Concrete Collections

Madhavan Mukund

https://www.cmi.ac.in/~madhavan

Programming Concepts using Java

Week 6

Built-in data types

- The `Collection` interface abstracts properties of grouped data
    - Arrays, lists, sets, . . .
    - But not key-value structures like dictionaries

Built-in data types

- The `Collection` interface abstracts properties of grouped data
  - Arrays, lists, sets, . . .
  - But not key-value structures like dictionaries
- Collections can be further organized based on additional properties
  - Are the elements ordered?
  - Are duplicates allowed?
  - Are there constraints on how elements are added, removed?

Built-in data types

- The Collection interface abstracts properties of grouped data
  - Arrays, lists, sets, ...
  - But not key-value structures like dictionaries
- Collections can be further organized based on additional properties
  - Are the elements ordered?
  - Are duplicates allowed?
  - Are there constraints on how elements are added, removed?
- In the spirit of indirection, these are captured by interfaces that extend Collection
  - Interface List for ordered collections
  - Interface Set for collections without duplicates
  - Interface Queue for ordered collections with constraints on addition and deletion

The `List` interface

- An ordered collection can be accessed in two ways
  - Through an iterator
  - By position — random access

The List interface

- An ordered collection can be accessed in two ways
  - Through an iterator
  - By position — random access
- Additional functions for random access

```
public interface List<E>
             extends Collection<E>{
  void add(int index, E element);
  void remove(int index);
  E get(int index);
  E set(int index, E element);
}
```

The `List` interface

- An ordered collection can be accessed in two ways
  - Through an iterator
  - By position — random access
- Additional functions for random access
- `ListIterator` extends `Iterator`
  - `void add(E element)` to insert an element before the current index
  - `void previous()` to go to previous element
  - `boolean hasPrevious()` checks that it is legal to go backwards

```
public interface List<E>
             extends Collection<E>{
  void add(int index, E element);
  void remove(int index);
  E get(int index);
  E set(int index, E element);

  ListIterator<E> listIterator();
}
```

The List interface and random access

- Random access is not equally efficient for all ordered collections
  - In an array, can compute location of element at index i
  - In a linked list, must start at the beginning and traverse i links

```
public interface List<E>
                extends Collection<E>{
  void add(int index, E element);
  void remove(int index);
  E get(int index);
  E set(int index, E element);

  ListIterator<E> listIterator();
}
```

The List interface and random access

- Random access is not equally efficient for all ordered collections
  - In an array, can compute location of element at index i
  - In a linked list, must start at the beginning and traverse i links
- Tagging interface RandomAccess
  - Tells us whether a List supports random access or not
  - Can choose algorithmic strategy based on this

```
public interface List<E>
             extends Collection<E>{
  void add(int index, E element);
  void remove(int index);
  E get(int index);
  E set(int index, E element);

  ListIterator<E> listIterator();
}
```

```
if (c instanceof RandomAccess) {
  // use random access algorithm
} else {
  // use sequential access algorithm
}
```

The AbstractList interface

- Recall that AbstractCollection is the “usable” version of Collection

The AbstractList interface

- Recall that `AbstractCollection` is the “usable” version of `Collection`
- Correspondingly, `AbstractList` extends `AbstractCollection`
    - Inherits default implementations

The `AbstractList` interface

- Recall that `AbstractCollection` is the “usable” version of `Collection`
- Correspondingly, `AbstractList` extends `AbstractCollection`
  - Inherits default implementations
- `AbstractSequentialList` extends `AbstractList`
  - A further subclass to distinguish lists without random access

The AbstractList interface

- Recall that AbstractCollection is the “usable” version of Collection
- Correspondingly, AbstractList extends AbstractCollection
  - Inherits default implementations
- AbstractSequentialList extends AbstractList
  - A further subclass to distinguish lists without random access
- Concrete generic class LinkedList<E> extends AbstractSequentialList
  - Internally, the usual flexible linked list
  - Efficient to add and remove elements at arbitrary positions

The AbstractList interface

- Recall that `AbstractCollection` is the "usable" version of `Collection`
- Correspondingly, `AbstractList` extends `AbstractCollection`
  - Inherits default implementations
- `AbstractSequentialList` extends `AbstractList`
  - A further subclass to distinguish lists without random access
- Concrete generic class `LinkedList<E>` extends `AbstractSequentialList`
  - Internally, the usual flexible linked list
  - Efficient to add and remove elements at arbitrary positions
- Concrete generic class `ArrayList<E>` extends `AbstractList`
  - Flexible size array, supports random access

Using concrete list classes

- Concrete generic class `LinkedList<E>` extends `AbstractSequentialList`
    - Not random access

Using concrete list classes

- Concrete generic class `LinkedList<E>` extends `AbstractSequentialList`
  - Not random access
  - But random access methods of `AbstractList` are still available

Using concrete list classes

- Concrete generic class LinkedList<E> extends AbstractSequentialList
  - Not random access
  - But random access methods of AbstractList are still available
  - This loop will execute a fresh scan from start to element i in each iteration!

```
LinkedList<String> list = new ...;

for (int i = 0; i < list.size(); i++)
  // do something with list.get(i);
```

Using concrete list classes

- Concrete generic class LinkedList<E> extends AbstractSequentialList
  - Not random access
  - But random access methods of AbstractList are still available
  - This loop will execute a fresh scan from start to element i in each iteration!

```
LinkedList<String> list = new ...;

for (int i = 0; i < list.size(); i++)
  // do something with list.get(i);
```

- Two versions of add() available
  - add() from Collection appends to the end of the list
  - add() from ListIterator inserts a value before current position of the iterator

Using concrete list classes

- Concrete generic class LinkedList<E> extends AbstractSequentialList
  - Not random access
  - But random access methods of AbstractList are still available
  - This loop will execute a fresh scan from start to element i in each iteration!

```
LinkedList<String> list = new ...;

for (int i = 0; i < list.size(); i++)
  // do something with list.get(i);
```

- Two versions of add() available
  - add() from Collection appends to the end of the list
  - add() from ListIterator inserts a value before current position of the iterator
- In Collection, add() returns boolean — did the add update the collection?
  - add() may not update a set, always works for lists

Using concrete list classes

- Concrete generic class `LinkedList<E>` extends `AbstractSequentialList`
  - Not random access
  - But random access methods of `AbstractList` are still available
  - This loop will execute a fresh scan from start to element `i` in each iteration!

```
LinkedList<String> list = new ...;

for (int i = 0; i < list.size(); i++)
  // do something with list.get(i);
```

- Two versions of `add()` available
  - `add()` from `Collection` appends to the end of the list
  - `add()` from `ListIterator` inserts a value before current position of the iterator
- In `Collection`, `add()` returns `boolean` — did the add update the collection?
  - `add()` may not update a set, always works for lists
- `add()` in `ListIterator` returns `void`

The Set interface

- A set is a collection without duplicates

The Set interface

- A set is a collection without duplicates

- `Set` interface is identical to `Collection`, but behaviour is more constrained
    - `add()` should have no effect, and return `false`, if the element already exists
    - `equals()` should return `true` if contents match after disregarding order

The Set interface

- A set is a collection without duplicates
- `Set` interface is identical to `Collection`, but behaviour is more constrained
  - `add()` should have no effect, and return `false`, if the element already exists
  - `equals()` should return `true` if contents match after disregarding order
- Two interfaces, same signature?

The Set interface

- A set is a collection without duplicates
- Set interface is identical to Collection, but behaviour is more constrained
  - add() should have no effect, and return false, if the element already exists
  - equals() should return true if contents match after disregarding order
- Two interfaces, same signature?
- Use Set to constrain values to satisfy additional constraints

The Set interface

- A set is a collection without duplicates
- `Set` interface is identical to `Collection`, but behaviour is more constrained
  - `add()` should have no effect, and return `false`, if the element already exists
  - `equals()` should return `true` if contents match after disregarding order
- Two interfaces, same signature?
- Use `Set` to constrain values to satisfy additional constraints
- `Set` implementations typically designed to allow efficient membership tests

The Set interface

- A set is a collection without duplicates
- `Set` interface is identical to `Collection`, but behaviour is more constrained
  - `add()` should have no effect, and return `false`, if the element already exists
  - `equals()` should return `true` if contents match after disregarding order
- Two interfaces, same signature?
- Use `Set` to constrain values to satisfy additional constraints
- `Set` implementations typically designed to allow efficient membership tests
- Ordered collections loop through a sequence to find an element

The Set interface

- A set is a collection without duplicates
- `Set` interface is identical to `Collection`, but behaviour is more constrained
  - `add()` should have no effect, and return `false`, if the element already exists
  - `equals()` should return `true` if contents match after disregarding order
- Two interfaces, same signature?
- Use `Set` to constrain values to satisfy additional constraints

- `Set` implementations typically designed to allow efficient membership tests
- Ordered collections loop through a sequence to find an element
- Instead, map the value to its position
  - Hash function

The Set interface

- A set is a collection without duplicates
- `Set` interface is identical to `Collection`, but behaviour is more constrained
  - `add()` should have no effect, and return `false`, if the element already exists
  - `equals()` should return `true` if contents match after disregarding order
- Two interfaces, same signature?
- Use `Set` to constrain values to satisfy additional constraints
- `Set` implementations typically designed to allow efficient membership tests
- Ordered collections loop through a sequence to find an element
- Instead, map the value to its position
  - Hash function
- Or arrange values in a two dimensional structure
  - Balanced search tree

The Set interface

- A set is a collection without duplicates
- `Set` interface is identical to `Collection`, but behaviour is more constrained
    - `add()` should have no effect, and return `false`, if the element already exists
    - `equals()` should return `true` if contents match after disregarding order
- Two interfaces, same signature?
- Use `Set` to constrain values to satisfy additional constraints

- `Set` implementations typically designed to allow efficient membership tests
- Ordered collections loop through a sequence to find an element
- Instead, map the value to its position
    - Hash function
- Or arrange values in a two dimensional structure
    - Balanced search tree
- As usual, concrete set implementations extend `AbstractSet`, which extends `AbstractCollection`

Concrete sets

- `HashSet` implements a hash table
  - Underlying storage is an array
  - Map value `v` to a position `h(v)`
  - If `h(v)` is unoccupied, store `v` at that position
  - Otherwise, collision — different strategies to handle this case

Concrete sets

- `HashSet` implements a hash table
  - Underlying storage is an array
  - Map value `v` to a position `h(v)`
  - If `h(v)` is unoccupied, store `v` at that position
  - Otherwise, collision — different strategies to handle this case
- Checking membership is fast — check if `v` is at position `h(v)`

Concrete sets

- `HashSet` implements a hash table
  - Underlying storage is an array
  - Map value `v` to a position `h(v)`
  - If `h(v)` is unoccupied, store `v` at that position
  - Otherwise, collision — different strategies to handle this case
- Checking membership is fast — check if `v` is at position `h(v)`
- Unordered, but supports `iterator()`

Concrete sets

- HashSet implements a hash table
  - Underlying storage is an array
  - Map value v to a position h(v)
  - If h(v) is unoccupied, store v at that position
  - Otherwise, collision — different strategies to handle this case
- Checking membership is fast — check if v is at position h(v)
- Unordered, but supports iterator()
- Scan elements in unspecified order

Concrete sets

- `HashSet` implements a hash table
  - Underlying storage is an array
  - Map value `v` to a position `h(v)`
  - If `h(v)` is unoccupied, store `v` at that position
  - Otherwise, collision — different strategies to handle this case
- Checking membership is fast — check if `v` is at position `h(v)`
- Unordered, but supports `iterator()`
- Scan elements in unspecified order
- Visit each element exactly once

Concrete sets

- HashSet implements a hash table
  - Underlying storage is an array
  - Map value v to a position h(v)
  - If h(v) is unoccupied, store v at that position
  - Otherwise, collision — different strategies to handle this case
- Checking membership is fast — check if v is at position h(v)
- Unordered, but supports iterator()
- Scan elements in unspecified order
- Visit each element exactly once

- TreeSet uses a tree representation
  - Values are ordered
  - Maintains a sorted collection

Concrete sets

- `HashSet` implements a hash table
  - Underlying storage is an array
  - Map value `v` to a position `h(v)`
  - If `h(v)` is unoccupied, store `v` at that position
  - Otherwise, collision — different strategies to handle this case
- Checking membership is fast — check if `v` is at position `h(v)`
- Unordered, but supports `iterator()`
- Scan elements in unspecified order
- Visit each element exactly once

- `TreeSet` uses a tree representation
  - Values are ordered
  - Maintains a sorted collection
- Iterator will visit elements in sorted order

Concrete sets

- HashSet implements a hash table
  - Underlying storage is an array
  - Map value v to a position h(v)
  - If h(v) is unoccupied, store v at that position
  - Otherwise, collision — different strategies to handle this case
- Checking membership is fast — check if v is at position h(v)
- Unordered, but supports iterator()
- Scan elements in unspecified order
- Visit each element exactly once

- TreeSet uses a tree representation
  - Values are ordered
  - Maintains a sorted collection
- Iterator will visit elements in sorted order
- Insertion is more complex than a hash table
  - Time $O(\log n)$ if the set has n elements

The `Queue` interface

- Ordered, remove front, insert rear

The Queue interface

- Ordered, remove front, insert rear
- Queue interface supports the following

```
boolean add(E element);
E remove();
```

  - If queue full, add() flags an error
  - If queue empty, remove() flags an error

The Queue interface

- Ordered, remove front, insert rear
- Queue interface supports the following

```
boolean add(E element);
E remove();
```

  - If queue full, add() flags an error
  - If queue empty, remove() flags an error
- Gentler versions of add(), remove()

```
boolean offer(E element);
E poll();
```

  - Return false or null, respectively, if not possible

The Queue interface

- Ordered, remove front, insert rear
- Queue interface supports the following

```
boolean add(E element);
E remove();
```

  - If queue full, add() flags an error
  - If queue empty, remove() flags an error

- Gentler versions of add(), remove()

```
boolean offer(E element);
E poll();
```

  - Return false or null, respectively, if not possible

- Inspect the head, no update

```
E element(); // Throws exception
E peek();    // Returns null
```

The Queue interface

- Ordered, remove front, insert rear
- Queue interface supports the following

```
boolean add(E element);
E remove();
```

  - If queue full, add() flags an error
  - If queue empty, remove() flags an error
- Gentler versions of add(), remove()

```
boolean offer(E element);
E poll();
```

  - Return false or null, respectively, if not possible
- Inspect the head, no update

```
E element(); // Throws exception
E peek();    // Returns null
```

- Interface Deque, double ended queue

```
boolean addFirst(E element);
boolean addLast(E element);
boolean offerFirst(E element);
boolean offerLast(E element);
E pollFirst();
E pollLast();
E getFirst();
E getLast();
E peekFirst();
E peekLast();
```

The Queue interface

- Ordered, remove front, insert rear
- `Queue` interface supports the following

  ```
  boolean add(E element);
  E remove();
  ```

  - If queue full, `add()` flags an error
  - If queue empty, `remove()` flags an error
- Gentler versions of `add()`, `remove()`

  ```
  boolean offer(E element);
  E poll();
  ```

  - Return `false` or `null`, respectively, if not possible
- Inspect the head, no update

  ```
  E element(); // Throws exception
  E peek();    // Returns null
  ```

- Interface `Deque`, double ended queue

  ```
  boolean addFirst(E element);
  boolean addLast(E element);
  boolean offerFirst(E element);
  boolean offerLast(E element);
  E pollFirst();
  E pollLast();
  E getFirst();
  E getLast();
  E peekFirst();
  E peekLast();
  ```

- Interface `PriorityQueue`
  - `remove()` returns highest priority item

The Queue interface

- Ordered, remove front, insert rear
- `Queue` interface supports the following

```
boolean add(E element);
E remove();
```

  - If queue full, `add()` flags an error
  - If queue empty, `remove()` flags an error
- Gentler versions of `add()`, `remove()`

```
boolean offer(E element);
E poll();
```

  - Return `false` or `null`, respectively, if not possible
- Inspect the head, no update

```
E element(); // Throws exception
E peek();    // Returns null
```

- Interface `Deque`, double ended queue

```
boolean addFirst(E element);
boolean addLast(E element);
boolean offerFirst(E element);
boolean offerLast(E element);
E pollFirst();
E pollLast();
E getFirst();
E getLast();
E peekFirst();
E peekLast();
```

- Interface `PriorityQueue`
  - `remove()` returns highest priority item
- Concrete implementations
  - `LinkedList` — implements `Queue`
  - `ArrayDeque` — circular array `Deque`

Summary

- Different types of `Collection` are specified by subinterfaces
  - `List`, `Set`, `Queue`
- `List` allows random access, more functional `ListIterator`
- `Set` constrains collection to not have duplicates
- `Queue` supports restricted add and remove methods
- Each interface has corresponding version under `AbstractCollection`
- Concrete implementations extend `AbstractList`, `AbstractSet` and `AbstractQueue`