The Good, Bad and Ugly of Java Generics

Venkat Subramaniam

venkats@agiledeveloper.com

http://www.agiledeveloper.com/download.aspx

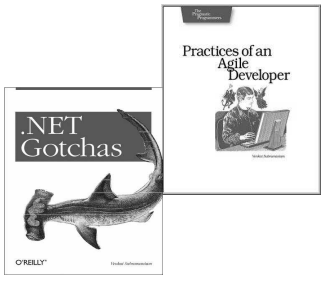
Abstract

Abstract Java introduced Generics in the 1.5 version (Java 5). What are the capabilities of Generics? How do you use it? Are there some gotchas in using it? In this example driven presentation, we will start at the basics of generics and look at its capabilities. We will then look at some of the under the hood details on generics implementation. We will then delve into the details of some of the changes to Java libraries to accommodate generics. Finally we will take a look at some restrictions and pitfalls that we need to be familiar with when it comes to practical and prudent use of generics.

About the Speaker *Dr. Venkat Subramaniam*, founder of Agile Developer, Inc., has trained and mentored thousands of software developers in the US, Canada and Europe. He has significant experience in architecture, design, and development of software applications. Venkat helps his clients effectively apply and succeed with agile practices on their software projects, and speaks frequently at conferences.

He is also an adjunct faculty at the University of Houston (where he received the 2004 CS department teaching excellence award) and teaches the professional software developer series at Rice University School of continuing studies.

Venkat has been a frequent speaker at No Fluff Just Stuff Software Symposium since Summer 2002.



The Good, Bad & Ugly of Java Generics

- **Need for Generics**
- Generics in Java
- Bounded Parameters
- Wildcard
- Restrictions
- Generics Implementation
- Effect of Erasure
- Java libraries changes
- Conclusion

Generics

- Remember the good old Templates in C++?
- Java went the route of using Object as generic type
  - Problem is when you pull some thing out of a collection, how do you call methods on it?
  - Only after casting it to the correct type right
  - Much worst if you are dealing with primitive types
    - These have to be boxed and unboxed
- Having collections that are type safe will eliminate this issue
  - Back to what C++ originally provided ☺

Need?!

- This is highly debatable
- First question is do we really need a type safe language
  - What about dynamically typed languages
- If we used dynamically typed languages, then we do not really care about generics!
- But then, we are talking about Java here
- So, how do we solve the issues with such a strongly typed language

The Good, Bad & Ugly of Java Generics

- Need for Generics
- **Generics in Java**
- Bounded Parameters
- Wildcard
- Restrictions
- Generics Implementation
- Effect of Erasure
- Java libraries changes
- Conclusion

Generics Type Safety

- Started as an experimental language – GJ
  - http://homepages.inf.ed.ac.uk/wadler/gj
  - Their slogan:
    "Making Java easier to type and easier to type"
- Java 1.5 (Java 5) provides support for Generics
  - I didn't say JVM supports Generics ☹
- Collections have been parameterized
- Simply create an instance of a Generic Collection class and use it like you would any other class
  - ArrayList<Integer> lst =
    new ArrayList<Integer>(); …

```
for(Integer val : lst)
{ total += val; }
```

Naming Convention

- Use single letter variables for types
  - Assuming you follow good coding practice, this will eliminate any confusion with your real classes ☺
- Use E for collection Elements
- T or other upper case single letters for general types

Generics Classes

- There is no instantiation of a class for each Generics parameter
- Vector<Integer> and Vector<Double> use the same class under the hood
- What is the consequence of this?
- Don't be fooled by **static** variables
  - you are not really dealing with different types under the hood

Liskov's Substitution Principle Honored

- ArrayList<Base> lst
  = new ArrayList<Derived>();
  //**ERROR**
- Collection of derived *is-not* a collection of base
- That is good, but why?
- If allowed, you may do the following
- lst.add(new Base()); into the ArrayList<Derived>

Generic Methods

- Methods may be parameterized as well
- Useful to express dependency among different parameters and the return type as well
- Use caution when mixing types, however
- Bends over back to accommodate to least common type, if possible

```
public static <T> void filter(
        Collection<T> in, Collection<T> out)
```

What if you send non-generic ArrayList as first argument?

The Good, Bad & Ugly of Java Generics

- Need for Generics
- Generics in Java
- **Bounded Parameters**
- Wildcard
- Restrictions
- Generics Implementation
- Effect of Erasure
- Java libraries changes
- Conclusion

Unbounded Type Parameters

- If you write a Generic like MyClass<T>, there is no restriction on what type argument may be used to instantiate it
- Well, almost. Java Generics do not allow instantiation with primitive types
  - Vector<int> is not allowed
  - We will learn *why?* later
- What if you want to expect a specific method to be available on the type?
  - You can constraint or bound the type your generic will accept

Constraints on Generics

- As an author of a Generic type, you can place restrictions on its usage
- For instance, you can ask the type to be of certain class or its sub class, or its super class.
- This is done using upper bound or lower bound constraint

```
public static <T extends Comparable>
        T max(T obj1, T obj2)
```

Types of Constraints

- Upper bound
  - T extends TypeName
  - Expects the type specified to instantiate Generics is of given type name or its sub class
- Lower bound
  - ? super TypeName
  - Expects the type specified to instantiate Generics is of given type name or its super class copy(listOfDerived, listOfBase);
- Multi-bound
  - T extends ClassTypeName1 & InterfaceType2 & InterfaceType3

The Good, Bad & Ugly of Java Generics

- Need for Generics
- Generics in Java
- Bounded Parameters
- **Wildcard**
- Restrictions
- Generics Implementation
- Effect of Erasure
- Java libraries changes
- Conclusion

Treating different Generics

- Each Generic type is separated from each other by the compiler
- What if you want to write a common method to work with different instantiation of a Generic class?
- For example, you want a method to use either a Vector<Integer> or a Vector<Double>
  - This is where a wildcard comes in

Wildcard

- Wildcard (?) gives you the capability to specify an ***unknown*** type
- This is a double edge sword
- It can help you treat different Generics instantiation as one
  - but may also lead to type safety issues
- One restriction – you can't write into a collection

```
abstract class Animal {
   public void playWith(
      Collection<Animal> playGroup) …
```

How to allow aDog.playWith(dogs); ???

but avoid aDog.playWith(books); ???

Wildcard capture

- You are not allowed to assign a unknown type to a specific type
- However, there are times when this may be useful
- Compiler makes an exception in these cases
- Wildcard capture allows compiler to infer the unknown type of a wildcard as a type argument to a generic method

```
public static <T extends Comparable>
   void print(Collection<T> coll)
      ...calling print with Collection<?>
```

Unchecked Warning

- This warning says that the compiler is not able to verify type safety
- Typically you would see this when you mix generic with non-generic code
  - Mixing Generics and non-generics code is dangerous
- Pay close attention to these warnings
- Best to treat warnings as errors
  - Better safe than sorry

Assigning generic to non-generic

Assigning non-generic to generic

The Good, Bad & Ugly of Java Generics

- Need for Generics
- Generics in Java
- Bounded Parameters
- Wildcard
- **Restrictions**
- Generics Implementation
- Effect of Erasure
- Java libraries changes
- Conclusion

Restrictions

- Array of collection of generics not allowed
- Array of collection of wildcard is allowed, but unsafe
- Can't use generic of primitive types
- Parameterized types not allowed for
  - static fields
  - static methods with type parameters
- Within a generics code you can't instantiate objects of that type using new as in new T[100]; or new T();
  - Why?
  - Let's understand how Generics are implemented to see why this restriction exists

Restrictions…

- Exceptions can't be generic (though you may throw generic exceptions)
- You can't inherit from a Parametric type though you may inherit from a Generic type
- You can't inherit from two instantiations of the same generic type

The Good, Bad & Ugly of Java Generics

- Need for Generics
- Generics in Java
- Bounded Parameters
- Wildcard
- Restrictions
- **Generics Implementation**
- Effect of Erasure
- Java libraries changes
- Conclusion

Implementation

- Generics are Java features
- No change to JVM was made for this
  - Advantage: Format of class file (byte code) not affected
  - Disadvantage: Not really that type safe!
- What does that mean?
- Why do we care?
  - Let's answer this first
  - Other languages are (slowly) coming to JVM
  - Groovy, for instance
  - But, of course, Groovy is a dynamic language, so it should not be an issue
  - What if we bring another type safe language to JVM?
  - Second, the type safety is only at the compiler level
    - Allows you to shoot yourself in the foot at runtime

type erasure

- Very fancy name
  - translated to English: **Generics are Macros**
- Generics information is present only at compile time
  - Compile and its forgotten! (erased that is)
  - Positive: No code bloat
  - Negative: Only compile time type safety
- What is erasure?
  - The generic type is replaced by a non-generic type during compilation time
  - Mostly with Object, but could be something else

Lacks type safety when treated as non-generic

type erasure…

- According to Java doc: ”The main advantage of this approach is that it provides total interoperability between generic code and legacy code that uses non-parameterized types (which are technically known as *raw* types). The main disadvantages are that parameter type information is not available at run time, and that automatically generated casts may fail when interoperating with ill-behaved legacy code. There is, however, a way to achieve guaranteed run-time type safety for generic collections even when interoperating with ill-behaved legacy code.”

Collections Wrapper Classes

- Collections outfitted with checked collection Wrapper classes
- These will make sure you are sending proper type

```
Collections.checkedCollection(col,
        Integer.class)
```

- Returns a Collection that will immediately throw an exception if a wrong type is added

The Good, Bad & Ugly of Java Generics

- Need for Generics
- Generics in Java
- Bounded Parameters
- Wildcard
- Restrictions
- Generics Implementation
- **Effect of Erasure**
- Java libraries changes
- Conclusion

Effect of erasure

- What is the type in the bytecode for:

- class MyList<T>
  - Object
- class MyList<T extends Vehicle>
  - Vehicle
- class MyList<T extends Comparable>
  - Comparable
- class MyList<T extends Object & Comparable>
  - Object
  - This is one reason to use Multi-bound constraints

Use javap –c to see what erasure actually did

Effect of Erasure

- Pre-erasure

```
ArrayList<Integer> lst
    = new ArrayList<Integer>();

lst.add(new Integer(1));
Integer val = lst.get(0);
```

- Erasure says

```
ArrayList lst = new ArrayList();
lst.add(new Integer(1));
Integer val = (Integer) lst.get(0);
```


Type comparison?

- List lst = new ArrayList<Integer>();
- lst instance of List
  - true
- lst instanceof ArrayList
  - true
- lst instance of ArrayList<Integer>
  - Compilation Error
- lst instanceof ArrayList<Double>
  - Compilation Error
- new ArrayList<Integer().getClass() == new ArrayList<Double>.getClass()
  - true

Type safety or lack of it

- ArrayList<Integer> aLst =
  new ArrayList<Integer>();
  List lst = aLst;
- aLst.add(new Double()); // ERROR
- lst.add(new Double()); // Oops!

Converting to Generics

- What if you want to convert legacy code to Generics
- Make sure the code after erasure is equivalent to non-generics
- Merely placing a <T> is not enough!

The Good, Bad & Ugly of Java Generics

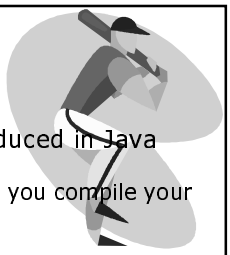
- Need for Generics
- Generics in Java
- Bounded Parameters
- Wildcard
- Restrictions
- Generics Implementation
- Effect of Erasure
- **Java libraries changes**
- Conclusion

Java Libraries Changes

- Java libraries have been changed to accommodate Generics

- Collections
- Reflection
  - java.lang.ref.WeakReference
- Lang
  - java.lang.ThreadLocal
  - java.lang.Class

Quiz Time



1. Generics are essential and it is high time they were introduced in Java
   false
2. A class is "written" in bytecode for each type of parameter when you compile your code
   No
3. To create an object of the parameterized type T, you simply write new T();
   No
4. How can you create an object of parameterized type, then?
   Pass a Class instance and use newInstance method
5. How do you specify that the type must implement Cloneable interface.
   T extends Cloneable
6. Is MyList<T> extends T valid?
   Nope
7. Is MyList2<T> extends MyList1<Integer> valid?
   Yes
8. The technique of ____ replaces parameterized type with a type during compilation
   erasure
9. Mention at least two problems with Generics in Java
   Dangerous to mix generics & non-generics, lacks runtime type-safety
10. Should you still have to learn and use Generics?
    You have no choice. It is being used in library and by others

The Good, Bad & Ugly of Java Generics

- Need for Generics
- Generics in Java
- Bounded Parameters
- Wildcard
- Restrictions
- Generics Implementation
- Effect of Erasure
- Java libraries changes
- **Conclusion**

Conclusion

- Good
  - Generics provide
    - improved type safety
    - better readability
  - Simpler to use
  - Nice job on bounded types
- Bad
  - Mixing Generic code with non-generic code is dangerous
  - Strongly advice to treat warnings as errors
  - Primitive types can't be used as parameters
  - No parameterized static variables
- Ugly
  - Still runtime overhead due to casting (remember erasure)
  - You can't rely on parameterized types when using reflection/meta programming
  - Erasure can lead to quite a bit of confusion

References

1. http://java.sun.com/j2se/1.5.0/download.jsp
2. JSR 14 http://jcp.org/en/jsr/detail?id=14
3. GJ http://homepages.inf.ed.ac.uk/wadler/gj
4. Examples, slides are for your download at **http://www.agiledeveloper.com/download.aspx**