Programming Languages and Techniques (CIS120)

Lecture 27

November 4, 2015

Generics, Collections, and Iterators
Announcements

• **Midterm 2 is Friday, November 6th in class**
  • Last names  A – L  Leidy Labs 10 (here)
  • Last names  M – Z  Cohen G17
    – Everything starting with mutable state in Ocaml to Subtyping in Java

• Review Session:
  TONIGHT
  Levine 100
  7-9PM
  Pizza!
Java Generics
public class Object {
    boolean equals(Object o) {
        ... // test for equality
    }
    String toString() {
        ... // return a string representation
    }
    ... // other methods omitted
}

• Object is the root of the class tree.
  – Classes that leave off the “extends” clause implicitly extend Object
  – Arrays also implement the methods of Object
  – This class provides methods useful for all objects to support

• Object is the highest type in the subtyping hierarchy.
Recap: Subtyping

- Interfaces extend (possibly many) interfaces
- Classes implement (possibly many) interfaces
- Classes (except Object) extend exactly one other class (Object if implicit)
- Interface types (and arrays) are subtypes “by fiat” of Object
**Subtype Polymorphism***

- **Main idea:**

  Anywhere an object of type A is needed, an object that is a subtype of A can be provided.

```java
void method(A obj) {
    // use obj at type A
}

method(new B());
```

- If B is a subtype of A, it provides all of A’s (public) methods.
- Due to dynamic dispatch, the behavior of the method depends on B’s implementation.
  - Behavior of B should be “compatible” with A’s behavior
  - Simple inheritance makes this easier

*polymorphism = many shapes*
Is subtyping good enough?

Subtype Polymorphism

vs.

Parametric Polymorphism
Subtype Polymorphism

```java
public interface ObjQueue {
    public void enq(Object o);
    public Object deq();
    public boolean isEmpty();
    ...
}
```

```java
ObjQueue q = ...;
q.enq(" CIS 120 ");
Object x = q.deq();
System.out.println(x.trim());
```

Does this line type check

1. Yes
2. No
3. It depends
Subtype Polymorphism

```java
public interface ObjQueue {
    public void enq(Object o);
    public Object deq();
    public boolean isEmpty();
    ...
}
```

ObjQueue q = ...;
q.enq(" CIS 120 ");
Object x = q.deq();
// System.out.println(x.trim());
q.enq(new Point(0.0,0.0));
___B___  y = q.deq();

What type for B?

1. Point
2. Object
3. ObjQueue
4. None of the above
Parametric Polymorphism (a.k.a. Generics)

• Big idea:

  Parameterize a type (i.e. interface or class) by another type.

```java
public interface Queue<E> {
    public void enq(E o);
    public E deq();
    public boolean isEmpty();
}
```

• The implementations of a parametric polymorphic interface can not depend on the implementation details of the parameter.
  – e.g. the implementation of enq should not invoke methods on ‘o’.
Generics (Parametric Polymorphism)

```java
public interface Queue<E> {
    public void enq(E o);
    public E deq();
    public boolean isEmpty();
    ...
}
```

```java
Queue<String> q = ...;
q.enq(" CIS 120 ");
String x = q.deq(); // What type of x? String
System.out.println(x.trim()); // Is this valid? Yes!
q.enq(new Point(0.0,0.0)); // Is this valid? No!
```
Subtyping and Generics
Subtyping and Generics*

• Java generics are invariant:
  – Subtyping of arguments to generic types does not imply subtyping between the instantiations:

    ```java
    Queue<String> qs = new QueueImpl<String>();
    Queue<Object> qo = qs;
    qo.enq(new Object());
    String s = qs.deq();
    ```

    * Subtyping and generics interact in other ways too. Java supports “bounded” polymorphism and wildcard types, but those are beyond the scope of CIS 120.

    Hardest part to learn about generics and subtyping...

Ok? Sure!
Ok? Let’s see...
Ok? I guess
Ok? Noooo!
The Java Collections Library

A case study in subtyping and generics

(Also very useful!)
Java Packages

• Java code can be organized into packages that provide namespace management.
  – Somewhat like OCaml’s modules
  – Packages contain groups of related classes and interfaces.
  – Packages are organized hierarchically in a way that mimics the file system’s directory structure.

• A .java file can import (parts of) packages that it needs access to:

```java
import org.junit.Test; // just the JUnit Test class
import java.util.*; // everything in java.util
```

• Important packages:
  – java.lang, java.io, java.util, java.math, org.junit

• See documentation at:
  http://download.oracle.com/javase/6/docs/api/index.html
Reading Java Docs

http://docs.oracle.com/javase/6/docs/api/java/util/package-summary.html
Interfaces* of the Collections Library

```
Iterable<E>                          Map<K,V>
    |
    v
Collection<E>                      SortedMap<K,V>
    |                                                |
    v                                                v
List<E>                             SortedSet<E>
    |                                        |
    v                                        |
Deque<E>                            Set<E>
    |                                            |
    v                                            |
Deque<E>                            Set<E>
```

*not all of them!
• We’ve already seen this interface in the OCaml part of the course.

• Most collections are designed to be mutable (like queues)

* Why not E? Internally, collections use the equals method to check for equality – membership is determined by o.equals, which does not have to be false for objects of different types. Most applications only store and remove one type of element in a collection, in which case this subtlety never becomes an issue.
Sequences

```
Iterable<E>
  /    
Collection<E>
    /   
List<E>  Deque<E>
```

```
LinkedList<E>
ArrayList<E>    ArrayDeque<E>
```

- **Extends**
- **Implements**
Sets and Maps*

*Read javadocs before instantiating these classes! There are some important details to be aware of to use them correctly.
Iterating over collections

iterators, while, for, for-each loops
interface Iterator<E> {
    public boolean hasNext();
    public E next();
    public void delete(); // optional
}

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

Challenge: given a List<Book> how would you add each book’s data to a catalogue using an iterator?
While Loops

Syntax:

```java
// repeat body until condition becomes false
while (condition) {
  body
}
```

Example:

```java
List<Book> shelf = ... // create a list of Books

// iterate through the elements on the shelf
Iterator<Book> iter = shelf.iterator();
while (iter.hasNext()) {
  Book book = iter.next();
  catalogue.addInfo(book);
  numBooks = numbooks + 1;
}
```
For Loops

syntax:

```
for (init-stmt; condition; next-stmt) {
  body
}
```

equivalent while loop:

```
init-stmt;
while (condition) {
  body
  next-stmt;
}
```

List<Book> shelf = … // create a list of Books

// iterate through the elements on the shelf
for (Iterator<Book> iter = shelf.iterator();
     iter.hasNext();) {
  Book book = iter.next();
  catalogue.addInfo(book);
  numBooks = numbooks+1;
}

For-each Loops

Syntax:

```java
// repeat body for each element in collection
for (type var : coll) {
    body
}
```
For-each can be used to iterate over arrays or any class that implements the `Iterable<E>` interface (notably `Collection<E>` and its subinterfaces).
public static void iteratorExample() {
    List<Integer> nums = new LinkedList<Integer>();
    nums.add(1);
    nums.add(2);
    nums.add(7);

    int numElts = 0;
    int sumElts = 0;
    Iterator<Integer> iter =
        nums.iterator();
    while (iter.hasNext()) {
        Integer v = iter.next();
        sumElts = sumElts + v;
        numElts = numElts + 1;
    }

    System.out.println("sumElts = " + sumElts);
    System.out.println("numElts = " + numElts);
}

What is printed by iteratorExample()?
1.  sumElts = 0  numElts = 0
2.  sumElts = 3  numElts = 2
3.  sumElts = 10  numElts = 3
4.  NullPointerException
5.  Something else

Answer: 3
public static void forEachExample() {
    List<Integer> nums = new LinkedList<Integer>();
    nums.add(1);
    nums.add(2);
    nums.add(7);

    int numElts = 0;
    int sumElts = 0;
    for (Integer v : nums) {
        sumElts = sumElts + v;
        numElts = numElts + 1;
    }

    System.out.println("sumElts = " + sumElts);
    System.out.println("numElts = " + numElts);
}
public static void nextNextExample() {
    List<Integer> nums = new LinkedList<Integer>();
    nums.add(1);
    nums.add(2);
    nums.add(7);
    int sumElts = 0;
    int numElts = 0;
    Iterator<Integer> iter = nums.iterator();
    while (iter.hasNext()) {
        Integer v = iter.next();
        sumElts = sumElts + v;
        v = iter.next();
        numElts = numElts + v;
    }
    System.out.println("sumElts = " + sumElts);
    System.out.println("numElts = " + numElts);
}

What is printed by nextNextExample()?
1. sumElts = 0 numElts = 0
2. sumElts = 3 numElts = 2
3. sumElts = 8 numElts = 2
4. NullPointerExcepEon
5. Something else

Answer: 5 NoSuchElementExcepEon
Method Overriding
public class C {
    public void printName() { System.out.println("I’m a C"); }
}

public class D extends C {
    public void printName() { System.out.println("I’m a D"); }
}

// somewhere in main
C c = new D();
c.printName();

What gets printed to the console?

1. I’m a C
2. I’m a D
3. NullPointerExcetion
4. NoSuchMethodException
A Subclass can **Override** its Parent

```java
public class C {
    public void printName() {
        System.out.println("I'm a C");
    }
}

public class D extends C {
    public void printName() {
        System.out.println("I'm a D");
    }

    // somewhere in main
    C c = new D();
    c.printName();
}
```

- Our ASM model for dynamic dispatch already explains what will happen when we run this code.
- Useful for changing the default behavior of classes.
- But... can be confusing and difficult to reason about if not used carefully.
Overriding Example

Workspace

```
C c = new D();
c.printName();
```

Stack

Heap

Class Table

<table>
<thead>
<tr>
<th>Class</th>
<th>Method</th>
<th>Implementation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Object</td>
<td>String toString()</td>
<td>{ ... }</td>
</tr>
<tr>
<td></td>
<td>boolean equals...</td>
<td>...</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>C</td>
<td>extends</td>
<td></td>
</tr>
<tr>
<td></td>
<td>() { }</td>
<td></td>
</tr>
<tr>
<td></td>
<td>void printName()</td>
<td>{ ... }</td>
</tr>
<tr>
<td>D</td>
<td>extends</td>
<td></td>
</tr>
<tr>
<td></td>
<td>() { ... }</td>
<td></td>
</tr>
<tr>
<td></td>
<td>void printName()</td>
<td>{ ... }</td>
</tr>
</tbody>
</table>
Overriding Example

Workspace

c.printName();

Stack

Heap

Class Table

Object

String toString(){
...
}

boolean equals...

...

C

extends

C() {
}

void printName(){...}

D

extends

D() {
...
}

void printName(){...}
Overriding Example

Workspace

Stack

Heap

Class Table

Object

String toString() {...

boolean equals...

...

C

extends

C() {}

void printName() {...

D

extends

D() {...

void printName() {...
System.out.println("I’m a D");

C

\texttt{C()} \{ \}
\texttt{void printName() \{ ... \}}

D

\texttt{D()} \{ ... \}
\texttt{void printName() \{ ... \}}
class C {
    
    public void printName() {
        System.out.println("I'm a " + getName());
    }

    public String getName() {
        return "C";
    }
}

class E extends C {
    
    public String getName() {
        return "E";
    }
}

// in main
C c = new E();
c.printName();

What gets printed to the console?
1. I’m a C
2. I’m a E
3. I’m an E
4. NullPointerExcepEon
Difficult with Overriding

```java
class C {
    public void printName() {
        System.out.println("I'm a " + getName());
    }

    public String getName() {
        return "C";
    }
}

class E extends C {
    public String getName() {
        return "E";
    }
}

// in main
C c = new E();
c.printName();
```

The `C` class might be in another package, or a library...

Whoever wrote `E` might not be aware of the implications of changing `getName`.

Overriding the method causes the behavior of `printName` to change!
- Overriding can break invariants/abstractions relied upon by the superclass.