Section 5

Java Generics

CSE 331 Spring 2013

May 16, 2013

Includes materials from Krysta Yousoufian, Jackson Roberts, Marty Stepp, David Notkin and Joshua Bloch's *Effective Java*.

Generics

(Example code will be posted on the course website)

Generic Types

- Each generic type defines a set of parameterized types.
  - Syntax: `public class ClassName<GENERIC PARAMS>`
  - `List<E>` defines `List<Color>`, `List<String>`, etc.
- Generic type information is lost during run-time:

```
List<String>.class;                          // Compile Error!

List<Long> foo = new ArrayList<Long>();
foo instanceof List<Long>;                   // Compile Error!
foo instanceof List;                         // Evaluates to true
```

- Raw types (e.g. `List`, `Set`) behave like normal Java objects, but should never be used in new code.

Generic Methods

- Use generics without creating a generic type.
- A generic method uses an arbitrary type as a parameter or return value.
- To declare a method as generic, put `<E>` (or `<T>` or …) before the return type:

```
public static <E> void add(Set<E> items, E element)
public static <T> Set<T> union (Set<T> s1, Set<T> s2)
```

- Example: `SetUtils.union()`

Generics and Arrays

- Generic types in Java are invariant; Arrays are covariant.
  - `Integer[]` is a Java subtype of `Number[]`
  - `List<Integer>` is not a Java subtype of `List<Number>`
- Arrays are reified – they enforce element types at runtime.

```
Set<Long>[] array = new Set<Long>[1];   // Compile Error!

T[] array = new T[10];   // Compile Error!
```

- As a result, implementing generic types using arrays is complicated.
  - Requires casting. Type safety must be proven manually.
  - *Effective Java* c.5 describes all of the messy details.
- **Use lists instead, unless you truly need an array.**

Let’s Break Java’s type system

- TypeBreaker.java

Implementing Generic ArrayList

- ArrayList.java

Why are Generic Types Invariant?

- If we (illegally) use ArrayList<Integer> in place of ArrayList<Number>, the add method type checks because:
  - `public void add(int index, Integer value)`

  is **weaker** than:
  - `public void add(int index, Number value)`
- But the get method fails:
  - `public Integer get(int index)`

  is **stronger** than:
  - `public Number get(int index)`

The Problem with Invariance

- What if we want to add?
  - `public void addTo(List<Integer>, Integer a)`
  - This doesn't work:
    - `List<Number> lst = new ArrayList<Number>();`
    - `Lst.addTo(lst, 5); // lst is not of type List<Integer>`
- Or retrieve?
  - `public Integer getFrom(List<Integer> lst, int index)`
  - This doesn't work:
    - `List<EvenInteger> lst = new ArrayList<EvenInteger>();`
    - `Integer a = lst.getFrom(lst, 2); // lst is not of type List<Integer>`

Bounded Wildcards

- Extends
  - Syntax: `Set<? extends Foo>`
  - Requires type `Foo`, or any subtype of `Foo`
  - Example: `unionBetter()`
- Super
  - Syntax: `Set<? super Foo>`
  - Requires type `Foo`, or any supertype of `Foo`
  - Example: `addAllBetter()`

PECS

"Producer-extends, Consumer-super"

- In general...
  - Producer methods should use `<? extends T>` for generic parameters.
  - Consumer methods generally should use `<? super T>` for generic parameters.
- PECS helps prevent unnecessary restrictions on generic parameters.
- Bottom line: Make your ADT parameters as flexible as possible. *This includes type parameters.*

Unbounded Wildcards

- You have an object of a generic type, but don't care what its type parameter is.
  - You care that you have a `Set`
  - You don't care if you have a `Set<String>` vs. `Set<Integer>`
- Usage:
  - Use `<?>` instead of `<E>`
  - Why not use raw type `Set` instead of wildcard `Set<?>` ?
  - (Almost) never use raw types – they aren't type safe!
- Example:
  - `public int size(List<?> lst);`
  - `public boolean contains(List<?> lst, Object o);`

Exercise: SetUtils

- How could we make the following method signatures more flexible by using (bounded) wildcards?

- `public static <E> Set<E> union(Set<E> s1, Set<E> s2)`

- `public static int intersectionCount(Set<E> s1, Set<E> s2)`

- `public static <E> void addAll(Set<E> source, Set<E> dest)`

Solution: SetUtils

- `public static <E> Set<E> unionBetter(Set<? extends E> s1, Set<? extends E> s2)`
- `public static int intersectionCount(Set<?> s1, Set<?> s2)`
- `public static <E> void addAllBetter(Set<E> source, Set<? super E> dest)`

When Not To Use Wildcards

- Type parameters which are used elsewhere.
- As return types for methods.
  - `Set<?>` and `Set<Object>` are not the same. What is their relationship?
  - Read `Set<?>` as "Set of some arbitrary type."
- Examples:
  - `union()` creates new `Set<E>`
  - `addAll()` adds items

Sort

- How does the type parameter of this method work?
- `public static <T> void sort(Collection<? extends Comparable<? super T>> coll)`

Exercise: Legal Ops

- Object o;
- Shape s;
- Rectangle r;
- SpecialRectangle q;
- List<? extends Rectangle> ler;

- Which of these is legal?
- ler.add(o);
- ler.add(s);
- ler.add(r);
- ler.add(q);
- ler.add(null);
- o = ler.get(0);
- s = ler.get(0);
- r = ler.get(0);
- q = ler.get(0);

Exercise: Legal Ops

- Object o;
- Shape s;
- Rectangle r;
- SpecialRectangle q;
- List<? super Rectangle> ler;

- Which of these is legal?
- ler.add(o);
- ler.add(s);
- ler.add(r);
- ler.add(q);
- ler.add(null);
- o = ler.get(0);
- s = ler.get(0);
- r = ler.get(0);
- q = ler.get(0);