Java: Collections and Maps

Madhavan Mukund, S P Suresh

Programming Language Concepts

Lecture 11, 28 February 2022

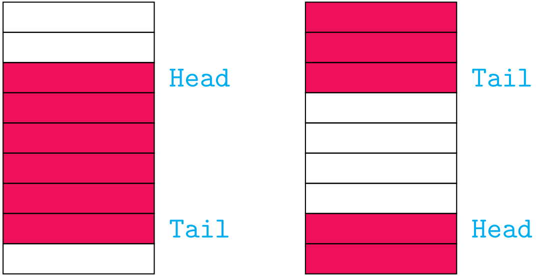
Abstract data types

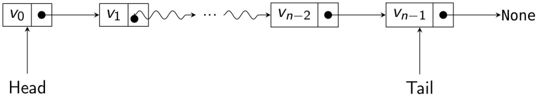
- Separate public interface from private implementation
- For instance, a (generic) queue

```
public class Queue<E> {
  public void add (E element){...};
  public E remove(){...};
  public int size(){...};
  ...
}
```

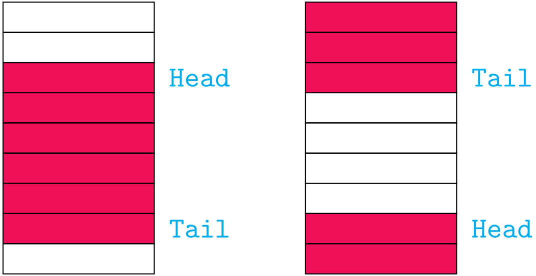
Abstract data types

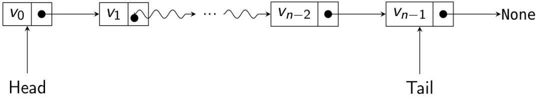
- Separate public interface from private implementation
- For instance, a (generic) queue
- Concrete implementation could be a circular array
- Or a linked list
- Implementer of class `Queue` can choose either one
- Public interface is unchanged





- Is the user indifferent to choice of implementation?
- Interface does not capture other aspects
- Efficiency
  - Circular array is better — one time storage allocation
- Flexibility
  - Linked list is better — circular array has bounded size
- Offer user a choice of implementation?





Multiple impementations

- Create two separate implementations
- User chooses

```
CircularArrayQueue<Date> dateq;
LinkedListQueue<String> stringq;

dateq =
   new CircularArrayQueue<Date>();
stringq =
   new LinkedListQueue<String>();
}
```

- What if we later realize we need a flexible size `dateq`?
- Change declaration for `dateq`
- And also every function header, auxiliary variable, . . . associated with it

```
public class CircularArrayQueue<E> {
  public void add (E element){...};
  public E remove(){...};
  public int size(){...};
  ...
}

public class LinkedListQueue<E> {
  public void add (E element){...};
  public E remove(){...};
  public int size(){...};
  ...
}
```

Adding indirection

- Instead, create a `Queue` interface
- Concrete implementations implement the interface
- Use the interface to declare variables

```
Queue<Date> dateq;
Queue<String> stringq;

dateq =
   new CircularArrayQueue<Date>();
stringq =
   new LinkedListQueue<String>();
}
```

- Benefit of indirection — to use a different implementation for `dateq`, only need to update the instantiation

```
public interface Queue<E> {
  abstract void add (E element);
  abstract E remove();
  abstract int size();
}
public class CircularArrayQueue<E>
    implements Queue<E> {
  public void add (E element){...};
  public E remove(){...};
  public int size(){...};
  ...
}
public class LinkedListQueue<E>
    implements Queue<E> {
  public void add (E element){...};
  public E remove(){...};
  public int size(){...};
  ...
}
```

The power of indirection

- Use interfaces to flexibly choose between multiple concrete implementations
  - Interfaces add a level of indirection
- Indirection in real life
  - Organization provides senior staff with an office car
  - Concrete: each official has an assigned car — what if it breaks down?
  - Indirection: a pool of office cars, use any that is available
  - Don't want to maintain a pool of cars? Contract with a taxi service
  - Don't want to negotiate tenders? Reimburse taxi bills

"Fundamental theorem of software engineering"

All problems in computer science can be solved by another level of indirection.

Butler Lampson, Turing Award 1992

Built-in data types

- Most programming languages provide built-in collective data types
  - Arrays, lists, dictionaries, ...
- Java originally had many such pre-defined classes
  - `Vector`, `Stack`, `Hashtable`, `Bitset`, ...
- Choose the one you need
- ...but changing a choice requires multiple updates
- Instead, organize these data structures by functionality
- Create a hierarchy of abstract interfaces and concrete implementations
  - Provide a level of indirection

The Collection interface

- The `Collection` interface abstracts properties of grouped data
  - Arrays, lists, sets, . . .
  - But not key-value structures like dictionaries
- `add()` — add to the collection
- `iterator()` — get an object that implements `Iterator` interface
- Use iterator to loop through the elements

```
public interface Collection<E>{
  boolean add(E element);
  Iterator<E> iterator();
  ...
}

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

Collection<String> cstr = new ...;
Iterator<String> iter = cstr.iterator();
while (iter.hasNext()) {
  String element = iter.next();
  // do something with element
}
```

Using iterators

- Use iterator to loop through the elements
- Java later added "for each" loop
  - Implicitly creates an iterator and runs through it
- Generic functions to operate on collections
- How does this line work?

  ```
  if (element.equals(obj))
  ```

- Later!

```
Collection<String> cstr = new ...;
Iterator<String> iter = cstr.iterator();
while (iter.hasNext()) {
  String element = iter.next();
  // do something with element
}
```

```
Collection<String> cstr = new ...;
for (String element : cstr){
  // do something with element
}
```

```
public static <E> boolean
       contains(Collection<E> c, Object obj) {
  for (E element : c)
    if (element.equals(obj))
      return true;
  return false;
}
```

Removing elements

- Iterator also has a `remove()` method
  - Which element does it remove?

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

Removing elements

- Iterator also has a `remove()` method
  - Which element does it remove?
- The element that was last accessed using `next()`

```
Collection<String> cstr = new ...;
Iterator<String> iter = cstr.iterator();
while (iter.hasNext()) {
  String element = iter.next();
  // Delete element if it has some property
  if (property(element)) {
     iter.remove();
  }
}
```

Removing elements

- Iterator also has a `remove()` method
  - Which element does it remove?
- The element that was last accessed using `next()`
- To remove consecutive elements, must interleave a `next()`

```
Collection<String> cstr = new ...;
Iterator<String> iter = cstr.iterator();
...
iter.remove();
iter.remove(); // Error
```

Removing elements

- Iterator also has a `remove()` method
  - Which element does it remove?
- The element that was last accessed using `next()`
- To remove consecutive elements, must interleave a `next()`

```
Collection<String> cstr = new ...;
Iterator<String> iter = cstr.iterator();
...
iter.remove();
iter.next();
iter.remove();
```

Removing elements

- Iterator also has a `remove()` method
  - Which element does it remove?
- The element that was last accessed using `next()`
- To remove consecutive elements, must interleave a `next()`
- To remove the first element, need to access it first

```
Collection<String> cstr = new ...;
Iterator<String> iter = cstr.iterator();

// Remove first element in cstr
iter.next();
iter.remove();
```

The `Collection` interface — the full story

- How does this line work?

  `if (element.equals(obj))`

- Actually, `Collection` defines a much larger set of abstract methods
  - `addAll(from)` adds elements from a compatible collection
  - `removeAll(c)` removes elements present in `c`
  - A different `remove()` from the one in `Iterator`
- To implement the `Collection` interface, need to implement all these methods!

```
public static <E> boolean
       contains(Collection<E> c, Object obj) {
  for (E element : c)
    if (element.equals(obj))
      return true;
    return false;
}
```

```
public interface Collection<E>{
  boolean add(E element);
  Iterator<E> iterator();
  int size() boolean isEmpty();
  boolean contains(Object obj);
  boolean containsAll(Collection<?> c);
  boolean equals(Object other);
  boolean addAll(Collection<? extends E> from);
  boolean remove(Object obj);
  boolean removeAll(Collection<?> c);
  ...
}
```

The AbsractCollection class

- To implement Collection, need to implement all these methods!
- "Correct" solution — provide default implementations in the interface
- Added to Java interfaces later!

```
public interface Collection<E>{
  boolean add(E element);
  Iterator<E> iterator();
  int size() boolean isEmpty();
  boolean contains(Object obj);
  boolean containsAll(Collection<?> c);
  boolean equals(Object other);
  boolean addAll(Collection<? extends E> from);
  boolean remove(Object obj);
  boolean removeAll(Collection<?> c);
  ...
}
```

The AbsractCollection class

- To implement `Collection`, need to implement all these methods!
- "Correct" solution — provide default implementations in the interface
- Added to Java interfaces later!
- Instead, `AbstractCollection` abstract class implements `Collection`
- Concrete classes now extend `AbstractCollection`
  - Need to define `iterator()` based on internal representation
  - Can choose to override `contains()`, ...

```
public abstract class AbstractCollection<E>
                         implements Collection<E> {
  ...
  public abstract Iterator<E> iterator();

  public boolean contains(Object obj) {
    for (E element : this)
      if (element.equals(obj))
        return true;
    return false;
  }
  ...
}
```

Concrete collections

- 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
- 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
- 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`
- 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
  - 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);
```

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
- 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