Java Generics

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

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 fruitList = new ArrayList();
fruitList.add(new Fruit(“Apple”));
fruitList.add(new Vegetable(“Carrot”));

Fruit f;
f = (Fruit) fruitList.get(0);
f = (Fruit) fruitList.get(1);
//  Runtime Error! (java.lang.ClassCastException)
```

Compiler doesn’t know that *fruitList* 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  GenericList<T>  {
   void  add(T  element)  {  …  }
   T  get(int  index)      {  …  }
   T  remove(int  index)  {  …  }
   …
}
```

Compiler knows that *GenericList* contains only objects of type T

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

Examples

```
public class ArrayList<T> {
   T get(int index);
}

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

public interface Comparator<T> {
    public int compare(T o1, T o2);
}
```

Java SE5: Now We're Talking

Now, Java code looks like this:

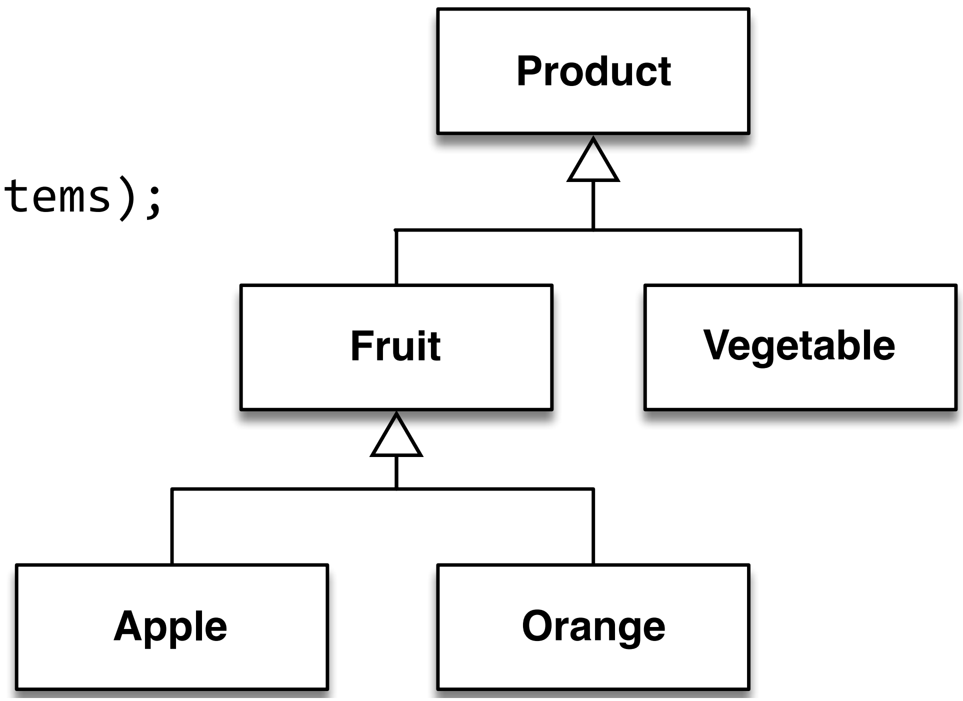
```
List<Fruit> l = new ArrayList<Fruit>();
l.add(new Fruit("Apple"));
l.add(new Vegetable("Carrot")); // Compile time error!
```

Summary

```
// Raw types: 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 types: Ok!
List<Fruit> l = new ArrayList<Fruit>();
l.add(new Fruit());
l.add(new Vegetable());  // Compile-time error
...
for (Fruit f : l) {
     ...
}
```

A practical example

```
public interface Shop<T> {
   private List<T> items;

   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 { }
…
```



One type works fine

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

// Individual purchase and resale
fs.buy(new Fruit());
Fruit fs = s.sell();

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

// You can sell a Fruit to Product shop
Shop<Product> ps = new Shop<Product>;
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)

```
// The fruit shop cannot store a list of fruits in a list of products
List<Product> myProducts = new ArrayList<Product>();
fs.sell(myProducts, 5); // Compile error

// The product shop cannot buy products from a list of fruits
List<Fruit> myFruits = new ArrayList<Fruit>();
ps.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 Product is a subtype of Object, is List<Product> a subtype of List<Object>?

```
List<Product> pl = new ArrayList<Product>();
List<Object> ol = pl;   // Does  this  compile? (Hopefully not!)
```

If that worked, we could put any object in our List<Product>. For example:

```
ol.add(new String("a crash is likely imminent"));
// OK because ol is a List<Object>

Product p = pl.remove(0);
// Would assign a String object to Product reference
// (ClassCastException)!
```

Wildcard Types

- So, what is List<Product> 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

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

ING

Bounded Wildcards to the Rescue

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

List<Fruit> myFruits = new ArrayList<Fruit>();
ps.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)

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

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 parametrize methods too. Generic methods are methods that introduce their own type parameters. This is similar to declaring a generic type, but the type parameter's scope is limited to the method where it is declared.
- For example, the fill() method (java.util.Collections) is used to replace all the elements of the specified list with the specified element.

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

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);
}
```

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) {
    ...
}
```

How Generics are Implemented

- Rather than undergoing major changes between Java 4 and Java 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 (angle brackets) is thrown out: List<String> -> List
  - Type variables are replaced by their upper bound (usually Object)
  - Casts are inserted to preserve type-correctness

Code Erasure

```
public class Stack<E> {
    private E[] stackContent;

    public Stack(int capacity) {
        this.stackContent = (E[]) new
Object[capacity];
    }

    public void push(E data) {
        // ..
    }

    public E pop() {
        // ..
    }
}
```

```
public class Stack {
    private Object[] stackContent;

    public Stack(int capacity) {
        this.stackContent = (Object[])
new Object[capacity];
    }

    public void push(Object data) {
        // ..
    }

    public Object pop() {
        // ..
    }
}
```

Code Erasure

```
public class BoundStack<E extends Comparable<E>>
{
    private E[] stackContent;

    public BoundStack(int capacity) {
        this.stackContent = (E[]) new
Object[capacity];
    }

    public void push(E data) {
        // ..
    }

    public E pop() {
        // ..
    }
}
```

```
public class BoundStack {
    private Comparable [] stackContent;

    public BoundStack(int capacity) {
        this.stackContent = (Comparable[]) new
Object[capacity];
    }

    public void push(Comparable data) {
        // ..
    }

    public Comparable pop() {
        // ..
    }
}
```

Code 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
*/
```

Pros and Cons of Code 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 generic code dealing with Fruits, but I want to call legacy library functions:
  - `void renameFruits(String name, List fruits);`

- I can pass in my generic *List<Fruit>* as the *fruits* parameter, which has the raw type List. That seems unsafe…
  - `renameFruits()` 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