

CIS 120 — Programming Languages and Techniques

Midterm II

Answer key

October 30, 2009

1. (3 points) Circle “true” or “false.”

(a) `List<String>` is a subtype of `List<Object>`.

Answer: False

(b) `SortedSet<String>` is a subtype of `Collection<String>`.

Answer: True

(c) `Map<String>` is a subtype of `Collection<String>`.

Answer: False

2. (6 points) Briefly explain the differences between the standard interfaces `InputStream`, `Reader`, and `BufferedReader`.

Answer: An `InputStream` is a source for bytes. A `Reader` is a source for Unicode characters. A `BufferedReader` is like a reader except that it stores a large chunk of text internally, for efficiency, and uses it to return characters one at a time when asked. `BufferedReader`s also offer convenience methods for reading whole lines at a time and for marking the current position and resetting to this position later.

Grading scheme:

2 points: `InputStream` gives bytes

2 points: `Reader` gives characters

2 points: any one of:

`BufferedReader` can read a line at a time

`BufferedReader` is more efficient

`BufferedReader` has `mark/reset`

1 point off for egregious incorrect information

3. (10 points) Circle the type that would be most appropriate for representing the data structure described in each of the following scenarios.

(a) From a given **Room** in an adventure game, it is possible to move in a number of directions (represented as **Strings**), each leading to another **Room**.

- i. `Map<Room, String>`
- ii. `Map<String, Set<Item>>`
- iii. `Map<String, Room>`
- iv. `Set<String>`

Answer: iii

(b) My favorite game of solitaire is played by laying out a row of 13 stacks of cards with four cards in each stack. The order of the stacks (and the cards in each stack) is important.

- i. `Set<Set<Card>>`
- ii. `Map<Card, List<Card>>`
- iii. `List<Card>`
- iv. `List<List<Card>>`

Answer: iv

(c) An address book can be used to find someone's phone number given their first name and their last name.

- i. `Map<Set<String>, Phone>`
- ii. `Map<Map<String, String>, Phone>`
- iii. `Map<Phone, Map<String, String>>`
- iv. `Map<String, Map<String, Phone>>`

Answer: iv

(d) Every year, Sally takes her kids to the worldwide tattoo festival. These days they have their own tattoos to show off, but when they were young she used to play a game with them to keep them quiet: the goal was to find as many different tattoo configurations as possible among the passersby, where a "configuration" is the set of particular body parts that somebody has tattooed.

- i. `Map<BodyPart, Set<BodyPart>>`
- ii. `Map<Integer, Set<BodyPart>>`
- iii. `Set<Set<BodyPart>>`
- iv. `List<List<BodyPart>>`

Answer: iii

(e) The Cddb database is used by iTunes and other music programs to look up the names of the songs on a CD, given the lengths of the tracks. (It is a little surprising that this works, but it turns out that knowing just the lengths of all the tracks, in order, is enough to uniquely identify pretty much any CD!) Since title information can be entered into the database several times (e.g., for international releases), some CDs are listed with several variants of the song names.

- i. `Map<Integer, List<List<String>>>`
- ii. `Map<List<Integer>, Set<List<String>>>`
- iii. `List<Map<Integer, String>>`
- iv. `Map<Integer, Map<Integer, String>>`

Answer: ii

4. (15 points) For reference, the `LinkedListSimpleCollection` class discussed in the lectures is reproduced at the end of the exam, together with the auxiliary class `Node` and the `SimpleCollection` and `SimpleIterator` interfaces.

Suppose we wanted to extend `LinkedListSimpleCollection` with a method `stutter` that duplicates each element in the collection. For example, if the collection is currently holding the three elements "foo", "bar", and "baz" (in that order), then after calling `stutter()` the collection will hold the six elements "foo", "foo", "bar", "bar", "baz", and "baz" (in that order).

Complete the definition:

```
public void stutter() {  
  
}
```

Answer:

```
public void stutter () {  
    Node<E> curr = first;  
    while (curr != null) {  
        Node<E> n = new Node<E>(curr.element, curr.next);  
        curr.next = n;  
        curr = n.next;  
    }  
}
```

*Grading scheme: 0-5 points: badly broken / confused
6-10 points: some elements of correct solution present
11-15 points: correct ideas with errors in execution*

5. (16 points) For your reference, the final version of the `ArraySimpleCollection` class from the lecture notes is reproduced at the end of the exam. Here is a variant with an incorrect implementation of the `contains` method:

```
public class ArraySimpleCollection<E> implements SimpleCollection<E> {
    ...
    public boolean contains(Object o) {
        for (int i = 0; i < elements.length; i++) {
            if (elements[i] == null) return (o == null);
            if (elements[i].equals(o)) return true;
        }
        return false;
    }
}
```

- (a) Briefly describe a situation where the bad `contains` will return `false` when the correct answer is `true`. (Feel free to provide a concrete example if you wish, but make sure you also explain in words why it is handled incorrectly.)

Answer: The bad version will wrongly yield false if the collection contains the element being searched for but also contains null (and the null was added earlier).

- (b) Write a JUnit test capturing this situation — i.e., fill in the body of the method `testContains` below so that `testContains` succeeds when the correct implementation of `contains` is used, and fails when the incorrect implementation is used. Your test should include a single call to `assertTrue`.

```
public class TestSimpleCollection1 extends TestCase {

    public void testContains () {
        SimpleCollection<String> c = new ArraySimpleCollection<String>();
        // Fill in here:
    }
}
```

Answer:

```
public void testContains () {
    SimpleCollection<String> c = new ArraySimpleCollection<String>();
    c.add(null);
    c.add("foo");
    assertTrue(c.contains("foo"));
}
}
```

- (c) Briefly describe a situation where the bad `contains` will return `true` when the correct answer is `false`.

Answer: The bad version will wrongly yield true for the argument null when, for example, the collection is empty.

- (d) Write a JUnit test capturing this situation — i.e., fill in the body of the method `testContains` below so that `testContains` succeeds when the correct implementation of `contains` is used, and fails when the incorrect implementation is used. Your test should include a single call to `assertFalse`.

```
public class TestSimpleCollection2 extends TestCase {

    public void testContains () {
        SimpleCollection<String> c = new ArraySimpleCollection<String>();
        // Fill in here:
    }
}
```

```
    }  
}
```

Answer:

```
public void testContains () {  
    SimpleCollection<String> c = new ArraySimpleCollection<String>();  
    assertFalse(c.contains(null));  
}
```

Grading scheme: Descriptions graded out of 5, JUnit code out of 3.

Part (b) still got full credit (3/3) if it correctly checked the situation described in (a), even if (a) was incorrect. Same for part (d).

-1 for type errors in the JUnit tests.

-1 for using assertEquals instead of assertTrue/assertFalse.

For Reference: SimpleCollection, SimpleIterator, and Node

```
public interface SimpleCollection<E> {
    boolean add(E element);
    boolean contains(Object o);
    SimpleIterator<E> iterator();
}
```

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

```
public class Node<E> {

    public E element;
    public Node<E> next;

    public Node (E element, Node<E> next) {
        this.element = element;
        this.next = next;
    }
}
```

For Reference: LinkedSimpleCollection class

```
public class LinkedSimpleCollection<E> implements SimpleCollection<E> {

    private Node<E> first = null;

    public boolean add(E element) {
        Node<E> newnode = new Node<E>(element, first);
        first = newnode;
        return true;
    }

    public boolean contains(Object o) {
        for (Node<E> current = first; current != null; current = current.next) {
            if ( (current.element == null && o == null)
                || (current.element != null && current.element.equals(o))) {
                return true;
            }
        }
        return false;
    }

    public SimpleIterator<E> iterator() {
        return new LinkedSimpleIterator<E>(first);
    }

}
```

For Reference: ArraySimpleCollection class

```
public class ArraySimpleCollection<E> implements SimpleCollection<E> {

    private E[] elements;
    private int last;

    public ArraySimpleCollection () {
        elements = (E[]) (new Object[5]);
        last = -1;
    }

    public boolean add(E element) {
        last++;
        if (last == elements.length) {
            E[] newelements = (E[]) (new Object[elements.length * 2]);
            for (int i = 0; i < elements.length; i++) {
                newelements[i] = elements[i];
            }
            elements = newelements;
        }
        elements[last] = element;
        return true;
    }

    public SimpleIterator<E> iterator() {
        return new ArraySimpleIterator<E>(elements, last);
    }

    public boolean contains(Object o) {
        for (int i = 0; i <= last; i++) {
            if ( (elements[i] == null && o == null)
                || (elements[i] != null && elements[i].equals(o))) return true;
        }
        return false;
    }
}
```