Diagnosing Java code: Java generics without the pain, Part 1

J2SE 1.5 -- code-named Tiger -- is scheduled for release near the end of 2003. I'm always in favor of gathering as much advance information on upcoming technology as possible, so this article is the first in a series on the new and reformatted features available in version 1.5. Specifically, I'd like to talk about generic types and highlight the changes and tweaks in Tiger designed to support them.

In many ways, Tiger promises to be the biggest leap forward in Java programming so far, including significant extensions to the source language syntax. The most visible change scheduled to occur in Tiger is the addition of generic types, as previewed in the JSR-14 prototype compiler (which you can download right now for free; see [Resources](http://www.ibm.com/developerworks/java/library/j-djc02113/index.html#resources)).

Let's start off with an introduction to what generic types are and what features are being added to support them.

**Casts and errors**

To understand why generic types are useful, we turn our attention to one of the most significant causes of bugs in the Java language -- the need to continually downcast expressions to datatypes more specific than their static types. (See "The Double Descent bug pattern," in [Resources](http://www.ibm.com/developerworks/java/library/j-djc02113/index.html#resources), for a discussion of some of the ways you can get into trouble with casts.)

Every downcast in a program is a potential hot spot for a ClassCastException and they should be avoided whenever possible. But they are often unavoidable in the Java language, even in very well-designed programs.

The most common reason to downcast in the Java language is that classes are often used in specialized ways that restrict the potential runtime types of arguments returned by method calls. For example, suppose we are adding and retrieving elements to and from a Hashtable. In a given program, the types of elements we use as keys, and the types of values we store in the hashtable, will not be arbitrary objects. Typically, all keys will be instances of a particular type. Similarly, the stored values will all share a common type more specific than Object.

But in the Java language versions that exist today, it is impossible to declare that the particular keys and elements of a hashtable have types more specific than Object. The type signatures on insertion and retrieval operations on hashtables tell us only that arbitrary objects are inserted and deleted. For example, the signatures of put and get operations are as follows:

|  |
| --- |
| class Hashtable {  Object put(Object key, Object value) {...}  Object get(Object key) {...}  ...  } |

Thus, when we retrieve an element from an instance of class Hashtable, even if we know that we haven't put anything into thatHashtable but, say, Strings, the type system will only know that the retrieved value is of type Object. Before we can do anything String-specific with that retrieved value, we have to cast it to a String, even when the retrieved element was added in the same code block!

|  |
| --- |
| import java.util.Hashtable;  class Test {  public static void main(String[] args) {  Hashtable h = new Hashtable();  h.put(new Integer(0), "value");  String s = (String)h.get(new Integer(0));  System.out.println(s);  }  } |

Notice the cast needed in the third line of the body of the main method. Because the Java type system is so weak, code tends to be riddled with casts like the one above. Not only do these casts make Java code wordier, they also diminish the value of static type checking (since each cast is a directive to selectively ignore static type checking). How can we extend the type system so that we don't have to circumvent it?

**Generic types to the rescue!**

A natural way to eliminate casts like the one above is to augment the Java type system with what are known as *generic types*. Generic types can be thought of as type "functions"; they are parameterized by type variables that can then be *instantiated* with various type arguments depending on context.

For example, instead of simply defining a class Hashtable, we could define a generic class Hashtable<Key, Value> in whichKey and Value are type parameters. The syntax for defining such generic classes in Tiger is just like that for ordinary class definitions, except that the class name is followed by a sequence of type parameter declarations enclosed in angle brackets. For example, we could define our own generic Hashtable class as follows:

|  |
| --- |
| class Hashtable<Key, Value> { ... } |

Then we can refer to these type parameters like we would ordinary types inside the body of the class definition, like this:

**Listing 4. Referencing type parameters like ordinary types**

|  |
| --- |
| class Hashtable<Key, Value> {  ...  Value put(Key k, Value v) {...}  Value get(Key k) {...}  } |

The scope of the type parameters is the body of the corresponding class definition, with the exception of static members. (In the next article, we'll discuss why a quirk of the Tiger implementation necessitates this restriction with static members. Stay tuned!)

When we create a new instance of a Hashtable, we have to pass type arguments to specify the types of Key and Value. How we do so depends on how we intend to use the Hashtable. In the example above, what we really wanted to do was to create an instance of a Hashtable that only mapped Integers to Strings. We could do so with our new Hashtable class:

|  |
| --- |
| import java.util.Hashtable;  class Test {  public static void main(String[] args) {  Hashtable<Integer, String> h = new Hashtable<Integer, String>();  h.put(new Integer(0), "value");  ...  }  } |

Now we don't need the cast anymore. Notice the syntax we've used to instantiate our generic class Hashtable. Just as the type parameters of a generic class are wrapped in angle brackets, the arguments of a generic type application are wrapped in angle brackets as well.

|  |
| --- |
| ...  String s = h.get("key");  System.out.println(s); |

Of course, it would be a significant amount of work for the programmer to have to redefine all of the standard utility classes -- such as Hashtable and List -- just to be able to use generic types. Luckily, Tiger provides users with generic versions of all of the Java collections classes, so we don't have to redefine them ourselves. What's more, these classes work seamlessly with both legacy code and new generic code (next month, we'll explain how that's possible).

**Tiger's "primitive" limitation**

One limitation to type variables in Tiger is that they must be instantiated with reference types -- primitive types won't work. So, in the example above, if we wanted to instead create a Hashtablemapping ints to Strings, we couldn't do it.

That's unfortunate, because it means that you have to wrap primitive types whenever you want to use them as arguments to a generic type. On the other hand, that's no worse than the current situation; you can't pass an int as a key to Hashtable because all keys must be of type Object.

What we'd really like to see would be automatic boxing and unboxing of primitive types, similar to what is done in C# (except better). Unfortunately, Tiger is not scheduled to include autoboxing of primitives (but one can always hope for Java 1.6!).

Good news! After this article was written, autoboxing was added to the Java 1.5 spec!

**Constrained generics**

Hashtable could be instantiated with any type arguments, which is what we'd like, but there are other classes where we will want to restrict the set of possible type arguments to subtypes of a given type *bound*.

For example, we may want to define a generic ScrollPane class that keeps a reference to an ordinary Pane that it decorates with scrolling functionality. The runtime type of the contained Pane will often be a subtype of class Pane, but the static type is simply Pane.

Sometimes we may want to retrieve the contained Pane with a getter, but we'd like the return type of the getter to be as specific as possible. We may want to add a type parameter MyPane to ScrollPane that can be instantiated with any subclass of Pane. Then we can place a bound on MyPane by annotating the declaration of MyPane with a clause of the form extends *Bound* :

|  |
| --- |
| class ScrollPane<MyPane extends Pane> { ... } |

Of course, we could simply leave off the explicit bound and just make sure that we never instantiate the type parameter with an inappropriate type.

Why bother putting bounds on type parameters? There are a couple of reasons. First of all, the bounds give us added static type checking. With it, we're guaranteed that every instantiation of the generic type adheres to the bounds we place on it.

Second, because we know that every instantiation of the type parameter is a subclass of the bound, we can safely call any methods on an instance of the type parameter that appear in the bound. If we place no explicit bound on the parameter, then by default the bound is Object, meaning that we can't call any methods on an instance of the bound that don't appear in classObject.

**Polymorphic methods**

In addition to parameterizing classes by type parameters, it is often useful to parameterize a method by type parameters as well. In generic Java programming parlance, methods parameterized by type are called *polymorphic methods*.

The reason polymorphic methods are useful is that sometimes there will be operations we want to perform where the type dependencies between the arguments and the return value are naturally generic, but the generic nature doesn't rely on any class-level type information and will change from method call to method call.

For example, suppose we want to add a factory method to a List class. This static method would take a single argument, intended to be the sole element of the List (until others are added). Because we'd like our Lists to be generic in the type of element they contain, we'd like our static factory method to take an argument of type variable T and return an instance ofList<T>.

But we'd really like this type variable T to be declared at the method level because it will change with every separate method call (also, as I will discuss in the next article, a quirk of the Tiger design dictates that static members are outside the scope of class-level type parameters). Tiger allows us to declare type parameters at the level of individual methods by prefixing them to method declarations. For example, we could do so for our factory method make as follows:

|  |
| --- |
| class Utilities {  <T extends Object> public static List<T> make(T first) {  return new List<T>(first);  }  } |

In addition to the added flexibility that polymorphic methods allow, there is an added benefit in Tiger. Tiger uses a type-inference mechanism to automatically infer the types of polymorphic methods based on the types of the arguments. This can greatly reduce the wordiness and complexity of a method call. For example, if we wanted to call our make method to construct a new instance of List<Integer> that contains an new Integer(0), we would simply write:

|  |
| --- |
| Utilities.make(Integer(0)) |

Then the type parameter instantiations would be inferred automatically from the method arguments.

**Generically speaking**

As we have seen, the addition of generic types to the Java language promises to greatly enhance our ability to leverage the static type system. Learning to use generic types is quite straightforward, but there are also pitfalls to be avoided. In the next few articles, we will discuss how to use to your advantage the particular incarnation of generic types that will appear in Tiger, as well as some of the pitfalls. We'll also examine the extensions to the generic Java type facilities that we can look forward to in versions of the Java platform still on the drawing boards.

# Diagnosing Java code: Java generics without the pain, Part 2

J2SE 1.5 -- code-named "Tiger" -- is scheduled for release near the end of 2003 and will include generic types (as previewed in the JSR-14 prototype compiler, available for download right now). In [Part 1](http://www.ibm.com/developerworks/java/library/j-djc02113.html), we discussed the basics of generic types and why they will be an important and much needed addition to the Java language. We also touched upon how the incarnation of generic types scheduled for Tiger includes several "kinks" that limit the contexts in which generic types can be used.

To help new programmers in their efforts to use generics effectively, I'll elaborate on exactly which usages of generic types are prohibited in Tiger and JSR-14, and I'll explain why the limitations are a necessary consequence of the implementation strategy used by JSR-14 (and consequently, Tiger) to compatibly implement generic types on the JVM.

**Limitations on generic types**

Let's start by reviewing the limitations on the use of generic types in Tiger and JSR-14:

* Enclosing type parameters should not be referred to inside static members.
* Generic type parameters can't be instantiated with primitive types.
* "Naked" type parameters can't be used in casts or instanceof operations.
* "Naked" type parameters can't be used in new operations.
* "Naked" type parameters can't be used in the implements or extends clauses of class definitions.

Why do these limitations exist? Because of the mechanism used by Tiger and JSR-14 to implement generic types on the JVM. Because the JVM doesn't provide any support for generic types, these compilers perform a "trick" to make it seem like support for generic types exists -- they type check all the code with the generic type information, but then "erase" all generic types and produce class files that include nothing but ordinary types.

For example, a generic type such as List<T> is erased to simplyList. "Naked" type parameters -- type parameters that appear alone rather than inside of a type, such as type parameter T in classList<T> -- are simply erased to their upper bounds (in the case of T, that would be Object).

This technique is extremely powerful; we get almost all of the increased precision of generic types, but we maintain compatibility with the JVM. In fact, we can even use non-generic legacy classes such as List with their generic counterparts (List<T>) interchangeably; both look the same at runtime.

Unfortunately, as the above limitations show, there is a price for this power. Erasing in this manner introduces holes in the type system that limit how we can safely use generic types.

To help clarify each limitation, we'll review an example of where it can occur. In this article, we'll discuss the first three limitations. The issues with the last two are so intricate that they need a more in-depth treatment, which we'll save for the next article.

**Enclosing type parameters in static members**

Referring to enclosing type parameters inside static methods and static inner classes is prohibited outright by the compiler. So, for instance, the following code is illegal in Tiger:

|  |
| --- |
| class C<T> {  static void m() {  T t;  }    static class D {  C<T> t;  }  } |

When this code is compiled, it generates two errors:

* An error for the illegal reference to T inside static method m
* An error for the illegal reference to T inside static class D

When defining static fields, things get more complicated. In both JSR-14 and Tiger, static fields in a generic class are shared across all instantiations of the class. Now in the JSR-14 compilers 1.0 and 1.2, if you refer to a type parameter in a static field declaration, the compiler doesn't complain but it should. The fact that the field is shared can easily lead to weird errors at runtime, such as a ClassCastException in code that doesn't include a cast.

For example, the following program will compile without warning under these versions of JSR-14:

|  |
| --- |
| class C<T> {  static T member;    C(T t) { member = t; }    T getMember() { return member; }    public static void main(String[] args) {  C<String> c = new C<String>("test");  System.out.println(c.getMember().toString());  new C<Integer>(new Integer(1));  System.out.println(c.getMember().toString());  }  } |

Notice that every time an instance of class C is allocated, the static field member is reset. What's more, the type of the object it is set to is dependent on the type of the instantiation of C! In the main method provided, the first instance, c, is of type C<String>. But the second is of type C<Integer>. Whenever the shared static field member is accessed from c, it is assumed that the type of member is String. However, after the second instance of type C<Integer> is allocated, member is of type Integer.

The result of running C's main method might surprise you -- it'll issue a ClassCastException! How can that be, since the source code doesn't include any casts? It turns out that the compiler actually inserts casts into the code during compilation to account for the fact that type erasure reduces the precision of the types of certain expressions. These casts are *supposed* to succeed, but in this case they don't.

This particular "feature" of JSR-14 1.0 and 1.2 should be considered a bug. It breaks the soundness of the type system, in other words, the fundamental contract that a type system should uphold with the programmer. It would be much better to simply prevent the programmer from referring to generic types in static fields, as is done in the case of static methods and classes.

Note that the problem with allowing such potentially explosive code is not that programmers could *intentionally* override the type system in their own code. The problem is that programmers could accidentally write such code (say by mistakenly including a static modifier in a field declaration, due to copy-and-pasting).

The type checker is supposed to help a programmer recover from exactly these sorts of mistakes, but in the case of static fields, the type system could actually help to confuse the programmer. How are we supposed to diagnose a bug like this one when the only error signaled is a ClassCastException in code that makes no use of casts? The situation is worse for a programmer who isn't aware of the implementation scheme used for generic types in Tiger and just assumes that the type system acts reasonably. In this case, it doesn't.

Luckily, the latest version of JSR-14 (1.3) outlaws the use of type parameters in static fields. Therefore, we can reasonably expect that they will be outlawed in static fields in Tiger as well.

**Generic type parameters and primitive types**

This restriction doesn't have the same potential pitfalls as what we just discussed, but it can make your code pretty wordy. For example, in the generic version of java.util.Hashtable there are two type parameters: one for the type of Keys and one for the type of Values. So, if we want a Hashtable mapping Strings to Strings, we can specify the new instance with the expression new Hashtable<String, String>(). However, if we want a Hashtable that maps Strings to ints, we have no choice but to create an instance of Hashtable<String, Integer> and wrap all int values in Integers.

Again, this aspect of Tiger follows naturally from the implementation scheme used. Since type parameters are erased to their bounds and the bounds can't be primitive types, there is no way that an instantiation with primitive types would make sense once the types are erased.

**"Naked" parameters in casts or instanceof operations**

Recall that by "naked" type parameters, we mean type parameters that lexically occur alone, not as a syntactic subcomponent of a larger type. For instance C<T> is not a naked type parameter, but (in the body of C), T is.

If you use casts or instanceof operations on naked type parameters inside your code, the compiler will issue what is called an "unchecked" warning. For example, the following code will generate the warning: Warning: unchecked cast to type T:

|  |
| --- |
| import java.util.Hashtable;  interface Registry {  public void register(Object o);  }  class C<T> implements Registry {  int counter = 0;  Hashtable<Integer, T> values;    public C() {  values = new Hashtable<Integer, T>();  }    public void register(Object o) {  values.put(new Integer(counter), (T)o);  counter++;  }  } |

You should take such warnings seriously because they indicate that your code could behave very strangely at runtime. In fact, they can make it extraordinarily difficult to diagnose bugs. In the previous code, we'd expect that if register("test") were called on an instance of C<JFrame>, a ClassCastException would be signaled. But it won't be; the computation will continue as if the cast had succeeded, signaling an error further into the computation or worse, completing with corrupt data but no outward signs of trouble. Similarly, instanceof checks on naked type parameters will result in an "unchecked" warning at compile time and the check will not occur as expected at runtime.

**A double-edged sword**

So, what's going on here? Because Tiger relies on type erasure, the naked type parameters in casts and instanceof tests are "erased" to their upper bounds (in the earlier case, that'll be type Object). So, casts to type parameters will turn into casts to the upper bound of the parameter.

Similarly, instanceof will check that the operand is an instanceof the bound of the parameter. That's not what we intended at all, and if it were, we would have simply cast to the bound explicitly. So, in general, avoid using casts and instanceof checks on type parameters.

Nevertheless, you will sometimes have to rely on casts to type parameters in order to get your code to compile. When that happens, just remember that, in that part of the code, there is no safety from type checking -- you're on your own.

Although generic types can be a powerful weapon for producing robust code, we've shown how their misuse can lead to code that is not only less robust but also extraordinarily hard to diagnose and fix. Next time, we'll cover the last two limitations of generic types in Tiger and discuss some of the issues that necessarily come up in any attempt to include them in a generic Java type system.

# Diagnosing Java code: Java generics without the pain, Part 3

In this installment of our series on the addition of generic types to Java programming, we'll consider one of the two limitations on the use of generics that we haven't discussed, namely the addition of support for new operations on "naked" type parameters (such as new T() in a class C<T>).

As I mentioned [last month](http://www.ibm.com/developerworks/java/library/j-djc03113.html), Tiger and JSR-14 implement generic types over the Java language using "type erasure." With type erasure, generic types are used for type checking only; afterwards, they are replaced with their upper bound. As is natural with this definition, erasure would interfere with expressions such as new T().

If the bound on T were, say Object, such an expression would be erased to new Object() and, regardless of how T was instantiated (String, List, URLClassLoader, and so on), the new operation would make a new instance of Object. Clearly, that's not what we want.

To add support for expressions such as new T(), as well as the other type-dependent operations we discussed last time such as casts and instanceof expressions, we'd have to adopt some implementation strategy other than type erasure (such as keeping a separate class for each generic instantiation). But in the case of new operations, there are other issues that must be addressed.

In particular, many fundamental language-design issues have to be decided upon to make such an addition to the Java language a reality.

**Valid constructor calls**

First off, in order to form a legal new expression on a type parameter, such as new T(), it's necessary to make sure that we're calling a constructor that is valid for every instantiation of T. But because all we know about T is that it's a subtype of its declared bound, we have no idea what constructors an instantiation of T will have. This problem could be handled in one of three ways:

1. Require all instantiations of a type parameter to include a zeroary constructor.
2. Throw an exception whenever a run-time instantiation of the generic class does not include a needed constructor.
3. Modify the language syntax to include more elaborate bounds on the type parameters.

**No. 1: Require a zeroary constructor**

We could simply require all instantiations of a type parameter to include a zeroary constructor. This solution has the advantage of being very simple. It also has a precedent.

Existing Java technologies, like JavaBeans technology, that deal with similar problems solve the problem by requiring a zeroary constructor. A big disadvantage to this approach, however, is that for many classes there is no reasonable zeroary constructor.

For instance, any class representing a non-empty container would naturally take arguments representing its elements in the constructor. Including a zeroary constructor would force us to first create an instance and only then do the initialization we would have done in the constructor call. But that practice can lead to problems (You might want to read the April 2002 installment of this column "The Run-on Initializer bug pattern" for details; see [Resources](http://www.ibm.com/developerworks/java/library/j-djc04093/index.html#resources).)

**No. 2: Throw an exception when needed constructor is missing**

Another way of handling this problem would be to throw an exception whenever a run-time instantiation of the generic class does not include a needed constructor. Notice that the exception must be thrown at run time. Because of the Java language's incremental compilation model, we can't statically determine all of the instantiations of a generic class that will occur at run time. For example, suppose we have a set of generic classes as follows:

**Listing 1. New operation on a "naked" type parameter**

|  |
| --- |
| class C<T> {  T makeT() {  return new T();  }  }  class D<S> {  C<S> makeC() {  return new C<S>();  }  } |

Now, in class D<S>, an instance of class C<S> is constructed. Then in the body of class C, a zeroary constructor on S will be called. Does such a zeroary constructor exist? The answer, of course, is that it depends on the instantiation of S!

If S is instantiated as, say, String, then the answer is yes. If it's instantiated as, say, Integer, then the answer is no. But when compiling classes D and C, we have no idea what instantiations of D<S> other classes may construct. Even if we had the whole program available for analysis (which we almost never have with Java programs), we would have to do a rather costly flow analysis to determine where potential problems with constructors could occur.

Furthermore, the sorts of errors we would get from this technique would be particularly hard for the programmer to diagnose and repair. For example, suppose the programmer is familiar only with the header of class D. He sees that the bound on D's type parameter is the default bound (Object). Given that information, he has no reason to believe that an instantiation of D that meets the declared type bound (such as D<Integer>) would cause an error. In fact, it may not cause an error for quite a long time, until finally someone calls method makeC, and (finally) calls method makeT on the instantiation of C. Then we will get a signaled error, but it will be long after the real problem occurred -- the bad instantiation of class D.

What's more, the stack trace of the signaled error may not even include any method calls on that faulty instance of D! Now let's suppose that the programmer doesn't have access to the source code for class C. He will be entirely in the dark as to what the problem is or how to fix his code unless he can manage to get hold of the maintainer of class C and get a clue.

**No. 3: Modify syntax for more elaborate bounds**

Another possibility is to modify the language syntax to include more elaborate bounds on the type parameters. These bounds could specify the set of available constructors that must be present in any instantiation of the parameter. Then, inside the generic class definition, the only constructors that could be called are those declared in the bound.

Also, client classes that instantiate our generic class must do so with classes that meet the declared constraint on what constructors exist. That way, the parameter declaration would serve as a contract between the class and its clients, and we could statically check that both obey the contract.

This approach has many advantages over the other two, allowing us to keep the expressiveness of the second approach and the same degree of static checking as in the first. But it also has problems to overcome.

For one thing, the type parameter declarations can easily become wordy. We'd probably need some form of syntactic sugar to make these augmented parameter declarations tolerable. Also, if augmented parameter declarations were added in a future version beyond Tiger, we'd have to ensure that these augmented declarations would be compatible with existing compiled generic classes.

If support for type-dependent operations on generic types is ever added to Java programming, it's not yet clear what form it would take. But from the perspective of which approach will keep Java code as robust as possible (and as easy to fix as possible when it does break), option three is definitely the way to go.

However, there is another problem with new expressions that is even more serious.

**Polymorphic recursion**

The more serious problem is the potential for *polymorphic recursion* in class definitions. Polymorphic recursion occurs when a generic class instantiates itself in its own body. For example, consider the following devious example:

**Listing 2. A self-referential generic class**

|  |
| --- |
| class C<T> {  public Object nest(int n) {  if (n == 0) return this;  else return new C<C<T>>().nest(n - 1);  }  } |

Suppose a client class creates a new instance of C<Object> and calls, say, nest(1000). Then in the process of executing method nest(), a new instantiation C<C<Object>> will be constructed and nest(999) will be called on it. Then an instantiationC<C<C<Object>>> will be constructed, and so on, until a thousand separate instantiations of class C are made. Of course, I choose the number 1000 arbitrarily; in general we will have no way of knowing what integers will be passed to method nest at run time. In fact, they could be passed in as user input.

Why is this a problem? Because if we support type-dependent operations on generic types by constructing a separate class for each instantiation, then we will have no way of knowing which classes we'll need to construct until the program is run. But how can that work if the class loader looks for existing class files for each class it loads?

Again, there are a couple of possibilities here:

1. Place an upper limit on the number of instantiations of a generic class that a program can make.
2. Statically forbid polymorphic recursion.
3. Construct new instantiation classes on demand while the program is running.

**No. 1: Place upper limit on instantiations**

We could put an upper limit on the number of instantiations of a generic class that a program can make. Then, during compilation, we can determine a finite bound on the set of legal instantiations and simply generate class files for all instantiations in this bound.

This approach is similar to what is done in the C++ standard template library (and that should give us reason to worry that it's not a good approach). The problem with this approach is that, like signaling errors for bad constructor calls, the programmer will have no way of predicting that a given run of his program will crash. For example, suppose the bound on the number of instantiations were 42 and the nest() method mentioned earlier was being called with a user-supplied argument. Then, so long as the user types numbers below 42, everything works. When the user types 43, our house of cards crashes around him. Now imagine the poor maintainer of the code who's given the job of putting the pieces back together and trying to find out what's so special about the magic number 42.

**No. 2: Statically forbid polymorphic recursion**

Why don't we issue a command to the compiler like "statically forbid polymorphic recursion." (Sigh. If only it were that simple.) Of course, many programmers, including myself, would object that such a policy would inhibit the use of many important design patterns.

For example, in a generic class List<T>, do you really want to prevent the construction of a List<List<T>>? Returning such a list from a method could be useful for building many very common data structures. In fact, it turns out that we couldn't prevent polymorphic recursion even if we wanted to. Just like static detection of bad generic constructor calls, forbidding polymorphic recursion conflicts with incremental class compilation. This fact may be obfuscated by our simple example earlier in which the polymorphic recursion occurs as a simple, direct self-reference. But in general, the self-reference may take arbitrary levels of indirection through a multitude of classes compiled at different times. Again, that's because one generic class may instantiate another with its own type parameters.

Here's an example involving polymorphic recursion across two classes:

**Listing 3. Mutually recursive polymorphic recursion**

|  |
| --- |
| class C<T> {  public Object potentialNest(int n) {  if (n == 0) return this;  else return new D<T>().nest(n - 1);  }  }  class D<S> {  public Object nest(int n) {  return new C<C<S>>().nest(n);  }  } |

There is no polymorphic recursion evident in either class C or D, but an expression like new D<C<Object>>().nest(1000) will cause 1000 instantiations of class C.

Potentially, we could add new attributes to class files indicating all the various generic type instantiations inside the class and then analyze these instantiations for recursion when compiling other classes. But still, we would have to provide the programmer with weird and counter-intuitive error messages.

In the code above, where would we signal an error? In the compilation of class D or in the compilation of a client class that included the meddlesome expression new D<C<Object>>().nest(1000)? Either way, unless the programmer has access to the source code for class C, he will have no way of predicting when compilation errors would occur.

**No. 3: Construct new instantiation classes on the fly**

Another approach is to construct new instantiation classes on demand while the program is running. At first, this approach may seem entirely incompatible with the Java runtime. But in fact, all that is necessary to implement this strategy is to use a modified class loader that constructs new instantiation classes from a "template" class file.

The JVM spec already allows programmers to use modified class loaders; in fact, they're used by many popular Java applications such as Ant, JUnit, and DrJava. The disadvantage of this approach is that the modified class loader would have to be distributed along with the application for it to run on older JVMs. Because class loaders tend to be small, this overhead should not be high.

Let's look at a working example of this approach.

**NextGen example: Modified class loader**

The previous approach -- to solve the problem of polymorphic recursion with a modified class loader that constructs generic type instantiations on demand -- is the approach adopted with the NextGen extension to the Java language. A modified class loader uses template files that look almost exactly like ordinary class files except for "holes" in the constant pool that are filled in at load time for each instantiation class. Non-generic classes are unaffected.

At the Rice University JavaPLT programming languages lab, we've recently released a prototype of the NextGen compiler, an extension of the GJ generic Java compiler that includes support for type-dependent operations (casts, instanceof tests, newexpressions) on type parameters. In this prototype implementation, we have employed a modified class loader to support polymorphic recursion. This prototype is available as a free download (see [Resources](http://www.ibm.com/developerworks/java/library/j-djc04093/index.html#resources)).

**Conclusion**

As the above considerations demonstrate, adding full-fledged run-time support to generic Java includes the solution to many subtle design issues. If these issues are not handled well, the benefits of generic types can easily be outweighed by decreases in expressiveness and robustness. Hopefully, Java programming will continue to develop in a direction that maintains a high degree of both of these attributes.

Next time, we'll finish up our discussion of generic types by discussing perhaps the most powerful way that generic types could be applied: to add mixins (classes with parametric parent type) to the language. We'll relate such a formulation of mixins to previous discussions of this powerful language feature, discussing the advantages and disadvantages of adding mixins through generic types.

# Diagnosing Java code: Java generics without the pain, Part 4

So far in this mini-series discussing generic types in JSR-14 and Tiger, we've covered:

* Generic types and the upcoming features designed to support them
* The limitations on primitive types, constrained generics, and polymorphic methods
* Several limitations imposed in these Java extensions
* How the limitations are necessitated by the implementation strategy used by the compilers of these extended languages
* The ramifications of adding support for new operations on "naked" type parameters in generic types

This month, we'll conclude our discussion of generic types in the Java language by covering the issues that need to be addressed before you can handle *mixins* -- perhaps the most powerful feature that generic types promise.

**Mixins versus wrapping**

Mixins are classes parameterized by their parent class. For example, consider the following generic class, which extends its own type parameter:

|  |
| --- |
| class Scrollable<T> extends T {...} |

The intention of class Scrollable is that it embeds the functionality necessary for adding scrollability to a GUI widget. Each application of this generic class would extend a distinct parent class. For example,Scrollable<JTextPane> would be a subclass of JTextPane andScrollable<JEditorPane> would be a subclass of JEditorPane. Contrast this way to embed functions with the current functionality in the Java Swing library in which a JComponent must be "wrapped" in a JScrollPane if we want to make it scrollable.

Not only does wrapping require adding forwarding methods to access the functionality of the wrapped class, it also prevents us from using the resulting scrollable object in contexts where an instance of the wrapped object is needed (for instance, we can't pass a JScrollPane to a method requiring an instance of JTextPane). By parameterizing Scrollable by its parent class, we are able to keep a single point of control for the functionality involved in scrolling while extending multiple superclasses. In this way, being able to use mixins gives us back some of the power of multiple inheritance but without the accompanying pathologies.

In the previous example, we could even put a constraint on a type parameter to prevent it from being used in inappropriate contexts. For instance, we might want to constrain the type parameter to be a subclass of JComponent:

|  |
| --- |
| class Scrollable<T extends JComponent> extends T {...} |

Then only GUI components could be extended by our mixin.

**Mixins and generic classes: A perfect match**

Often, mixins are added to a language as an independent language feature, as is done in Jam. But it is appealing, almost seductive, to incorporate mixins as part of a generic type system. The reason: both mixins and generic classes can be thought of as functions mapping existing classes to new classes.

Generic classes can be viewed as functions mapping their arguments to new instantiations. Mixins can be viewed as functions mapping existing classes to new subclasses. By incorporating mixins using generic types, we are able to work around many of the key limitations of other formulations of mixins.

In the Jam extension to the Java language, the type of the superclass of a mixin has no name; we simply can't refer to it in the body of the mixin. This limitation snowballs to include all sorts of other problems. For example, in Jam, the programmer is not allowed to pass this as an argument to a method; there is no way to type check such calls. That limitation is crippling because many of the most common design patterns rely on being able to pass this as an argument.

Consider the visitor pattern in which a visitor class is defined with a for method for each class in a composite hierarchy. Typically the class being visited includes an accept method that takes a visitor and calls a method on that visitor, passing inthis. Thus, in Jam, the visitor pattern can't be used with mixins.

With mixins formulated as generic classes, we always have a handle on the parent class, the type parameter that the class extends. For example, we can refer to the parent class of Scrollable as type T. As a result, there are no fundamental difficulties with allowing this to be passed as a type argument.

However, there are other significant difficulties with formulating mixins as generic types. Just to give you a taste of some of the difficulties that can arise, we'll discuss a few prominent ones and some potential solutions.

**Mixins and type erasure**

Before discussing any other problems, we should point out that, like the feature extensions of generic types discussed last month, support for mixins can't be added to the Java language using the simple type erasure strategy used by JSR-14 and Tiger.

To see why, consider what would happen when a class that extended a type parameter was erased. It would end up extending the bound of the type parameter! For example, every instantiation of class Scrollable in the previous example would end up extending class JComponent. That's clearly not what we want.

To support mixins through generic types, we need to have run-time representations of the generic type instantiations available. Fortunately, there are ways of encoding this information that are actually backward compatible with Tiger. Such a backward-compatible encoding scheme is the hallmark trait of the NextGen formulation of Generic Java (in the [Resources](http://www.ibm.com/developerworks/java/library/j-djc05133/index.html#resources) section).

**Available constructors of the superclass**

An immediate and pressing problem that arises as soon as we want to allow for classes that extend a type parameter is to decide what super-constructors are we able to call? Recall that every Java class constructor must call a constructor of the superclass. Normally, the type checker ensures that these super-constructor calls will succeed by looking up the superclass and making sure that a matching super-constructor exists.

But when all we know about our superclass is that it's some instantiation of a type parameter, we have no idea what constructors will be available for a given instantiation. Also notice that the type checker can't even check that every instantiation of a mixin will result in valid super-constructor calls. The reason: a mixin's parameters may be instantiated with type parameters bound in some other context.

For example, a generic class JSplitPane<T> may create an instance of Scrollable<T>. We can't know whether the super-constructors called in Scrollable<T> are valid unless we know all the ways in which type parameter T is instantiated forJSplitPanes. But because Java coding allows for separate class compilation, we can't know all of the instantiations ofJSplitPane during type-checking.

The various solutions to this problem correspond exactly to the solutions proposed for checking new expressions on type parameters we discussed in [Part 3](http://www.ibm.com/developerworks/java/library/j-djc04093.html) last month, because both super-constructor calls and new expressions reference the same class constructors of a given class. Let's review those solutions:

* Require a zeroary constructor for all type parameter instantiations.
* Throw an exception at run time when there is no matching constructor.
* Include additional annotations on type parameters telling us which constructors those instantiations must contain.

As in the case of new expressions, the first two solutions have serious drawbacks. Often, it just doesn't make sense to include a zeroary constructor in a class definition. Also, it's not ideal to simply throw an exception when no matching constructor exists. After all, the whole point of static type checking is to prevent exactly that sort of exception.

The third solution can be wordy, but it has many advantages. Annotate type parameters with a set of constructors that all instantiations must have. These annotations tell us exactly what constructors we can reliably call on a type parameter. Thus, when a type parameter T is used as the superclass of a generic class, the annotation on T tells us exactly what super-constructors we can call. If T doesn't include an annotation, then the type checker disallows its use as the superclass.

**Accidental method overriding**

One really big problem that arises with any formulation of mixins is that the method names of a particular mixin may clash with the method names of a potential instantiation of its superclass. For example, suppose that class Scrollable contained a method getSize that took no arguments and returned a Size object that encoded its horizontal and vertical dimensions. Now let's suppose that class MyTextPane (a subclass of JComponent) also included a method getSize that took no arguments but returned an int representing the screen area of the object on which it was called.

The resulting classes are shown as follows:

**Listing 1. An example of an accidental method override**

|  |
| --- |
| class Scrollable<T extends JComponent> extends T {  ...  Size getSize() {...}  }  class MyTextPane extends JComponent {  ...  int getSize() {...}  }  new Scrollable<MyTextPane>() |

Then the mixin instantiation Scrollable<MyTextPane> would contain two methods getSize with identical (empty) parameter types, but incompatible return types! Because we could not have expected this problematic override of getSize to be foreseen by either the programmer of class Scrollable or by the programmer of MyTextPane (after all, they may not even be on the same development team), we call it an *accidental* override.

When mixins are formulated as generic classes, the problem of accidental overrides is particularly nasty. Because a mixin's parent may be instantiated with a type parameter, there is no way for the type checker to determine all cases of accidental method overriding. What's more, throwing a run-time exception when an accidental override occurs is not acceptable because there is no way for a client programmer to predict when such an exception will be thrown. If we want to write reliable programs, we can't allow for unpredictable errors to occur at run time.

Another solution would be to simply hide one of these clashing methods and resolve all matching method calls to refer to the method not hidden. The problem with this solution is that we'd like a mixin instantiation such as Scrollable<MyTextPane> to be used in both contexts in which a Scrollable object is called for and in contexts in which a MyTextPane object is called for. Hiding either one of the getSize methods would prevent the use of Scrollable<MyTextPane>s in both of these contexts.

In the context of mixins outside of generic types, Felleisen, Flatt, and Krishnamurthi proposed a good solution to this problem at the 1998 ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (see [Resources](http://www.ibm.com/developerworks/java/library/j-djc05133/index.html#resources)): to resolve references to clashing methods based on the context in which the mixin instantiation is used. In this solution, a mixin is associated with a view that determines which method to call in the case of a name clash.

In the case of mixins as generic types, we can apply the same solution. We just have to devise some notion of *view* that works in the context of generic types and also allows for backward compatibility with the JVM. At the Rice JavaPLT labs, we've proposed one such solution in the paper "A First-Class Approach to Genericity" (see [Resources](http://www.ibm.com/developerworks/java/library/j-djc05133/index.html#resources)).

**With power comes problems**

As the examples, problems, and potential solutions demonstrate, extending generic types in Java programming to include support for mixins results in a powerful language, but it also introduces problems to overcome. This is typical of programming-language design: a desirable feature can be added only by complicating many existing features. In the world of programming languages, there's no such thing as a free lunch.

Ref: <http://www.ibm.com/developerworks/java/library/j-djc05133/index.html>