latex input: mmd-article-header Title: Java Collections Notes Author: Ethan C. Petuchowski Base Header Level: 1 latex mode: memoir Keywords: Java, programming language, syntax, fundamentals CSS: http://fletcherpenney.net/css/document.css xhtml header: copyright: 2014 Ethan Petuchowski latex input: mmd-natbib-plain latex input: mmd-article-begin-doc latex footer: mmd-memoir-footer

For all the "optional" methods, it means that if you're not implementing it, you should throw an `UnsupportedOperationException`. This sounds like a sloppy way to set up the Collections hierarchy, but the point was to make there be only a few simple interfaces so that it's easier to learn.

Iterator

```
interface Iterator<E> {
    boolean hasNext()
    E next()

    /* [optional]
     * removes most recent element
     * you *must* have just called next() or previous()
     * can't call it twice in a row
     */
    void remove()
}
```

ListIterator

These can be extracted from a `List<E>`

```
interface ListIterator<E> {
    int nextIndex()
    boolean hasPrevious()
    E previous()
    int previousIndex()
    void add(E)             // optional
    void set(E)             // optional
}
```

Iterable

Implementing `Iterable<T>` allows you to use the "foreach" loop construct.

```
interface Iterable<T> {
    Iterator<T> iterator()
}
```

Collection

"The root interface in the *collection heirarchy*." A lot of freedom is left up to the implementer to provide whatever specifics they want.

```
interface Collection<E> extends Iterable<E> {
    boolean add(E)                              // optional
    boolean addAll(Collection<? extends E>)     // optional
    void clear()                                // optional
    boolean contains(Object)
    boolean containsAll(Collection<?>)
    boolean equals(Object)
    int hashCode()
    boolean isEmpty()
    Iterator<E> iterator()
    boolean remove(Object)                      // optional
    boolean removeAll(Collection<?>)            // optional
    int size()
    Object[] toArray()
    <T> T[] toArray(T[] a) // you can specify runtime type of array
}
```

List

An *ordered* collection (aka. "sequence"), meaning **it has indexes**.

```
interface List<E> extends Collection<E> {
    void add(int, E)
    boolean addAll(int, Collection<>)
    E get(int)
    int indexOf()
    int lastIndexOf(Object)
    ListIterator<E> listIterator()
    ListIterator<E> listIterator(int)
    boolean retainAll(Collection<?>)
    E set(int, E)                       // optional
    List<E> subList(int, int)
}
```

AbstractCollection

Implements most of `Collection` based on (`Iterator<E> iterator()`) and (`int size()`), so that those are *all that remain* for *you* to get your class to `implement Collection`.

This example is for `contains(Object)`. Note that you can always override this implementation if you want to.

```
abstract class AbstractCollection<E> implements Collection<E> {
    public boolean contains(Object obj) {
        for (E element : this) // calls iterator()
            if (element.equals(obj))
                return true;
        return false;
    }
}
```

Map

From the docs:

- An object that maps keys to values
- A map cannot contain duplicate keys
- Each key can map to at most one value

Doesn't extend anything. Comments are mine.

```
interface Map<K, V> {

    /* find & retrieve */
    boolean             containsKey(Object)
    boolean             containsValue(Object)
    V                   get(Object key)

    /* size */
    int                 size()
    boolean             isEmpty()

    /* alternate view */
    Set<K>              keySet()
    Collection<V>       values()

    // from this you call entry.getKey|Value()
    Set<Map.Entry<K,V>> entrySet()

    boolean             equals(Object)
    int                 hashCode()

    /* insert */
    // overwrites, returns *previous* value
    V                   put(K, V)      // optional
    void                putAll(Map<>)  // optional

    /* remove */
    void clear()        // optional
    V remove(Object)    // optional
}
```

HashMap

- **no guarantees** about iteration order

NavigableMap (interface)

- A `SortedMap` which allows searches for "closest matches"
    - `lowerEntry`, `floorEntry`, `ceilingEntry`, `higherEntry`, ⇒ `Map.Entry<K,V>`
    - `lowerKey`, ..., ⇒ `K`
- **can be traversed in either direction**.
- The `descendingMap` method returns a view of the map with the senses of all relational and directional methods inverted.
- Iterates/sorted according to `compareTo()` (or supplied `Comparator`)

TreeMap

- It's **a Red-Black tree based `NavigableMap`** (above)

LinkedHashMap

- **Iterates in order of insertion**
- Contains both a hashmap *and* a **double-linked-list** in ordered by *insertion-order*
    - In other words, when you call `map.entrySet()` the *first* one out is the *first* one you *inserted*
- If you (re-)insert a `key` that the map already `contains`, the order is *not* affected

IdentityHashMap

1. Implements `Map` interface, but uses `k1==k2` for comparisons instead of `k1==null ? k2 == null : k1.equals(k2)` (thereby violating `Map`'s general contract).
2. Useful for *topology-preserving object graph transformations*, such as serialization or deep-copying
    - I found it in Spark's SizeEstimator.scala for calculating the `sizeof` a Java object
3. Permits `null` values and the `null` key (woudn't have guessed that.)
4. No ordering guarantees, including ordering may not be preserved between different traversals
5. Uses hash table to achieve constant-time performance for `get` and `put` given well-chosen identity hash codes for the objects stored
    - This is implemented using *linear-probing* rather than *chaining*
6. To *synchronize* its operations easily, use

```
Map m = Collections.synchronizedMap(new IdentityHashMap(maxSiz
```

7. (JavaDocs)

ConcurrentHashMap

1. (JavaDocs)
2. A hash table supporting full concurrency of retrievals and high expected concurrency for updates.
3. The *big win*: retrieval does *not* entail locking or block in general
4. Retrievals can be concurrent with updates

5. Retrievals reflect the most recently *completed* update operations holding upon their nset (ie. *happens-before* relation [thanks Leslie & Maurice])
6. `putAll` and `clear` are not atomic with respect to retrievals
7. Iterators will *not* throw `ConcurrentModificationException`
8. `null` may be neither used as *key* nor *value*
9. It's often not obvious whether using this will increase the speed of your application. Try it and see if it's any better, then you may want to play with the parameters by first setting them to the extremes and seeing which cases are more performant.

Queue

- Often FIFO, but not always
    - Example exception to the rule: *priority queue* is ordered by the supplied comparator
- The *tail* is where you put elements, the *head* is where you get elements from
- Often has a bounded max-size

```
interface Queue<E> extends Collection<E> {
    // Inserts the specified element if possible, returning true
    // upon success, and throwing an IllegalStateException
    // if no space is currently available.
    boolean add(E e);

    // Like add but returns false instead of throwing an exception
    boolean offer(E e);

    // Retrieves, but does not remove, the head of this queue; throw
    // exception on empty queue.
    E element();

    // Like element but returns null on empty queue
    E peek();

    // Retrieves and removes the head of this queue, or throws an ex
    E remove();

    // Like remove, but returns null if this queue is empty.
    E poll();
}
```

AbstractQueue

- **Abstract class** *extends* `AbstractCollection<E>` *implements* `Queue<E>`
- Provides skeletal implementations of some Queue operations.
- Only makes sense to use if `null` elements are not allowed in the queue.

- Implementers must override at *least* `offer`, peek, `poll`, `size`, and `iterator`.

BlockingQueue

- **Interface** *extends* `Queue<E>`
- Adds methods to also allow you to `put` & `take` elements, which means you will wait until the performing action on the queue is physically possible
    - I.e. wait until there is empty space, or an element to remove
- Adds versions of `offer` and `poll` that take a time limit. If that time limit is reached, they return the normal `false` or `null`, respectively.
- Designed primarily to be used for producer-consumer queues
- Implementations are *thread-safe*, meaning effects are made atomic using internal locks or whatever

LinkedBlockingQueue

- **Class** *extends* `AbstractQueue<E>` *implements* `BlockingQueue<E>`
- Optionally bounded [in size]
- Built upon a FIFO-ordered *linked-list* of nodes