301AA - Advanced Programming [AP-2017]

Lecturer: Andrea Corradini andrea@di.unipi.it
Tutor: Lillo Galletta galletta@di.unipi.it

Department of Computer Science, Pisa
Academic Year 2017/18

AP-2018-15: Java Generics
301AA - Advanced Programming

Lecturer: Andrea Corradini
andrea@di.unipi.it
http://pages.di.unipi.it/corradini/

Course pages:
http://pages.di.unipi.it/corradini/Didattica/AP-18/

AP-2018-15: Java Generics
Outline

• Java generics
• Type bounds
• Generics and subtyping
• Covariance, contravariance in Java and other languages
• Subtyping and arrays in Java
• Wildcards
• Type erasure
• Limitations of generics
Classification of Polymorphism

- Universal
  - Parametric
    - Explicit
    - Bounded
      - Covariant
      - Invariant
      - Contravariant
  - Implicit
- Coercion
- Inclusion
  - Overriding
  - Overloading
- Ad hoc
- Polymorphism
Java Generics
Explicit Parametric Polymorphism

• Classes, Interfaces, Methods can have type parameters
• The type parameters can be used arbitrarily in the definition
• They can be instantiated by providing arbitrary (reference) type arguments
• We discuss only a few issues about Java generics...

interface List<E> {
    boolean add(E n);
    E get(int index);
}

List<Integer>
List<Number>
List<String>
List<List<String>>>
...

Tutorials on Java generics:
https://docs.oracle.com/javase/tutorial/java/generics/index.html
http://thegreyblog.blogspot.it/2011/03/java-generics-tutorial-part-i-basics.html
Generic methods

- Methods can use the type parameters of the class where they are defined, if any.
- They can also introduce their own type parameters.

```
public static <T> T getFirst(List<T> list)
```

- Invocations of generic methods must instantiate all type parameters, either explicitly or implicitly.
  - A form of type inference.
Bounded Type Parameters

interface List<E extends Number> {
    void m(E arg) {
        arg.asInt(); // OK, Number and its subtypes
        // support asInt()
    }
}

• Only classes implementing Number can be used as type arguments
• Method defined in the bound (Number) can be invoked on objects of the type parameter
Type Bounds

<TypeVar extends SuperType>
  – *upper bound*; SuperType and any of its subtype are ok.

<TypeVar extends ClassA & InterfaceB & InterfaceC & …>
  – *Multiple* upper bounds

<TypeVar super SubType>
  – *lower bound*; SubType and any of its supertype are ok

• Type bounds for methods guarantee that the type argument supports the operations used in the method body

• Unlike C++ where overloading is resolved and can fail after instantiating a template, in Java type checking ensures that overloading will succeed
A generic algorithm with type bounds

public static <T> int countGreaterThan(T[] anArray, T elem) {
    int count = 0;
    for (T e : anArray)
        if (e > elem) // compiler error
            ++count;
    return count;
}

public interface Comparable<T> {
    // classes implementing
    public int compareTo(T o); // Comparable provide a
} // default way to compare their objects

public static <T extends Comparable<T>>
    int countGreaterThan(T[] anArray, T elem) {
    int count = 0;
    for (T e : anArray)
        if (e.compareTo(elem) > 0) // ok, it compiles
            ++count;
    return count;
}
Generics and subtyping

• **Integer** is subtype of **Number**

• Is **List<Integer>** subtype of **List<Number>**?

• NO!
What are Java rules?

• Given two concrete types \( A \) and \( B \), \( \text{MyClass}<A> \) has no relationship to \( \text{MyClass}<B> \), regardless of whether or not \( A \) and \( B \) are related.
• Formally: subtyping in Java is invariant for generic classes.
• Note: The common parent of \( \text{MyClass}<A> \) and \( \text{MyClass}<B> \) is \( \text{MyClass}<?> \): the “wildcard” \( ? \) Will be discussed later.
• On the other hand, as expected, if \( A \) extends \( B \) and they are generic classes, for each type \( C \) we have that \( A<C> \) extends \( B<C> \).
• Thus, for example, \( \text{ArrayList<Integer>} \) is subtype of \( \text{List<Integer>} \)
List<Number> e List<Integer>

interface List<T> {
    boolean add(T elt);
    T get(int index);
}

type List<Number> has:
    boolean add(Number elt);
    Number get(int index);

type List<Integer> has:
    boolean add(Integer elt);
    Integer get(int index);

Is the **Substitution Principle** satisfied in either direction? Thus List<Number> is neither a supertype nor a subtype of List<Integer>: Java rules are adequate here
But in more specific situations...

```java
interface List<T> {
    T get(int index);
}

type List<Number>: Number get(int index);

type List<Integer>: Integer get(int index);

A covariant notion of subtyping would be safe:
    - List<Integer> can be subtype of List<Number>
    - Not in Java

• In general: covariance is safe if the type is read-only
```
Viceversa... contravariance!

interface List<T> {
    boolean add(T elt);
}

type List<Number>:
    boolean add(Number elt);

type List<Integer>:
    boolean add(Integer elt);

A **contravariant** notion of subtyping would be safe:

- List<Number> can be a subtype of List<Integer>
- But Java .....  

In general: **contravariance** is safe if the type is **write-only**
Generics and subtypes in C#

• In C#, the type parameter of a generic class can be annotated **out** (covariant) or **in** (contravariant), otherwise it is **invariant**. Examples:
  
  • **Ienumerator** is covariant, because the only method returns an enumerator, which accesses the collection in read-only

```csharp
public interface IEnumerable<out T> : [...] {
    public [...] IEnumerator<out T> GetEnumerator();
}
```

• **IComparable** is contravariant, because the only method has an argument of type **T**

```csharp
public interface IComparable<in T> {
    public int CompareTo(T other);
}
```
Co- and Contra-variance in Scala

• Also Scala supports co/contra-variance annotations (− and +) for type parameters:

```scala
class VendingMachine[+A]{…}
class GarbageCan[−A]{…}

trait Function1[−T, +R] extends AnyRef
{ def apply(v1: T): R }
```

http://blog.kamkor.me/Covariance-And-Contravariance-In-Scala/
A digression: Java arrays

• Arrays are like built-in containers
  – Let \( \text{Type1} \) be a subtype of \( \text{Type2} \).
  – How are \( \text{Type1}[\ ] \) e \( \text{Type2}[\] \) related?

• Consider the following generic class, mimicking arrays:

```java
class Array<T> {
    public T get(int i) { ... “op” ... }
    public T set(T newVal, int i) { ... “op” ... }
}
```

According with Java rules, \( \text{Array<Type1>} \) and \( \text{Array<Type2>} \) are not related by subtyping
But instead...

• In Java, if **Type1** is a subtype of **Type2**, then **Type1**[] is a subtype of **Type2**[]. Thus Java arrays are **covariant**.

• Java (and also C#, .NET) fixed this rule before the introduction of generics.

• Why? Think to **void sort(Object[] o)**;

• Without **covariance**, a new sort method is needed for each reference type different from Object!

• But sorting does not insert new objects in the array, thus it cannot cause type errors if used covariantly
Problems with array covariance

Even if it works for `sort`, covariance may cause type errors in general

```java
Apple[] apples = new Apple[1];
Fruit[] fruits = apples; //ok, covariance
fruits[0] = new Strawberry(); // compiles!
```

This breaks the general Java rule: For each reference variable, the **dynamic type** (type of the object referred by it) must be a **subtype** of the **static one** (type of declaration).
Java’s design choices

(1) Apple[] apples = new Apple[1];
(2) Fruit[] fruits = apples;  // ok, covariance
(3) fruits[0] = new Strawberry();  // compiles!

• The dynamic type of an array is known at runtime
  – During execution the JVM knows that the array bound to 
    fruits is of type Apple[] (or better [LApple; in JVM 
    type syntax)

• Every array update includes a run-time check

• Assigning to an array element an object of a non-
  compatible type throws an 
  ArrayStoreException
  – Line (3) above throws an exception
Recalling "Type erasure"

All type parameters of generic types are transformed to **Object** or **to their first bound** after compilation

- Main Reason: backward compatibility with legacy code
- Thus at run-time, all the instances of the same generic type have the same type

```java
List<String> lst1 = new ArrayList<String>();
List<Integer> lst2 = new ArrayList<Integer>();
lst1.getClass() == lst2.getClass() // true
```
Array covariance and generics

- Every Java array-update includes run-time check, but
- Generic types are not present at runtime due to *type erasure*, thus
- **Arrays of generics are not supported in Java**
- In fact they would cause type errors not detectable at runtime, breaking Java strong type safety

```java
List<String>[] lsa = new List<String>[10]; // illegal
Object[] oa = lsa;    // OK by covariance of arrays
List<Integer> li = new ArrayList<Integer>();
li.add(new Integer(3));
oa[0] = li;  // should throw ArrayStoreException,
             // but JVM only sees “oa[0]:List = li:ArrayList”
String s = lsa[0].get(0); // type error !!
Wildcards for covariance

- Invariance of generic classes is restrictive
- Wildcards can alleviate the problem
- What is a “general enough” type for `addAll`?

```java
interface Set<E> {
    // Adds to this all elements of c
    // (not already in this)
    void addAll(?? c);
}
```

- `void addAll(Set<E> c) // and List<E>?`
- `void addAll(Collection<E> c)`
  // and collections of T <: E?
- `void addAll(Collection<? extends E> c); // ok`
Wildcards, for both co- and contra-variance

- wildcard = anonymous variable
  - `?` Unknown type
  - Wildcard are used when a type is used exactly once, and the name is unknown
  - They are used for use-site variance (not declaration-site variance)

- Syntax of wildcards:
  - `? extends Type`, denotes an unknown subtype of `Type`
  - `?`, shorthand for `? extends Object`
  - `? super Type`, denotes an unknown supertype of `Type`
The “PECS principle”: 
Producer Extends, Consumer Super

When should wildcards be used?

– Use `? extends T` when you want to get values (from a producer): supports **covariance**

– Use `? super T` when you want to insert values (in a consumer): supports **contravariance**

– Do not use `?` (`T` is enough) when you both obtain and produce values.

Example:
```
<T> void copy(List<? super T> dst,
    List<? extends T> src);
```
What about type safety?

• Arrays covariance:

Apple[] apples = new Apple[1];
Fruit[] fruits = apples;
fruits[0] = new Strawberry();
// JVM throws ArrayStoreException

• Covariance with wildcards:

List<Apple> apples = new ArrayList<Apple>();
List<? extends Fruit> fruits = apples;
fruits.add(new Strawberry());
// compile-time error!!!
The price to pay with wildcards

• A wildcard type is anonymous/unknown, and almost nothing can be done:

```
List<Apple> apples = new ArrayList<Apple>();
List<? extends Fruit> fruits = apples;  //covariance
fruits.add(new Strawberry());       // compile-time error!  OK
Fruits f = fruits.get(0);            // OK
fruits.add(new Apple());            // compile-time error???
fruits.add(null);                    //ok, the only thing you can add 😃
```

```
List<Fruit> fruits = new ArrayList<Fruits>();
List<? super Apples> apples = fruits;  //contravariance
apples.add(new Apple());             // OK
apples.add(new FujiApple());         // OK
apples.add(new Fruit());             // compile-time error, OK
Fruits f = apples.get(0);            // compile-time error???
Object o = apples.get(0);            //ok, the only way to get
```
Limitations of Java Generics

Mostly due to "Type Erasure":

• Cannot Instantiate Generic Types with Primitive Types
  `ArrayList<int> a = ... //does not compile`

• Cannot Create Instances of Type Parameters

• Cannot Declare Static Fields Whose Types are Type Parameters
  `public class C<T>{ public static T local; ...}`

• Cannot Use casts or instanceof With Parameterized Types
  `(list instanceof ArrayList<Integer>) // does not compile`
  `(list instanceof ArrayList<?>) // ok`

• Cannot Create Arrays of Parameterized Types

• Cannot Create, Catch, or Throw Objects of Parameterized Types

• Cannot Overload a Method Where the Formal Parameter Types of Each Overload Erase to the Same Raw Type
  `public class Example { // does not compile`
  `public void print(Set<String> strSet) { }`
  `public void print(Set<Integer> intSet) { } }`