More than you ever wanted to know about

Java Generics

Jeff Meister

CMSC 420 Summer 2007

The obligatory review of the boring stuff, or…

GENERICS: A YOUNG LADY’S ILLUSTRATED PRIMER IN FOUR SLIDES

Java 1.4: Life Before Generics

Java code used to look like this:

```
List listOfFruits = new ArrayList();
listOfFruits.add(new Fruit(“Apple”));
Fruit apple = (Fruit) listOfFruits.remove(0);
listOfFruits.add(new Vegetable(“Carrot”)); // Whoops!
Fruit orange = (Fruit) listOfFruits.remove(0); // Run-time error
```

Problem: Compiler doesn’t know listOfFruits should only contain fruits

A Silly Solution

We could make our own fruit-only list class:

```
class FruitList {
    void add(Fruit element) { … }
    Fruit remove(int index) { … }
    …
}
```

But what about when we want a vegetable-only list later? Copy-paste? Lots of bloated, unmaintainable code?

Java 1.5: Now We’re Talking

Now, Java code looks like this:

```
List<Fruit> listOfFruits = new ArrayList<Fruit>();
listOfFruits.add(new Fruit(“Apple”));
Fruit apple = listOfFruits.remove(0);
listOfFruits.add(new Vegetable(“Carrot”)); // Compile-time error
```

Hooray! Compiler now knows listOfFruits contains only Fruits

- So remove() must return a Fruit
- And add() cannot take a Vegetable

You guys remember this, right?

Here’s how we’d write that generic List class:

```
class List<T> {
    void add(T element) { … }
    T remove(int index) { … }
    …
}
```

Problem solved! Simply invoke List<Fruit>, List<Vegetable>, and so on. I’m sure you’ve written code like this before, so let’s move to…

Hope you planned on being confused today, because it’s time for…

THE FUN STUFF

Abandon All Hope…

- Generics implement *parametric polymorphism*
  - Parametric: The type parameter (e.g., <T>)…
  - Polymorphism: …can take many forms
- However, if we're going to program with parameterized types, we need to understand how the language rules apply to them
- Java generics are implemented using *type erasure*, which leads to all sorts of wacky issues, as we'll see

Subtyping

Since Apple is a subtype of Object, is List<Apple> a subtype of List<Object>?

```
List<Apple> apples = new ArrayList<Apple>();
List<Object> objs = apples; // Does this compile?
```

Seems harmless, but no! If that worked, we could put Oranges in our List<Apple> like so:

```
objs.add(new Orange()); // OK because objs is a List<Object>
Apple a = apples.remove(0); // Would assign Orange to Apple!
```

An Aside: Subtyping and Java Arrays

- Java arrays actually have the subtyping problem just described (they are covariant)
- The following obviously wrong code compiles, only to fail at run-time:

```
Apple[] apples = new Apple[3];
Object[] objs = apples; // The compiler permits this!
objs[0] = new Orange(); // ArrayStoreException
```

- Avoid mixing arrays and generics (trust me)

Wildcard Types

- So, what *is* List<Apple> a subtype of?
- The supertype of all kinds of lists is List<?>, the List of unknown
- The ? is a wildcard that matches anything
- We can’t add things (except null) to a List<?>, since we don’t know what the List is really of
- But we can retrieve things and treat them as Objects, since we know they are at least that

Bounded Wildcards

- Wildcard types can have upper and lower bounds
- A List<? extends Fruit> is a List of items that have unknown type but are all at *least* Fruits
  - So it can contain Fruits and Apples but not Peas
- A List<? super Fruit> is a List of items that have unknown type but are all at *most* Fruits
  - So it can contain Fruits and Objects but not Apples

Bounded Wildcards Example

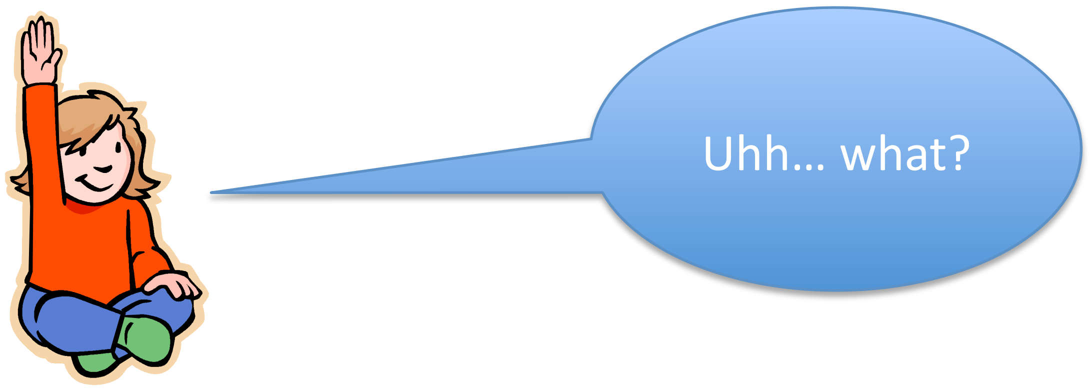
```
class WholesaleVendor<T> {
    void buy(int howMany, List<? super T> fillMeUp) { … }
    void sell(List<? extends T> emptyMe) { … }
    …
}

WholesaleVendor<Fruit> vendor = new WholesaleVendor<Fruit>();
List<Food> stock = …;
List<Apple> overstockApples = …;

// I can buy Food from the Fruit vendor:
vendor.buy(100, stock);
// I can sell my Apples to the Fruit vendor:
vendor.sell(overstockApples);
```

Josh Bloch's Bounded Wildcards Rule

- Use <? extends T> when parameterized instance is a T producer (for reading/input)
- Use <? super T> when parameterized instance is a T consumer (for writing/output)



Generic Methods

- You can parameterize methods too. Here’s a signature from the Java API:

```
static <T> void fill(List<? super T> list, T obj);
```

- Easy enough, yeah? Try this one on for size:

```
static <T extends Object & Comparable<? super T>> T
   max(Collection<? extends T> coll);
```

- You don’t need to explicitly instantiate generic methods (the compiler will figure it out)

How Generics are Implemented

- Rather than change every JVM between Java 1.4 and 1.5, they chose to use *erasure*
- After the compiler does its type checking, it *discards* the generics; the JVM never sees them!
- It works something like this:
  - Type information between angle brackets is thrown out, e.g., List<String> ➔ List
  - Uses of type variables are replaced by their upper bound (usually Object)
  - Casts are inserted to preserve type-correctness

Pros and Cons of Erasure

- Good: Backward compatibility is maintained, so you can still use legacy non-generic libraries
- Bad: You can’t find out what type a generic class is using at run-time:

```
class Example<T> {
    void method(Object item) {
        if (item instanceof T) { … } // Compiler error!
        T anotherItem = new T(); // Compiler error!
        T[] itemArray = new T[10]; // Compiler error!
    }
}
```

Using Legacy Code in Generic Code

- Say I have some generic code dealing with Fruits, but I want to call this legacy library function:

```
Smoothie makeSmoothie(String name, List fruits);
```

- I can pass in my generic List<Fruit> for the fruits parameter, which has the *raw type* List. But why? That seems unsafe… makeSmoothie() could stick a Vegetable in the list, and that would taste nasty!

Raw Types and Generic Types

- List doesn’t mean List<Object>, because then we couldn’t pass in a List<Fruit> (subtyping, remember?)
- List doesn’t mean List<?> either, because then we couldn’t assign a List to a List<Fruit> (which is a legal operation)
- We need both of these to work for generic code to interoperate with legacy code
- Raw types basically work like wildcard types, just not checked as stringently
  - These operations generate an unchecked warning

The Problem with Legacy Code

- “Calling legacy code from generic code is inherently dangerous; once you mix generic code with non-generic legacy code, all the safety guarantees that the generic type system usually provides are void. However, you are still better off than you were without using generics at all. At least you know the code on your end is consistent.” – Gilad Bracha, Java Generics Developer

My Advice on Generics

- Don’t try to think about generic code abstractly; make an example instantiation in your head and run through scenarios using it
- Generics are a valuable tool to ensure type safety, so use them! Let the compiler help you
- However, generics also complicate syntax, and they can generate some nasty errors that are a pain to understand and debug

An Analogy: Functions

- Problem: I want to perform the same computation on many different input values without writing the computation over and over.
- Solution: Write a function! Use a variable to represent the input value, and write your code to perform the computation on this variable in a way that does not depend on its value. Now you can call the function many times, passing in different values for the variable. Easy stuff.

Generics Provide Another Abstraction

- Problem: I want to use the same class (or method) with objects of many different types without writing the class over and over or sacrificing type safety.
- Solution: Generify the class! Use a variable T to represent the input type, and write your code to operate on objects of type T in a way that does not depend on the actual value of T. Now you can instantiate the class many times, passing in different types for T.
- See? It’s not so bad. Generics just allow you to abstract over types instead of values.