The Case for Run-time Types in Generic Java

Eric Allen and Robert Cartwright

17th March 2002

Rice University
6100 Main St., MS-132
Houston TX 77005-1892
{eallen, cork}@rice.edu

Abstract

Since the advent of the Java Programming Language in 1995, many thoughtful proposals have been made for adding generic types to the Java programming language. Generic types are a glaring omission from the existing language, forcing programmers to use permissive, non-parametric type signatures for fields and methods of "naturally generic" classes (such as java.util.Vector) and repeatedly cast the results of operations on these classes to the more specific types. The JSR14 extension of Java proposed by Sun Microsystems (based on GJ) addresses this problem by adding generic types to the language, but prohibits operations that depend on run-time generic type information. This prohibition relegates generic types to "second-class" status where they are invisible at run-time, which is inconsistent with the status of types already in the language, including parametric arrray types. We have implemented a generalization of JSR14 called NextGen that supports the same syntactic extensions of Java as JSR14 yet eliminates the prohibition on operations that depend on run-time generic type information. In NextGen, generic types are "first-class": they can be used in exactly the same contexts as conventional types. The implementation of the NextGen compiler is derived from the same prototype GJ compiler as the JSR14 compiler and shares most of its desirable properties including a high degree of compatibility with legacy code. In this paper, we show through of a series of programming examples that first-class generic types play an important role in writing clean generic code.

1 Introduction

One of the most significant limitations of the Java Programming Language is its lack of support for generic types (*e.g.*, templates in C++, or parameterized classes in Eiffel). Without such types, the Java programmer is forced to work around this lack of expressiveness by relying on a clumsy idiom in which generic types are replaced with their upper bounds (typically Object) and casts are inserted when referring to specific uses of these types.

Sun Microsystem has announced its intention to incorporate generic types into Java. The recent release of the JSR14 compiler (adapted from Odersky's GJ compiler) demonstrates that generic types can be added to the language without sacrificing compatibilty with the JVM, or even with compiled binaries. However, the JSR14 compiler achieves these goals through the process of *type erasure,* where

No license: PDF produced by PStill (c) F. Siegert - http://www.this.net/~frank/pstill.html

generic types are replaced at compile-time with their upper bounds (a process quite similar to what Java programmers must do manually to work-around the lack of generic types). As a result, generic types and their instantiations have no representation at run-time, preventing run-time type operations such as **instanceof**, casts, and **new** operations of "naked" parametric type.[3] In short, generic types have "second-class" status in comparison with conventional types.

The NextGen programming language, as defined by Cartwright and Steele, overcomes this limitation by creating a separate class to represent each distinct instantiation of a generic type.[4] In essence, NextGen supports the same "Generic Java" language as JSR14, but eliminates the restrictions against using operations that depend on run-time generic type information. In NextGen, generic types are "first-class" and can be used anywhere that conventional types can. The authors have constructed a compiler for NextGen (adapted from Odersky's GJ compiler) that demonstrates that run-time generic types can be supported with little or no overhead in comparison with type erasure[2]. The NextGen compiler employs type erasure as an *optimization technique* to avoid code replication, but this optimization has no impact on the semantics of NextGen programs. In this paper, we will show through a series of short coding examples that run-time generic types matter because they enable Java programmers to write cleaner, more reliable code.

2 The Singleton Pattern

One of the most widely applicable design patterns in programming practice is the Singleton Pattern.[5] In this pattern, all instances of a class are indistinguishable, allowing them to be represented by a single instance which can be bound to a static field of the class. The only way to "generate" an instance of this class is to refer to the field (to enforce this practice, the class constructors can be declared private).

For example, consider the following class hierarchy for representing binary trees:

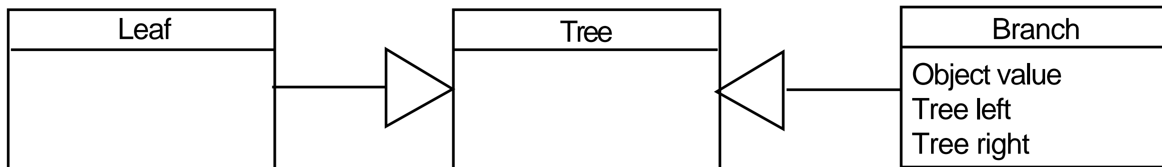

Figure 1. Simple Tree Class Hierarchy

Notice that class Leaf contains no fields. All instances of this class are functionally equivalent. Therefore it would wasteful to construct a new instance of this class every time a Leaf is needed. Instead, we can apply the Singleton Pattern and include an extra static field ONLY in class Leaf ("*static final*" is abbreviated with "*sf*"):

No license: PDF produced by PStill (c) F. Siegert - http://www.this.net/~frank/pstill.html

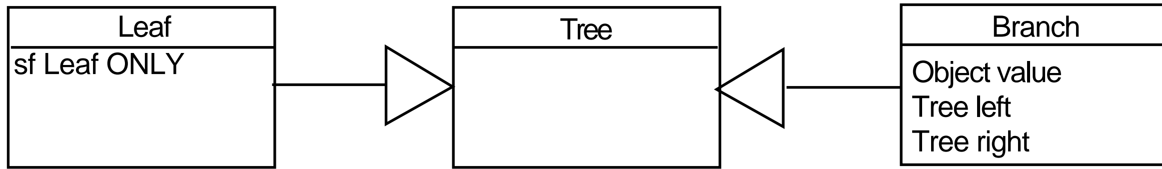

Figure 2. Leaf with Singleton Field

Now, this field can be referred to whenever a Leaf is needed. In the class definition, we would define the value of ONLY as

```
public static final Leaf ONLY = new Leaf();
```

If we express this class hierarchy in Generic Java, we obviously should parameterize the classes by the type of the elements contained in Branch nodes, as follows:

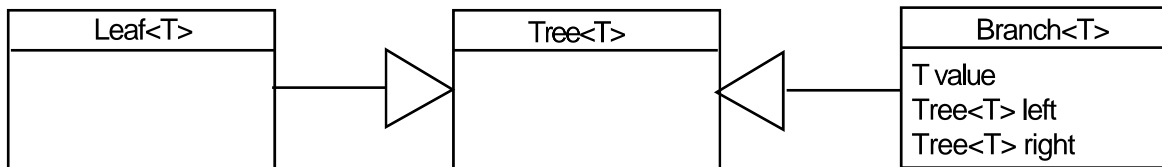

Figure 3. Parametric Trees

Now, consider the static field ONLY we added to class Leaf in the non-parametric class hierarchy. What should be the type of this field?

In the JSR14 extension of Java, a static field of generic class is shared across all instantiations of the class.[3] This design choice is dictated by the fact that JSR14 uses type erasure to implement generic types: a generic class is translated to a conventional class–called a *base class*–by converting all references to generic types to their upper bounds (which are conventional types). Hence, there can only be one static ONLY field for all of the instantiations of the generic type Leaf<T>, *e.g.* Leaf<Integer>, Leaf<String>, ... But such a static field shared across all instantiations cannot have type Leaf<T>, because no Java object can belong to type Leaf<T> for *all* types T. In generic Java, the types Leaf<Integer>, Leaf<String>, ... are all disjoint. For this reason, the JSR14 compiler prohibits the type parameters of a generic class from appearing within the type of a static field of the class.

The only way to support the Singleton Pattern in the JSR14 extension of Java is to declare a separate static field for each anticipated instantiation of a generic class as follows:

No license: PDF produced by PStill (c) F. Siegert - http://www.this.net/~frank/pstill.html

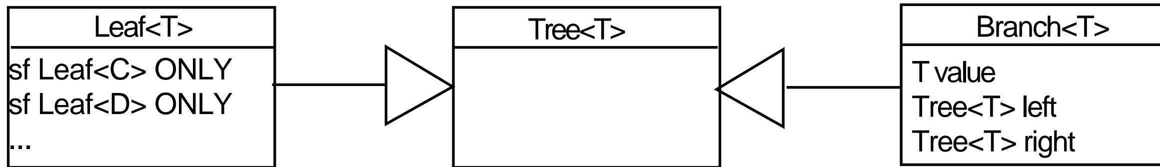

Figure 4. Parametric Trees with Multiple Singletons

This solution is clumsy, requires code duplication, and is not stable under program extension. If a program requires an instantiation of Leaf<T> for which no singleton ONLY field has been declared, the generic class Leaf<T> must be modified to include a new static field. In contrast, NextGen allows for the specification of *per-instantiation* static fields with generic types. [4] Because NextGen implements each instantiation of a generic type with a separate instantiation class, each instantiation class can contain its own copy of the generic field. In the case of class Leaf<T>, this allows us to apply the Singleton Pattern in the natural way:

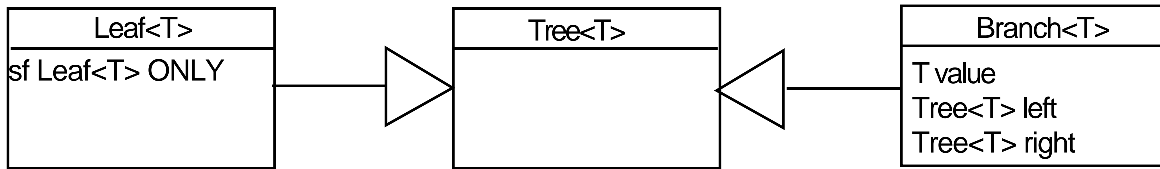

Figure 5. Parametric Trees with Parametric Singleton

The value of this field can be declared parametrically as follows:

```
public static final Leaf<T> ONLY = new Leaf<T>();
```

Each instantiation of class Leaf will then contain its own field ONLY of the appropriate type.

3 Casting and Catching Generic Types

In JSR14, casting to a generic type **C<E>** is prohibited unless parametric information **E** can be statically deduced by the type checker (enabling the compiler to implement the cast **(C<E>)** by its erasure **(C)**). Similarly, the declaration of generic exception types is prohibited because catch clauses cannot match generic types (only their erasures).

For a good illustration of the awkwardness resulting from the inability to dynamically confirm or query the generic type of an object, consider the task of implementing the **Cloneable** interface in a generic class. The method **Object clone()** is inherited by all classes from class **Object** but it throws an exception unless the invoking class implements the **Cloneable** interface. Of course, to make much use of the **clone()** method

No license: PDF produced by PStill (c) F. Siegert - http://www.this.net/~frank/pstill.html

for a class **C**, a client must cast the result to type **C**. If **C** is generic, then the cast must be to a generic type **C<E>**, which is prohibited in JSR14 unless the parametric type information **E** can be statically inferred (an uncommon occurrence in practice). In NextGen, all generic casts are legal so the **Cloneable** interface naturally applies to generic classes in exactly the same way it does to conventional classes.

4 Functional Mapping over Arrays

When working with arrays and other compound data structures, it is often useful to map a transformation over the constituent elements of the data structure, producing a new structure consisting of the resultant elements. Generic types allow us to perform such transformations with far less casting. For example, consider the following parametric Command interface:

```
public interface Mapper<A,B> {
  public B map(A element);
}
```

Suppose we wanted to apply a particular implementation of this Mapper interface to an array. In the process, we would like to create a new array of type **B[]** to hold the resulting elements. We could write the code to do this in NextGen as follows:

```
public class ArrayMapper {
  <A,B> public static B[] mapArray<A,B> aMapper) {
    B[] out = new B[in.length];
    for (int i = 0; i < in.length; i++) {
      out[i] = aMapper.map(in[i]);
    }
  }
}
```

But since JSR14 does not support run-time generic type operations such as **new B[]**, the preceding code is invalid in JSR14. Some other approach would be needed, such as passing an extra argument **dummy** of type **B[]** to the **map** method and using the static method invocation **Array.newinstance(dummy.getClass(), in.length)** to create the resulting array.

5 Legacy Classes and Interfaces

Any viable extension of Java with generic types must have an efficient implementation on top of the existing Java Virtual Machine and must interoperate with existing compiled binaries. JSR14 accomplishes these goals through the use of type erasure. Since generic types are erased at compile-time, they are identical at run-time to ordinary Java classes. Consequently, the JSR14 compiler can be fooled into compiling generic class references into references to existing Java classes *if* the class path is set during compilation to include generic "stub" classes with the same signatures as the erasures of the non-generic classes they spoof. This feature enables JSR14 to re-interpret existing "naturally generic" library classes or other binaries as classes with generic signatures.

No license: PDF produced by PStill (c) F. Siegert - http://www.this.net/~frank/pstill.html

In addition, JSR14 allows Generic Java code to refer directly to the base classes corresponding to generic classes. However, this feature breaks the type soundness of Generic Java: a JSR14 program can pass type-checking, yet generate a run-time type error (a CastException) at a point in the source program where there is no cast! [7, 3] The run-time error is produced by a cast automatically inserted by the JSR14 compiler. When this happens, how is the programmer supposed to diagnose what went wrong much less repair it?

In NextGen, there is a clear distinction between instances of a "naturally generic" legacy (non-generic) class and instances of a corresponding generic class. The latter can be defined as subclasses of the former, but they cannot be freely used in place of one another. To support the interoperation of code using a "naturally generic" legacy class and Generic Java code using a corresponding generic class, the programmer must write explicit conversion routines that convert legacy class instances to generic class instances and vice-versa[1] and perform the conversions whenever "naturally generic" data is passed between legacy code and new code.[2] This task superficially looks like extra work, but it provides scaffolding for checking that legacy (non-generic) data passed to a site requiring data of generic type actually satififies the invariant assocatiated with that generic type.

6 Transparent Debugging

In the JSR14 extension of Java, generic type information cannot be used in the program debugging process because it is erased by the compiler. Hence, at a breakpoint, a debugger can only report the erased types of data objects, not their generic types. Information that may be essential to efficiently diagnosing a bug is hidden from the programmer. For example, an empty **List<Integer>** will simply be reported as an empty list. In contrast, NextGen preserves all generic type information at run-time so a debugger can report the precise types of all data objects. The output of the debugger is completely consistent with the source code.

7 Conclusion

Generic types hold great promise for simplifying the application of many design patterns and coding practices in Java. But, as the preceding examples have shown, their full value cannot be realized without support for run-time generic types. If such operations that depend on run-time generic types are excluded from the language, programmers will find Java frustrating and awkward in many situations. Moreover, programmers who are not intimately familiar with the type erasure model for supporting generic types will be perplexed as to when they can or cannot use a generic type. Finally, as we have demonstrated in the last example, support for run-time type operations is critical for type-safe compatibility with Java legacy code and the transparent debugging of Generic Java code.

Since the NextGen compiler demonstrates that run-time generic type operations can be supported with little additional overhead, we strongly advocate their incorporation in a future version of Java.

[1]For naturally generic classes in the standard Java libraries, the NextGen extension of the libraries would obviously include the required conversion methods.

[2]If the new generic class is defined as a subclass of the old "naturally generic" class, conversion is required whenever old data is passed to a new context or new data is passed to an old context that mutates the data.

No license: PDF produced by PStill (c) F. Siegert - http://www.this.net/~frank/pstill.html

References

[1] Agesen, Freund, Mitchell. Adding Parameterized Types to Java. In ACM Symposium on Object-Oriented Programming, Systems, Languages, and Applications. 1997.

[2] Allen, Cartwright, Stoler. Efficient Implementation of Run-time Generic Types for Java Technical Report, Rice Computer Science Department, March 2002. Submitted to IFIP WG2.1 Working Conference on Generic Programming, 2002.

[3] Bracha, Odersky, Stoutamire, Wadler. Making the Future Safe for the Past: Adding Genericity to the Java Programming Language. In OOPSLA Proceedings. October 1998.

[4] Cartwright, Steele. Compatible Genericity with Run-time Types for the Java Programming Language. In Proceedings of the 13th ACM Conference on Object-Oriented Programming, Systems, and Applications. October 1998.

[5] Gamma, Helm, Johnson, Vlissides. Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley. Reading, Massachusetts. 1995.

[6] Gosling, Joy, Steele. The Java Language Specification. Addison-Wesley. Reading, Massachusetts. 1996.

[7] Igarashi, Pierce, Wadler. Featherweight Java: A Minimal Core Calculus for Java and GJ. In OOPSLA Proceedings, November, 1999.

No license: PDF produced by PStill (c) F. Siegert - http://www.this.net/~frank/pstill.html