DEPARTMENT OF COMPUTER SCIENCE

UCL

COMP2007 Coursework 1

Purpose: Warm-up coursework to get back into Java programming. In particular, make sure you understand interfaces, abstract classes, inheritance, overriding methods and using exceptions. You need to do this coursework properly if you are going to follow the rest of the course. There are a number of important concepts covered here, make sure you understand them all.

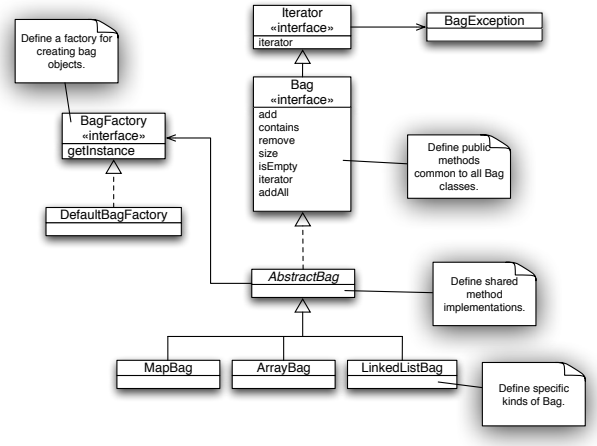
Submission: Complete this coursework by midnight Wednesday 21st October and submit online via Moodle (instructions will be sent by email).

Marking: This is a *collaborative* coursework that will be binary marked and is worth 2% of the overall module mark. Collaborative means that it is all right to work together on doing this coursework. The absolutely essential thing is that you individually fully understand the programming ideas and concepts involved. And that you are able to individually write the Java code and get it fully working. I can't emphasise enough how important this is. If you simply copy what someone else has done without understanding what is going on, you will end up failing the module.

Core Questions (must complete these)

This coursework is about implementing Bag classes, making use of interfaces, an abstract class, inheritance, overridden methods and exceptions. A Bag is a data structure that holds an unordered collection of objects. Duplicates are allowed and the bag knows how many occurrences (i.e., count) of each duplicate are in its collection. The inspiration for Bags comes from the Smalltalk language. The Java language features needed to implement these classes were all covered in course COMP1008 but it would be a good idea to read up on them first.

The goal is to set up a simple framework that will support several classes providing different concrete Bag implementations. The framework structure is shown below:



This framework is based on the principles used by the data structure classes in the standard Java libraries but is intended to illustrate a range of programming concepts rather than be a practical design for a real framework.

The Iterator interface is defined in the standard Java library and declares the iterator method, so that each value stored in a data structure can be accessed in sequence. Implementing the Iterator interface allows a data-structure object to be used with the enhanced for loop (foreach).

The Bag interface defines the common methods for all Bag classes. Bag extends the Iterator interface, so note that one interface can extend another. A variable of type Bag can reference any Bag object of a class that implements Bag (or inherits the implements relationship). This enables 'Programming to an Interface', allowing code to be written using the Bag type without needing to be concerned about the exact type of Bag of objects it is working with at runtime.

The Bag interface, documented using documentation comments, is defined as follows:

```
public interface Bag extends Iterable
{
  /**
   * The fixed maximum size of a bag.
   * This determines the maximum number of distinct values that can
   * be stored in a bag, not the number of occurrences of each value.
   */
  static final int MAX_SIZE = 1000;

   /**
    * Add a value to a bag.
    * @param object The value to add.
    * @throws BagException If the bag is full.
    */
  void add(Object object) throws BagException;

   /**
    * Add the given number of occurrences of value to a bag.
    * @param object The value to add.
    * @param occurrences The number of occurrences of the value.
    * @throws BagException If the bag is full.
    * Note that the bag holds a single copy of a given value, along with
    * the number of occurrences of that value. It does not store multiple
    * copies of the same value.
    */
  void addWithOccurrences(Object object, int occurrences) throws BagException;

  /**
   * Check if the bag contains a value.
   * @param object The value to look for.
   * @return True if the bag contains the value, false otherwise.
   */
  boolean contains(Object object);

  /**
   * Return the number of occurrences (count) of a value in the bag.
   * @param object The value to look for.
   * @return The number of occurrences.
   */
  int countOf(Object object);

  /**
   * Remove an occurrence of value from the bag. If the last occurrence is
removed,
   * remove the value as well. Do nothing if the value is not in the bag.
   * @param object The value to remove.
```

```
   */
  void remove(Object object);

  /**
   * Determine the number of distinct values stored in the bag. The number of
   * occurrences of each value is not taken into account.
   * @return The number of distinct values in the bag.
   */
  int size();

  /**
   * Check if the set is empty.
   * @return True if the set is empty, false otherwise.
   */
  boolean isEmpty();

  /**
   * Create a new Bag containing the contents of this and the argument Bag.
   * @param b The bag to add.
   * @return The new Bag.
   * @throws BagException If the bag becomes full while adding.
   */
  Bag addAll(Bag b) throws BagException;
}
```

Note that several methods are declared as throwing exceptions of type BagException. Also a default maximum size is imposed on all bags of 1000 distinct values (don't confuse a value, with occurrences of a value). A different maximum size can be specified when a bag object is created. If an attempt is made to to create a bag with size greater than the maximum size, or less than 1, an exception should be thrown. In addition, if adding a value to a bag using the add or addAll methods would cause it to exceed the maximum size an exception should be thrown.

The BagException class is:

```
public class BagException extends Exception
{
  /**
    * Use the default message.
    */
   public BagException()
   {
     super("Bag error");
   }

   /**
    * Provide a custom message.
    * @param message The message to store in the exception object.
    */
   public BagException(String message)
   {
     super(message);
   }

}
```

The AbstractBag class is an abstract class that provides common methods for concrete subclasses. This avoids duplicating the methods in the subclasses as they can just be inherited. An example addAll method is implemented, which creates a new bag and adds the contents of the bag it is called for and the argument bag to the new bag. The new bag object returned is created using a *Factory*. A factory is a class that specialises in creating objects. The use of a factory avoids

code having to name a specific concrete set class and decouples object creation from class naming. Factory is an example of a Design Pattern.

The AbstractBag class follows:

```
/**
 * This class implements methods common to all concrete bag implementations
 * but does not represent a complete bag implementation.<br />
 *
 * New bag objects are created using a Factory. By default a DefaultBagFactory
 * instance is used but this can be replaced with any factory implementing the
 * BagFactory interface.
 */
import java.util.Iterator;

public abstract class AbstractBag implements Bag
{
  private static BagFactory factory = new DefaultBagFactory();

  private Bag getNewBagInstance(Bag kind) throws BagException
  {
    return factory.getInstance(kind.getClass());
  }

  /**
   * Set the factory to be used for bag creation. Note this is a static
   * method so <em>all</em> bag objects created from now on
   * will be created by the new factory.
   * @param aFactory The factory to use.
   */
  public static void setFactory(BagFactory aFactory)
  {
    factory = aFactory;
  }

  public Bag addAll(Bag b) throws BagException {
    Bag result = getNewBagInstance(this);
    for (Object o : this)
    {
      result.add(o);
    }
    for (Object o : b)
    {
      result.add(o);
    }
    return result;
  }

}
```

Note that the AbstractBag class uses the DefaultMySetFactory but this can be changed at runtime by providing a different factory. As the factory is referenced via a static variable the same factory is used for creating all bag objects. Allowing this variability is a common strategy to facilitate testing, as test code can provide a special factory for testing purposes.

The interface BagFactory defines the public methods needed by a factory. This again allows the Programming to an Interface concept to used in code that uses a factory.

```
/**
 * This interface defines the public methods that a Bag factory must
 * implement.
```

```
 */
public interface BagFactory
{
  /**
   * Create a bag that is an instance of the same class as the parameter object.
   * @param bagClass A class object used to determine the class of the bag to
instantiate.
   * @return The new bag.
   * @throws BagException If the class is not recognised as one from
   * which a bag object can be created.
   */
  public Bag getInstance(Class bagClass) throws BagException;

  /**
   * Create a bag that is an instance of the same class as the parameter object,
with the
   * given maximum bag size.
   * @param  bagClass A class object used to determine the class of the bag
   *  to instantiate
   * @param maxSize The maximum size of the new bag.
   * @return The new bag.
   * @throws BagException If the class is not recognised as one from
   * which a bag object can be created.
   */
  public Bag getInstance(Class bagClass, int maxSize) throws BagException;

}
```

Note that the getInstance factory methods take a value of type Class, which is used to determine the class of the bag object to be created. Java represents classes at runtime with objects of type Class and an object can be asked for its class.

The DefaultBagFactory class looks like this:

```
/**
 * The default bag factory. The getInstance method will return a new instance of
 * the Bag implementation class determined by the class of the parameter.
 */
public class DefaultBagFactory implements BagFactory
{

  public Bag getInstance(Class bagClass) throws BagException
  {
    return getInstance(bagClass, Bag.MAX_SIZE);
  }

  public Bag getInstance(Class bagClass, int size) throws BagException
  {
    if (bagClass.equals(ArrayBag.class))
    {
      return new ArrayBag(size);
    }
    throw new BagException
      ("Attempting to use DefaultBagFactory to create something that is not a
Bag");
  }
}
```


To illustrate how to implement a concrete bag class as a subclass of AbstractBag, here is a class that uses an Arraylist of Elements as the private internal data structure used to store the values in the bag. Each elements holds a value (object reference) and an occurrence count.

```
import java.util.Iterator;
import java.util.ArrayList;

public class ArrayBag extends AbstractBag
{
  private static class Element
  {
    public int count;
    public Object value;
    public Element(int count, Object object)
    {
      this.count = count;
      this.value = object;
    }
  }

  private int maxSize;
  private ArrayList<Element> contents;

  public ArrayBag() throws BagException
  {
    this(MAX_SIZE);
  }

  public ArrayBag(int maxSize) throws BagException
  {
    if (maxSize > MAX_SIZE)
    {
      throw new BagException("Attempting to create a Bag with size greater than
maximum");
    }
    if (maxSize < 1)
    {
      throw new BagException("Attempting to create a Bag with size less than
1");
    }
    this.maxSize = maxSize;
    this.contents = new ArrayList<Element>();
  }

  public void add(Object object) throws BagException
  {
    for (Element element : contents)
    {
      if (element.value.equals(object))
      {
        element.count++;
        return;
      }
    }
    if (contents.size() < maxSize)
    {
      contents.add(new Element(1,object));
    }
    else
    {
      throw new BagException("Bag is full");
```

```
    }
  }

  public void addWithOccurrences(Object object, int occurrences) throws
BagException
  {
    for (int i = 0 ; i < occurrences ; i++)
    {
      add(object);
    }
  }

  public boolean contains(Object object)
  {
    for (Element element : contents)
    {
      if (element.value.equals(object))
      {
        return true;
      }
    }
    return false;
  }

  public int countOf(Object object)
  {
    for (Element element : contents)
    {
      if (element.value.equals(object))
      {
        return element.count;
      }
    }
    return 0;
  }

  public void remove(Object object)
  {
    for (int i = 0 ; i < contents.size() ; i++)
    {
      Element element = contents.get(i);
      if (element.value.equals(object))
      {
        element.count--;
        if (element.count == 0)
        {
          contents.remove(element);
          return;
        }
      }
    }
  }

  public boolean isEmpty()
  {
    return contents.size() == 0;
  }

  public int size()
  {
```



```
    return contents.size();
  }

  private class ArrayBagIterator implements Iterator
  {
    private int index = 0;
    private int count = 0;

    public boolean hasNext()
    {
      if (index < contents.size()) {
        if (count < contents.get(index).count) return true;
        if ((count == contents.get(index).count) && ((index + 1) < contents.size
())) return true;
      }
      return false;
    }

    public Object next()
    {
      if (count < contents.get(index).count)
      {
        Object value = contents.get(index).value;
        count++;
        return value;
      }
      count = 1;
      index++;
      return contents.get(index).value;
    }

    public void remove()
    {
    }
  }

  public Iterator iterator()
  {
    return new ArrayBagIterator();
  }
}
```

Note the way that a private nested class is used to implement the Iterator interface to enable iteration of the bag object. Don't confuse Iterable with Iterator. Iterable defines the iterator method, while Iterator defines the next, hasNext and remove methods.

Note that class ArrayBag throws exceptions *but does not catch any of its own exceptions*. A bag throws exceptions, while code using a bag catches any exceptions using try/catch blocks. Make sure you understand the point being made here.

Finally, here is some example code that uses the classes and interfaces above:

```
/**
 * Example code illustrating the use of Bag objects.
 */
public class Main
{
  private BagFactory factory = new DefaultBagFactory();

  public void print(Bag bag)
  {
```

```
    boolean first = true;
    System.out.print("{");
    for (Object obj : bag)
    {
      if (!first) { System.out.print(" , "); }
      first = false;
      System.out.print(obj);
    }
    System.out.println("}");
  }

  public void go()
  {
    Bag bag1;
    Bag bag2;
    try
    {
      bag1 = factory.getInstance(ArrayBag.class);
      bag1.add(1);
      bag1.add(2);
      bag1.add(3);
      print(bag1);
      bag2 = factory.getInstance(ArrayBag.class);
      bag2.add(2);
      bag2.add(2);
      bag2.add(4);
      print(bag2);
      System.out.print("AddAll: ");
      print(bag1.addAll(bag2));
    }
    catch (BagException e)
    {
      System.out.println("====> Bag Exception thrown...");
    }
  }

  public static void main(String[] args)
  {
    new Main().go();
  }
}
```

Q1. Implement a concrete class MapBag that uses an HashMap as its private data structure.

Q2. Implement a concrete class LinkedListBag that uses a chain of linked list elements as its data structure. Do not use any linked list classes from the standard Java class libraries. A linked list element looks like this:

```
  private static class Node
  {
    public Object value;
    public int occurrences;
    public Node next;

    public Node(Object value, int occurrences, Node next)
    {
      this.value = value;
      this.occurrences = occurrences;
      this.next = next;
    }
```



```
  }
```

This should be a nested class inside the LinkedListBag class.

Q3. Add a method called toString() to the AbstractBag class, that returns a string representation of the contents of a bag in this style: [value: occurrences, value: occurrences, ...]

```
    String toString();
```

Note, no code in the concrete subclasses must be changed. The method in the abstract class should be inherited unchanged by subclasses.

Q4. Add a bag removeAllOccurrences method to class AbstractBag and confirm it works with all the concrete classes. The method should have this signature and be declared in the Bag interface:

```
    void removeAllOccurrences(Object object);
```

Q4. Add a bag subtract method, again in class AbstractBag, that returns a new Bag containing all values and occurrences that occur in the `this` bag but not the argument bag.

```
    Bag subtract(Object object);
```

Challenge (hard)

Q5. Convert the Bag framework to implement generic classes and interfaces. This would allow you to write code such as:

Bag<Integer> bag = new Bag<Integer>();

The code for the framework described can be downloaded from Moodle. Note, I make no guarantee that this code is free from errors! Also, some parts, such as the nested iterator class in ArrayBag could do with some refactoring.