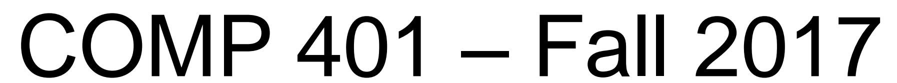

COMP 401 – Fall 2017

Recitation 7: Factories and Lists

Agenda

- High-level introduction to Factories
- Factory Example/Exercise
- Introduction to Lists
- List Performance Exercise
- Quiz

Copyright (C) 2017 Alex Blate. All Rights Reserved

Recitation Source Code

- Please download and import the project of:

  `recitation7_source_project.zip`

- You should see the following source files:

```
COMP401F17_Recitation7/src/recitations/recitation7/ALine.java
COMP401F17_Recitation7/src/recitations/recitation7/InstanceGenerator.java
COMP401F17_Recitation7/src/recitations/recitation7/Line.java
COMP401F17_Recitation7/src/recitations/recitation7/ListOfLinesDemo.java
COMP401F17_Recitation7/src/recitations/recitation7/ListPerformanceDemo.java
COMP401F17_Recitation7/src/recitations/recitation7/RandomUtils.java
COMP401F17_Recitation7/src/recitations/recitation7/Timer.java
```


Copyright (C) 2017 Alex Blate. All Rights Reserved

For `instance`

- I want an instance of an object of type `Foo` (or something implementing `Foo`)
- Our current method: Instantiate some concrete implementation of a `Foo` using a constructor, e.g.:

```
Foo foo = new FooImpl(…);
```

- This requires:
  - Knowing one or more specific implementations for `Foo`
  - Knowing the semantics for their constructors
  - "Locking in" our choice of implementation
- In many cases, one or more of these is undesirable.

Copyright (C) 2017 Alex Blate. All Rights Reserved

Factory Pattern

- Suppose we have want an instance of some Interface, class, or derived class of type `Foo`
- In the Factory Pattern, we are provided (or build) a "factory" for `Foo`s, say `FooFactory`, which provides one or more (usually, but not always) `static` methods that return instances of Foo.
  - NB: see also the Builder pattern which similar to but not the same as Factory
- In this example, `FooFactory` might contain methods:

```
FooFactory.getInstance()
FooFactory.getInstance(int initialCapacity)
```

  etc.
- In some cases, it may not be possible to directly instantiate object returned – the Factory is the only means of obtaining an instance.

Copyright (C) 2017 Alex Blate. All Rights Reserved

Example/Exercise

- Have a look at `InstanceGenerator`
  - Is it a Factory?
  - For what data types is it a Factory?
  - What method(s) comprise its Factory “interface”
  - What concrete `List` implementation is returned by `getListInstance()`?

- Run `ListOfLinesDemo` (opens a `List` of `Line`s in OE)
  - What do you see?
  - Try fiddling with the code

Copyright (C) 2017 Alex Blate. All Rights Reserved

Aside: `RandomUtils`

- After building the factory, I decided to factor out the methods that assist in generating random lines
- This content is in class `RandomUtils`
- Note the instantiation of static member rng, around line 21:

```
    public static Random rng =
            new Random(System.currentTimeMillis());
```

- What does the argument to the constructor do? (see also the comments above the declaration)
- Are there cases where I might want to do something else here?

Copyright (C) 2017 Alex Blate. All Rights Reserved

Collections

- I recommend that you spend a bit of time studying the Java Collections Framework (JCF): https://docs.oracle.com/javase/7/docs/technotes/guides/collections/index.html
- Aside from containing almost every basic data structure you will need for general tasks, I think it is a good example of a well-designed class-interface hierarchy and a good exposition of Generics

Copyright (C) 2017 Alex Blate. All Rights Reserved

Generics (very brief intro)

- Java 1.5 introduced Generics, which provide a degree of polymorphism
- Idea: I want to write a class or a method that operates on objects, but the implementation doesn't really care what specific objects they are.
- You could do this by simply operating on instances of `Object`; that's how we did things for a long time.
- However, I would like to take advantage of strong type-checking at compile time and avoid the proliferation of type-casting

Copyright (C) 2017 Alex Blate. All Rights Reserved

Generics (brief intro, continued)

- Semantically, I want to declare that my class or method operates on some type T, but I don't care what the actual type of T is.
- Semi-trivial example:

```
public <T> T getArrayElement(T[] array, int index) {
  if(index >= 0 && index < array.length) {
    return array[index];
  } else {
    return null;
}
```

- By the way, what does this do?

Copyright (C) 2017 Alex Blate. All Rights Reserved

Lists

- A good definition of the semantics of a list is given in the JavaDoc for the `List` interface:
  - "An ordered collection (also known as a sequence). The user of this interface has precise control over where in the list each element is inserted. The user can access elements by their integer index (position in the list), and search for elements in the list."

    Source: https://docs.oracle.com/javase/7/docs/api/java/util/List.html
- There are many concrete implementations of `List`, several of which are included in the Standard Platform Libraries; the most commonly-used of these are:
  - `ArrayList`
  - `LinkedList`

Copyright (C) 2017 Alex Blate. All Rights Reserved

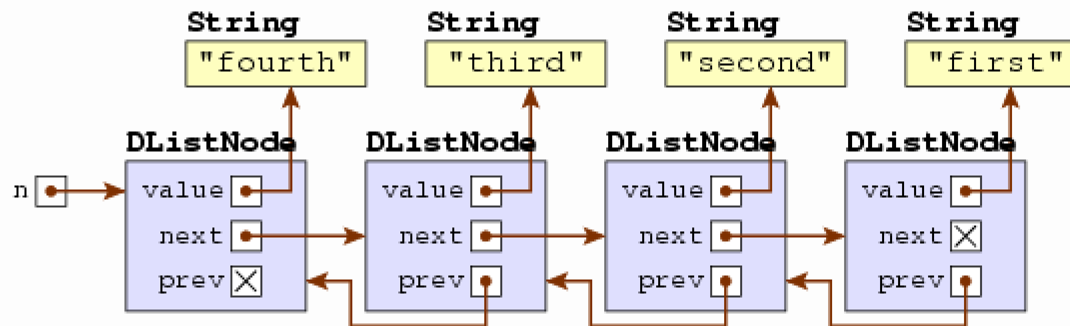
`ArrayList`

- The ordered sequence is stored in an array. Said another way, the backing store for the sequence is an array.
- The array is dynamically resized as elements are added (and possibly removed)
- `ArrayList` provides constant-time random access to elements by index (as you would expect) and (essentially) constant-time insertion at and removal from the end of the list.
- (Roughly) Linear-time insertion and removal at arbitrary points in the list.
- `ArrayList` has a constructor that allows you to specify its initial capacity… why might this be helpful?

Copyright (C) 2017 Alex Blate. All Rights Reserved

Linked Lists

- The ordered sequence is stored in a "chain" of objects called nodes.
- Each node has a reference to the next (and possibly) previous node, allowing sequential traversal
- Graphical Example:



https://goo.gl/images/m8vBpC

- This is basically a degenerate digraph.

Copyright (C) 2017 Alex Blate. All Rights Reserved

Linked Lists (continued)

- Constant-time insertion and removal at the head and (usually) the tail of the list.
- An efficient way to implement (de)queues, stacks, and other fun data structures where operations occur at the ends of the list.
- Using a List Iterator, or equivalent, constant-time insertion and removal at arbitrary locations in the list
- *Linear time* access to elements by index
- Node objects must be allocated (and reaped) for each element added (or removed)

Copyright (C) 2017 Alex Blate. All Rights Reserved

Lists

- I have alluded to performance differences between two list implementations.
- Consider the following operations:
  - Adding *N* elements to a list
  - Removing the first element of a list *N* times
- How do we think the performance of these operations will (or won't) differ when applied to `ArrayList` and `LinkedList`?

Copyright (C) 2017 Alex Blate. All Rights Reserved

List Performance Example/Exercise

- Open up and run `ListPerformanceDemo`
- Try running it for different values of `TEST_SIZE_LINES`
  - Think about the relationship between the list size and the time taken for each operation
- Hint: Read through the rest of the source code and maybe play with the several methods towards the end (this may help on the quiz ☺)
- Exercise: Modify method `runPerformanceTestForList` such that it also prints the operation rate (operations per unit time) and time per operation for each test.

Copyright (C) 2017 Alex Blate. All Rights Reserved