CS1007: Object Oriented Design and Programming in Java

Lecture #19
Mar 30
Shlomo Hershkop
shlomo@cs.columbia.edu

Announcements

• Any question on the homework?
Question

• Why is the homework practical for this course?

Outline

• Practical Example
• Generics
• Complicated example
• More complicated example
• Java beans (not lunch)
Question to get you thinking

• Say I have a million numbers, I’m looking for the max (highest value) number…..

• Can you write some pseudo code to find it?
• How many times do you have to go through the numbers?
• How much memory do you need to use?

• What about the 3rd largest number?

• Same 3 questions
• Now get together with your neighbors….how would you find the 1000’th largest number

• Any ideas?

---

Practical Application

• Want to write a general method to implement an algorithm
• Example:
  – You are reprogramming you home alarm system
  – You sprinkle a network of sensors across the system
  – Given a set of readings: you would like to reconstructs what the sensors are seeing
  – You want to make some money off of it 😊
With some basic knowledge

```
public void Alert Foo(alarmInterface ai) {
    Some cool code
}
```

- Works
- But has some specific limitations:

Generics

- By default java has treated collections as general as possible
- Getting an object requires a cast
  - Can end up with type cast exception..runtime

- New to java 1.5 (originally planned for 1.6)
- Allows you generalize a type for more specific use than just Object class
Another advantage

- Allows you to program algorithms which don’t work in the dark
- Allows you to setup constraints on the objects you allow to be passed to your methods

Example 1

```java
class Example1<T> {
    T ob;

    Example1(T o) {
        ob = o;
    }

    T getob() {
        return ob;
    }

    void showType() {
        System.out.println("Type of T is "+
            ob.getClass().getName());
    }
}
```
Usage

```java
Example1<Integer> e1;

e1 = new Example1<Integer>(88);
e1.showType();

int v = e1.getob();

Example1<String> e2 =
    new Example1<String>("Generics Test");

String str = e2.getob();
```

Example2

```java
class Example2<T, V> {
    T ob1;
    V ob2;

    Example2(T o1, V o2) {
        ob1 = o1;
        ob2 = o2;
    }

    T getob1() {
        return ob1;
    }

    V getob2() {
        return ob2;
    }
}
```
Example 3

class Example3<Q extends Number> {
    Q[] nums; // array of Number or subclass
    Stats(Q[] o) {
        nums = o;
    }
    double sum() {
        double sum = 0.0;
        for(int i=0; i < nums.length; i++)
            sum += nums[i].doubleValue();
        return sum;
    }
}

Usage

Double dnums[] = { 1.1, 2.2, 3.3, 4.4, 5.5 };
Example3<Double> dob = new Example3<Double>(dnums);

double w = dob.sum();
System.out.println("dob sum is " + w);
Example4

class TwoD {
    int x, y;
    TwoD(int a, int b) {
        x = a;
        y = b;
    }
}

class ThreeD extends TwoD {
    int z;
    ThreeD(int a, int b, int c) {
        super(a, b);
        z = c;
    }
}

class FourD extends ThreeD {
    int t;
    FourD(int a, int b, int c, int d) {
        super(a, b, c);
        t = d;
    }
}

class Coords<T extends TwoD> {
    T[] coords;
    Coords(T[] o) { coords = o; }
}
static void showXY(Coords<?> c) {
    System.out.println("X Y Coordinates: ");
    for (int i = 0; i < c.coords.length; i++)
        System.out.println(c.coords[i].x + " " +
                            c.coords[i].y);
    System.out.println();
}

static void showXYZ(Coords<? extends ThreeD> c) {
    System.out.println("X Y Z Coordinates: ");
    for (int i = 0; i < c.coords.length; i++)
        System.out.println(c.coords[i].x + " " +
                            c.coords[i].y + " " +
                            c.coords[i].z);
    System.out.println();
}
static void showAll(Coords<? extends FourD> c) {
    System.out.println("X Y Z T Coordinates:");
    for(int i=0; i < c.coords.length; i++)
    {
        System.out.println(c.coords[i].x + " " +
            c.coords[i].y + " " +
            c.coords[i].z + " " +
            c.coords[i].t);
    }
    System.out.println();
}

Using in methods

static <T, V extends T> boolean isIn(T x, V[] y) {
    for(int i=0; i < y.length; i++)
    {
        if(x.equals(y[i])) return true;
    }
    return false;
}
class GenCons {
    private double val;
    <T extends Number> GenCons(T arg) {
        val = arg.doubleValue();
    }
    void showval() {
        System.out.println("val: " + val);
    }
}

public class GenConsDemo {
    public static void main(String args[]) {
        GenCons test = new GenCons(100);
        GenCons test2 = new GenCons(123.5F);
        test.showval();
        test2.showval();
    }
}

Can also use it in your interface

interface MinMax<T extends Comparable<T>> {
    T min();
    T max();
}
Inheritance

• You have to be careful on how things are passed down an inheritance tree

```java
interface BaseInterface<A> {
    A getInfo();
}

class ParentClass implements BaseInterface<Integer> {
    public Integer getInfo()
    {
        return(null);
    }
}

class ChildClass extends ParentClass implements BaseInterface<String> {
}

class BadParents {
    public static void main(String args[]) {
        Vector<Integer> v1 = new Vector<Integer>();
        Vector v2;
        v1 = v2;
        v2 = v1;
    }
}
Wildcard usage

- You can use a wildcard to say anything
- ?
- Problem:
- If you use it as a parameter in a method, you won’t be able to disambiguate the ?
  - Can’t create a type (same problem like original)
  - The question mark isn’t a type

Wildcards

- Ok to call a method:
  \[ ? \text{ extends } E \text{ get(int i)} \]

- Can create any object and use it as a return value to an element of type E
Wildcards

- Wildcards can go both ways
- `? super F` matches any supertype of `F`

```java
public static <F> void append(
    ArrayList<? super F> a, ArrayList<F> b, int count)
{
    for (int i = 0; i < count && i < b.size(); i++)
        a.add(b.get(i));
}
```

- Safe to call `ArrayList<? super F>` since:
  - `boolean add(? super F newElement)`
  - Can pass any element of type `F`

- Code:

```java
public static <E extends Comparable<E>> E getMax(ArrayList<E> a)
{
    E max = a.get(0);
    for (int i = 1; i < a.size(); i++)
        if (a.get(i).compareTo(max) > 0) max = a.get(i);
    return max;
}
```

- `E extends Comparable<E>` so that we can call `compareTo` inside our method
• Too restrictive--can't call with ArrayList<GregorianCalendar>
• GregorianCalendar does not implement Comparable<GregorianCalendar>, only Comparable<Calendar>

This will fit

• Wildcards to the rescue:

public static <E extends Comparable<? super E>> E getMax(ArrayList<E> a)
Advantage/disadvantage

- Really good to move errors to compile time and not run time
  - Allow the programmer more chances not to mess up
  - Allows your code to run faster
  - Allows more robust code generation

Issues

- One of the big issues with any programming language extensions
  - How to add new ideas without having to redo everything else

  - How can we make generics backwards compatible?
Erasure

• Java Virtual machine does not know about generic types...why is that good?
• Type variables are erased--replaced by type bound or Object if unbounded
• Ex. ArrayList<E> becomes
  
```
public class ArrayList
{
    public Object get(int i) { . . . }
    public Object set(int i, Object newValue) { . . . }
    . . .
    private Object[] elementData;
}
```

• Ex. getmax becomes
  
```
public static Comparable getMax(ArrayList a)
    // E extends Comparable<?
    super E> erased to Comparable
```

• Erasure necessary to interoperate with legacy (pre-JDK 5.0) code
Limitations of Generics

- Cannot replace type variables with primitive types
- Cannot construct new objects of generic type

```java
a.add(new E());
```

Work around

- Pass in class objects

```java
public static <E> void fillWithDefaults(
    ArrayList<E>, Class<? extends E> cl, int count)
throws InstantiationException, IllegalAccessException {
    for (int i = 0; i < count; i++)
        a.add(cl.newInstance());
}
```

- Usage:
  - `fillWithDefaults(a, Something.class, count)`
For next time

- Reading: 8.2, 9.1

- We will cover applet programming, multi process/threaded programming and background for homework