Assignment 1:
Benchmarking data structures from the Java API

This assignment is due on: Tuesday, February 11th 23.00.

Anything that is turned in after this deadline, will not *be checked.*

assistent(s) Auke Wiggers en Tim van Rossum
email tim.vanrossum@student.uva.nl en wiggers.auke@gmail.com

author(s) Marten Lohstroh, Stephan Schroevers

University of Amsterdam

Department of Computer Science

Faculty of Science

1 Introduction

The Java API already offers a vast selection of Abstract Data Type (ADT) interfaces and implementations which can be conveniently used by the programmer. The accompanying course materials and lectures mainly focus on:

- acquaintance with several data structures;
- how some of their implementations work;
- why one implementation is better (faster) than another for a specific purpose, and how this can be expressed in terms of computational complexity[1].

The practical assignments—which will consist of Java programming exercises—aim at helping you understand the theory behind data structures while teaching you how to make the best use of what the Java API has to offer.

2 Getting Started

This first assignment is intended for you to refresh your Java programming skills and to collect the necessary additional knowledge that is required to successfully fulfill the followup assignments. It is assumed that you have basic knowledge of the Java programming language and the concepts of Object Oriented Programming (OOP). To refresh your memory with regard to basics such as variables (primitive types, default values, literals, ...), operators, expressions, statements, blocks, and control flow statements, have a look at the Sun Java tutorial page Lesson: Language Basics. To fresh up your OOP knowledge, check out Lesson: Object-Oriented Programming Concepts, where you can read about objects, class inheritance and interfaces.
Furthermore, we would like you to be aware of the Java coding conventions and we expect you to apply them properly. All handed out classes are delivered with Javadoc HTML pages that are derived from their source code comments. Reading this documentation is essential for you in order to accomplish each assignment. We also expect you to comment your own code sufficiently, but we don't expect you to write full-fledged Javadoc. However, if you are interested in writing (and compiling) Javadoc, you can read about it here.
Other things you might not yet be familiar with are generic and parameterized types and methods, collectively known as *generics*. These constitute the most recent major extension of the Java programming language. Since many of the most popular JDK classes are generic since Java 5.0, there is no way around them. Although generics are considered very useful, they are conceptually rather unintuitive and require a substantial learning curve. Therefore, it is highly recommended that you read Sun's Generics Tutorial.
Also note that you must use (at least) **JDK 1.5** for all assignments. Hence the code must be compatible with the **Java 5.0** (or higher) standards.

3 Description

In this assignment you will be timing the mutations (population and depopulation) of various data structures provided by the Java standard library that either implement the `List` or `Queue` interface. It is your task to write a Java program for this experiment, solely based on the Javadoc description of its classes—which is available online here.
First of all, you must construct the timing framework which consists of an abstract class called `CollectionTimer` and the two concrete classes `ListTimer` and `QueueTimer` that both extend the abstract class. Though `CollectionTimer`

[1]Note that besides the time complexity of an algorithm, also its space complexity can be evaluated.

implements the `insert()`, `extract()`, and `time()` method, the remaining methods stay abstract since they require a specific implementation for each distinct ADT interface. In this case we use lists and queues, hence we need the subclasses `ListTimer` and `QueueTimer`. Using the provided documentation, writing these classes should be fairly straightforward.
Furthermore, let us have a look at the `Assignment1` class which provides the main method of the program. Here you must read parameters from the commandline and execute the timing experiment accordingly. When you read its class description, it probably won't go unnoticed that the field summary shows the following rather peculiar field declaration:

```
private ArrayList<List<Integer>> lists;
```

This tells us that the class variable `lists` is not just of the type `ArrayList`, but it's moreover parameterized with `List`, which in turn is parameterized with `Integer`. This indeed sounds a bit confusing, but it expresses the following constraints:

- `lists` is of type `ArrayList`;
- `lists` may only contain `List` elements;
- each `List` element may only contain `Integer` elements.

Notice that `List` is not a concrete class but an interface. Moreover, it is a parameterized interface and should thus be referred to as `List<E>`. Therefore, we could also just say that `lists` may only contain elements that implement `List<Integer>`. A schematic view of the relations between the mentioned class and interface:

```
          Collection<E>
                |
  List<E> extends Collection<E>
                |
 ArrayList<E> implements List<E>
```

In Java, interface names and the names of their implementations are often used interchangeably, since the interface defines the feature set of its implementation. Similarly, you are probably familiar with subtyping and supertyping; e.g. since the `Integer` class extends the `Object` class, it's legal to cast an `Integer` to an `Object` and treat it like one. However, for some generic type declaration `G<E>`, it is not the case that `G<Integer>` is a subtype of `G<Object>`. For example, let's say you are interested in declaring a collection that may only contain elements that *extend* the `AbstractList<Integer>` class. Looking at the above example, your first guess would probably be something like this:

```
private ArrayList<AbstractList<Integer>> lists;
```

However, this expression will *not* enforce your requirement. Instead, you will need a bounded wildcard like this:

```
private ArrayList<?  extends AbstractList<Integer>> lists;
```

There are no constructions involving wildcards in this particular assignment though. Aside from this note about generics, the class description for `Assignment1` in the Javadoc mentioned above really provides all information that is required for you to implement it properly.

4 Tips

- Enhanced for-loops (also referred to as for-each loops) are available since Java 1.5.
- `Object.getClass().getSimpleName()` can be used to request the class name of an object contained in a generic container.
- For the execution of `Assignment1.java`, you should use the `-Xlint` parameter (which disables the HotSpot JIT compiler) in order to make the results more accurately comparable.

5 Criteria

Please be aware that compile warnings will cost you valuable points and compile errors are *not* tolerated (no points will be accredited). Furthermore, you must take care of proper exception handling throughout the program (e.g. an attempt to extract more elements from an ADT than were previously inserted into it must be handled properly).
The bottom line of this assignment is that your code must be in exact accordance with the specifications in the Javadoc class descriptions. If, however, you experience much trouble handling the commandline arguments, you may opt to replace the optional `-s` switch with a mandatory first argument, yielding the following synopsis:

```
java Assignment1 seed [mutations]
```

Note that in this situation the `seed` argument is not between square brackets: it must be mandatory, since it would otherwise not be possible to determine whether the first argument is the seed or the first mutation.
Additionally we would like you to answer the following question:

> "How will the implementation of `removeElement()` affect the performance of the different ADTs?"

Please answer this question in a comment at the appropriate place(s) in the code!

6 Hand-in Instructions

The assignment must be submitted on BlackBoard. Please turn in your files structured like this:

```
assignment_1 (folder)
        |_*.java
```

To submit this on BlackBoard you will have to compress the folder using either TAR, ZIP or RAR. Other formats are *not* allowed. The file you turn in should be named like `student-id_'assignment'_#.(tar / zip / rar)`. Example:

```
1234567_assignment_1.zip
```

All files will be automatically decompressed before checking. So, turning in files named in a different way can cause difficulties! Also, please take care that you only turn in **.java** files and NO *.class* files or temporary files made by your editor.
The deadline is February 15th, 23.00. This deadline is *very strict*. Also, beware that the assignment is strictly individual. We will use fraud detection software to reduce the possibility of fraud. If caught, severe penalties apply. Lastly, if you are really concerned about your Java skills, refer to the other lessons of the Sun Java tutorial.