

Subscribe  
 RSS / Newsletters  
 Digital Library  
 Log in  
 Go Parallel

Home Articles News Blogs Source Code Dobbs on DVD Dobbs TV Webinars & Events



## CHANNELS

Architecture & Design  
 C/C++  
 Database  
 Development Tools  
 Embedded Systems  
 High Performance Computing  
 Java  
 Jolt Awards  
 Mobility  
 Open Source  
 Security  
 Web Development  
 Windows/.NET  
 Visual Studio 2010

## INFO-LINK

Stay in step with Parallel computing trends and news. Go Parallel!



## Java Generics and C++ Templates

By Vladimir Batov, July 01, 2004

0 Comments

**Who says Java Generics are better than C++ templates? Well, someone did, and Vladimir examines both to find out for himself.**

*Vladimir Batov has been developing software for nuclear power stations, air traffic control, military radars, and many other things for nearly 25 years. He can be reached at [vladimir.batov@ca.com](mailto:vladimir.batov@ca.com).*

Time has shown that early decisions to keep the Java language simple by not including support for important programming concepts such as generic programming and operator overloading has not satisfied the requirements for modern general-purpose programming languages. Consequently, Java extensions in support of generic programming have been debated for several years. More recently, proposals for Java Generics [1] are close to being accepted and are included in the Java 2 SDK, Standard Edition (J2SE), Version 1.5.

I rank support for generic programming as one of the most important attributes of modern programming languages, particularly considering the precise and meticulous implementation of that mechanism in C++ templates. Not surprisingly, I was keen to learn what Java Generics had to offer, especially in terms of the numerous (direct and indirect) claims that Java Generics was better than C++ templates.

### A Closer Look At Java Generics

In the heart of the Java Generics implementation lies a technique called "type erasure" [1], which is largely adopted from the GJ Java extension [2]. The main characteristic of the technique is the elimination of the parameterization types during source-code translation to JVM bytecodes. For example, the following code illustrates how you might use type-safe Java Generics containers:

```
Vector<Integer> iv = new Vector<Integer>();
Vector<String> sv = new Vector<String>();

iv.add(new Integer(1));
```

## FREE Newsletter

- Timely news
- Colorful commentary
- Timeless technical features

Sign up today!



### Recent Articles

Survey Shows Unethical Behavior Rampant Inside IT Development Teams  
 Making Two-Way Tables in iOS  
 This Week's Developer Reading List  
 JVM Concurrency and Actors with GPar  
 Designing for Reliability



### Most Popular

Stories Blogs

Endless Projects: Where's the Fun Anymore?  
 Introduction to OpenMP  
 The Boost.Threads Library  
 QuickThread: A New C++ Multicore Library  
 Survey Shows Unethical Behavior Rampant Inside IT Development Teams

- Colorful commentary

- Timeless technical features

Dr.Dobb's  
ONLINE

```
sv.add(new String("abc"));

Integer ival = iv.get(0);
String  sval = sv.get(0);
```

During compilation, the code is rewritten by the Java Generics (pre-)compiler into familiar-looking code without any traces of parameterization types (**Integer** and **String**) in the parameterized type (**Vector<>**). Moreover, all instantiations of the parameterized class (**Vector<>**) are replaced with its nongeneric alternative (**Vector**):

```
Vector iv = new Vector();
Vector sv = new Vector();

iv.add(new Integer(1));
sv.add(new String("abc"));

Integer ival = (Integer) iv.get(0);
String  sval = (String) sv.get(0);
```

Essentially, the compiler does what Java programmers used to do by hand. However, the compiler does that job better and safer. That simple (almost preprocessor-style) technique provides a type-safe interface to the excessively polymorphic (and inherently type-unsafe) **Vector**. The compiler performs some additional, fairly sophisticated type checks to ensure proper cast match. Therefore, attempts to add, say, a **String** instance into **Vector<Integer>** will fail during compilation:

```
// Both lines are illegal.
sv.add(new Integer(1));
iv.add(new String("abc"));
```

Without Java Generics, bad-cast programming errors often lead to fatal runtime errors (since **ClassCastException** is an unchecked exception). With Java Generics, those programming errors are caught during compilation.

Admittedly, all this may not sound like a big deal to C++ programmers accustomed to C++ templates. However, for Java it is a serious step forward in the area of commercial software development. There is no need to elaborate on the difference and the impact of fixing a bug discovered at a customer site as opposed to the comfort of a developer's cubicle.

As all Java parameterization information is wiped out during compilation, "all parameterized types share the same class or interface at runtime" [1]. In the aforementioned example, instantiations of **Vector<Integer>** and **Vector<String>** are replaced with their erasure type **Vector** and unsurprisingly share the same implementation of **Vector** at runtime. A pleasant side effect is that, as William Grosso puts it, "the code bloat problem associated with C++ templates simply doesn't exist" [3]. Sure, the economical Java approach looks attractive, but everything comes at a price.

### Code Bloat? What Code Bloat?

Compared to C++, Java has a simple memory and object-access model. All objects are created exclusively on the heap and handled via Java references, which are essentially plain pointers with a few perks (or rather without). Therefore, say **Vector<Integer>** and **Vector<String>** are both containers of pointers and can safely share a single implementation (that happens to be **Vector**). The same technique—the C++ style—has been well described by Bjarne Stroustrup [4]. To avoid reciting the book, I'll briefly reiterate that:

```
template<class T>
class Vector {...}
```

[View All Videos](#)

### This month's Dr. Dobb's Journal



**This month**, in the May issue of Dr. Dobb's Journal, we focus on mobile development. Tom Thompson shows you how to create two-way tables in iOS; we offer a guide for Java programmers on how to quickly grasp the main features of Objective-C; and we help you determine whether your project really needs to be open, extensible, and standards-compliant. Plus, from our vault - ternary search trees, [and much more!](#)

[Download the latest issue today. >>](#)

### Most Recent Premium Content

#### Digital Issues

[Dr. Dobb's ALM Sourcebook](#)  
[Dr. Dobb's Journal February Digital Issue](#)  
[Dr. Dobb's Journal March Digital Issue](#)  
[Dr. Dobb's Journal April Digital Issue](#)

is a general declaration of a general container type;

```
template<>
class Vector<void*> {...}
```

is a specialization of **Vector** for **void** pointers and a single implementation for all **Vectors** of pointers; and

```
template<class T>
class Vector<T*> : private Vector<void*> {...}
```

is a partial specialization used for every pointer type.

**Vector<T\*>** is simply a type-safe interface to the common implementation of **Vector<void\*>**.

**Vector<T\*>** is implemented exclusively through derivation and inline expansion. It is routinely optimized away by the compiler without the dreaded "code bloat." If, in the unlikely case you are still wondering what it is all about, revisit section 13.4 of Stroustrup's book.

The described technique has been around for a while and the implementations of that technique by C++ and Java look similar on the surface. The difference is that Java only pretends to be generic, while C++ actually is. That last sentence sounds somewhat harsh and disrespectful towards Java. However, that is exactly how the proposed Java Generics extension is intended to work. That was deliberate to prevent any modifications of and to retain backward compatibility with the existing JVM.

### How Generic Is Java Generics Anyway?

Taking into account the constraints they faced, the Java Generics team has done an excellent job of making Java a better language. Java Generics adds only a modest level of complexity. This is well balanced by a more expressive and type-safe system that substantially reduces the negative impact of the grossly abused polymorphism of the single-rooted class hierarchy. Better yet, Java Generics is still improving. Notably, its latest (at the time of this writing) 1.2 prototype fixed the famous bug in the inference system ([http://fpl.cs.depaul.edu/ Problem.java](http://fpl.cs.depaul.edu/Problem.java)). However, there is much more to generic programming than type-safe containers. Consider:

```
bool test() // C++
{
    vector<int>    intv;
    vector<string> strv;

    return typeid(intv)
           == typeid(strv);
}

boolean test() // Java
{
    Vector<Integer> intv;
    Vector<String>  strv;

    return intv.getClass()
           == strv.getClass();
}
```

The C++ test correctly returns **false** and the Java test yields **true**. Similarly, if you had a **Vector<Integer>** and asked if it was an instance of **Vector<String>**, Java would tell you that it was. That trivial example demonstrates that parameterization in Java is literally "skin-deep" (or rather, compiler-deep). Without going into an elaborate analysis of Java Generics' specification, let me highlight some of the most notable limitations and niceties.

Java Generics does not allow primitives (**int**, **float**, **double**, and so on) as parameterization types. There are discussions that automatic conversion of primitives to and from their class-based equivalents (say, an **int** to/from an **Integer**) might resolve that problem. Although it works for containers, I do not see it working in general, as it will

require support for operator overloading as well. Consequently, you cannot have, for instance, a time-and-space efficient hash table with integers as keys and without the heap allocation of an int-to-Integer conversion—unless you don't have anything better to do than to write your own implementations for **IntHashTable**, **DoubleHashTable**, and so on. More so, for algorithms designed to work with primitives (such as **Complex** or **Rational**), you still have to duplicate your code for every type used.

Java Generics does not allow specialization of a generic class; that is, you cannot make special cases that apply to certain types. Specialization is important even if only for optimization purposes. Evidently, there is far more to specialization than that. An expectation that one implementation of an algorithm, function, or class can satisfy the myriad of parameterization types is naive at best.

Some widely used generic programming idioms (available in C++) are impossible in Java due to the absence of parameterization and parameterized types at runtime. The following snippets are not legal in Java Generics:

```
class Allocator<T>
{
    public T allocate()
    {
        return new T();
    }
}
class Foo<T>
{
    public void foo(T inst)
    {
        ...
        inst.do_something();
    }
}
```

To achieve this, you have to use a **Factory** model for the **Allocator** example and impose unnecessary polymorphism in the **Foo** example. Such workarounds significantly diminish the value of Java Generics for generic programming.

There is no RTTI mechanism for parameterized types. This is no surprise remembering that those types do not exist beyond the source code.

An exception mechanism for parameterized types is somewhat incomplete for the very same reason. Although parameterization types are allowed in throw clauses, they are not allowed in catch clauses. Again, you have to work around that restriction.

Unsafe semantics are allowed to enable interfacing with legacy code. The following assignment from **Vector** to **Vector<String>** is unsafe (since the vector has a different element type) but compiles with a warning:

```
Vector<Integer> intv;
Vector vector = intv;
Vector<String> strv = vector;
```

Life is full of compromises and this certainly looks like one of them. The only hesitation I have is, when I have to use legacy libraries, I will likely be confronted with a large volume of those warnings. From my experience, voluminous warnings of the same type are tiresome and eventually ignored. That creates a loophole where a few hard-to-find bad-cast errors can easily slip through.

Java Generics has a convenient **<T extends C>** construct to restrict **T** parameterization only for C-derived types. Although introduced out of necessity, this is programmer friendly and expressive. A C++ analogue might be:

```
template <T : C>
```

Unfortunately, C++ does not have it. C++ achieves this through implementation (rather than declaration like Java) and in a somewhat obscure way. The assignment line (implicit upcast) fails to compile if **T** is not derived from C:

```
template<class T>
void foo(T const& inst)
{ ...
  C const* test = &inst;
}
```

Java grammar was refined to allow two consecutive closing angle brackets, as in **Vector<Vector<String>>**, which is very thoughtful. Remembering all the clever stuff that C++ compilers have to perform, the need to have that annoying space between two closing brackets is a nuisance.

### Whose Code Bloat Is It Anyway?

The "code bloat" moniker refers to potential multiple instantiations of a function template for different argument parameterization types. It is fair to say that in the early days, C++ compilers met that potential in a grand style. Combined with a lack of template-based design experience, that resulted in astonishingly big executables. Modern C++ compilers are much better in that regard, particularly when the expansions can be inlined.

However, this is not the area where Java stands out compared to C++. In C++, function template instantiation can generate many versions of the function for each set of template arguments used. To achieve this (function overloading based on argument type) in Java, the programmer currently has to overload that function by hand, usually by copying code, for all argument types used. This is the very same duplication with the additional and laborious code maintenance requirement. Java Generics only partially addresses the issue for Java programmers.

In both languages, good design comes to the rescue. The standard programming technique to minimize duplication is to pull out common functionality into a separate, nonparameterized function or class and to provide a parameterized interface to that functionality, as demonstrated earlier in the article. C++ again goes one step further by allowing inline parameterized interfaces. Well-written C++ templates generally lead to code reduction at the cost of longer compilation time.

To the contrary, Java Generics often adds an extra method for every parameterization of a method. The following example is taken from [1]:

```
class C<A>
{
    abstract A id(A x);
}
class D extends C<String>
{
    String id(String x);
}
```

This will be translated (rewritten) to:

```
class C
{
    abstract Object id(Object x);
}
class D extends C
{
    String id(String x);
    Object id(Object x); // added.
}
```

### Conclusion

Despite their deceptively similar syntax, C++ and Java are two very different languages. Likewise, the original goals, programming models, and under-currents shaping their continued evolution are very different. Not surprisingly, their approaches to support generic programming are very different—"simple and comprehensible" [5] in Java and meticulous and comprehensive in C++.

Java Generics substantially improves type safety, encourages software reuse, and adds some basic support for generic programming. For Java, it is a serious step forward in the area of commercial software development. However, Java Generics in its proposed form does not qualify to be called generics (as in generic programming) as yet.

Although some generic programming concepts are difficult in C++, compared to Java, C++ is well advanced and Java has a lot to catch up with. Consequently, claims of Java Generics being "better than C++ templates" are naive and ungrounded at best.

Nevertheless, both languages have their applications and their devoted users. There is enough room for everyone.

### Acknowledgments

Thanks to Alan Eliassen and the many people who contributed their time and energy, shared their knowledge and views on the matter in `comp.lang.c++.moderated`, `comp.lang.java.programmer`, and Java Developer Connection forums, and provided many interesting insights.

### References

1. [1] Bracha, Gilad et al. "Adding Generics to the Java Programming Language: Participant Draft Specification" (<http://www.jcp.org/aboutJava/communityprocess/review/jsr014/index.html>).
2. [2] GJ Language Home Page (<http://www.research.avayalabs.com/user/wadler/pizza/gj/>).
3. [3] Grosso, William. "Generics and Method Objects" (<http://www.onjava.com/pub/a/onjava/2001/11/14/rmi3.html?page=1>).
4. [4] Stroustrup, Bjarne. *The C++ Programming Language*, Third Edition, Addison-Wesley, 2000.
5. [5] *Java Developer's Journal*. "An Interview with James Gosling" (<http://www.sys-con.com/java/javaone/interviews/gosling.html>).
6. [6] Tomlins, Alex and Chris Jackson. Java Generics: Final Report (<http://www.iis.ee.ic.ac.uk/~frank/surp99/report/caj97/index.html>).

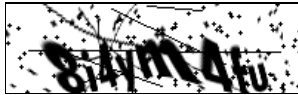
## Care to Comment?

**Subject (max length: 75):**

**Comments:**



Captcha:



Type the characters you see in the picture above.

## AROUND THE WEB

### Shrink Your JavaScript With Google's Source-to-Source Compiler

Google's Closure compiler speeds up your JavaScript by minimizing its size. It strips out comments, shrinks variable names, and eliminates extraneous JavaScript elements...while validating the syntax.

[Quick Read](#)

### Cross-Platform Mobile Toolkits Should Be Avoided

Martin Fowler explains why the usual limitations of cross-platform toolkits are greatly amplified on mobile platforms.

[Quick Read](#)

### Building and Graphing Binary Trees in Ruby

How to build a binary search tree in Ruby, traverse it, and generate a graph using HTML5's canvas — clearly and simply explained.

[Quick Read](#)

### Erlang vs. Java Memory Architecture

Erlang's actor model gives it a very different memory architecture from Java.

[Quick Read](#)

### Building a Queue With Two Stacks

How to use pair of stacks to build a queue in the same way they're created in functional languages.

[Quick Read](#)

### Mercurial vs. Git

One developer's experience using the two primary DVCSs.

[Quick Read](#)

# FREE Newsletter



*Enabling People and Organizations to Harness the Transformative Power of Technology*

**CIOs & IT Professionals**

[Black Hat](#)  
[BYTE](#)  
[Cloud Connect](#)  
[Dark Reading](#)  
[Enterprise 2.0](#)  
[Enterprise Connect](#)  
[Enterprise Efficiency](#)  
[HDI](#)  
[InformationWeek](#)  
[InformationWeek 500](#)  
[InformationWeek 500 Conference](#)  
[InformationWeek Analytics](#)  
[InformationWeek Events](#)  
[InformationWeek Global CIO](#)  
[InformationWeek Healthcare](#)  
[InformationWeek India](#)  
[InformationWeek SMB](#)  
[Interop](#)  
[Network Computing](#)  
[No Jitter](#)  
[Plug into the Cloud](#)  
[TechWeb.com](#)  
[The BrainYard](#)

**Software Developers**

[Dr. Dobbs](#)  
[Dr. Dobbs M-Dev](#)  
[Dr. Dobbs Digest](#)  
[Dr. Dobb's Update](#)  
[TechWeb.com](#)

**Web & Digital Professionals**

[Internet Evolution](#)  
[Web 2.0 Expo](#)  
[Web 2.0 Summit](#)  
[TechWeb.com](#)

**Government Officials**

[GTEC Ottawa](#)  
[InformationWeek Government](#)  
[TechWeb.com](#)

**Vertical Markets**

[Advanced Trading](#)  
[Bank Systems & Technology](#)  
[CreateYourNextCustomer](#)  
[InformationWeek Government](#)  
[InformationWeek Healthcare](#)  
[Insurance & Technology](#)  
[Light Reading / Telecom](#)  
[The CMO Site](#)  
[Wall Street & Technology](#)

**Game Industry Professionals**

[Gamasutra.com](#)  
[Game Developers Conference \(GDC\)](#)  
[Independent Games Festival](#)  
[Game Developer Magazine](#)  
[GDC Europe](#)  
[GDC China](#)  
[Game Career Guide](#)  
[Game Advertising Online](#)

**Global Communications  
Service Providers**

[Heavy Reading](#)  
[Heavy Reading Insiders](#)  
[Pyramid Research](#)  
[Light Reading](#)  
[Light Reading Mobile](#)  
[Light Reading Cable](#)  
[Light Reading Europe](#)  
[Light Reading Asia](#)  
[Ethernet Expo](#)  
[TelcoTV](#)  
[Tower Summit](#)  
[Light Reading Live & Virtual Events](#)  
[Webinars](#)

**Most Popular**

[Cable Catchup](#)  
[Cloud Connect Blog](#)  
[Digital Life](#)  
[Evil Bytes](#)  
[InformationWeek Analytics](#)  
[Interop Blog](#)  
[Monkey Bidness](#)  
[Over the Air](#)  
[Personal Tech](#)  
[The Philter](#)  
[Valley Wonk](#)

**UBM TechWeb Reader Services**

[About UBM TechWeb](#) [Advertising Contacts](#) [Technology Marketing Solutions](#) [Contact Us](#) [Feedback](#)  
[Reprints](#) [TechWeb Digital Library / White Papers](#) [TechWeb Events Calendar](#) [TechWeb.com](#)

[Terms of Service](#) | [Privacy Statement](#) | Copyright © 2011 UBM TechWeb, All rights reserved.