CSC 207 2014S: Extra Session, Week 8

Overview

- Admin
- You ask questions.
- I try to give answers.

Admin

- Sam hopes to get stuff ready for the exam today.

Questions

When should I use anonymous inner classes?

Anonymous: I need an object with particular behavior. I only need it only in this place (or places in which I send the object). So don’t bother making a separate class. The code is where you need it, so you don’t have to look elsewhere. And you don’t need to name the class.

Inner: You may want to refer to the fields of the enclosing class. We could do this by adding a field to the new class.

Innerness is determined by placement.

```java
public class Outer {
    int value;

    Object anonymous = new Object() {
        public String toString() {
            return "<" + value + ">";
        } // toString
    }; // new Object

    Object samIsVerbose = new Object() {
        public String toString() {
            return "<" + Outer.this.value + ">";
        } // toString
    }; // new Object

    Object namedInner = new NameInner();

    Object partner = new Partner(this);
```
class NamedInner
{
    public String toString()
    {
        return "<" + value + ">";
    } // toString()
} // NamedInner
} // class Outer

class Partner
{
    Outer link;
    public Partner(Outer link)
    {
        this.link = link;
    } // Partner(Outer)
    public String toString()
    {
        return "<" + link.value + ">";
    } // toString()
} // class Partner

- Inner classes: Easier to refer to the enclosing object.
- Partner class: More general.
- Anonymous classes: Less to write, no need to define name, code is where you want it
- Named classes: Easier to make more copies; Easier to clone

What should we do with comparators?

Arrays.sort(myValues, new Comparator<String>()
{
    public int compare(String left, String right)
    {
        return left.charAt(1).compareTo(right.charAt(1));
    } // compare(String, String)
};

vs.

class CompareByCharacterOne
    implements Comparator<String>
{
    public int compare(String left, String right)
    {
        return left.charAt(1).compareTo(right.charAt(1));
    } // compare(String, String)
} // CompareByCharacterOne

Arrays.sort(myValues, new CompareByCharacterOne());

vs.
class CompareByCharacterOne
    implements Comparator<String>
{
    Comparator<String> ORDER = new CompareByCharacterOne();

    private CompareByCharacterOne()
    {
    } // CompareByCharacterOne()

    public int compare(String left, String right)
    {
        return left.charAt(1).compareTo(right.charAt(1));
    } // compare(String, String)
} // CompareByCharacterOne

Arrays.sort(myValues, CompareByCharacterOne.ORDER);

// The following is disallowed
Arrays.sort(myValues, new CompareByCharacterOne());

This technique ensures that there is only one copy of this object. It’s called a Singleton Class.

Here’s a more complex example

Integer[] big = ArrayUtils.select(myValues, new Predicate<Integer>()
{
    public boolean test(Integer val)
    {
        return val > 10;
    } // test(Integer)
});

vs.

public class AtLeast
    implements Predicate<Integer>
{
    int bottom;
    public AtLeast(int bottom)
    {
        this.bottom = bottom;
    } // AtLeast(int)
    public boolean test(Integer val)
    {
        return val > bottom;
    } // test(Integer)
} // class AtLeast

Integer[] big = ArrayUtils.select(myValues, new AtLeast(10));

vs. (syntax may not be quite right

Predicate<Integer> atLeast(int bot)
{
    return new Predicate<Integer>()
    {

3
int bottom = bot;
public boolean test(Integer val)
{
    return val > bottom;
} // test(Integer)
}; // new Predicate<Integer>
} // atLeast(int)

Integer[] big = ArrayUtils.select(myValues, atLeast(10));

When should I use generics?

- Whenever you want your code to work with more than one type, and you still want type checking.
- Probably whenever you create a collection.
- Generic vs. polymorphism

Do I write

    public static Addable sum(Addable x, Addable y)
    {
        return x.add(y);
    } // sum(Addable,Addable)

Or do I write

    public <T extends Addable<T>> static T sum(T x, T y)
    {
        return x.add(y);
    } // sum(T,T)

- It depends on what you think about the add method.

Where do we put the type variable? Don’t we put it in the class/interface?

Yes, we put it in the class/interface

    public interface Collection<T>
    {
        ...
    }

    public class SkipList<T>
    implements Collection<T>
    {
        ...
    }

Can we also put them in static methods?
public static <T> Collection<T> collect(T val) {
    Collection<T> collection = new Collection<T>();
    collection.add(val);
    return collection;
} // collect(T)

Copyright (c) 2013-14 Samuel A. Rebelsky.

This work is licensed under a [Creative Commons Attribution 3.0 Unported License](http://creativecommons.org/licenses/by/3.0/) To view a copy of this license, visit [http://creativecommons.org/licenses/by/3.0/] or send a letter to Creative Commons, 543 Howard Street, 5th Floor, San Francisco, California, 94105, USA.