Programming Languages and Techniques (CIS120)

Lecture 27
November 5, 2018

Java Generics
Collections and Equality
Chapters 25 & 26
Announcements

• HW7: Chat Server
  – Available on Codio / Instructions on the web site
  – Due Tuesday, November 13th at 11:59pm

• Midterm 2 is this Friday, in class
  • Last names  A – F  Stiteler Hall B21
  • Last names  G – S  Leidy Labs 10 (here)
  • Last names  T – Z  Fagin 118

• Review Session: Wednesday November 7th at 6:00pm, Towne 100

• Coverage:
  – Mutable state (in OCaml and Java)
  – Objects (in OCaml and Java)
  – ASM (in OCaml and Java)
  – Reactive programming (in Ocaml)
  – Arrays (in Java)
  – Subtyping, Simple Extension, Dynamic Dispatch (in Java)

• Makeup exam request form: on the course web pages
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*
Recap: Subtyping

- Interfaces extend (possibly many) interfaces
- Classes implement (possibly many) interfaces
- Classes (except Object) extend exactly one other class (Object by default)
- Interface types (and arrays) are subtypes “by fiat” of Object
Is subtype polymorphism enough?
Mutable Queue ML Interface

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();
}

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
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());

ANSWER: No

trim is a method of the String class (removes extra spaces)

Does this line type check
1. Yes
2. No
3. It depends
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
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`
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:

  ```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.
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
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
Reading Java Docs

java.util

https://docs.oracle.com/javase/8/docs/api/java/util/package-summary.html
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.
Buggy Use of TreeSet implementation

```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)
```
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. ...
public interface Comparable<T>

This interface imposes a total ordering on the objects of each class that implements it. This ordering is referred to as the class's natural ordering, and the class's compareTo method is referred to as its natural comparison method. ...
## Method Summary

<table>
<thead>
<tr>
<th>Modifier and Type</th>
<th>Method and Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>int</td>
<td><strong>compareTo(T o)</strong></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 $\text{sgn}(x.\text{compareTo}(y)) = -\text{sgn}(y.\text{compareTo}(x))$ for all $x$ and $y$. (This implies that $x.\text{compareTo}(y)$ must throw an exception iff $y.\text{compareTo}(x)$ throws an exception.)

The implementor must also ensure that the relation is transitive. (i.e., $x.\text{compareTo}(y) < 0$, $y.\text{compareTo}(z) < 0$ implies $x.\text{compareTo}(z) < 0$.)
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
Method Overriding
A Subclass can *Override* its Parent

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

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. NullPointerException
4. NoSuchMethodException
A Subclass can Override its Parent

---

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

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

Stack

Heap

Class Table

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

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(){...}
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");
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. NullPointerException
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.