

CIS 120

Midterm 2 — November 8, 2007

SOLUTION

1. These multiple-choice questions test your knowledge of Java collections. All the interactions assume that the appropriate `import` statements have been used to make `List`, `ArrayList`, `Map`, and `HashMap` available. Circle the correct outcome from the interactions:

(a) `List<String> l = new ArrayList<String>();` *4 points*
`l.add("a");`
`l.add("b");`
`l.add("c");`
`l.remove(0);`
`l.remove(0);`
`l.get(1)`
`"a" "b" "c" exception`

ANSWER: exception

(b) `Map<String,Integer> m = new HashMap<String,Integer>();` *4 points*
`m.put("A", 1);`
`m.put("B", 2);`
`m.put("C", 3);`
`m.put("B", 4);`
`m.size()`
`1 2 3 4`

ANSWER: 3

(c) `List<Integer> l = new ArrayList<Integer>();` *4 points*
`for (int i = 0; i < 3; i++) l.add(i);`
`l.add(0, 3);`
`l.get(3)`
`1 2 3 exception`

ANSWER: 2

(d) `List<String> l = new ArrayList<String>();` *4 points*
`List<String> r = new ArrayList<String>();`
`l.add("a");`
`l.add("b");`
`l.add("c");`
`for (String s: l) r.add(0, s);`
`r.get(2)`
`"a" "b" "c" exception`

ANSWER: "a"

(e) `Map<Integer,Integer> m = new HashMap<Integer,Integer>();`
`for (int i = 0; i < 10; i++)`
`m.put(i, i+2);`
`int s = 0;`
`while (m.containsKey(s)) {`
`s = m.remove(s);`
`}`
`m.size()`
 loops 0 5 10

4 points

ANSWER: 5 (removes every other)

(f) `List<String> l = new List<String>();`
`StringBuffer b = new StringBuffer("a");`
`for (int i = 0; i < 3; i++) {`
`l.add(0, b.toString());`
`b.append("b");`
`}`
`Collections.sort(l);`
`l.get(2);`
 exception "a" "ab" "abb"

4 points

ANSWER: "abb"

2. For each of the following scenarios, write the full type of the data structure you'd use to store the data. Use some combination of the generic `Map`, `Set`, and `List`. Remember to include their generic types (eg. `List<String>`, not `List`) and that `Maps`, `Sets`, and `Lists` can be nested. You may assume that classes representing each of the objects in the scenario exist, and that all of the classes implement `equals` and `hashCode` appropriately so that they can be used in sets and as keys in maps. However do not assume that these objects store any particular data. Remember to use fast data structures when you don't need the features of the slower ones. Finally, you can call the classes anything reasonable (`Title`, `Artist`, `Price`,...).

Example: You want to organize the current roster of each NBA team so that players can be found by searching for their team. The answer is

```
Map<Team, Set<Player>>
```

(do not assume, for instance, that the `Team` object contains the set of players on that team)

- (a) An online music store needs to keep track for each artist of the artist's titles and respective prices. *4 points*

```
ANSWER: Map<Artist, Map<Title, Price>>
```

- (b) The wish list data structure for the music store keeps track for each customer of the set of titles the customer would like to have. *4 points*

```
ANSWER:  
Map<Customer, Set<Title>>  
Map<Customer, List<Title>>
```

- (c) We keep track of the popularity of titles in the music store by maintaining an ordered table of titles "graded on the curve": which titles are in the top 10% of sales, which are in the top 20% but not in the top 10%, which are in the top 30% but not in the top 20%, ... In other words, we have an ordered table of bins where each contains the titles in the appropriate sales bracket. *4 points*

```
ANSWER:  
List< Set<Title>>  
Map<Integer, Set<Title>>
```

- (d) The music store is getting into socially-based recommendations, and so it wants a data structure that stores for each customer what other customers have music interests in common with that customer. *4 points*

```
ANSWER:  
Map<Customer, Set<Customer>>
```

- (e) The store's new recommendation algorithm generates automatically for each customer some playlists, each containing some titles in a play order, based on information about the customer's tastes. These recommendations are stored in a data structure that can be efficiently accessed when a customer logs in to retrieve the playlists generated for that customer. *4 points*

ANSWER:

```
Map<Customer, Map<Playlist, List <Title>>>
```

```
Map<Customer, Map<String, List <Title>>>
```

3. In these questions, we will implement some `Iterators` based on other `Iterators` or collections. The `Iterator` interface is given in the handout for reference. In your implementations, you may need to throw some of the exceptions mentioned in class and described in the `Iterator` documentation in the handout.

- (a) Implement a *generic* iterator `ReverseIterator<E>` that returns the elements of a list (type `List<E>`) in reverse order. The list is passed as an argument to the iterator's constructor. The iterator should implement its `remove` method by using the appropriate `remove` method for `List<E>` (the documentation for `List<E>` is included for reference in the handout). 10 points

```
import java.util.*;

public class ReverseIterator<E> implements Iterator<E> {
    private List<E> list;
    private int index;
    public ReverseIterator(List<E> list) {
        this.list = list;
        index = list.size();
    }
    public boolean hasNext() {
        return index > 0;
    }
    public E next() {
        return list.get(--index);
    }
    public void remove() {
        list.remove(index);
    }
}
```

- (b) Implement a *generic* iterator `ZipperIterator<E>` that “zips” together two instances of `Iterator<E>` **14 points** provided as arguments to the `ZipperIterator` constructor to create an iterator over elements of type `E`. More specifically, if the first iterator would return in order the elements $x_0, x_1, \dots, x_n, \dots$ and the second iterator would return in order the elements $y_0, y_1, \dots, y_n, \dots$, the zipper iterator would return in order the elements $x_0, y_0, x_1, y_1, \dots, x_n, y_n, \dots$. In other words, the zipper alternates elements from the two argument iterators (thus its name). The zipper runs out of elements (`hasNext` is false) when the argument iterator that it would get the next element from has run out of elements. The zipper’s `remove` method removes the last returned element from the iterator that supplied it. Here are some DrJava interactions that show `ZipperIterator` in action:

```
> import java.util.*;
> List<String> l = new ArrayList<String>();
> List<String> r = new ArrayList<String>();
> l.add("a");
> l.add("b");
> l.add("c");
> r.add("A");
> Iterator<String> z = new ZipperIterator<String>(l.iterator(),r.iterator());
> while (z.hasNext()) {
    System.out.println(z.next());
    z.remove();
    System.out.println(l);
    System.out.println(r);
}
a
[b, c]
[A]
A
[b, c]
[]
b
[c]
[]
```

Write below (continue on the next page if needed) the definition of `ZipperIterator`. The main trick you need is to maintain an instance variable that indicates which of the two argument iterators will provide the next element to return.

ANSWER ON NEXT PAGE

```

import java.util.*;

public class ZipperIterator<E> implements Iterator<E> {
    private Iterator<E> left, right;
    private boolean nextIsLeft;
    public ZipperIterator(Iterator<E> left, Iterator<E> right) {
        this.left = left;
        this.right = right;
        nextIsLeft = true;
    }
    public boolean hasNext() {
        return nextIsLeft ? left.hasNext() : right.hasNext();
    }
    public E next() {
        E next;
        next = nextIsLeft? left.next() : right.next();
        nextIsLeft = !nextIsLeft;
        return next;
    }
    public void remove() {
        if (nextIsLeft)
            right.remove();
        else
            left.remove();
    }
}

```

4. In this problem, we will create a new `Library` class to hold a set of `Books` (we assume there are no multiple copies of the same book in our libraries), and some kind of catalog to allow searching efficiently for books meeting certain criteria. In this library, the catalog will simply allow us to search for books that contain certain index terms, taking advantage of the index that each `Book` provides. We will build the `Library` class step by step. You should refer to the handout for the Java collections interfaces that will be useful in these questions. Also, be careful with maps: if a key does not exist in a map, `get` for that key returns `null`, and *null is not the same as the empty set*. Before using the return value of `get`, you should check if the map contains the key. Finally, take advantage of convenient collection methods like `addAll`, `retainAll`, and the like, which can simplify hugely your answers.

- (a) The library data structures should include a collection of books, and a catalog data structure providing efficient way of finding which books contain a given index term. We don't care how many times a term occurs in a book, or where in the book, just whether the book contains the term. The same term may occur in multiple books, so your catalog data structure should allow for that. The `Library` constructor takes a `Collection` of `Books` as its single parameter, and saves a copy of that collection (no need to copy the books themselves, just the collection) in its own book collection data structure. In addition, the constructor should construct the catalog by examining each of the given books for index terms. When the constructor you will write below finishes, the `Library` instance should contain: a book collection, filled with the books in the collection passed in as the constructor argument, and a catalog mapping terms to the set of books in the library whose index contains the term. 14 points

```
import java.util.*;

public class Library {
    private Set<Book> collection;
    private Map<String,Set<Book>> termIndex;
    public Library(Collection<Book> books) {
        collection = new HashSet<Book>(books);
        termIndex = new HashMap<String,Set<Book>>();
        for (Book b: collection) {
            for (String term: b.getIndex().keySet()) {
                if (termIndex.containsKey(term))
                    termIndex.get(term).add(b);
                else {
                    Set<Book> idx = new HashSet<Book>();
                    idx.add(b);
                    termIndex.put(term,idx);
                }
            }
        }
    }
}
```

- (b) Write a method `Set<Book> searchSome(String[] terms)` that returns a set of the books in the library that contain *at least one* of the terms in the given array of terms. If the array is empty, an empty set should be returned. **8 points**

```
public Set<Book> searchSome(String[] terms) {
    Set<Book> found = new HashSet<Book>();
    for (String term: terms)
        if (termIndex.containsKey(term))
            found.addAll(termIndex.get(term));
    return found;
}
```

- (c) Write a method `Set<Book> searchAll(String[] terms)` that returns a set of the books in the library that contain *all* of the terms in the given array of terms. If the array is empty, the whole library collection should be returned. **10 points**

```
public Set<Book> searchAll(String[] terms) {
    if (terms.length == 0)
        return collection;
    Set<Book> found = new HashSet<Book>(termIndex.get(terms[0]));
    for (int i = 1; i < terms.length && !found.isEmpty(); i++) {
        if (termIndex.containsKey(terms[i]))
            found.retainAll(termIndex.get(terms[i]));
        else
            found.clear();
    }
    return found;
}
```