.
  - partial implementations to help write your own

General-purpose Implementations

All general-purpose implementations:

- implement all optional methods
- allow null elements, keys and values
- are unsynchronised (i.e. not suitable for threads)
- are serializable (can be written to files)
- have public clone methods to make copies
- have *fail-fast* iterators
  - If the collection is modified directly (not through an iterator) the iterator throws ConcurrentModificationException
    - But only if iterator detects the modification
  - Better than unpredictable behaviour of iterating over modified collection (“fail later”)

General-purpose Implementations

All General-purpose Implementations

Interfaces	Implementations					
	Hash table	Resizable array	Tree	Linked list	Hash table + Linked list	Priority Heap
`Set`	`HashSet`		`TreeSet`		`LinkedHashSet`	
`List`		`ArrayList`		`LinkedList`		
`Map`	`HashMap`		`TreeMap`		`LinkedHashMap`	
`Queue`				`LinkedList`		`PriorityQueue`

- Names are in 2 parts: implementation/interface

Selected General-purpose Implementations

Selected General-purpose Implementations

Interfaces	Implementations			
	Hash table	**Resizable array**	**Tree**	**Linked list**
`Set`	`HashSet`		`TreeSet`	
`List`		`ArrayList`		`LinkedList`
`Map`	`HashMap`		`TreeMap`	

- These are the ones you should know

JCF Set Implementations

- HashSet
  - No guarantees about order of traversal
  - Most efficient version (uses a hash table)
  - Default initial size is 16
  - Rule of thumb: initialise to twice size you expect to need
- TreeSet
  - Visits elements in order of their value
  - Much slower than HashSet (uses a red-black tree)
- LinkedHashSet
  - Visits elements in order of their insertion
  - Slightly slower than HashSet (uses a hash table and a linked list)

JCF Map Implementations

- HashMap, TreeMap and LinkedHashMap
- Just like the corresponding Set implementations

JCF List Implementations

- The JFC has 2 implementations of List
  - ArrayList and LinkedList
- They have very different run-time behaviour
  - ArrayList has random access
  - LinkedList has sequential access
- Making List fit both of them means
  - List is not very cohesive
  - For both ArrayList and LinkedList there are methods in List which are very inefficient
    - Of course the smaller the collection the less it matters
    - Inefficient use is probably only noticeable if you have hundreds of elements and process them intensely
    - But if the collection is big it can make a huge difference

Random vs. Sequential Access

- Random access
  - Jumps straight to any element (think of an array)
- Sequential access
  - Starts at one end and visits every element until it reaches the one you want (think of a linked list)
  - If you want element 1000 you must visit the 1000 elements that come before it – slow!
- List methods with an index argument
  - are fast with ArrayList
  - are very slow with LinkedList

How ArrayList is Implemented

- ArrayList delegates storage to a private array
  - This is called the *backing* array
  - Because it supports (backs up) the ArrayList
- You may not have known that, even if you've used it
  - It's hidden behind the public interface of ArrayList
- Usually it's good to separate the interface and implementation
  - But sometimes it is useful to know about the implementation
  - E.g. so you can choose one that is efficient for the current problem

Efficiency of Insertion and Deletion

- If the array is not full, adding to the end is efficient
- But insertions and deletions which cause reordering are inefficient
  - If the array has 1000 elements and you insert at position 500 it has to shift all elements between 500 to 1000
- If the array is too small
  - A bigger one is created
  - The smaller one is copied into it
- In contrast, insertion and deletion in a LinkedList always involve 1 element
  - Very efficient

LinkedList or ArrayList?

- Use the one that suits your problem
  - Usually ArrayList
- If you're not sure try them both
  - Easy if your code treats them both as type List
- The JCF could have split List into two interfaces
  - One for linked lists and one arrays (plus a List parent)
  - Then it would be clearer how to use them efficiently
  - But they wanted to minimise the number of interfaces

“If you think you want to use a LinkedList, measure the performance of your application with both LinkedList and ArrayList before making your choice; ArrayList is usually faster.”

Josh Bloch, primary JCF designer
Quote from the Collections Trail

Other JCF Implementations

- We have seen only general-purpose ones so far
  - Here are a few others
  - They may not implement the “optional” methods
- EnumSet
  - High-performance Set for Enums
  - Allows easy iteration over ranges

```
for (Day d : EnumSet.range(Day.MONDAY, Day.FRIDAY))
    System.out.println(d);
```

- EnumMap
  - High-performance Map using Enums as keys
- Synchronization Wrappers
  - Add synchronization (thread safety) to Collection classes
- Unmodifiable Wrappers
  - Provide read-only access to Collection classes

JCF Algorithms

JCF Algorithms

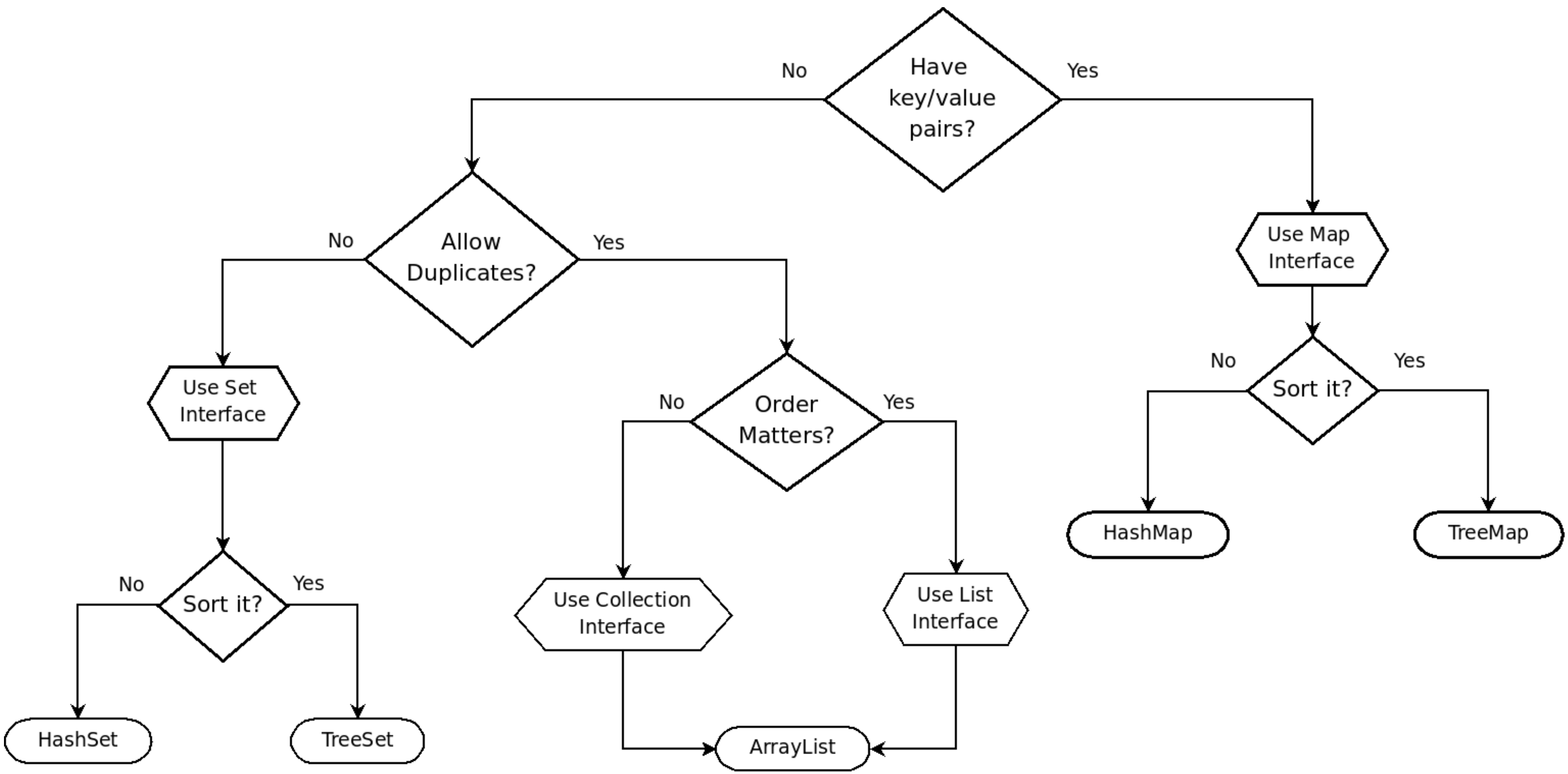
- Collection**s** class (not Collection interface; note -s)
  - Provides static methods
  - First argument is a Collection to work on
  - Most apply to List, some to Collection
- Has algorithms for
  - Sorting (fast, stable version of merge sort)
  - Shuffling (randomises order of a list)
  - Routine data manipulation (reverse, fill, copy, swap, addAll)
  - Searching
  - Composition
    - frequency() counts number of times element occurs
    - disjoint() determines whether two collections are disjoint
  - Finding extreme values (max and min)

Choosing a Collection

Choosing Interfaces and Classes

- Not always easy to choose the right one
- Different approaches to choosing
- Simple approach: what properties do you want the collection to have?
  - No duplicates?
  - Keys that map to values?
  - Sorted into a particular order?
- Next slide has a flowchart
  - Covers only the main cases
  - A bit simplistic

Choosing using Collection Properties



Choosing Collections: Operations

- A more sophisticated approach
- Decide which operations you need
- Think about which classes support them
  - and how efficiently
- Two kinds of operations:
  - Those that don't change the collection (queries)
  - Those that do (mutations)
- But first we quickly review linked lists and hash tables
  - If you understand how they work, you can work out which queries they handle efficiently
    - Easier than trying to memorise which queries are efficient

Linked Lists

- A linked list is built out of *nodes*
  - The first is called the *head*, the last the *tail*
- Each node holds:
  - an element
  - a reference to the next node (except the tail)
- To add an element, Java:
  - makes a new node n
  - makes an existing node (if there is one) refer to n
  - makes n refer to the following node (if there is one)
- We can insert n anywhere in the list
  - same time cost no matter where
    - assuming we're already at the right position
- To delete an element, Java:
  - makes the node *before* it to refer to the node *after* it
  - heads and tails are special cases

Hash Tables

- Hash tables are an efficient way of implementing some collections (e.g. `HashSet`, `HashMap`)
- Basically, a hash table is an array of lists
  - These lists are called buckets
  - The user does not specify an index

    `hashTable[0] = myObj;` // no
  - Instead the index is computed automatically for each object

    `hashTable.add(myObj);` // yes
- Looking up an entry is efficient
  - We do not need to search the entire array (and all lists)
  - Java computes the index and searches only that list

    `hashTable.contains(myObj);` // searches only 1 list

Queries

- "What is in the collection?"
  - Supported by all collections
  - Get an iterator and go through elements one at a time
- "Is x in the collection?"
  - Supported by all collections
  - Efficient with hash tables, less efficient with binary search of sorted collections, inefficient using iteration

Queries

- "What value is associated with x?"
  - Only supported by maps
  - Efficient (implemented with hash table)
- "How many of these are in the collection?"
  - Supported by all using iterating and counting (inefficient)
  - Special cases that are efficient:
    - Sets either have 0 or 1
    - Maps can sometimes use special value objects:
      - value is a collection storing all the relevant objects, or
      - value is an integer indicating the number of objects (instead of storing them)

Positional Queries

- "What is at integer position x?"
  - Only supported by List
  - Efficient with ArrayList, inefficient with LinkedList
- "What integer position does x have?"
  - Only supported by List
  - Requires iteration over collection (inefficient)
- "What comes before/after x?"
  - I.e. search for x, then find element before/after it
  - Supported by List, TreeMap & TreeSet
  - Use iteration to find out (inefficient)
- "What comes next?"
  - List, TreeMap & TreeSet support iteration in a defined order
    - List also supports moving iterator backwards
  - Iteration starts at the beginning
    - List can also start at the end, or at an integer position

Mutating Collections

- Insert/delete
  - ArrayList: inefficient except at end
    - But forcing it to grow is inefficient
  - LinkedList: efficient*
  - Sorted collections: not very efficient as sorting is needed
  - Efficient with collections based on hash tables
- Mutating elements is efficient*
  - Assignment to an ArrayList index points it at a new object
  - map.entry.setValue points key to a new value
  - Mutating internal state of an element

*Finding the position/element to work on may not be efficient!

Mutating Collections

- Swapping position of elements
  - Efficient with ArrayList (use 2 assignments)
  - Efficent with hashMap (use setValue twice)
  - Sorted collections use insert/delete
    - less efficient
  - LinkedList uses insert/delete
    - inefficient

Miscellanea

Hash Codes

- An object's index in a hash table is called its *hash code*
  - Computed by the `hashCode` method
  - In maps, only keys are hashed, not values
  - Search is faster when lists are shorter
    - In extreme case, array has length 1 and all element are in the same list
    - This is equivalent to using a list instead of a hash table!
  - So we want to distribute elements to lists more or less equally
- `hashCode` is implemented by class `Object`
  - Based on memory location of the object
    - But manipulated to fit length of array
  - So 2 different objects (usually) have different hash codes
  - Sometimes you want a different behaviour

hashCode() and equals()

- Sometimes we want different objects to be equivalent

```
Rectangle r1 = new Rectangle(0,0,5,5);
Rectangle r2 = new Rectangle(0,0,5,5);
r1.equals(r2); // should be true
```

- We need to override `Object`'s `equals()` to do this
- Similarly, if we want 2 objects to have the same hash code we must override `Object`'s `hashCode()`
  - E.g. a set should not contain both r1 and r2
  - They must have same hash code, or e.g. `contains` will not search the right list
  - Compute hash code from object's state instead of address
- Check the `Object` class in the API for documentation

Rules on hashCode() and equals()

- Two objects that are equivalent must have the same hash code
  - (But not all objects with same hash code are equivalent)
- The default `equals()` and `hashCode()` use identity (memory address)
  - If you override `equals()` to use equality you must override `hashCode()` to do the same
  - If you do NOT override `equals()`, do NOT override `hashCode()`
  - I.e. make "`hashCode` consistent with `equals`"
- In a `HashMap` only keys are used to look things up
  - Values do not need a hash code
- A special case: overriding equals in a subclass when the superclass already overrides it
  - See http://www.ibm.com/developerworks/java/library/j-jtp05273.html

New Syntax in Java 7

- Java 7 is due in late 2010
- Type parameters in constructors can be inferred from variable types e.g.

```
ArrayList<String> friends = new ArrayList<>();
Map<String, Long> phoneBook = new HashMap<>();
```

No types!

- [ ] syntax for accessing ArrayList
  - Allows us to write

```
friends[0] = "Rachel";
```

instead of

```
friends.set(0, "Rachel");
```

and

```
System.out.println(friends[0]);
```

instead of

```
System.out.println(friends.get(0));
```

Java 7: Collection Literals

- A literal is code that defines an immutable value
  - Unlike a variable, which can change
- Before Java 7 we could make string and number literals

```
System.out.println(“Hello, number ” + 2);
```

- Java 7 allows collection literals of type List, Set and Map
  - Constructs and initialises collection at same time
  - These collections are immutable

```
[“Rachel”, “Caroline”, “Sophie”]; // a List
{“Rachel”, “Caroline”, “Sophie”}; // a Set
{“Rachel” : 1, “Caroline” : 2, “Sophie” : 3}; // a Map
```

Using Collection Literals

- Use as a parameter to a method

```
names.addAll([“Rachel”, “Caroline”, “Sophie”]);
```

- As parameter to a constructor

```
ArrayList<String> newFriends = new
  ArrayList<>([“Sam”]);
```

- Assign to a variable
  - Type of variable must be the relevant interface type
  - We don't actually know the implementation type!

```
List<String> friends = [“Penny”, “Imi”];
```

Interface type

Constructs an object of some unknown type that implements List

Older Classes

- The JCF was new in Java 1.2
  - Pre-1.2 code uses older classes (“legacy collections”)
  - Some library code uses these old classes
  - They are not deprecated; there are still useable
  - But only use them if you need to
  - See the “Interoperability” part of the Collections Trail
- `Hashtable`
  - Note lowercase t
  - A synchronized (thread-safe) version of `HashMap`
  - Only use if you use threads
- `Vector`
  - A synchronized version of `ArrayList`
  - Only use if you use threads

More Legacy Collections

- `Enumeration`
  - Replaced by `Iterator`
  - Only use if you're working on old code that uses it
- arrays
  - Still useful and widely used
    - but sometimes the newer collections are better
  - Many libraries still use them even when a collection is better
  - `Collection` interface has methods to convert to array
- `Arrays` class
  - Has static methods for manipulating arrays (sorting...)
  - Like `Collections` does for collections
  - `asList` returns a list backed by the array

Conclusion

Key Points

- Adding and removing can confuse existing iterators
  - Fail-fast iterators throw `ConcurrentModificationException` *if* they detect their collection has been modified in an unauthorised way
- ArrayList and LinkedList both implement List
  - Each is inefficient for some uses
  - Usually you want an ArrayList
- equals() is important
  - Sets define duplicates with equals()
  - For sorted collections and hash tables to work equals must be defined correctly
- hashCode must be “consistent with equals”
  - equals() and hashCode() must both test for identity or both test for equality
- Sorting with Comparator and Comparable must be “consistent with equals”

Key Points

- The hash code of a key must not change
  - Otherwise the value will be lost
  - Do not mutate the state of a key if hash code is based on state
- Do not introduce duplicates into sets by mutating elements
  - It's safest to keep elements immutable
- The general-purpose JCF implementations have certain properties (allow null elements, fail-fast iterators etc.)
  - Other implementations are different – be careful!

Exercises

- Assume you have code to read a book from a file
- Think about how to implement the following:
  - List the words used in the book
    - i.e. each word only appears once in your list
  - List them in alphabetic order
  - Also list the number of times each word appears
  - Also list the pages each word appears on

- The last is called a concordance
  - People used to do them by hand!
  - This could take years

Layers of JCF Understanding

- The JCF is a minefield
  - You may have used a collection successfully in the past
  - But often doing something a little different causes a bug
    - e.g. changing state of an element can cause problems
  - The solution is often complex
- Roughly, there are 3 levels of knowledge
  - Basic: standard use of ArrayList, HashMap, Set and Iterator
    - Covered in *Objects First* and most textbooks
    - Iterators vs. for-loops with indexes: http://en.wikipedia.org/wiki/Iterator
  - Intermediate: more details about special cases
    - This lecture
    - The trail on which this lecture is based
    - For a JCF overview and design FAQ see also http://java.sun.com/javase/6/docs/technotes/guides/collections/index.ht
    - *Core Java* is a good reference book with a JCF chapter

Advanced Resources

- Advanced: Full details about all the special cases, efficiency issues, implementing your own collections...
  - Specialised books that go into the JCF in detail e.g.
    - *Java Generics and Collections* by Maurice Naftalin and Philip Wadler
    - *Data Structures and the Java Collections Framework* by William Collins
    - I haven't read either...
  - Many resources on the web
    - many articles on specific subjects
    - just search for them
    - e.g. equals and hashCode
      - http://www.ibm.com/developerworks/java/library/j-jtp05273.html

Related Resources

- There are textbooks on data structures with example Java code
  - But they mostly show you how to implement your own data structures
  - Not much about how to use the JCF
- See *Big Java* on how to implement hash tables and linked lists yourself
  - You are not likely to need this since the JCF provides them
  - But it can be useful to understand how they work