JOURNAL OF OBJECT TECHNOLOGY

Online at http://www.jot.fm. Published by ETH Zurich, Chair of Software Engineering ©**JOT, 2004**

Vol. 3, No. 8, September-October 2004

Case Study: Converting to Java 1.5 Type-Safe Collections

Wes Munsil, Lexonics, Inc., Denver, Colorado, USA

1 INTRODUCTION

The next edition of the Java 2 Platform, Standard Edition (J2SE), formerly known as Java 1.5 but now known as J2SE 5, includes lots and lots of changes: "15 component JSRs with nearly 100 other significant updates," according to [1] (and see [2] for a link to this and other interesting pages). Just one of those changes is the introduction of generic types. Generic types allow reference types (class types, interface types, and array types) to be parametrized by other types, and are used to good effect in the J2SE 5 collections framework. But modifying existing code that uses the collections framework to take advantage of the new capabilities can involve a fair amount of work. This note describes my experience in porting the Edsel object model and parser ([3]), a heavy user of the collections framework, to the new type-safe collections framework of J2SE 5. Maybe some of what I learned will be useful to you, if you face a similar challenge.

In the following, I assume you are familiar with the definition of generic types in J2SE 5. If you aren't, [4] provides a good overview.

For the examples in this note, I used J2SE 5.0 Beta 2, downloadable from [5].

2 WARNINGS OUT THE WAZOO

For backward compatibility, code that uses the previous, non-generified, collection types is still valid with J2SE 5. These types—`List`, `Set`, and so on—are now called *raw types*. Their semantics is similar to the semantics of the corresponding J2SE 5 wildcard types—`List<?>`, `Set<?>`, etc.—with one important difference: non-type-safe code using them is flagged with warnings, not error messages.

Here is an example of what I mean. If the type of the variable `foo` is `List<?>` (list of unknown), then the statement `foo.add (3);` results in the error message

Cite this column as follows: Wes Munsil: "Case Study: Converting to Java 1.5 Type-Safe Collections", in *Journal of Object Technology*, vol. 3, no. 8, September-October 2004, pp. 7-13. http://www.jot.fm/issues/issue_2004_09/column1

```
Foo.java:9: cannot find symbol
symbol  : method add(int)
location: interface java.util.List<capture of ?>
                foo.add (3);
                    ^
```

But if `foo` is of type `List`, the same statement results in the warning

```
Note: Foo.java uses unchecked or unsafe operations.
Note: Recompile with -Xlint:unchecked for details.
```

You can compile with the option `-Xlint:unchecked` to get a more informative message:

```
Foo.java:9: warning: [unchecked] unchecked call to add(E) as a
member of the raw type java.util.List

                foo.add (3);
                        ^
```

What is going on here? In the first place, the reason we can add 3 to the list, instead of, say, `new Integer (3)`, is that J2SE 5's new *boxing conversion* will convert the `int` value `3` to a reference to an `Integer` object. Fine. So is there a problem with adding an `Integer` reference to a list? Previously, no—the elements of a list could be objects of any class at all. But now, declaring `foo` as a `List<?>` means that expressions of type `List<T>`, for any particular type `T`, can be assigned to it. The compiler takes this into account when doing static type checking. For all the compiler knows, `foo` could have been assigned a `List<String>` expression, for example—so the attempt to add an `Integer` reference has to be disallowed. In the case of the raw type `List`, though, this attempt only merits a warning, not an error, so that legacy code will continue to compile.

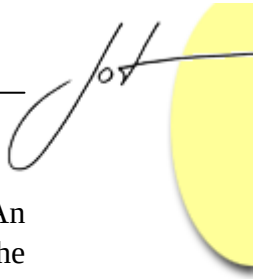
The first lesson in converting your code, then, is to compile with `-Xlint:unchecked`. This will let you see all the places where the compiler cannot guarantee type safety. You will want to modify these places to use parametrized types.

One thing to keep in mind: J2SE 5 has generic types, not generic classes. Regardless of the types of expressions that refer to them in source code, objects at runtime actually have only what we might call *raw classes* (my term). You might declare a variable to have type `ArrayList<String>`, but at runtime it will refer to objects of class `ArrayList`, period. This implementation strategy, called *type erasure*, prevents the "code bloat" common with C++ templates. But it has other consequences that might prove surprising—consequences that are outside the scope of this note.

Edsel consists of 2,600 lines of Java. When I compiled it with `-Xlint:unchecked`, I got 28 warnings, all on invocations of methods such as `List.add` and `Map.put`. The Edsel regression tests consist of another 1,448 lines, which yielded another 62 warnings.

3 MAKE THE WARNINGS GO AWAY

How do you use parametrized types to make the warnings go away? Let's look at a typical example.



The Edsel class `Attribute` had a private final `List` field called `valueList`. An `Attribute` constructor had a formal parameter `value` of type `String`, and the statement

```
this.valueList.add (value);
```

This statement got the "unchecked call" warning. Well, no problem—I wrote this code, and I knew that `valueList` was in fact always a list of `Strings`. So, just change the type in the declaration of `valueList` to `List<String>`, and recompile. Problem solved, right?

Yes and no. Now, instead of 28 warnings, I got 30. I got rid of one, but replaced it with three others. You may have the same experience, particularly for the first few warnings you eliminate. Generally speaking, if you re-type a variable, you should expect a "ripple effect" to introduce other problematic situations.

As an aside, remember that whether there are 28 or 30, they are still only warnings. The code that worked before will still work under J2SE 5. But our goal is to use the J2SE 5 type-safe collections as much as we can, to get the maximum benefit out of the improved compile-time error detection that they make possible.

The three new warnings were all "unchecked conversion" warnings. They were on these three statements:

```
this.valueList = emptyList;
this.valueList = new ArrayList ();
this.valueList = Util.clone (valueList);
```

where the type of `emptyList` and the result type of `Util.clone` were the raw type `List`.

To resolve the first, I knew that `emptyList` was only used in this context, so I changed its name to `emptyStringList`, its type to `List<String>`, and its initialization value from `new ArrayList ()` to `new ArrayList<String> ()`.

To resolve the second, I changed the right-hand side of the assignment to `new ArrayList<String> ()`.

To resolve the third, I observed that the method `Util.clone` was actually not necessary. Its only function was to make a shallow copy of a list; and, in this case, `new ArrayList<String> (valueList)` would do exactly that. This was my first example of how going through this exercise would actually make my code better—in this case, because it wasn't particularly good to begin with!

Hmm. Still 28 warnings, one on that very line from which I had just taken out `Util.clone`. It turned out that the `valueList` that was passed to `Util.clone` was not the instance variable `valueList`, but a `List` parameter to an `Attribute` constructor. So I changed the parameter's type to `List<String>` and that warning went away.

The ripple effect introduced another warning elsewhere, so I still had 28—but now none of them were in the `Attribute` class.

You can do this kind of analysis and code modification for all of the "unchecked" warnings you get. It will take a little time, but pay off in the long run. I did that for Edsel, and eventually got code that compiled with no warnings.

4 MAKE THE RAW TYPES GO AWAY

Once you have warning-free code, are you done? Oh, would that it were so! There still might be occurrences of raw types that, for whatever reason, were not considered problematic by the compiler, and not flagged. But since you want to be a good citizen of the J2SE 5 world, you need to fix them too.

Finding the remaining occurrences of raw types may be a little tricky. My coding style helped me here. Since, in my code, names like `List` are always followed by a space unless they are parametrized (in which case they are followed by a less-than sign), I was able to use UNIX tricks like

```
find . -name \*.java -exec grep "\<List " {} /dev/null \;
```

to find them all. You may have to be more devious.

5 EXPLOITING TYPE SAFETY

Once all the warnings and unwarned uses of raw types are gone, you can exploit the type-safety of the J2SE 5 collections to get some actual code improvements. For example, with type-safe collections, it is rarely (if ever) necessary to cast elements drawn from a collection, since the compiler knows what type they are. Furthermore, with the new `for`-statement syntax, it is rarely (if ever) necessary to use `Iterators` explicitly to draw elements from a collection.

Prior to J2SE 5, a common looping idiom looked like this:

```
for (Iterator i = valueList.iterator (); i.hasNext (); )
{
    String s = (String) i.next ();

    // use s
}
```

You can now replace that idiom with this one:

```
for (String s: valueList)
{
    // use s
}
```

Look through your code for all uses of `Iterator` to see where you can make improvements like this.

6 REMOVING INSTANCEOF—WITH CAUTION!

As we've seen, legacy code may have contained casts of collection elements, casts that are no longer necessary. Likewise, legacy code may have contained `instanceof` checks of collection elements, to ensure that casts would succeed. In many cases, you can remove these too—look through your code for all uses of `instanceof` to find them.

But be very careful! The `instanceof` operator always returns `false` if its first operand is `null`, and you need to be sure you take this into account in preserving your code's behavior.

Here's an example of what I mean. The `Attribute` class in Edsel has a constructor that takes a `valueList` parameter, and initializes the `valueList` instance variable with a reference to a shallow copy of the list referenced by the parameter. Lacking type-safe collections, it had to have a runtime check to be sure that only `Strings` were in the list. That code looked like this:

```
for (Iterator i = valueList.iterator (); i.hasNext (); )
{
    Object o = i.next ();

    if (! (o instanceof String))
    {
        throw new IllegalArgumentException
          ("an element of the value list is not a String");
    }
}
```

A naïve J2SE 5 analysis would say that this `for`-statement could now be deleted, since the type of the parameter is now `List<String>`. But the old code also kept `nulls` out of the list; so the correct translation is actually

```
for (String s: valueList)
{
    if (s == null)
    {
        throw new IllegalArgumentException
          ("an element of the value list is null");
    }
}
```

Of course, whenever you change the semantics of a method like this, be sure you change its javadocs accordingly.

The moral of the story is this: before embarking on a type-safe collection conversion exercise, be sure you have good regression tests! Otherwise, it would be all too easy to modify your code's behavior so that it treats `nulls` differently, through careless removal of `instanceof` operators—and not find out until much later.

7 SPLITTING

In the course of your conversion, you may at times find it desirable to "split" a local variable into two or more local variables. For example, some Edsel methods had a `Set` local variable that referred at different times to a `Set<String>` or a `Set<Node>`. In those cases, I split the `Set` variable into a `Set<String>` variable and a `Set<Node>` variable, and used each in its appropriate context.

An alternative to splitting is to re-type such variables with a wildcard type, bounded with a common supertype. In the above `Set` example, I could have re-typed the local variable as `Set<?>` (which is actually equivalent to `Set<? extends Object>`. But the extra precision that splitting provides is preferable.

A similar situation may arise with method parameters. A `Set` formal parameter may at times correspond to a `Set<String>` or `Set<Node>` actual parameter, for example. Again, you may decide to split the method into two, or re-type the formal parameter with a bounded wildcard type.

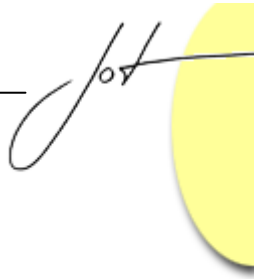
8 CONCLUSION

By modifying your collections-framework code to use the new J2SE 5 type-safe collections, you can make it more efficient and easier to understand, and detect more errors at compile-time instead of runtime. But you can expect to need several hours to do the modification. Based on my experience with Edsel, my recommendations are these:

1. Be sure you have good regression tests in place—and use them.
2. Compile with `-Xlint:unchecked`.
3. Remove the "unchecked" warnings by introducing parametrized types.
4. Find all other uses of raw types, and replace them by parametrized types too.
5. Look for all uses of `Iterator` and type casts, and replace them by the new `for`-statement syntax when possible.
6. Look for all uses of `instanceof`, and remove them when possible, or replace them by `null` tests—be very careful!
7. Change the javadocs for a method when you change its semantics.
8. Be alert to the need to split variables and methods into two or more.

REFERENCES

1. http://java.sun.com/developer/technicalArticles/releases/j2se15/.
2. http://java.sun.com/j2se/1.5.0/index.jsp.
3. http://www.lexonics.com/Edsel/.



4. http://java.sun.com/j2se/1.5/pdf/generics-tutorial.pdf.
5. http://java.sun.com/j2se/1.5.0/download.jsp

About the author

Wes Munsil is president of Lexonics, Inc. (http://www.lexonics.com), a reviewer at reviews.com, and until recently an instructor at the University of Colorado in Denver. Wes has 30 years of experience in the software industry, 20 of which have been in private practice as a consultant. His main interests are the design and implementation of programming languages. Wes holds graduate degrees from Caltech and the University of Cambridge, and a Ph.D. from the University of Colorado.