Java Generics

Università di Modena e Reggio Emilia

Prof. Nicola Bicocchi (nicola.bicocchi@unimore.it)

Java SE4: Life Before Generics

Ordinary classes and methods work with specific types: either primitives or class types. If you are writing code that might be used across more types, this rigidity can be overconstraining. (*)

*Angelika Langer's Java Generics FAQ
(www.angelikalanger.com/GenericsFAQ/JavaGenericsFAQ.html)

Java SE4: 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
```

- 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 SE5: Now We’re Talking

Here’s how we would write that generic class:

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

Problem solved! Simply invoke List<Fruit>, List<Vegetable>, and so on!

Java SE5: 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”)); // error
```

- Compiler now knows listOfFruits contains only Fruits!!
  - remove() must return a Fruit
  - add() cannot take a Vegetable

Recap

```
// Raw Type: Evil
Collection coinCollection = new ArrayList();
coinCollection.add(new Stamp()); // Succeeds but should not
...
for (Object o : coinCollection) {
        Coin c = (Coin) o;  // Throws exception at runtime
        ...
}

// Generic type: Good
Collection<Coin> coinCollection = new ArrayList<Coin>();
coinCollection.add(new Stamp());  // Won't compile
...
for (Coin c : coinCollection) {
        ...
}
```

A practical example

```
// Method names are from the perspective of customer
public interface Shop<T> {
    T buy();
    void sell(T myItem);
    void buy(int numItems, Collection<T> myItems);
    void sell(Collection<T> myItems);
}
class Model { }
class ModelPlane extends Model { }
class ModelTrain extends Model { }
```

Works Fine if You Stick to One Type

```
// Individual purchase and sale
Shop<ModelPlane> modelPlaneShop = ... ;
ModelPlane myPlane = modelPlaneShop.buy();
modelPlaneShop.sell(myPlane);

// Bulk purchase and sale
Collection<ModelPlane> myPlanes = ... ;
modelPlaneShop.buy(5, myPlanes);
modelPlaneShop.sell(myPlanes);
```

Simple Subtyping Works Fine

```
// You can buy a model from a train shop
Model myModel = modelTrainShop.buy();

// You can sell a model train to a model shop
modelShop.sell(myTrain);

public interface Shop<T> {
	T buy();
	void sell(T myItem);
	void buy(int numItems, Collection<T> myItems);
	void sell(Collection<T> myItems);
}
```

Collection Subtyping Doesn't Work!

```
// You can't buy a bunch of models from the train shop
modelTrainShop.buy(5, myModels); // Won't compile

// You can't sell a bunch of trains to the model shop
modelShop.sell(myTrains); // Won't compile

public interface Shop<T> {
	T buy();
	void sell(T item);
	void buy(int numItems, Collection<T> myStuff);
	void sell(Collection<T> lot);
}
```

Subtyping and Collection

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!
```

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 Types

- 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 Objects
- 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 to the Rescue

```
// You can buy a bunch of models from the train shop
modelTrainShop.buy(5, myModels); // Compiles
// You can sell your train set to the model shop;
modelShop.sell(myTrains); // Compiles

public interface Shop<T> {
    T buy();
    void sell(T item);
    void buy(int numItems, Collection<? super T> myStuff);
    void sell(Collection<? extends T> lot);
}
```

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);
```

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

Pros and Cons of Erasure

```
import java.util.*;
public class ErasedTypeEquivalence {
    public static void main(String[] args) {
        Class c1 = new ArrayList<String>().getClass();
        Class c2 = new ArrayList<Integer>().getClass();
        System.out.println(c1 == c2);
    }
}
/*
Output:
true
*/
```

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!

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 than you were without using generics at all. At least you know the code on your end is consistent." – *Gilad Bracha, Java Generics Developer*

Subtyping and Arrays

- Java arrays actually have the subtyping problem just described (covariant arrays*)
- 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
```

**http://en.wikipedia.org/wiki/Covariance_and_contravariance_%28computer_science%29#Covariant_arrays_in_Java_and_C.23*