Java Generics

Università di Modena e Reggio Emilia
Prof. Nicola Bicocchi (nicola.bicocchi@unimore.it)

ING

Java SE4: Life Before Generics

- Generics was added in Java 5 to provide compile-time type checking and removing risk of ClassCastException that was common while working with collection classes.
- The whole collection framework (JCF) was re-written to use generics for type-safety.

Java 4: Life Before Generics

```
List l = new ArrayList();
l.add(new Fruit(“Apple”));
l.add(new Vegetable(“Carrot”));

Fruit f;
f = (Fruit) listOfFruits.get(0);
f = (Fruit) listOfFruits.get(1);  //  Runtime 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  f)  { … }
    Fruit get(int index) { … }
    Fruit remove(int  index)  { … }
    …
}
```

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

Java 5: Now We're Talking

Here's how we would write that generic class:

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

Compiler knows that List contains only objects of type T

- remove() must return T
- add() accepts only T

Java SE5: Now We're Talking

Now, Java code looks like this:

```
List<Fruit> l = new  ArrayList<Fruit>();
l.add(new Fruit("Apple"));
l.add(new Vegetable("Carrot")); // Compile time error
```

Summary

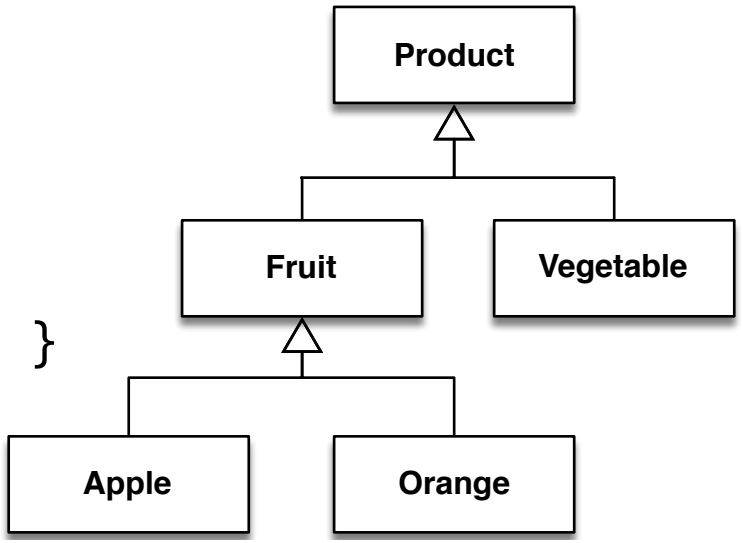
```
// Raw Type: Evil
List l = new ArrayList();
l.add(new Fruit());
l.add(new Vegetable()); // Succeeds but should not
...
for (Object o : l) {
     Fruit f = (Fruit) o;
      // Downcast eventually leading to run-time error
     ...
}

// Generic type: Good
List<Fruit> l = new ArrayList<Fruit>();
l.add(new Fruit());
l.add(new Vegetable());  // Compile-time error
...
for (Fruit f : l) {
     ...
}
```

The Comparable Interface

```
package java.lang;
import java.util.*;

public interface Comparable<T> {
    public int compareTo(T o);
}
```

A practical example

```
public interface Shop<T> {
    T sell();
    void buy(T item);

    void sell(Collection<T> item, int nItems);
    void buy(Collection<T> item);
}

class Product { }
class Fruit extends Product { }
class Vegetable extends Product { }
```



ING

One type works fine

```
Shop<Fruit> fruitShop = new Shop<Fruit>;

// Individual purchase and resale
fruitShop.buy(new Fruit());
Fruit f = fruitShop.sell();

// Bulk purchase and resale
List<Fruit> fruits = new ArrayList<Fruit>();
fruitShop.buy(fruits);
fruitShop.sell(fruits, 5);

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

Single-object subtyping (ok)

```
// You can buy a Product from a Fruit shop
Product p = fruitShop.sell();

// You can sell a Fruit to Product shop
productShop.buy(new Fruit());

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

Collection subtyping (!ok)

```
// You can't buy (a list of) products from the fruit shop
List<Product> myProducts = new ArrayList<Product>();
fruitShop.sell(myProducts, 5); // Compile error

// You can't sell (a list of) fruits to the product shop
List<Fruit> myFruits = new ArrayList<Fruit>();
productShop.buy(myFruits);    // Compile error

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

Subtyping and Collections

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

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

Seems harmless, but it is not!

If that worked, we could put Vegetables in our List<Fruit> like so:

```
objs.add(new Vegetable());   // OK because objs is a List<Object>
Fruit f = fruits.remove(0);  // Would assign Vegetables to Fruits!
```

Wildcard Types

- So, what is List<Fruit> a subtype of?
- The supertype of all kinds of lists is not List<Object> but **List<?> (the list of unknown)**
- The **? is a wildcard** matching with anything
  - We can't add things (except null) to a List<?>, since we don't know what the List is really of. However, we can retrieve things and treat them as Objects, since we know they are at least that

ING

Wildcards Types (Bounded)

- 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

```
List<Product> myProducts = new ArrayList<Product>();
fruitShop.sell(myProducts, 5); // OK!

List<Fruit> myFruits = new ArrayList<Fruit>();
productShop.buy(myFruits);     // OK!

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

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)

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:

```
Fruit[] fruits = new Fruit[16];
Object[] objs = fruits;  // The  compiler  permits  this!
objs[0] = new Apple();   // ArrayStoreException
```

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

Generic Methods

- You can parameterize methods too. The fill method (java.util.Collections) is used to replace all of the elements of the specified list with the specified element.

```
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)

Generic Methods

Imagine you want to write a method for copying an array into a Collection.

```
// wrong! Cannot add objects to Collection<?>
static void fromArrayToCollection(Object[] a, Collection<?> c) {
    for (Object o : a) {
        c.add(o); // compile-time error
    }
}

// ok! using generic methods
static <T> void fromArrayToCollection(T[] a, Collection<T> c) {
    for (T o : a) {
        c.add(o); // Correct
    }
}
```

Wildcards or not?

```
// Java API
interface Collection<E> {
    public boolean containsAll(Collection<?> c);
    public boolean addAll(Collection<? extends E> c);
}

// Alternative legitimate version
interface Collection<E> {
    public <T> boolean containsAll(Collection<T> c);
    public <T extends E> boolean addAll(Collection<T> c);
    // Hey, type variables can have bounds too!
}
```

Wildcards or not?

```
// Java API
class Collections {
    public static <T> void copy(List<T> dest,
      List<? extends T> src) {
    ...
}

// Alternative legitimate version
class Collections {
    public static <T, S extends T> void copy(List<T>
      dest, List<S> src) {
    ...
}
```

Wildcards or not?

Rule of thumb: Generic methods allow type parameters to be used to express dependencies among the types of one or more arguments to a method and/or its return type. If there isn't such a dependency, a generic method should not be used.

How Generics are Implemented

- Rather than change every JVM between Java 4 and 5, engineers chose to use code 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

How Generics are Implemented

```
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
*/
```

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

Using Legacy Code in Generic Code

- Say I have some generic code dealing with Fruits, but I want to call legacy library functions:
  - `void foo(String name, List fruits);`

- I can pass in my generic List<Fruit> for the fruits parameter, which has the raw type List? That seems unsafe…
  - foo() 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