Java Feature: Experiences With the New 1.5 Java Language Features

Quite a few articles have been written introducing the new language features in JDK 1.5. In this article, we're going to go a little deeper and provide tips on how to effectively use those features.

Introduction
During the beta period for JDK 1.5, we worked on a 1.5 Java compiler for BEA's Java IDE. As we implemented various new features, people would begin exploiting them in new ways, some clever, some clearly candidates for a list of what not to do. The compiler itself used 1.5 features, so we gained direct experience in maintaining 1.5 code as well.

As we mentioned, this is not an introductory article. You should know roughly what the new features are, and we'll talk about some of the interesting, hopefully non-obvious implications and uses. These tips are a somewhat random collection of things we ran into, loosely grouped by language feature.

We'll start with the simplest features and work our way toward the most advanced ones. Generics is an especially rich subject and occupies about half of this article.

For-Each Loop
The new for-each loop provides a simple, consistent syntax for iterating over collections and arrays. There are just a couple of interesting items to mention.

Init Expression
The initialization expression is evaluated only once inside the loop. This means that you can often remove a variable declaration. In this example, we had to create an integer array in order to hold the results of computeNumbers() to prevent reevaluation of that method on each pass through the loop. You can see the bottom code is a little cleaner than the above, and doesn't leak the variable "numbers."

Without For Each:

```java
int sum = 0;
Integer[] numbers = computeNumbers();
for (int i=0; i < numbers.length ; i++)
    sum += numbers[i];
```

With:

```java
int sum = 0;
for (Integer number : computeNumbers())
    sum += number;
```
int sum = 0;
for ( int number: computeNumbers() )
    sum += number;

Limitations
Sometimes you need access to the iterator or index during iteration. Intuitively it seems like the for-each loop should allow this. It doesn't. Take the following example:

for (int i=0; i < numbers.length ; i++) {
    if (i != 0) System.out.print(",");
    System.out.print(numbers[i]);
}

We want to print out a comma-separated list of the values in the array. We need to know whether we're on the first item in order to know if we should print a comma. With for-each, there's no way to get at this info. We'd need to keep an index ourselves, or a boolean indicating whether or not we've past the first item.

Here's another example:

for (Iterator<Integer> it = n.iterator() ; it.hasNext() ; )
    if (it.next() < 0)
        it.remove();

In this case, we want to remove the negative items from a collection of integers. In order to do this, we need to call a method on the iterator, but when using for-each, this iterator would be hidden from us. Instead, we need to just use the pre-1.5 method of iterating.

Note, by the way, that Iterator is generic, so the declaration is Iterator<Integer>. A lot of people seem to miss this and use Iterator in its raw form.

Annotations
Annotation processing is a very large topic. We're not going to cover all of the possibilities and pitfalls of it, as we're limiting our article to core language features.

We will, however, discuss the built-in annotations and the limitations of annotation processing in general.

Suppress Warnings
This annotation turns off compiler warnings at a class or method level. Sometimes you know better than the compiler that your code must use a deprecated method, or perform some action that cannot be statically determined to be typesafe, but in fact is.

@SuppressWarnings("deprecation")
public static void selfDestruct() {
    Thread.currentThread().stop();
}
This is probably the most useful of the built-in annotations. Unfortunately, javac doesn't support it as of 1.5.0_03. It is supported in 1.6, however, and Sun is working on back-porting it to 1.5.

It is supported in Eclipse 3.1, and possibly other IDEs as well. This allows you to keep your code entirely warning-free. If a warning shows up when compiling, you can be certain that you just added it - helping to keep you aware of possibly unsafe code. With the addition of generics, this is even more desirable.

Deprecated
Unfortunately, this one is a little less useful. It's meant to replace the @deprecated javadoc tag, but since it doesn't have any fields, there's no way to suggest to the user of a deprecated class or method what they should be using as a replacement. Most uses will require both the Javadoc tag and this annotation.

Override
This indicates that the method it annotates should be overriding a method with the same signature in a superclass.

@Override
public int hashCode() {
    ...
}

Take the above example - if you were to fail to capitalize the "C" in hashCode, you wouldn't get an error at compile time but, at runtime, your method would not be called as you expected. By adding the Override tag, the compiler will complain if it doesn't actually perform an override.

This also helps in the case where the superclass changes. If, say, a new parameter were added to this method and the method itself renamed, the subclass will suddenly fail to compile, as it no longer overrides anything in the super.

Other Annotations
Annotations can be extremely useful in other situations as well. They work best for frameworks like EJB and Web services, when behavior is not directly modified but rather enhanced, especially in the case of adding boiler-plate code.

Annotations cannot be used as a preprocessor. Sun's design specifically precludes modifying the byte code of a class directly because of an annotation. This is so that the results of the language can be properly understood and tools like IDEs can perform deep code analysis and functions like refactoring.

Annotations are not a silver bullet. When first running across them, people are tempted to try all sorts of tricks. Take this next proposal we got from someone:

public class Foo {
    @Property
    private int bar;
The idea here was to automatically create getter and setter methods for the private field "bar." Unfortunately, this is a bad idea for two reasons: (1) it doesn't work, and (2) it makes this code harder to read and deal with.

It can't be done because as we mentioned, Sun specifically precludes modifying the class that an annotation appears in.

Even if it were possible, it's not a good idea because it makes this code harder to understand. Someone looking at this code for the first time will have no idea that this annotation creates methods. Also, if in the future you need to do something inside one of those methods the annotation is useless.

In summary, don't try to use annotations to do what regular code would do.

**Enumeration**

 Enums are a lot like public static final ints that have been used for many years as enum values. The biggest and most obvious improvement over ints is type safety - you cannot mistakenly use one type of enum in place of another, unlike ints, which all look the same to the compiler. With very few exceptions, you should replace all enum-like int constructs with enum instances.

 Enums offer a few additional features as well. There are two utility classes, EnumMap and EnumSet, which are implementations of standard collections optimized specifically for enums. If you know your collection will contain only enums, you should use these specific collections instead of HashMap or HashSet.

 For the most part you can do a drop-in replacement of any public static final ints in your code with enums. They're comparable, and can be statically imported so that references to them look identical - even in the case of an inner class (or inner enum). Note that when comparing enums, the order they are declared indicates their ordinal value.

 **"Hidden" Static Methods**

 There are two static methods that appear on all enum declarations that you write. They don't appear in the javadoc for java.lang.Enum, as they are static methods on enum subclasses, not on Enum itself.

 The first, values(), returns an array of all of the possible values for an enum.

 The second, valueOf(), returns an enum for the provided string, which must match the source code declaration exactly.

 **Methods**

 This is one of our favorite aspects of enums: they can have methods. In the past you might have some code that performed a switch on a public static final int in order to translate from a database type into a JDBC URL. Now, you can have a method directly on the enum itself, which can clean up code dramatically. Here's an example of how this is done, with an abstract method on the DatabaseType enum and implementations provided in each enum instance:
public enum DatabaseType {
    ORACLE {
        public String getJdbcUrl() {...}
    },
    MYSQL {
        public String getJdbcUrl() {...}
    }
    public abstract String getJdbcUrl();
}

Now your enum can provide its utility method directly. For instance:

    DatabaseType dbType = ...;
    String jdbcURL = dbType.getJdbcUrl();

Previously you would have had to know where the utility method was for getting the url.

Varargs
Varargs can really clean up some ugly code, when used correctly. Here's the canonical example: a log method that takes a variable number of string arguments.

    Log.log(String code)
    Log.log(String code, String arg)
    Log.log(String code, String arg1, String arg2)
    Log.log(String code, String[] args)

The interesting item to discuss about varargs is the compatibility if you replace the first four examples with a new, vararged one:

    Log.log(String code, String... args)

All of them are source compatible. That is, if you recompile all callers of the log() method, you can just replace all four methods directly. If, however, you need backward binary compatibility, you'll need to leave in the first three. Only the final method, taking an array of strings, is equivalent to, and thus can be replaced by, the vararged version.

Casting
You should avoid casting with varargs in cases where you simply expect the caller to know what the types should be. Take this example, where the first item is expected to be a string, and the second an exception:

    Log.log(Object... objects) {
        String message = (String)objects[0];
        if (objects.length > 1) {
            Exception e = (Exception)objects[1];
            // Do something with the exception
        }
    }
Instead, your method signature should be like the following, with the string and exception declared separately from the vararg parameter:

```java
Log.log(String message, Exception e, Object... objects) {...}
```

Don't try to be too clever. Don't use varargs to subvert the type system. If you need strong typing, use it. PrintStream.printf() is one interesting exception to this rule: it provides type information as its first argument so that it can accept those types later.

### Covariant Returns

The primary use of covariant returns is to avoid casts when an implementation's return type is known to be more specific than the API's. In this example, we've got a Zoo interface that returns an Animal object. Our implementation returns an AnimalImpl object, but before JDK 1.5 it had to be declared to return an Animal object:

```java
public interface Zoo {
    public Animal getAnimal();
}
```

```java
public class ZooImpl implements Zoo {
    public Animal getAnimal(){
        return new AnimalImpl();
    }
}
```

The use of covariant returns replaces three anti-patterns:

1. Direct field access. In order to get around the API restriction, some implementations would expose the subclass directly as a field:
   ```java
   ZooImpl._animal
   ```

2. An additional form was to perform the downcast in the caller, knowing that the implementation was really this specific subclass:
   ```java
   ((AnimalImpl)ZooImpl.getAnimal()).implMethod();
   ```

3. The last form I've seen is a special method that avoids the problem by coming up with a different signature entirely:
   ```java
   ZooImpl._getAnimal();
   ```

All of these have their problems and limitations. Either they're ugly or expose implementation details that should not be necessary.

### With Covariance

The covariant return pattern is cleaner, safer, and easier to maintain. No casts or special methods or fields are required:
public AnimalImpl getAnimal() {
    return new AnimalImpl();
}

Using the result:

ZooImpl.getAnimal().implMethod();

Generics
Generics is split into using generics and writing generics. These include both generic types and
types. We're not going to talk about the obvious use of List, Set, and Map. Suffice it to say that
generic collections are great and should always be used.

We are going to cover using generic methods and how the compiler infers the types. Usually this
will just work for you, but when it doesn't the error messages are fairly inscrutable and you will
need to know how to fix the problem.

Generic Methods
In addition to generic types, 1.5 introduced generic methods. In this example from
java.util.Collections, a singleton list is constructed. The element type of the new list is inferred
based on the type of the object passed into the method:

\[
\text{static } <T> \text{ List<T> Collections.singletonList(T o)}
\]

Example usage:

```java
public List<Integer> getListOfOne() {
    return Collections.singletonList(1);
}
```

In the example usage, we pass in an int. The return type of the method is then List<Integer>. The
compiler infers Integer for T. This is different from generic types because you don't generally need
to explicitly specify the type argument.

This also shows interaction of autoboxing with generics. Type arguments must be reference types,
that's why we get List<Integer> and not List<int>.

Generic Methods Without Parameters
The emptyList() method was introduced with generics as a type-safe replacement for the
EMPTY_LIST field:

```
static <T> List<T> Collections.emptyList()
```

Example usage:

```java
public List<Integer> getNoIntegers() {
    return Collections.emptyList();
}
```
Unlike the previous example, this one has no parameters, so how does the compiler infer the type for T? Basically, it will try once using the parameters. If that does nothing, it tries again using the return or assignment type. In this case, we are returning List<Integer>, so T is inferred to be Integer.

What if you are invoking a generic method in a place other than in a return statement or assignment statement? Then the compiler is unable to do the second pass of type inferencing. In this example, emptyList() is invoked from within the conditional operator:

```java
public List<Integer> getNoIntegers() {
    return x ? Collections.emptyList() : null;
}
```

The compiler cannot see the return context, and cannot infer T, so it would give up and assume Object. You would see an error message like "cannot convert List<Object> to List<Integer>".

To fix this, you explicitly pass the type argument to the method invocation. Then the compiler won't try to infer the type arguments for you, and you get the right result:

```java
return x ? Collections.<Integer>emptyList():null;
```

The other place where this will happen frequently is in method invocation. If a method takes a List<String> and you try to call this passing emptyList() for that param, you would also need to use this syntax.

**Beyond Collections**

Here are three examples of generic types that are not collections that use generics in a novel way. All of these come from the standard Java libraries.

- **Class<T>:** Class is parameterized on the type of the class. This make it possible to construct a new instance without casting.
- **Comparable<T>:** Comparable is parameterized by the actual comparison type. This provides stronger typing on compareTo() invocations. For example, String implements Comparable<String>. Invoking compareTo() on anything other than a String will fail at compile time.
- **Enum<E extends Enum<E>>:** Enum is parameterized by the enum type. An enum called Color would extend Enum<Color>. The getDeclaringClass returns the class object for the enum type. It's different from getClass(), which may return an anonymous class.

**Wildcards**

The most complex part of generics is understanding wildcards. We'll cover the three types of wildcards and why you might want to use them.

First let's look at how arrays work. You can assign a Number[] from an Integer[]. If you attempt to write a Float into the Number[], it will compile but fail at runtime with an ArrayStoreException:

```java
Integer[] ia = new Integer[5];
Number[] na = ia;
```
na[0] = 0.5; // compiles, but fails at runtime

If we try to translate that example directly into generics, it fails at compile time because the assignment isn't allowed:

```java
List<Integer> iList = new ArrayList<Integer>();
List<Number> nList = iList; // not allowed
nList.add(0.5);
```

With generics, you will never get runtime ClassCastException as long as you have code that compiles without warnings.

**Upper Bounded Wildcards**
What we want is a list whose exact element type is unknown.

- A `List<Number>` is a list whose element type is the concrete type Number - exactly.
- A `List<? extends Number>` is a list whose exact element type is unknown. It is Number or a subtype.

**Upper Bounds**
If we update our original example, and assign to a `List<? extends Number>`, the assignment now succeeds:

```java
List<Integer> iList = new ArrayList<Integer>();
List<? extends Number> nList = iList;
Number n = nList.get(0);
nList.add(0.5); // Not allowed
```

We can get Numbers out of the list because no matter what the exact element type of the list is (Float, Integer, or Number), we can still assign it to Number.

We still can't insert floats into the list. This fails at compile time because we can't prove that this is safe. If we were to add a float into the list, it would violate the original type safety of `iList` - that it stores only Integers.

Wildcards give us more expressive power than is possible with arrays.

**Why Use Wildcards**
In this example, a wildcard is used to hide type information from the user of the API. Internally, the set is stored as `CustomerImpl`. To the user of the API, all they know is that they are getting a set from which they can read customers.

Wildcards are necessary here because you can't assign from a `Set<CustomerImpl>` to a `Set<Customer>`.

```java
public class CustomerFactory {
    private Set<CustomerImpl> _customers;
```
public Set<? extends Customer> getCustomers() {
    return _customers;
}

Wildcards and Covariant Returns
Another common use for wildcards is with covariant returns. The same rules apply to covariant returns as assignment. If you want to return a more specific generic type in an overridden method, the declaring method must use wildcards:

public interface NumberGenerator {
    public List<? extends Number> generate();
}
public class FibonacciGenerator extends NumberGenerator {
    public List<Integer> generate() {
        ...
    }
}

If this were to use arrays, the interface could return Number[] and the implementation could return Integer[].

Lower Bounds
We've talked mostly about upper bounded wildcards. There is also a lower bounded wildcard. A List<? super Number> is a list whose exact "element type" is unknown, but it is number or a super type of number. So it could be a List<Number> or a List<Object>.

Lower bounded wildcards are not nearly as common as upper bounded wildcards. But when you need them, they are essential.

Lower vs Upper Bounds
List<? extends Number> readList = new ArrayList<Integer>();
Number n = readList.get(0);

List<? super Number> writeList = new ArrayList<Object>();
writeList.add(new Integer(5));

The first list is a list that you can read numbers from. The second list is a list that you can write numbers to.

Unbounded Wildcard
Finally, the List<?> is a list of anything. Almost the same as List<? Extends Object>. You can always read Objects, but you cannot write to the list.

Wildcards in Public APIs
To summarize, wildcards are great for hiding implementation details from callers as we saw a few
slides back, but even though lower bounded wildcards appear to provide read-only access, they do not due to non-generic methods like remove(int position). If you want a truly immutable collection, use the unmodifiableCollection(), etc.

Be aware of wildcards when writing APIs. In general, you should try to use wildcards when passing generic types. It makes the API accessible to a wider range of callers.

In this example, by accepting a List<? extends Number> instead of List<Number>, the method can be called by many different types of lists:

```java
void removeNegatives(List<? extends Number> list);
```

## Constructing Generic Types

Now we'll cover constructing your own generic types. I'll show example idioms where type safety can be improved by using generics, as well as common problems that occur when trying to implement generic types.

### Collection-Like Functions

This first example of a generic class is a collection-like example. Pair has two type parameters, and the fields are instances of the types:

```java
public final class Pair<A,B> {
    public final A first;
    public final B second;

    public Pair(A first, B second) {
        this.first = first;
        this.second = second;
    }
}
```

This makes it possible to return two items from a method without having to write special-purpose classes for each two-type combo. The other thing you could have done is return Object[], which isn't type-safe or pretty.

In the usage here, we return a File and a Boolean from a method. The client of the method can use the fields directly without casting:

```java
public Pair<File,Boolean> getFileAndWriteStatus(String path){
    // create file and status
    return new Pair<File,Boolean>(file, status);
}
```

Pair<File,Boolean> result = getFileAndWriteStatus("...");
File f = result.first;
boolean writeable = result.second;

## Beyond Collections

Here is an example where generics are used for additional compile-time safety. By parameterizing
the DBFactory class by the type of Peer it creates, you are forcing Factories to return a specific
subtype of Peer:

```java
public abstract class DBFactory<T extends DBPeer> {
    protected abstract T createEmptyPeer();

    public List<T> get(String constraint) {
        List<T> peers = new ArrayList<T>();
        // database magic
        return peers;
    }
}
```

By implementing DBFactory<Customer>, the CustomerFactory is forced to return a Customer from
createEmptyPeer:

```java
public class CustomerFactory extends DBFactory<Customer>{
    public Customer createEmptyPeer() {
        return new Customer();
    }
}
```

**Generic Methods**
Whenever you want to place constraints on a generic type between parameters or a parameter and a
return type, you probably want to use a generic method.

For example, if you write a reverse function that reverses in place, you don't need a generic method.
However, if you want reverse to return a new list, you'd like the element type of the new list to be
the same as the list that was passed in. In that case, you need a generic method:

```java
<T> List<T> reverse(List<T> list)
```

**Reification**
When implementing a generic class, you may want to construct an array, T[]. Because generics is
implemented by erasure, this is not allowed.

You might try to cast an Object[] to T[]. This is not safe.

**Reification - Solution**
The solution, courtesy of the generics tutorial, is to use a "Type Token". By adding a Class<T>
parameter to the constructor, you force clients to supply the correct class Object for the type
parameter of the class:

```java
public class ArrayExample<T> {
    private Class<T> clazz;

    public ArrayExample(Class<T> clazz) {
        this.clazz = clazz;
    }
}
```
public T[] getArray(int size) {
    return (T[])Array.newInstance(clazz, size);
}

To construct an ArrayExample<String>, the client would have to pass String.class to the constructor because the type of String.class is Class<String>.

Having the class objects makes it possible then to construct an array with exactly the right element type.

**Migrating to Generics**

Finally, we'll just briefly talk about migrating 1.4 code to using Generics:

- You can generify existing classes and interfaces.
- You can convert raw collection uses to generic collections, even in public methods. This will not break clients who override.
- If you need to pass a collection to a 1.4-level library, you can use checked collections. These will fail fast if that library attempts to put something in a collection that shouldn't belong. This is an interesting example of the "Type Token" pattern we mentioned.
- In general, if you want to know if something is safe, look at the standard libraries. There are plenty of examples of migrating classes while maintaining source and binary compatibility.

**Conclusion**

In summary, the new language features make for a substantial change to Java. By understanding when and how to use them, you'll write better code.

© 2006 SYS-CON Media Inc.