Programming Languages and Techniques (CIS120)

Lecture 27
March 28, 2018

Java Generics
Collections and Equality
Chapters 25 & 26
Announcements

• HW7: Chat Server
  – Available on Codio / Instructions on the web site
  – Due Tuesday, April 3rd
  – No penalty for late submission (Thursday, 4/5 is a HARD deadline!)

• What do you need to know about Java for this assignment?
  – Subtyping (last week)
  – Dynamic dispatch (last Monday)
  – Generics, Collections, equality, overriding, overloading, enumerations (Today & Friday)

• CIS Diversity Summit, Friday 2-4PM, Glandt Forum (Singh center)
• CIS Advising session tonight with Dr. Val Tannen
• Survey about retention in CS
• Exam grades will be available this afternoon
Java Generics

Subtype Polymorphism

vs.

Parametric Polymorphism
Review: Subtype Polymorphism*

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

```java
// in class C
public static void times2(Counter c) {
    c.incBy(c.get());
}

// somewhere else, Decr extends Counter
C.times2(new Decr(3));
```

- If B is a subtype of A, it provides all of A’s (public) methods.

*polymorphism = many shapes*
Is subtype polymorphism enough?
Mutable Queue ML Interface

```ml
module type QUEUE =
  sig
    (* type of the data structure *)
    type 'a queue

    (* Make a new, empty queue *)
    val create : unit -> 'a queue

    (* Add a value to the end of the queue *)
    val enq : 'a -> 'a queue -> unit

    (* Remove the front value and return it (if any) *)
    val deq : 'a queue -> 'a

    (* Determine if the queue is empty *)
    val is_empty : 'a queue -> bool
  end
```

How can we translate this interface to Java?
Java Interface using Subtyping

module type QUEUE =
sig

  type 'a queue

  val create : unit -> 'a queue

  val enq : 'a -> 'a queue -> unit

  val deq : 'a queue -> 'a

  val is_empty : 'a queue -> bool
end

interface ObjQueue {

  // no constructors
  // in an interface

  public void enq(Object elt);

  public Object deq();

  public boolean isEmpty();
}

Subtype Polymorphism

interface ObjQueue {
    public void enq(Object elt);
    public Object deq();
    public boolean isEmpty();
}

ObjQueue q = ...;
q.enq(" CIS 120 ");
__A__ x = q.deq();

What type for A?

1. String
2. Object
3. ObjQueue
4. None of the above

ANSWER: Object
### Subtype Polymorphism

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

// Is this valid?
No!

// What type for B?
Object

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

```java
Obj Queue 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

**ANSWER: No**

trim is a method of the String class (removes extra spaces)
interface ObjQueue {
    public void enq(Object elt);
    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

ANSWER: Object
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 implementation of a parametric polymorphic interface cannot depend on the implementation details of the parameter.
  – the implementation of enq cannot invoke any methods on ‘o’ except those inherited from Object
  – i.e. the only thing we know about E is that it is a subtype of Object
Generics (Parametric Polymorphism)

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

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:

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

Hardest part to learn about generics and subtyping...

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

Which of these are true, assuming that class QueueImpl<E> implements interface Queue<E>?

1. QueueImpl<Queue<String>> is a subtype of Queue<Queue<String>>
2. Queue<QueueImpl<String>> is a subtype of Queue<Queue<String>>
3. Both
4. Neither

Answer: 1
Other subtleties with Generics

- Unlike OCaml, Java classes and methods can be generic only with respect to reference types.
  - Not possible to do:  Queue<int>
  - Must instead do:   Queue<Integer>

- Java Arrays cannot be generic
  - Not possible to do:

```java
class C<E> {
    E[] genericArray;
    public C() {
        genericArray = new E[];
    }
}
```
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://docs.oracle.com/javase/8/docs/api/](http://docs.oracle.com/javase/8/docs/api/)

- You should read this documentation in preparation for HW 7
Interfaces* of the Collections Library

*not all of them!
We’ve already seen a similar 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.
```java
import java.util.*;

class Point {
    private final int x, y;
    public Point(int x0, int y0) { x = x0; y = y0; }
    public int getX() { return x; }
    public int getY() { return y; }
}

public class TreeSetDemo {
    public static void main(String[] args) {
        Set<Point> s = new TreeSet<Point>();
        s.add(new Point(1,1));
    }
}
```

Exception in thread "main" java.lang.ClassCastException: Point cannot be cast to java.base/java.lang.Comparable
at java.base/java.util.TreeMap.compare(TreeMap.java:1291)
at java.base/java.util.TreeMap.put(TreeMap.java:536)
at java.base/java.util.TreeSet.add(TreeSet.java:255)
at TreeSetDemo.main(TreeSetDemo.java:14)
A Crucial Detail of TreeSet

Constructor Detail

TreeSet

public TreeSet()

Constructs a new, empty tree set, sorted according to the natural ordering of its elements. All elements inserted into the set must implement the Comparable interface. Furthermore, all such elements must be mutually comparable: e1.compareTo(e2) must not throw a ClassCastException for any elements e1 and e2 in the set.

...
### Method Summary

<table>
<thead>
<tr>
<th>Modifier and Type</th>
<th>Method and Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>int</td>
<td>compareTo(T o)</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>COMPARES THIS OBJECT WITH THE SPECIFIED OBJECT FOR ORDER.</td>
</tr>
</tbody>
</table>

### Method Detail

**compareTo**

```java
int compareTo(T o)
```

COMPARES THIS OBJECT WITH THE SPECIFIED OBJECT FOR ORDER. RETURNS A NEGATIVE INTEGER, ZERO, OR A POSITIVE INTEGER AS THIS OBJECT IS LESS THAN, EQUAL TO, OR GREATER THAN THE SPECIFIED OBJECT.

The implementor must ensure `sgn(x.compareTo(y)) == -sgn(y.compareTo(x))` for all `x` and `y`. (This implies that `x.compareTo(y)` must throw an exception iff `y.compareTo(x)` throws an exception.)

The implementor must also ensure that the relation is transitive. (i.e. `compareTo(x)` == 0, `compareTo(y)` == 0, or `compareTo(z)` == 0 for all objects `x, y, z`.)
import java.util.*;

class Point implements Comparable<Point> {
    private final int x, y;
    public Point(int x0, int y0) { x = x0; y = y0; }
    public int getX(){ return x; }
    public int getY(){ return y; }
    @Override
    public int compareTo(Point o) {
        if (this.x < o.x) {
            return -1;
        } else if (this.x > o.x) {
            return 1;
        } else if (this.y < o.y) {
            return -1;
        } else if (this.y > o.y) {
            return 1;
        }
        return 0;
    }
}

Point p1 = new Point(0,1);
Point p2 = new Point(0,2);
p1.compareTo(p2);  // -1
p2.compareTo(p1);  // 1
p1.compareTo(p1);  // 0