CS 310: HW 1, Junit, 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`: construct and analyze

HW 1 Is Up

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

- Implement board for Gomoku
- `DenseBoard`: Dense arrangement of pieces (2D array or ArrayList)
- Architecture makes Boards usable with new types of games
- Target complexities for some methods
- Board Expansion can be tricky
- Analyze 1D approach in a text file
- Milestone due next week Saturday
- Final Deadline 1 week later
- Start early like today

Running Unit Tests

- Grading for HW 1

5%	Milestone Automated Tests	Early deadline
45%	Final Automated Tests	Final deadline
50%	Final Manual Inspection	Final deadline

- Series of testing files `HW1MilestoneTests.java`, `HW1FinalTests.java`
- Must be able to compile and run these from command line
- Require the JUnit library in `junit-cs310.jar`
- Baked into DrJava and probably most other IDEs
- IDEs usually provide niceness (Test button in DrJava)
- Command line - like running any other class

JUnit on the Command Line

Under unix (file separator is a colon, Linux, Mac OS X, BSD, etc)

```
demo$ javac -cp .:junit-cs310.jar *.java            # compile al
demo$ java  -cp .:junit-cs310.jar HW1MilestoneTests # run tests
```

Under windows cmd.exe (file separator is a semicolon)

```
demo$ javac -cp .;junit-cs310.jar *.java              # compile
demo$ java  -cp .;junit-cs310.jar HW1MilestoneTests   # run test
```

- What's `-cp`?
- What's a `.jar` file?

jUnit in IDES

DrJava

- Press the "Test" button
- Output correspondences kind of suc

Other IDEs

- Lots of material online
- Ask Piazza questions if having trouble but course staff may not be able to answer

Review: So far in the land of Big-O

- What does the following statement mean?

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

- What does the following statement mean?

  The method reverseArrayE(a) requires O(N) memory to complete while reversel(a) requires O(1) memory where N is the length of the array a.

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

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

```
String  sa[] = new String[2];
Integer ia[] = new Integer[2];
```

- Create/Assign/Access semantics don't change for different types

```
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`

```
public class ObjectPair {
  Object first, second;
  public ObjectPair(Object f,
                    Object s)
  {
    this.first = f;
    this.second = s;
  }
  public Object getFirst(){
    return first;
  }
  public Object getSecond(){
    return second;
  }
}
```

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());
```

- `Pair` parameterized on a type
- Inspect implementation in `Pair.java`
- Contrast with `MixedPair.java`: two type parameters

Pair and MixedPair

First/Second must be the same

```
public class Pair<T>{
  private T first, second;
  public Pair(T f, T s){
    this.first = f;
    this.second = s;
  }
  public T getFirst(){
    return first;
  }
  public T getSecond(){
    return second;
  }
}

main(){
  Pair<Integer> pi;
  pi = new Pair<Integer>(1,2);
  Pair<String> ps;
  ps = new Pair<String>("a","B");
}
```

First/Second can be different

```
public class MixedPair<F,S>{
  private F first;
  private S second;
  public MixedPair(F f, S s){
    this.first = f;
    this.second = s;
  }
  public F getFirst(){
    return this.first;
  }
  public S getSecond(){
    return this.second;
  }
}
main(){
  MixedPair<String,Integer> psi;
  psi =
   new MixedPair<String,Integer>("a",2);
  MixedPair<Integer,Integer> ps;
  ps = new MixedPair<>(5,9);
}
```

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>();
MixedPair<double,char> p = new MixedPair<double,char>(1.2,'c
```

Use the Capitalized Class version of primitive types instead

```
ArrayList<Integer> al = new ArrayList<Integer>();
MixedPair<Double,Character> p =
  new MixedPair<Double,Character>(1.2,'c');
```

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`

  ```
  public class SomeClass <T extends OtherClass>
  ```

- `CharSequence` is an interface, `E` must implement it

  ```
  public class CharLister <E extends CharSequence>
  ```

- All `T` must be `Comparable` to one another

  ```
  public class SomeTree <T extends Comparable<T>>
  ```

- `T` must descend from something `Comparable`

  ```
  public class SomeTree <T extends Comparable<? super T>>
  ```

90% of the time it will be

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

in CS 310. Occasionally

```
public class SomeTree <T extends Comparable<T>>
```