CSE 331 Section 5

Java Generics

Jackson Roberts

CSE 331 Spring 2012
April 26, 2012

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

Homework Questions

- HW3: Lingering questions?
- HW4:
  - What was difficult or problematic?
  - What did you find valuable?
  - Any topics you would like to see covered?

Commenting

- Have a look at the Java Style Guide on the course website.
- Clarity is the primary goal of comments.
- Know your audience: Other programmers!
- Good code is usually readable with few comments.
- Javadoc can be verbose out of necessity, but should be as concise as possible.

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

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

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.

- As a result, implementing generic types using arrays is complicated.
  - Necessary casting eliminates compile-time type checking.
  - 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.

Generic Methods

- Use generics without creating a generic type.
- A generic method uses some unknown type (i.e. 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()`

Generic 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: `SetUtils.intersectionCount()`

When Not To Use Wildcards

- Type parameters which are used elsewhere.

- As return types for methods.
  - `Set<?>` and `Set<Object>` are not the same.
  - Read `Set<?>` as "Set of some arbitrary type."

- Examples:
  - `union()` creates new `Set<E>`
  - `addAll()` adds items

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.*