Java 5 Features: Part-II

Venkat Subramaniam
venkats@agiledeveloper.com
http://www.agiledeveloper.com/download.aspx

Abstract

In Part I we discussed the Java 5 features of autoboxing and foreach. In this Part-II, we discuss varargs and static import. Since each of the other features enum, annotation, and generics deserve separate/special treatment, we will cover these in other articles.

varargs

Assume you want to invoke a method with variable number of arguments. The option we had on hand in Java 1.4 or earlier was to pass an array. Let's consider a simple example as shown below:

```
public static int max(int[] values)
{
    int max = values[0];

    for(int aValue : values)
    {
        if (aValue > max) max = aValue;
    }

    return max;
}
```

We can invoke this method by passing an array with different number of values as illustrated below:

```
max(new int[] {1, 7, 2, 9, 8});
max(new int[]{8, 12, 87, 23, 1, 3, 6, 9, 37});
```

In order to invoke a method that takes variable number of arguments, we had to bundle the parameters into an array. While this works, it is not elegant. The varargs introduced in Java 5 addresses the elegance issue.

Before we look at that, let's talk about how C++ handled this. In C++ you could pass variable number of arguments to methods by using the concept of ellipsis (…). While that was an interesting concept it has two significant problems. First, the code to get different parameters within the method is not simple. You have to use a special set of function (va_list, va_args, etc) to pull the parameters from the stack. Second, there was no guaranteed way to determine what type of data is on the stack. Generally I discourage developers from using ellipsis in C++ and instead really on operator overloading and concatenation.

The varargs concept in Java 5 does not suffer from these issues. It is not only elegant, it also is very type safe. Let's take the `max` method and modify it to use the varargs.

```
public static int max(int... values)
```

Notice that the only thing I changed is `int[]` to `int…` and I did not modify the implementation of the `max` method. If the previous implementation of the method still works after the change to the parameter type, then what is the new syntax and how is it related to the array?

A type followed by … is simply a syntax sugar–it's nothing but an array. However, the compiler allows you to pass either an array or a discrete set of values to this method. For example, now I can call the modified `max` method as follows:

```
max(new int[] {1, 7, 2, 9, 8});
max(new int[]{8, 12, 87, 23, 1, 3, 6, 9, 37});

max(1, 7, 2, 9, 8);
max(8, 12, 87, 23, 1, 3, 6, 9, 37);
```

There's less clutter in the bottom two lines than in the top two lines. But what's going on when you pass discrete values? The compiler simply rolls the values into an array. So, the code:

```
max(1, 7);
```

is compiled into:

```
max(new int[] {1, 7});
```

as you can see from the following byte code:

```
0:   iconst_2
1:   newarray int
3:   dup
4:   iconst_0
5:   iconst_1
6:   iastore
7:   dup
8:   iconst_1
9:   bipush  7
11:  iastore
12:  invokestatic    #2; //Method max:([I)I
```

In the above example we passed an array of int. What if we want to pass different types of data? Sure we can. Consider the example below:

```
public static void print(Object... values)
{
    for(Object obj : values)
    {
        System.out.printf("%s is of type %s\n",
                    obj, obj.getClass().getName());
    }
```

```
    }
```

The above code receives a varargs of type `Object`. You can invoke it with different types as shown below:

```
print(1, "test", 'a', 2.1);
```

The output from this call is:

```
1 is of type java.lang.Integer
test is of type java.lang.String
a is of type java.lang.Character
2.1 is of type java.lang.Double
```

The first line should be no surprise if you kept autoboxing in mind–that is the reason for the type to be `Integer` instead of `int`. We've also used the `printf` statement which directly benefits from the varargs concept.

You are not restricted to having only varargs as parameters, that is, you can have regular parameters and varargs, if you like. However, varargs, if present, must be trailing. In other words, place any regular parameters you like and then place the varargs as shown in the following example:

```
public static void print(String msg, Object... values)
```

Pros and Cons:

- varargs comes in handy when you want to pass variable number of arguments. Use it if you need that flexibility and elegance.
- You loose some compile time type safety if your varargs is of type `Object`. However, if it is specific type (like `int…`), then you have type safety.
- If in your application you expect only three or four parameters to be passed to the method, then don't bother to use varargs.
- If you are not careful, you may have trouble with method overloading as the varargs may increase the chance of parameter collision when methods are overloaded.

static imports

An `import` statement allows you to give a hint to the compiler as to which class you are referring to in your code. It gives you the convenience of using the short name for a class (like `JButton`) instead of the long, fully qualified name (like `javax.swing.JButton`). Remember that `import` does not tell the compiler where the class resides (*classpath* does that). It only tells the compiler which class you mean to use.

Of course if classes in multiple package/namespace collide, then we will get an error. In this case, you can resort to using the fully qualified name for the class.

While `import` gives you convenience, it does have a disadvantage. It does not clearly indicate to the reader as to where the class being used comes from. If you have several

imports at the top of you code, then it may take a while for a reader of your code to figure out which classes belong to which package/namespace.

Static import is a concept in Java 5 that unfortunately aggravates this. Let's take a look at the following code:

```
package com.agiledeveloper.com;

public class Test
{
    public static void main(String[] args)
    {
        System.out.println("Math.abs(-2) = " + Math.abs(-2));
        System.out.println("Math.ceil(3.9) = " + Math.ceil(3.9));
    }
}
```

Looking at this code, you can quickly figure out that I am calling the methods `abs()` and `ceil()` of the `Math` class. However, I had to type `Math.` for both the calls. Assume you have a need to call the `abs()` function ten times in your code. Then, you could argue that, it would reduce clutter by calling `abs()` instead of `Math.abs()`. Static import allows you to do that as shown below:

```
package com.agiledeveloper.com;

import static java.lang.Math.abs;
import static java.lang.Math.ceil;
import static java.lang.Runtime.*;

public class Test
{
    public static void main(String[] args)
    {
        System.out.println("Math.abs(-2) = " + abs(-2));
        System.out.println("Math.ceil(3.9) = " + ceil(3.9));

        System.out.printf("Free memory is = %d",
                          getRuntime().freeMemory());
    }
}
```

In the above code, we are making a call to `abs()`, `ceil()`, and `getRuntime()` as if these are static methods of the `Test` class (which they are not). The static import gives us the impressions that we have brought these methods into the current namespace.

Baring the issue of difficulty to identify where methods have come from, static import has some advantages. For instance, in EasyMock 2.0, static import is used to reduce the verbosity of the code. In using the mock, you would write code like:

```
SomeInterface mock = (SomeInterface) createMock(SomeInterface.class);
expectCall(mock.foo()).andReturn(5);
replay(mock);
```

…

The underlined methods above all are part of the `EasyMock` class and can be used in the code above as shown, provided we added

```
import static org.easymock.EasyMock.*;
```

Pros and Cons:

- Static imports are useful to remove the tedium in calling a set of static methods over and over.
- The significant disadvantage is that it can make the code hard to read–you may have difficulty figuring out where certain methods came from. For this reason, we recommend that you use static import very sparingly and keep it to less than a couple per file. Using several static imports is not a good idea.

enum, annotation, and generics

The concepts of enum, annotation, and generics deserve their own coverage. enum is pretty interesting in Java 5 and it is a realization of Jashua Bloch's recommendation[6] for writing enum well.

Other features

In this article we have focused on the language features in Java 5. Java 5 has some other interesting features as well. Listed below are some select features:

- StringBuilder
  - StringBuffer eliminates object creation overhead, but has synchronization overhead
  - StringBuilder removes that overhead
- Client vs. Server side differences in garbage collection, more adaptive collection
- Improved Image I/O for performance and memory usage
- Reduced application startup time and footprint using shared archive
- Enhancement to Thread Priority
- Ability to get stack trace for a thread or all threads
- UncoughtExceptionHandler on a Thread
- Improved error reporting on fatal exceptions
- System.nanoTime() for nanoseconds granularity for time measurements
- ProcessBuilder
  - Easier than Runtime.exec() to start process
- Formatter and Formattable provide ability to format output in printf like style
- Scanner for easier conversion to primitive types – based on regex
- java.lang.instrument allows byte code enhancement at runtime to instrument code
- Collections Framework has Queue, BlockingQueue, and ConcurrentMap interfaces and implementations. Some classes modified to implements new interfaces
- Reflection API supports annotation, enum. Class has been generified
- System.getenv() undeprecated!

Conclusion

In this Part-II of the Java 5 features article we discussed the benefits of varargs and saw how it can make our code elegant and easier to use. It is simply a syntax sugar where you can pass an array or discrete values to a method. We also looked at the static imports, which we recommend that you use sparingly.

References

1. http://java.sun.com
2. http://www.agiledeveloper.com/download.aspx Look in "Presentations" section for "Java 5 Features" examples and slides.
3. "Good, Bad, and Ugly of Java Generics," Part-I, Part-II, and Part-III http://www.agiledeveloper.com/download.aspx.
4. "enum in Java 5," TBA.
5. "Annotations in Java 5," TBA.
6. Joshua Bloch, *Effective Java Programming Language Guid*, Addison-Wesley, 2001.