Collections & Generics

John Jannotti

/course/cs0320/www/lectures/

Feb 9, 2017
Course Announcements

- Stars is due Monday.
- Then you should start thinking about your teams.
- Stars grades should be out by next Monday.
- Finalize your teams and ideas by next Wednesday.
- I’m available to help flesh out ideas. Email for a meeting.
- AJAX lab is out. How you make interactive webby GUIs.
Providing constants is a bit verbose.

```java
double r = Math.cos(Math.PI * theta);
```

You might prefer:

```java
double r = cos(PI * theta);
```

Static import pulls in names directly from a `class`.

One at a time:

```java
import static java.lang.Math.PI;
```

Or all at once:

```java
import static java.lang.Math.*;
```

Obviates an “anti-pattern”: the Constant Interface
class Person {
    String confess(Object msg) {
        return msg.toString();
    }
}

class Spy extends Person {
    String confess(String msg) {
        return "Never!";
    }
}

Person p = new Person();
p.confess("The dock, at 11am");
Spy s1 = new Spy();
s1.confess("The airport, at 12pm");
Person p2 = s1;
p2.confess("The depot, at 6pm");
Spy s3 = new Spy();
s3.confess(new StringBuilder("The stadium, at 3pm"));
Generics

- Many abstractions are “generic” across types.
  - Holders: Optional, ThreadLocal
  - Factory methods:
    - ImmutableList.of(), DB.getObject(Company.class, id)
  - Collections: List<String>, Map<Label, Person>

- Before Generics, these abstractions took/returned Object.
- Return values were cast to the (statically known) return type.
- The compiler couldn’t help with mistakes, refactoring.
  - ClassCastException was a common error.
Generic comparison

```java
/**
 * @param words must contain only String
 */

void swear(Collection words) {
  for (Object o : words) {
    String word = (String)o;
    if (word.length() == 4)
      System.out.println(word);
  }
}

void swear(Collection<String> words) {
  for (String word : words)
    if (word.length() == 4)
      System.out.println(word);
}
```
Writing generic abstractions

- You’ve probably all used generic abstractions.
- Have you written any? What?
- Simple things are pretty simple.
- Complex things can be very, very complex.
public class Pair<A, B> {

private final A left;
private final B right;

public Pair(A a, B b) {
    left = a;
    right = b;
}

@Override
public String toString() {
    return "(" + left + ".." + right + ")";
}
}
Type constraints

Why isn’t this legal?

```java
1 List a = ImmutableList.of(new Pair<>(1.0, 2.0),
2 new Pair<>(2.0, 1.0),
3 new Pair<>(2.0, 1.5));
4 Collections.sort(a);
```
Type constraints

Why isn’t this legal?

```
List a = ImmutableList.of(new Pair<>(1.0, 2.0),
                new Pair<>(2.0, 1.0),
                new Pair<>(2.0, 1.5));
```

Because `Pair<Double,Double>` does not implement `Comparable<Pair<Double,Double>>`.
Does this work?

```java
public int compareTo(Object other) {
}
```
Does this work?

```java
public int compareTo(Object other) {
}
```

No. The types A & B might not, themselves, be Comparable\_\_A\_\_ and Comparable\_\_B\_\_.

John Jannotti (cs32)  Collections & Generics  Feb 9, 2017  10 / 21
We can’t implement compareTo

Another route to sorting? Use Comparators.

```java
List a = ImmutableList.of(new Pair<>(1.0, 2.0),
                         new Pair<>(2.0, 1.0),
                         new Pair<>(2.0, 1.5));
Collections.sort(a, a.leftToRight());
```

How can we implement Pair.leftToRight()? What needs to be true of A and B, and how can we enforce it?
Generic methods

- Note the introduction of X,Y (not using A,B).
- But why does it even exist?
- Allow more concise construction.
  - Java 6:
    ```java
    Pair<Integer,String> p = new Pair<Integer,String>(7, "john")
    ```
  - With make():
    ```java
    Pair<Integer,String> p = Pair.make(7, "john")
    ```
  - Java 7:
    ```java
    Pair<Integer,String> p = new Pair<>(7, "john")
    ```
Manipulating generic objects

- Suppose your `printAll()` handles any `Collection`.
  - Iterates over the members and calls `toString()`
- What’s the signature of `printAll()`?

```java
void printAll(Collection c);
```

What if you want to call `draw()` on a list of any `Shapes`?

```java
draw(List<Shape> shapes);
```

Can you pass in a `List<Circle>`?

```java
draw(List<?>);
```

```java
draw(List<? extends Shape>);
```
Manipulating generic objects

- Suppose your `printAll()` handles any Collection.
  - Iterates over the members and calls `toString()`
- What’s the signature of `printAll()`?
  - `void printAll(Collection c);`?
  - `void printAll(Collection<Object> c);`?
  - `void printAll(Collection<?> c);`?
Manipulating generic objects

- Suppose your `printAll()` handles any `Collection`.
  - Iterates over the members and calls `toString()`
- What’s the signature of `printAll()`?
  - `void printAll(Collection c);`?
  - `void printAll(Collection<Object> c);`?
  - `void printAll(Collection<?> c);`?

What if you want to call `draw()` on a list of any `Shapes`?
- `draw(List<Shape> shapes);`?
- Can you pass in a `List<Circle>`?
- `draw(List<?>);`?
- `draw(List<? extends Shape>);`?
Manipulating generic objects

- Suppose your `printAll()` handles any Collection.
  - Iterates over the members and calls `toString()`
- What's the signature of `printAll()`?
  - `void printAll(Collection c);` ?
  - `void printAll(Collection<Object> c);` ?
  - `void printAll(Collection<?> c);`
- What if you want to call `draw()` on a list of any Shapes?
  - `draw(List<Shape> shapes);` ?
Manipulating generic objects

- Suppose your `printAll()` handles any `Collection`.
  - Iterates over the members and calls `toString()`

- What’s the signature of `printAll()`?
  - `void printAll(Collection c);` ?
  - `void printAll(Collection<Object> c);` ?
  - `void printAll(Collection<?> c);`

- What if you want to call `draw()` on a list of any `Shapes`?
  - `draw(List<Shape> shapes);` ?
Manipulating generic objects

- Suppose your `printAll()` handles any `Collection`.
  - Iterates over the members and calls `toString()`

- What’s the signature of `printAll()`?
  - `void printAll(Collection c);` ?
  - `void printAll(Collection<Object> c);` ?
  - `void printAll(Collection<?> c);`

- What if you want to call `draw()` on a list of any `Shapes`?
  - `draw(List<Shape> shapes);` ?
  - Can you pass in a `List<Circle>`?
Manipulating generic objects

- Suppose your `printAll()` handles any Collection.
  - Iterates over the members and calls `toString()`
- What's the signature of `printAll()`?
  - `void printAll(Collection c);`
  - `void printAll(Collection<Object> c);`
  - `void printAll(Collection<? c);`
- What if you want to call `draw()` on a list of any Shapes?
  - `draw(List<Shape> shapes);`
  - Can you pass in a `List<Circle>`?
  - `draw(List<?>);`
  - `draw(List<? extends Shape>);`
Manipulating generic objects

- Suppose your `printAll()` handles any `Collection`.
  - Iterates over the members and calls `toString()`
- What’s the signature of `printAll()`?
  - `void printAll(Collection c);`?
  - `void printAll(Collection<Object> c);`?
  - `void printAll(Collection<?> c);`
- What if you want to call `draw()` on a list of any `Shapes`?
  - `draw(List<Shape> shapes);`?
  - Can you pass in a `List<Circle>`?
  - `draw(List<?>);`?
  - `draw(List<? extends Shape>);`
You should have a generic KDTree.

- You’ll thank me later.
- What makes KDTree<T> harder than List<T>?
Your KDTree

- You should have a generic KDTree.
  - You’ll thank me later.
  - What makes KDTree<T> harder than List<T>?
- KDTrees are not *completely* generic.
- Items *must* have positions.
You should have a generic KDTree.  
▶ You’ll thank me later.  
▶ What makes KDTree<T> harder than List<T>?

KDTrees are not completely generic.  
Items must have positions.  
In 3D space?
Your KDTree

- You should have a generic KDTree.
  - You’ll thank me later.
  - What makes KDTree<T> harder than List<T>?
- KDTrees are not *completely* generic.
- Items *must* have positions.
- In 3D space?
- You want a constraint like draw() has.
A little food for thought on Stars

- “Should my program do this obviously correct thing?” Yes.
- What do you do if two stars have the same coordinate?
- What do you do if two stars have the same name?
- Can you report more than one error in a CSV?
- How do you report where an error in CSV occurred?
- How easily can you add new “commands”?
- Is your command processing reusable?
- How did you write tests for your command processing?
- Can you add() to your KdTree? remove()?
Consider implementing Collection.
Someone else has thought about “completeness.”
You will interoperate with more code.
But (it seems like) you need to implement 15 methods!
Consider implementing Collection.

Someone else has thought about “completeness.”

You will interoperate with more code.

But (it seems like) you need to implement 15 methods!

You can extend AbstractCollection to get a leg up.

▶ You’ll “just” need size and iterator.
▶ You ought to provide a constructor from Collection.
▶ How about add?

That iterator is a bit challenging.

Allows for (Star s : tree) { }
Enhanced For Loop

// Old style
static void swear(Collection<String> c) {
    for (Iterator<String> i = c.iterator(); i.hasNext();)
        String word = i.next();
    if (word.length() == 4)
        System.out.println(word);
}

// New style
static void swear(Collection<String> c) {
    for (String word : c)
        if (word.length() == 4)
            System.out.println(word);
}
How does the new for loop work?

- It’s just “syntactic sugar”
- Essentially compiles to the Iterator code.
- What’s required?
  - c.iterator()
  - A new interface: Iterable
- Key point: c is Iterable, not an Iterator.
- An Iterator abstracts one iteration, Iterable abstracts the ability to be iterated over.
- When will you still use the iterator code pattern?
  - When you need access to Iterator.remove()
Why should you bother?

- Java made many things Iterable
  - Arrays
  - Collections or all kinds
- So why should you bother making *your* types Iterable?
  - You could just return one of the iterable types.
  - And clients would still get to use the nice for-loops.
  - Remember to copy or wrap modifiable objects.
Why should you bother?

- Java made many things Iterable
  - Arrays
  - Collections or all kinds

- So why should you bother making your types Iterable?
  - You could just return one of the iterable types.
  - And clients would still get to use the nice for-loops.
  - Remember to copy or wrap modifiable objects.

- But consider allowing iteration over a binary tree.
  - Or your KDTree.
  - Why can’t we lean on the existing collections?
Why should you bother?

● Java made many things Iterable
  ▶ Arrays
  ▶ Collections or all kinds

● So why should you bother making *your* types Iterable?
  ▶ You could just return one of the iterable types.
  ▶ And clients would still get to use the nice for-loops.
  ▶ Remember to copy or wrap modifiable objects.

● But consider allowing iteration over a binary tree.
  ▶ Or your KDTree.
  ▶ Why can’t we lean on the existing collections?
  ▶ You don’t want to have to copy your (possibly big) tree into another collection for each call.
Making BitSet Iterable

- Java library provides BitSet
  - A compact representation for a set of (small) integers.
  - Uses an array of integers internally.
  - Each 32-bit integer records the presence/absence of 32 integers.

- But it isn’t Iterable

- How would we make an Iterable BitSet?
  - What primitives do you need BitSet to have?
  - Anything else you would like?
  - How do you structure the classes?
```java
@Test public void testConstruction() {
    IntSet set = new IntSet(21, 10, 13);
    Set<Integer> hs = Sets.newHashSet(13, 10, 21);
    assertEquals(hs.size(), set.cardinality());
    assertTrue(set.get(21));
    assertTrue(set.get(10));
    assertTrue(set.get(13));
    assertFalse(set.get(0));
    assertFalse(set.get(1));
    assertFalse(set.get(11));

    for (int i : set) {
        assertTrue(hs.contains(i));
        hs.remove(i);
    }

    assertTrue(hs.isEmpty());
}
```
IntSet, an Iterable BitSet

```java
public class IntSet extends BitSet implements Iterable<Integer> {
    public IntSet(Integer... ints) {
        super();
        for (int i : ints)
            set(i);
    }
    public Iterator<Integer> iterator() {
        return new Itr();
    }
    private class Itr implements Iterator<Integer> {
        ...
    }
}
```
private class Itr implements Iterator<Integer> {
    int cursor = -1;

    public boolean hasNext() {
        return nextSetBit(cursor + 1) >= 0;
    }

    public Integer next() {
        int bit = nextSetBit(cursor + 1);
        if (bit < 0)
            throw new NoSuchElementException();
        cursor = bit;
        return cursor;
    }

    public void remove() {
        clear(cursor);
    }
}