CS 310: HW 1, Code Complexity, Generics Review

Chris Kauffman

Week 2-1
Logistics

At Home

- Read Weiss Ch 5: Big-O
- Read Weiss Ch 15: ArrayList implementation
- HW 1: Posted, due end of next week

Reminder to DrJava Users

- Consider Using GMU Edition of DrJava
- Download here: https://cs.gmu.edu/~kauffman/drjava/

Goals

- Overview of HW 1
- Review of Generics (Weiss Ch 4.6-4.7)
- ArrayList: first building and analysis
HW 1 Is Up

It’s a good one:
http://www.cs.gmu.edu/~kauffman/cs310/hw1.html

- Inifinite Gomoku
- DenseBoard: Dense arrangement of generic elements
- ExpandableBoard: Interface, likely used for HW2
- Target complexities for some methods
- Elements can be set once
- Determining longest sequence is interesting
- Analyze an expansion approach
- Due Sunday next week but **start early** like today
- Unit Tests up by end of this week
What does the following statement mean?

The method addRow() should run in $O(C)$ time where $C$ is the number of columns.

What are some tips/tricks for analyzing code for its Big-O complexity?

Name some "relatives" of Big-O and how they differ
An Informative Example

Weiss Exercise 5.44

```java
public static String makeLongString1( int N ){
    String result = "";
    for( int i = 0; i < N; i++ )
        result += "x"; // trouble
    return result;
}
```

- What is the Big-O complexity of this method?
- Is there a more efficient way to concatenate strings?
Where did this data come from?
Does this plot confirm our analysis?
How would we check?
Analysis

Linear

> summary(linmod)

Coefficients:  
  Estim   Pr(>|t|)
(Intercept)  7.26 <2e-16 ***
poly(N, 1)  16.25 <2e-16 ***
poly(N, 2)  -0.34   0.287 
poly(N, 3)  -0.01   0.962

Why these coefficients?

Quadratic

> summary(quadmod)

Coefficients:  
  Estim   Pr(>|t|)
(Intercept)  83.89 <2e-16 ***
poly(N, 1)  278.16 <2e-16 ***
poly(N, 2)  54.75 <2e-16 ***
poly(N, 3)  -0.24   0.562
Review of Java Generics

- Generics allow type parameters for classes
- Important for data structures which should be able to hold any data type
- Data structures concern the layout of the container, not the contained type
- Java has grown
  - In old java (version < 1.5) a container that holds anything had to hold Object
  - Jealousy of C++ Templates inspired java Generics
- I presume that this is not your first time seeing java generics
  - Review Weiss Ch 4.6-7 as needed
Type parameters

Arrays are a container that can hold anything

- Arrays are parameterized on the type of their elements
  ```java
  String sa[] = new String[2];
  Integer ia[] = new Integer[2];
  ```

- Create/Assign/Access semantics don’t change for different types
  ```java
  sa[0] = new String("Yeah");
  sa[1] = sa[0];
  ia[0] = new Integer(5);
  ia[1] = ia[0];
  ```

- Algorithms based on assign/access are independent of type parameter, won’t change if the type changes

**Generics** allow type parameters for our own classes
Consider the Pair

Make a class that holds a pair of anything in Java

- In java < version 1.5, done with Object
- Irritating casting required
- Dangerous runtime exceptions easily result
- Examine ObjectPair.java
Generic Pair

I want a *pair* of Strings

Pair<String> sp = new Pair<String>("One","Two");
System.out.println(sp.getFirst());
System.out.println(sp.getSecond());

I want a *pair* of Integers

Pair<Integer> ip = new Pair<Integer>(1,2);
System.out.println(ip.getFirst());
System.out.println(ip.getSecond());

- Pairs are parameterized on a type
- Pairs are defined in Pair.java
- Inspect implementation in Pair.java
- Contrast with Pair2.java: two type parameters
Restriction: Boxed v Unboxed

- Generics require a uniform memory model
- Restricted Reference Types Only
- Primitives not allowed

Compile errors:

```
ArrayList<int> al = new ArrayList<int>();
Pair2<double, char> p = new Pair<double, char>(1.2, ’c’);
```

So what does the alternative do differently?

```
ArrayList<Integer> al = new ArrayList<Integer>();
Pair<Double, Character> p =
    new Pair2<Double, Character>(1.2, ’c’);
```

Draw a memory diagram to understand.
Defining Parameterized Classes

Angle brackets: type parameters
- A compile-time variable
- Class definitions <T> or <K,V> or whatever

public class MyContainer <T>
public class MyMap <K,V>
Generics Get Weird

Constraints can show up in angle brackets

- Can only use a T that descends from OtherClass
  
  ```java
  public class SomeClass <T extends OtherClass>
  ```

- CharSequence is an interface, E must implement it
  
  ```java
  public class CharLister <E extends CharSequence>
  ```

- All T must be Comparable to one another
  
  ```java
  public class SomeTree <T extends Comparable<T>>
  ```

- T must descend from something Comparable
  
  ```java
  public class SomeTree <T extends Comparable<? super T>>
  ```

90% of the time it will be

```java
public class MyContainer <T>
public class MyMap <K,V>
```

in CS 310. Occasionally

```java
public class SomeTree <T extends Comparable<T>>
```
Next Time

- Build our own ArrayList: MyArrayList
- Be reading Weiss Ch 15
- Be working on HW1