Implementing Java Generics

Print Article

Rough edges aside, see how the addition of Java Generics adds substantial expressive power to the Java programming language

by Klaus Kreft and Angelika Langer

Posted March 10, 2004

Editor's Note: *This is the second of two installments presenting an overview of Java Generics, a new language feature that will be supported in the upcoming release of Java 2 Platform, Standard Edition 1.5. The first installment—"Language Features of Java Generics," Java Pro Online, March 3, 2004—discussed Java collections and generic treatment of types and classes. This concluding installment will look at generic treatment of methods as well as implementation and use of Java Generics.*

Types aren't the only components that can be parameterized. In addition to generic classes and interfaces, we can define generic methods. Static and nonstatic methods as well as constructors can be parameterized in pretty much the same way as we described parameterizing types previously (see "Language Features of Java Generics," *Java Pro* Online, March 3, 2004). The syntax is a little different, as you'll see. Everything said about type variables of parameterized types applies to type variables of parameterized methods in the exact same way. See Listing 1 for an example of a parameterized static method: max().

Parameterized methods are invoked like regular nongeneric methods. The type parameters are inferred from the invocation context. In our example, the compiler would invoke max<Byte>() automatically. The type inference algorithm is significantly more complex than this simple example suggests, and exhaustive coverage of type inference is beyond the scope of this discussion.

For the sake of completeness let's touch briefly on wildcards. So far we have been instantiating parameterized types using a concrete type that replaces the type parameter in the instantiation. In addition, so-called wildcards can be used to instantiate a parameterized type. A wildcard instantiation looks like this:

```
List<? extends Number> ref =
  new LinkedList<Integer>();
```

In this statement List<? extends Number> is a wildcard instantiation, while LinkedList<Integer> is a regular instantiation. There are three types of wildcards: "? extends Type", "? super Type", and "?". Each wildcard denotes a family of types. The "? extends Number" wildcard, for instance, is the family of subtypes of type Number; "? super Integer" is the family of supertypes of type Integer; and "?" is the set of all types. Correspondingly, the wildcard instantiation of a parameterized type stands for a set of instantiations; for example, List<? extends Number> refers to the set of instantiations of List for types that are subtypes of Number.

Wildcard instantiations can be used for declaration of reference variables, but they cannot be used for creation of objects. Reference variables of a wildcard instantiation type can refer to an object of a compatible type, however. Compatible in this sense are concrete instantiations from the family of instantiations denoted by the wildcard instantiation. In a way, this is similar to interfaces: we cannot create objects of an interface type, but a variable of an interface type can refer to an object of a compatible type, and by "compatible" we

mean a type that implements the interface.

Similarly, we cannot create objects of a wildcard instantiation type, but a variable of the wildcard instantiation type can refer to an object of a compatible type—"compatible" meaning a type from the corresponding family of instantiations. Access to an object through a reference variable of a wildcard instantiation type is restricted. Through a wildcard instantiation with "extends" we must not invoke methods that take arguments of the type that the wildcard stands for:

```
List list =
  new LinkedList<Integer>();
list.add(new Integer(25));
  // compile-time error
```

The add() method of type List takes an argument of the element type, which is the type parameter of the parameterized List type. Through a wildcard instantiation such as List<? extends Number> it is not permitted to invoke the add() method. Similar restrictions apply to wildcards with "super"; methods where the return type is the type that the wildcard stands for are prohibited. And both restrictions apply for reference variables with a "?" wildcard.

This brief overview of wildcard instantiations is far from comprehensive; exhaustive coverage of wildcards is beyond the scope of this discussion. In practice, wildcard instantiations will show up most frequently as argument or return types in method declarations, and only rarely in the declaration of variables. The most useful wildcard is the "extends" wildcard. Examples for the use of this wildcard can be found in the J2SE 1.5 platform libraries. One example is the method Boolean addAll(Collection<? extends ElementType> c) of class java.util.List. It allows the addition of elements to a List of element type ElementType, where the elements are taken from a collection of elements that are of a subtype of ElementType.

Now we have discussed all major language features related to Java generics: parameterized types, parameterized methods, bounded type parameters, and wildcard instantiations. Though there are many more details not covered here, we want to forge ahead by exploring some of the underlying principles of JG, in particular the translation of paramterized types and methods into Java byte code. While this sounds pretty technical and mainly like a compiler builder's concern, an understanding of these principles aids understanding of many of the less obvious effects related to JG.

Under the Hood

How is JG implemented? What does the Java compiler do with our Java source code that contains definitions and usages of parameterized types and methods? Well, as usual, the Java compiler translates the Java source code into Java byte code. Here we intend to take a look under the hood of the compilation process to understand the effects and side effects of JG.

A compiler that must translate a parameterized type or method (in any language) has in principle two choices:

- *Code specialization* – The compiler generates a new representation for every instantiation of a parameterized type or method. For instance, the compiler would generate code for a list of integers and additional, different code for a list of strings.
- *Code sharing* – The compiler generates code for only one representation of a parameterized type or method and maps all the concrete instantiations of the generic type or method to the one unique representation, performing type checks and type conversions where needed.

Code specialization is the approach that C++ takes for its templates. The C++ compiler generates executable code for every instantiation of a template. The downside of code specialization of generic types is its potential for code bloat. A list of integers and a list of strings would be represented in the executable code as two implementations of two different types.

This bloat is particularly wasteful in cases where the elements in a collection are references (or pointers) because all references (or pointers) are of the same size and internally have the same representation. There is no need for generation of mostly identical code for a list of references to integers and a list of references to strings. Both lists could be represented internally by a list of references to any type of object. The compiler just has to add a couple of casts whenever these references are passed in and out of the generic type or method. Since in Java most types are reference types, it seems natural that Java chooses code sharing as its technique for translation of generic types and methods. (C#, by the way, uses both translation techniques for its generic types: code specialization for the value types and code sharing for the reference types.)

One downside of code sharing is that it creates problems when primitive types are used as parameters of generic types or methods. Values of primitive type are of different size and require that different code is generated for a list of int and a list of double, for instance. It's not feasible to map both lists onto a single list implementation. There are two solutions to this problem:

- *No primitive types* – Primitive types are prohibited as type arguments of parameterized types or methods, that is, the compiler rejects instantiations such as a List.
- *Boxing* – Primitive type values are boxed automatically so that, internally, references to the boxed primitive value were used. (Boxing is the process of wrapping a primitive value into its corresponding reference type, and unboxing is the reverse [see Resources].) Naturally, boxing has a negative effect on performance because of the extra box and unbox operations.

Java Generics uses the first approach and restricts instantiation of generics to reference types, hence, a LinkedList<int> is illegal in Java. (In C++ and C# primitive types are allowed as type arguments because these languages use code specialization: in C++ for all instantiations and in C# at least for instantiations on primitive types).

Type Erasure

Let's turn to the details of the code sharing implementation of JG. The key question is: how exactly does the Java compiler map different instantiations of a parameterized type or method onto a single representation of the type or method?

The translation technique used by the Java compiler can be imagined as a translation from generic Java source code back into regular Java code. The translation technique is called *type erasure*; the compiler removes all occurrences of the type variables and replaces them by their leftmost bound or type Object, if no bound had been specified. For instance, the instantiations LinkedList<Integer> and a LinkedList<String> in a previous example (see Listing 2) would be translated into a LinkedList<Object>, or LinkedList for short, and the methods max<Integer>() and max<String>() (see Listing 1) would be translated to max<Comparable>(). In addition to removal of all type variables and replacing them by their leftmost bound, the compiler inserts a couple of casts in certain places and adds so-called bridge methods where needed.

Java designers deliberately chose the translation from generic Java code into regular Java code. One key requirement to all new language features in Java 1.5 is their compatibility with previous versions of Java. In particular, it is required that a pre-1.5 Java Virtual Machine must be capable of executing 1.5 Java code, which is achievable only if the bytecode resulting from a 1.5 Java source looks like regular byte code resulting from pre-1.5 Java code. Type erasure meets this requirement: after type erasure there is no difference anymore between a parameterized and a regular type or method.

For explanatory reasons we described the type erasure as a translation not from generic Java code into regular nongeneric Java code. However, this is not exactly true; the translation is from generic Java code directly to Java byte code. Nevertheless, we will subsequently refer to the type erasure process as a translation from generic Java to nongeneric Java.

Listing 3 illustrates the translation by type erasure and shows the type erasure of our previous example of generic types (see Listing 2). As you can see, all occurrences of the type variable A are replaced by type Object. The implementation of our generic collection is now exactly like an implementation that uses the traditional Java technique for genericity—namely, implementation in terms of Object references.

The sample code also gives you an example of an automatically inserted cast. In the main() method, where a linked list of strings is used, the compiler added a cast from Object to String. See Listing 4 for a type erasure of our parameterized max() method from Listing 1.

Again, all occurrences of type variables are replaced by either type Object (in the Comparable interface) or the leftmost bound (type Comparable in the max() method). We see the inserted cast from Object to Byte in the main() method, where the generic method is invoked for a collection of Bytes, and we see an example of a bridge method in class Byte.

The compiler inserts bridge methods in subclasses to ensure overriding works correctly. In the example, class Byte implements interface Comparable<Byte> and must therefore override the superinterface's compareTo() method. The compiler translates the compareTo() method of the generic interface Comparable<A> to a method that takes an Object, and translates the compareTo() method in class Byte to a method that takes a Byte. After this translation, method Byte.compareTo(Byte) is no overriding version of method Comparable<Byte>.compareTo(Object) any longer because the two methods have different signatures as a side effect of translation by erasure. To enable overriding, the compiler adds a bridge method to the subclass. The bridge method has the same signature as the superclass's method that must be overridden and delegates to the other methods in the derived class that was the result of translation by erasure.

Exploration

We've provided an overview of all major language features related to parameterized types and methods. Naturally, coverage of a fairly complex language feature such as JG could be given far more space. There are many more details to be explored and understood before JG can be used reliably and effectively. The greatest difficulties in using and understanding JG stem perhaps from the type erasure translation process by which the compiler elides all occurrences of the type parameters, which leads to quite a number of surprising effects. For example, arrays of parameterized types are prohibited in Java, that is, Comparable<String>[] is an illegal type, while Comparable[] is permitted.

This characteristic is surprising at best and turns out to be quite a nuisance in practice. It boils down to the fact that arrays are best avoided and replaced by collections as soon as the element type is a parameterized type. This realization and many other tips and techniques demand thorough exploration before the new language feature can be exploited to its capacity. Despite the rough edges here and there, the addition of JG adds substantial expressive power to the Java programming language. Our own experience is that once you've been using generics for a while, you'll miss them badly if you have to return to nongeneric Java with its unspecific types and countless casts and runtime checks.

About the Authors

Klaus Kreft is a senior consultant and software architect for Siemens Business Services. He has delivered consultancy services to large-scale industry projects for almost 20 years and used Java since 1995. Angelika Langer is a freelance trainer/consultant. She was involved in compiler construction in the '90s and has been watching closely the development of C++, Java, and C#. Her teaching is backed by over 12 years of work as a developer in the software industry and over 8 years of training and consulting. Angelika is also author of an independent curriculum of introductory and advanced C++ and Java courses. The authors are speakers at international conferences, they authored *Standard C++ IOStreams and Locales*, and they wrote numerous articles about C++ and Java, including columns for *C++ Report* and *JavaSpektrum*. Visit www.langer.camelot.de for further information. Contact Klaus at klaus.kreft@siemens.com, and contact

Angelika at langer@camelot.de.