Extreme Java!

Mastering the Java Language

Exercise book

Version 1.5

01a Multi-Threading

Detecting Inconsistencies

Exercise description:

Below is a program that:
a. Creates a list populated with 10,000 identical items (int values). (Increase the list size if the program runs too quickly for effective measurements).
b. Creates 20 "reader" threads that repeatedly check if the list is consistent, by verifying that every item is identical to the first one.
c. Creates one "writer" thread that repeatedly updates the list.

Since no synchronization mechanisms are used, the "reader" threads obviously detect multiple cases of inconsistency. For each inconsistency detected, the reader threads print an exclamation mark.

Step 1:

Explain why the inconsistency problem cannot be fixed using the Collections.synchronizedList method.

Step 2:

Solve the inconsistency problem using the standard **synchronized** mechanism.

Step 3:

Use a **semaphore** so that the main program will know when all other threads have finished, so it will measure and print the total runtime after all threads are done.

Step 4:

Replace the synchronization mechanism from Step 2 with a Readers/Writer Lock. Measure the performance difference obtained by this change.

Will the difference be greater or smaller in a multi-processor system?

Step 5 (optional):

Can you further improve the program's behavior by changing it so it will *not* use autoboxing?

Program source code:

```
public class RWDemo {
    public static void main(String[] args) throws Exception {
        List<Integer> list = new ArrayList<Integer>();
        for (int j = 0; j < 10000; j++) {
            list.add(0);
        }

        new WriterThread(list).start();
        for (int i = 0; i < 20; i++) {
            new ReaderThread(list).start();
        }
        System.out.println("All created.");
    }
}

class ReaderThread extends Thread {
    public static final int READER_LOOP = 100;

    List<Integer> list;

    public ReaderThread(List<Integer> l) {
        list = l;
    }

    public void run() {
        for (int i = 0; i < READER_LOOP; i++) {
            int first = list.get(0);
            for (int n = 1; n < list.size(); n++) {
                  int current = list.get(n);
                if (current != first) {
                    System.out.print("!"); // Inconsistent view detected.
                    break;
                }
            }
        }
        System.out.println("Reader Done.");
    }
}

class WriterThread extends Thread {
    public static final int WRITER_LOOP = 20;

    List<Integer> list;

    public WriterThread(List<Integer> l) {
        list = l;
    }

    public void run() {
        for (int i = 0; i < WRITER_LOOP; i++) {
            int newVal = (int)(Math.random() * 100000);
            for (int j = 0; j < list.size(); j++) {
                list.set(j, newVal);
            }
        }
        System.out.println("Writer Done.");
    }
}
```

01b Multi-Threading

Reader-Writer List

Exercise description:

Create a static method, similar to **Collections.synchonizedList**, that will return a synchronized wrapper for a given list; however, this wrapper will use a Readers/Writer lock for its synchronization.

Can you use this method to solve exercise 01a?

Can you explain why didn't the JRE developers upgrade **Collections.synchronizedList** in version 1.5 to use a Readers/Writer lock?

02 Java IO

Advanced Object Serialization

Exercise description:

Applications that serialize data structures are very useful and common. Unfortunately, sometimes these data structures contain objects that are not **serializable** (For example **java.nio.ByteBuffer**)
We want to write a solution that will enable serialization of such data structures.

Step 1:

Write a **ByteBufferWrapper** that wraps a **ByteBuffer**. The wrapper should implement the **externalizable** interface and include methods for reading and writing from and to object streams.

```
public class ByteBufferWrapper  implements Externalizable {
     private ByteBuffer  buf;
     ...
}
```

Check your code by reading and writing a Map containing **ByteBuffer** objects.

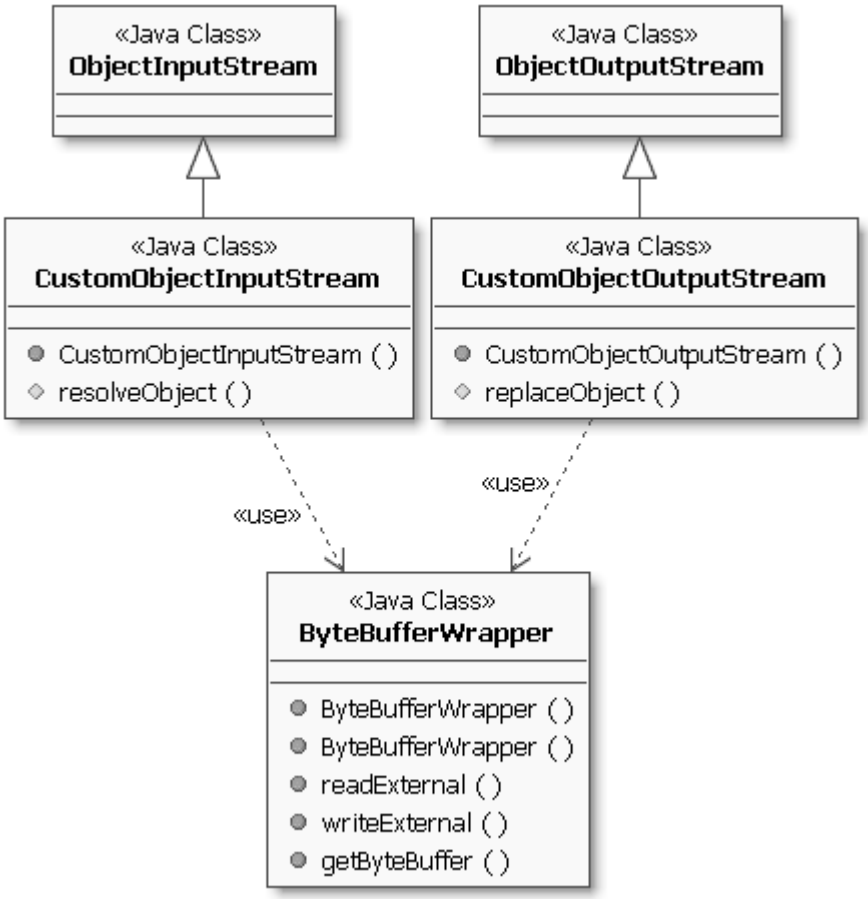
Step 2:

Write a **CustomObjectOutputStream** and a **CustomObjectInputStream** that will extend the **ObjectStream** objects and add the ability to handle **ByteBuffer** objects.

```
class CustomObjectOutputStream extends ObjectOutputStream
class CustomObjectInputStream extends ObjectInputStream
```

Check your code by reading and writing a Map containing **ByteBuffer** objects.

Class Diagram:



03 Memory Management

Object Pool implementation

Exercise description:

Object pools are a useful method to reduce the cost of object creation and garbage collection. Usually we want to implement a generic object pool that will work with any object needed for pooling.

Step 1:

Write a pool that uses a factory, which is in charge of object creation and initialization. Using a factory instance will allow loose coupling between the pooled objects and the pool implementation.

```
interface ObjectFactory<E> {
     E newObject();                  // allocate new object
     void objectActivated(E obj);    // called when object it is
                                     // handed out to consumers
     void objectDeactivated(E obj); // called when object is
                                     // put back into the pool
     ...
}

class BasicPool<E> {
     private ObjectFactory<E> factory;
}
```

Step 2:

Add dynamic capabilities to the pool.
When creating the pool, some dynamic properties are passed:
InitSize – Number of objects to create when initializing the pool
Limit – The maximum number of objects allowed
Mode – one of 3 available modes: IGNORE_LIMIT, ERROR_WHEN_LIMIT_REACHED, and WAIT_WHEN_LIMIT_REACHED. Explained below.
Timeout – a Timeout for the WAIT_WHEN_LIMIT_REACHED mode

Available modes:
IGNORE_LIMIT – Continue creating objects even if the limit has been reached. Take the risk of an OutOfMemoryError thrown from the JVM
ERROR_WHEN_LIMIT_REACHED – Throw an exception when the limit is exceeded
WAIT_WHEN_LIMIT_REACHED – Block the thread until an object is available. The thread will block for a maximum time of timeout.

Question:

What is problematic in the following code?

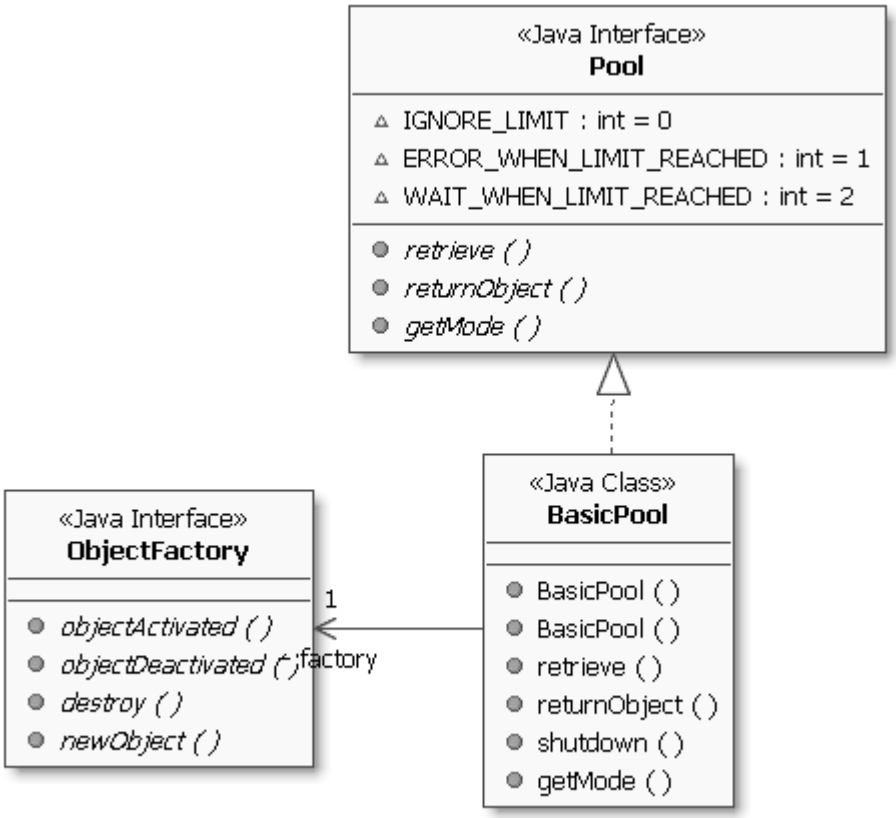
```
Pool myPool<Object> = …

// Retrieve 1000 objects (assuming business logic indeed
// requires 1000 objects simultaneously):
int size = 1000;
List<Object> list = new ArrayList<Object>(size);
for(int j = 0; j < size; j++)
     list.add(myPool.retrieve());

// … use the objects

// Return to pool:
for(int j = 0; j < size; j++){
     Object obj= list.remove(list.size() - 1);
     myPool.returnObject(obj);
}
```

Class Diagram:



04 Reflection

Overriding Class Behavior

Exercise description:
By changing classes at load time, we can change their behavior in every way we wish (almost). In this exercise, we will prevent a class from using System.exit.

Step 1:
Write a simple HelloWorld class that prints something, and then calls System.exit(1).

Step 2:
Create and use a class-loader that will prevent HelloWorld from using System.exit. Every time System.exit should have been called, the program will instead output "Exit attempted with status code *n*" (where *n* is the argument to System.exit), and keep on running (i.e., no exit takes place).

Tip:
Check how class names are stored inside class files (e.g. by using a hex viewer).

05 Data Structures

LRU Cache implementation

Exercise description:

A cache mechanism is very useful to improve performance, but usually there is a limitation in size and number of objects stored in the cache. For smart caching, there should be a policy for putting and removing objects. For this exercise, we implement a **Least-Recently-Use** policy.

The cache will hold key-value pairs, which are used through out the application.

Usage example:

```
Double sin = cache.get(angle);
if (sin == null){
     sin = Math.sin(angle);
     cache.put(angle,  sin);
}
// use sin …
```

Implement using inheritance or composition:

```
class Cache<K,V> {
     Cache(int maxSize)

     void put(K key, V value)

     V get(K key)
}
```

Notes:

1. All actions performed by the cache on data structures should perform with **O(1)**!
2. JDK 1.4 has a new collection class called **LinkedHashMap** that can be used to implement an LRU cache mechanism. For this exercise, do not use it!

Class Diagram: