CPSC 324
Topics in Java Programming

Lecture 20

Today …

• Wrap up Exercise 10 …
• Finish up Exercise 11
• Intro to Generics
• Readings
  – Core: Ch. 12 pp. 613-625
Exercise 10 Part 3

• Here is an example for Point:
  public int hashCode() {
    Integer tmpX = new Integer(x);
    Integer tmpY = new Integer(y);
    return tmpX.hashCode() + 31 * tmpY.hashCode();
  }

• And here is an example for ColorPoint:
  public int hashCode() {
    return point.hashCode() + 31 * color.hashCode();
  }

Exercise 10

How should we test hashCode()?

• Check that it is consistent, e.g.,
  assert p1.hashCode() == p1.hashCode()

• Check that it matches equals()
  assert p1.equals(p2) && p1.hashCode() == p2.hashCode()

• Note these are not “proofs”
Exercise 11

Note that you need to turn in your exercise at the end of the lab

Generics & Collections

• The plan
  – Using type parameters
  – Creating simple generic classes
  – The Java Collection API
  – More advanced use of type variables (if time …)

  The "newer" features of Java all work well together:
  • Generics
  • Collections
  • Boxing and Unboxing
  • The new “foreach” loop
  • Varargs

So we’ll talk about these as we go …
Advantages

Before these features …

• Summing a list of integers:

  Create array \( [1, 2, 3] \) of Integer objects

  Integer[] intArray = \{new Integer(1), new Integer(2),
  new Integer(3)\};

  Create a list out of the array

  List ints = Arrays.asList(intArray);

  Loop through the list, sum the contained primitive values

  int sum = 0;
  for(int i = 0; i < ints.size(); i++)
    sum += ((Integer)ints.get(i)).intValue();
  assert sum == 6;

  * Somewhat contrived example ... since we are using an “unbounded” list

• Summing a list of integers using Iterators:

  Iterator objects simplify traversing collections

  boolean hasNext()
  * Object next()
  * void remove()
Advantages

After the new features ...

• Summing a list of integers the "new way":

```java
List<Integer> ints = Arrays.asList(1, 2, 3);
int sum = 0;
for(int n : ints)
    sum += n;
assert sum == 6;
```

* Even though the example is contrived ... This is way better!!!

• If we only used arrays:

```java
int[] ints = {1, 2, 3};
int sum = 0;
for(int n : ints)
    sum += n;
```

Collections are more flexible
• Grow and shrink collection
• Switch data structure

These are very similar ...
Why use the first version?
Generics

- The List interface is in the Java “Collections Framework”
- The List interface is now “generic”
  - List<E> indicates a list containing elements of type E
  - So List<Integer> aList indicates a list of Integer objects
  - Java now casts for us: Integer n = aList.get(i);
- Before generics
  - No way to indicate the type of elements within a list
  - … unless we extended the List interface
  - Developers had to cast:
    Integer n = (Integer)aList.get(i);

Type Parameter Syntax

- We pass parameters into the <>’s
- These parameters are Java types (classes or interfaces)
- Type parameters can themselves have type parameters
- Examples for the List interface
  
<table>
<thead>
<tr>
<th>List&lt;T&gt;</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>List&lt;Integer&gt;</td>
<td>A list of integers</td>
</tr>
<tr>
<td>List&lt;String&gt;</td>
<td>A list of strings</td>
</tr>
<tr>
<td>List&lt;List&lt;Integer&gt;&gt;</td>
<td>A list of lists-of-integers</td>
</tr>
<tr>
<td>List&lt;List&lt;String&gt;&gt;</td>
<td>A list of lists, where sublists contain</td>
</tr>
<tr>
<td></td>
<td>lists of strings</td>
</tr>
<tr>
<td>List&lt;List&lt;List&lt;Integer&gt;&gt;&gt;</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Generics

- Generics are implemented by “erasure”
- This means:
  - Your program is rewritten at compile time
  - List<String>, List<Integer>, and List<List<String>> are each represented as List types
  - The rewriting changes these to List and adds correct casts
  - This is called “erasure” … since type parameters are erased (and in this case casts are added)
  - The correct casts are (mostly) guaranteed not to fail!

Defining Generic Classes

**Problem:** Lets say we want to have a class to represent a simple pair \((x_1, x_2)\) where \(x_1\) and \(x_2\) can be of any type

- We’ll create three approaches
- The first won’t be “generic”
- The other two will be generic
Defining Generic Classes

First approach: The “old-fashioned” way

```java
public class Pair {
    private Object first;
    private Object second;
    public Pair(Object first, Object second) {
        this.first = first; this.second = second;
    }
    public Object getFirst() {return first;}
    public Object getSecond() {return second;}
}
```

Can first & second be any object type? Why make pair generic here?

Defining Generic Classes

Second approach: A simple generic class

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

A type "variable" 
first and second are of the type passed in as T

Can first and second be any object type? 
Yes ... But they both have to be the same type (T)
Defining Generic Classes

Second approach: A simple generic class

// create some pairs:

Pair<String> p1 = new Pair<String>("its", "it");
assert (p1.getFirst() + p1.getSecond()).equals("itsit");

Pair<Integer> p2 = new Pair<Integer>(1, 2);
assert 3 == p2.getFirst() + p2.getSecond();

Pair<Pair<String>> p3 = new Pair<Pair<String>>(new Pair<String>("a", "b"), new Pair<String>("a", "c"));
assert "b".equals(p3.getFirst().getSecond());

// this:

Pair<Pair<String>> p3 = new Pair<Pair<String>>(new Pair<String>("a", "b"), new Pair<String>("a", "c"));
assert "b".equals(p3.getFirst().getSecond());

// would be this without the generic class (or type params):

Pair p3 = new Pair(new Pair("a", "b"), new Pair("a", "c"));
assert "b".equals((String)((Pair)p3.getFirst()).getSecond());
Defining Generic Classes

**Third approach:** Use *two* type variables

```java
public class Pair<T1,T2> {
    private T1 first;
    private T2 second;
    public Pair(T1 first, T2 second) {
        this.first = first; this.second = second;
    }
    public T1 getFirst() {return first;}
    public T2 getSecond() {return second;}
}
```

Now both first and second can be of any (and different) types.

Defining Generic Classes

**Third approach:** Use *two* type variables

```java
// creating pairs
Pair<String,Integer> p1 = new Pair<String,Integer>("gr", 8);
assert "gr8".equals(p1.getFirst() + p1.getSecond());

Pair<Integer,Integer> p2 = new Pair<Integer,Integer>(1, 2);
assert 3 == p2.getFirst() + p2.getSecond();
```
Generics and Foreach Loops

Foreach can be used on objects implementing Iterable<E>

```java
public interface Iterable<E> {
    public Iterator<E> iterator();
}
```

Again based on compile-time rewriting

- Assume we have our generic ints list from before:
  ```java
  for(int n : ints)
      sum += n;
  ```
- This becomes:
  ```java
  for(Iterator<Integer> it = ints.iterator(); it.hasNext();)
      sum += it.next();
  ```

Generics and Foreach Loops

The Iterator<E> interface provides three methods:

```java
public interface Iterator<E> {
    public boolean hasNext();
    public E next();
    public void remove();
}
```

- The foreach loop does only the most basic iteration
  - For example, can't use remove
  - Can't check if more elements exist within a list
  - For these and other cases you can use the Iterator interface

- All collections in the Collection API plus arrays are “Iterable”
- We can make our own classes Iterable if needed …
Exercise 12