

Java 1.5 Tiger

*A Developer's
Notebook™*

Brett McLaughlin
David Flanagan

O'REILLY®

varargs

In this chapter:

- *Creating a Variable-Length Argument List*
- *Iterating Over Variable-Argument Lists*
- *Allowing Zero-Length Argument Lists*
- *Specify Object Arguments Over Primitives*
- *Avoiding Automatic Array Conversion*

One of the coolest features of Java, and of any object-oriented language, is method overloading. While many might think Java's strengths are its typing, or all the fringe APIs it comes with, there's just something nice about having the same method name with a variety of acceptable arguments:

```
Guitar guitar = new Guitar("Bourgeois", "Country Boy Deluxe",
    GuitarWood.MAHOGANY, GuitarWood.ADIRONDACK,
    1.718);

Guitar guitar = new Guitar("Martin", "HD-28");

Guitar guitar = new Guitar("Collings", "CW-28"
    GuitarWood.BRAZILIAN_ROSEWOOD, GuitarWood.ADIRONDACK,
    1.718,
    GuitarInlay.NO_INLAY, GuitarInlay.NO_INLAY);
```

This code calls three versions of the constructor of a (fictional) Guitar class, meaning that information can be supplied when it's available, rather than forcing a user to know everything about their guitar at one time (many professionals couldn't tell you their guitar's width at the nut). Here are the constructors used:

```
public Guitar(String builder, String model) {
}

public Guitar(String builder, String model,
    GuitarWood backSidesWood, GuitarWood topWood,
    float nutWidth) {
}

public Guitar(String builder, String model,
    GuitarWood backSidesWood, GuitarWood topWood,
    float nutWidth,
    GuitarInlay fretboardInlay, GuitarInlay topInlay) {
}
```

Enums, which are used in these examples, are detailed in Chapter 3.

However, things start to get a little less useful when you want to add information that isn't finite. For example, suppose you want to allow additional, unspecified features to be added to this constructor. Here are some possible invocation examples:

```
Guitar guitar = new Guitar("Collings", "CW-28"  
    GuitarWood.BRAZILIAN_ROSEWOOD, GuitarWood.ADIRONDACK,  
    1.718,  
    GuitarInlay.NO_INLAY, GuitarInlay.NO_INLAY,  
    "Enlarged Soundhole", "No Popsicle Brace");  
  
Guitar guitar = new Guitar("Martin", "HD-28V",  
    "Hot-rodged by Dan Lashbrook", "Fossil Ivory Nut",  
    "Fossil Ivory Saddle", "Low-profile bridge pins");
```

For these two cases alone, you'd have to add another constructor that takes two additional strings, and yet another that takes four additional strings. Try and apply these same versions to the already-overloaded constructor, and you'd end up with 20 or 30 versions of that silly constructor!

It's here where *variable arguments*, more often called *varargs*, come in. Another of Tiger's additions, varargs solve the problem detailed here once and for all, in a pretty slick way. This chapter covers this relatively simple feature in all its glory, and will have you writing better, cleaner, more flexible code in no time.

All of the new formatting methods, which are detailed in Chapter 9, use varargs.

Creating a Variable-Length Argument List

Variable arguments allow you to specify that a method can take multiple arguments of the same type, and don't require that the number of arguments be pre-determined (at compile- or runtime). This is one of the integral parts of Tiger, in fact, as several of the new features of the language actually incorporate varargs..

How do I do that?

First, get used to typing the ellipsis (...). Those three little dots are the key to varargs, and you'll be typing them quite often. Here's a version of the Guitar constructor that uses varargs to allow for an indeterminate number of String features:

```
public Guitar(String builder, String model, String... features);
```

All these constructors are shown, completed, in the source code for the com.oreilly.tiger.ch05.Guitar class.

The argument `String...` features indicates that any number of `String` arguments may be supplied. So all of the following invocations are legal:

```
Guitar guitar = new Guitar("Martin", "HD-28V",
                           "Hot-rodged by Dan Lashbrook", "Fossil Ivory Nut",
                           "Fossil Ivory Saddle", "Low-profile bridge pins");

Guitar guitar = new Guitar("Bourgeois", "OMC",
                           "Incredible flamed maple bindings on this one.");

Guitar guitar = new Guitar("Collings", "OM-42",
                           "Once owned by Steve Kaufman--one of a kind");
```

You could add the same variable-length argument to the other constructors:

```
public Guitar(String builder, String model,
              GuitarWood backSidesWood, GuitarWood topWood,
              float nutWidth, String... features)

public Guitar(String builder, String model,
              GuitarWood backSidesWood, GuitarWood topWood,
              float nutWidth,
              GuitarInlay fretboardInlay, GuitarInlay topInlay,
              String... features)
```

Example 5-1 shows a simple class that puts this all together, and even uses delegation to pass some varargs around.

Example 5-1. Using varargs in constructors

```
package com.oreilly.tiger.ch05;

public class Guitar {

    private String builder;
    private String model;
    private float nutWidth;
    private GuitarWood backSidesWood;
    private GuitarWood topWood;
    private GuitarInlay fretboardInlay;
    private GuitarInlay topInlay;

    private static final float DEFAULT_NUT_WIDTH = 1.6875f;

    public Guitar(String builder, String model, String... features) {
        this(builder, model, null, null, DEFAULT_NUT_WIDTH, null, null, features);
    }

    public Guitar(String builder, String model,
                  GuitarWood backSidesWood, GuitarWood topWood,
                  float nutWidth, String... features) {
        this(builder, model, backSidesWood, topWood, nutWidth, null, null, features);
    }
```

Example 5-1. Using varargs in constructors (continued)

```
}

public Guitar(String builder, String model,
              GuitarWood backSidesWood, GuitarWood topWood,
              float nutWidth,
              GuitarInlay fretboardInlay, GuitarInlay topInlay,
              String... features) {

    this.builder = builder;
    this.model = model;
    this.backSidesWood = backSidesWood;
    this.topWood = topWood;
    this.nutWidth = nutWidth;
    this.fretboardInlay = fretboardInlay;
    this.topInlay = topInlay;
}
}
```

What just happened?

When you specify a variable-length argument list, the Java compiler essentially reads that as “create an array of type *<argument type>*”. You typed:

```
public Guitar(String builder, String model, String... features)
```

However, the compiler interprets this as:

```
public Guitar(String builder, String model, String[] features)
```

This means that iteration over the argument list is simple (as shown in “Iterating Over Variable-Length Argument Lists”), as is any other programming tasks you need to undertake. You can work with varargs just as you would with arrays.

However, there are some limitations. First, you can only use one ellipsis per method. Thus, the following is illegal:

```
public Guitar(String builder, String model,
              String... features, float... stringHeights)
```

Additionally, the ellipsis must appear as the *last* argument to a method.

What about...

...if you don't have any features to pass in? That's fine. Just call the constructor in the old way:

```
Guitar guitar = new Guitar("Martin", "D-18");
```

You'll get a compiler error from this, and one that's not all that descriptive of the real problem.

Look closely, though—there is no constructor with the following signature:

```
public Guitar(String builder, String model)
```

So, what gives? Well, as an added bonus to varargs, *not* passing in an argument is a legitimate option. So when you see `String... features`, you should think “zero or more `String` arguments.” That saves you from creating another constructor without the varargs parameter.

Iterating Over Variable-Length Argument Lists

All this varargs business is well and good, but unless you can actually use them in your methods, it’s obviously just eye-candy and window dressing. However, you can work with vararg parameters just as you do an array, making usage a piece of cake.

How do I do that?

Make sure you read “Creating a Variable-Length Argument List,” which lets you know the most important piece of information relating to vararg methods—variable-length arguments are treated just as arrays. So, continuing with the previous example, you could do something like this:

```
public Guitar(String builder, String model,
              GuitarWood backSidesWood, GuitarWood topWood,
              float nutWidth,
              GuitarInlay fretboardInlay, GuitarInlay topInlay,
              String... features) {
```

The for/in loop is covered in detail in Chapter 7.

```
    this.builder = builder;
    this.model = model;
    this.backSidesWood = backSidesWood;
    this.topWood = topWood;
    this.nutWidth = nutWidth;
    this.fretboardInlay = fretboardInlay;
    this.topInlay = topInlay;
```

This example is yanked straight out of Java in a Nutshell, Fifth Edition (O'Reilly).

```
    for (String feature : features) {
        System.out.println(feature);
    }
}
```

This isn’t particularly sexy, but it should get the point across. As another example, here’s a simple method that calculates the maximum from a set of numbers:

```

public static int max(int first, int... rest) {
    int max = first;
    for (int i : rest) {
        if (i > max)
            max = i;
    }
    return max;
}

```

Simple enough, right?

What about...

...storing variable-length arguments? Since the Java compiler treats these like arrays, an array is obviously a great choice for storage, as seen in Example 5-2, which is a modified version of Example 5-1.

Example 5-2. Storing variable-length arguments as member variables

```

package com.oreilly.tiger.ch05;

public class Guitar {

    private String builder;
    private String model;
    private float nutWidth;
    private GuitarWood backSidesWood;
    private GuitarWood topWood;
    private GuitarInlay fretboardInlay;
    private GuitarInlay topInlay;
    private String[] features;

    private static final float DEFAULT_NUT_WIDTH = 1.6875f;

    public Guitar(String builder, String model, String... features) {
        this(builder, model, null, null, DEFAULT_NUT_WIDTH, null, null, features);
    }

    public Guitar(String builder, String model,
                  GuitarWood backSidesWood, GuitarWood topWood,
                  float nutWidth, String... features) {
        this(builder, model, backSidesWood, topWood, nutWidth, null, null, features);
    }

    public Guitar(String builder, String model,
                  GuitarWood backSidesWood, GuitarWood topWood,
                  float nutWidth,
                  GuitarInlay fretboardInlay, GuitarInlay topInlay,
                  String... features) {

```

Example 5-2. Storing variable-length arguments as member variables (continued)

```
this.builder = builder;
this.model = model;
this.backSidesWood = backSidesWood;
this.topWood = topWood;
this.nutWidth = nutWidth;
this.fretboardInlay = fretboardInlay;
this.topInlay = topInlay;
this.features = features;
    }
}
```

You could also store these in Java collection classes easily:

```
// Variable declaration
private List features;

// Assignment in method or constructor body
this.features = java.util.Arrays.asList(features);
```

*The java.util.
Arrays class has
several nice
methods for
working with
arrays, all of
which are of
interest in varargs
methods.*

Allowing Zero-Length Argument Lists

One particularly nice feature about varargs is that a variable-length argument can take from *zero* to *n* arguments. This means that you can actually invoke one of these methods *without* any parameters, and things still behave. On the other hand, this means that, as a programmer, you better realize you must safeguard against this condition.

How do I do that?

Remember in “Iterating Over Variable-Length Argument Lists,” you saw this simple method:

```
public static int max(int first, int... rest) {
    int max = first;
    for (int i : rest) {
        if (i > max)
            max = i;
    }
    return max;
}
```

You can call this method in several ways:

```
int max = MathUtils.max(1, 4);
int max = MathUtils.max(1, 2, 3, 4, 5, 6, 7, 8, 9, 10);
int max = MathUtils.max(18, 8, 4, 2, 1, 0);
```


What's not so nice is that there are many cases where you may already have the numbers to pass in stored as an array, or at least in some collected form:

```
// Get the numbers from some method
int[] numbers = getListOfNumbers();
```

It's impossible to just pass these numbers on to the `max()` method. You would need to check the list length, and strip off the first object (if it's available), then check the type to ensure it's an `int`. That would be passed in, along with the rest of the array (which can be iterated over, or converted manually to a suitable format). In general, this process is a real pain and is a lot of work for what should be trivial. To get around this, remember that this method is treated by the compiler as the following:

```
public static int max(int first, int[] rest)
```

So, by extension, you could convert `max()` to look like this:

```
public static int max(int... values) {
    int max = Integer.MIN_VALUE;
    for (int i : values) {
        if (i > max)
            max = i;
    }
    return max;
}
```

Autounboxing helps some, as "Integer" objects are freely converted to "int" primitives. Autounboxing is covered in Chapter 4.

You've now created a method that can easily be used with arrays:

```
// Get the numbers from some method
int[] numbers = getListOfNumbers();

int max = MathUtils.max(numbers);
```

While using a single variable-length argument made this task easier, it introduces problems if you pass in a zero-length array—in the best case, you're going to get unexpected results. To account for this, you now need a little error checking. Example 5-3 is a complete code listing for the `MathUtils` class, which at this point is more of a `MathUtil` class!

Example 5-3. Handling zero-argument methods

```
package com.oreilly.tiger.ch05;
```

```
public class MathUtils {
```

```
    public static int max(int... values) {
        if (values.length == 0) {
            throw new IllegalArgumentException("No values supplied.");
        }
    }
```

Example 5-3. Handling zero-argument methods (continued)

```
int max = Integer.MIN_VALUE;
for (int i : values) {
    if (i > max)
        max = i;
}
return max;
}
```

Anytime you have the possibility for a zero-length argument list, you need to perform this type of error checking. Generally, a nice informative `IllegalArgumentException` is a great solution.

*Whatever you do,
please don't
throw a checked
exception—you
just add hassle
for programmers
using your code,
and for what is a
fringe case,
rather than a
normal problem.*

What about...

...invoking this same method with normal non-array arguments? That's perfectly legal, of course. The following are all legitimate ways to invoke the `max()` method:

```
int max = MathUtils.max(myArray);
int max = MathUtils.max(new int[] { 2, 4, 6, 8 });
int max = MathUtils.max(2, 4, 6, 8);
int max = MathUtils.max(0);
int max = MathUtils.max();
```

Specify Object Arguments Over Primitives

As discussed in Chapter 4, Tiger adds a variety of new features through *unboxing*. This allows you, in the case of varargs, to use object wrapper types in your method arguments.

How do I do that?

Remember that every class in Java ultimately is a descendant of `java.lang.Object`. This means that any object can be converted to an `Object`; further, because primitives like `int` and `short` are now automatically converted to their object wrapper types (`Integer` and `Short` in this case), any Java type can be converted to an `Object`.

Thus, if you want to accept the widest variety of argument types in your vararg methods, use an object type as the argument type. Better yet, go with `Object` for the absolute most in versatility. For example, take a method that did some printing:

```
private String print(Object... values) {
    StringBuilder sb = new StringBuilder();
    for (Object o : values) {
        sb.append(o)
          .append(" ");
    }
    return sb.toString();
}
```

The basic idea here is to print anything and everything. However, the more obvious way to declare this method is like this:

```
private String print(String... values) {
    StringBuilder sb = new StringBuilder();
    for (Object o : values) {
        sb.append(o)
          .append(" ");
    }
    return sb.toString();
}
```

The problem here is that now this method won't take Strings, ints, floats, arrays, and a variety of other types, all of which you might want to legitimately print.

By using a more general type, `Object`, you obtain the ability to print anything and everything.

Avoiding Automatic Array Conversion

Tiger adds all sorts of automatic conversions and conveniences, which is pretty cool...about 99% of the time. Unfortunately, there are times when all those helps turn into hindrances. The conversion of `Object...` to `Object[]` in a `varargs` method can be one of those cases, and you'll find that in rare cases, you need to work around Java.

How do I do that?

Before getting into the details of getting around this issue, be sure you understand the problem. Take Java's new `printf()` method, a real convenience:

```
System.out.printf("The balance of %s's account is $%(,6.2f\n",
    account.getOwner().getFullName(), account.getBalance());
```

printf(), along with the other new Tiger formatting methods, are detailed in Chapter 9.

I realize this isn't the most common scenario. Then again, if all I covered were common scenarios, we'd all be debugging right now, wouldn't we?

If you look at the Javadoc for `printf()`, you'll see it's a varargs method, with two parameters: a `String` for the formatting string, and then `Object...` for all the arguments passed in for use in that formatting string:

```
PrintStream printf(String format, Object... args)
```

By now, you can mentally convert this to the following:

```
PrintStream printf(String format, Object[] args)
```

All good, right? Well, most of the time. Consider the following code:

```
Object[] objectArray = getObjectArrayFromSomewhereElse();
out.printf("Description of object array: %s\n", obj);
```

This might seem a bit far-fetched—however, consider this as normal fare for *introspective code*. That's a ten-cent word for code that investigates other code. If you are writing a code analysis tool, or an IDE, or anything else that might use reflection or a similar API to figure out what objects an application uses, this suddenly becomes a normal usecase. Here, you're not really interested in the contents of the object array as much as you are with the array itself. What type is it? What's its memory address? What is its `String` representation? Keep in mind that all these questions apply to the array itself, and *not* to the contents of the array. For example, let's say the array is something like this:

```
public Object[] getObjectArrayFromSomewhereElse() {
    return new String[] {"Hello", "to", "all", "of", "you"};
}
```

In that case, you might write some code like this to begin to answer some questions about this array:

```
out.printf("Description of object array: %s\n", obj);
```

However, the output isn't what you expect:

```
run-ch05:
[echo] Running Chapter 5 examples from Java Tiger: A Developer's
Notebook

[echo] Running VarargsTester...
[java] Hello
```

What in the world? This is hardly what you'd expect to see—however, the compiler did just what it always did—it converted `Object...` in the `printf()` method to `Object[]`. When it read your method invocation, it saw an argument that was, in fact, `Object[]`! So instead of treating the array as an object itself, it broke it up into its various parts. The first argument became the `String` "Hello", which was passed to the format string (`%s`), and the result was "Hello" being printed out.

To get around this, you need to tell the compiler that you want the entire object array, `obj`, treated as a *single* object, and not as a grouping of arguments. Here's the magic bullet:

```
out.printf("Description of object array: %s\n", new Object[] { obj });
```

Alternatively, here's an even shorter approach:

```
out.printf("Description of object array: %s\n", (Object)obj);
```

In both cases, the compiler no longer sees an array of objects, it simply sees a single `Object` (which just happens to be an array of objects). The result is what you should want (at least in this rather odd scenario):

```
run-ch05:
[echo] Running Chapter 5 examples from Java Tiger: A Developer's
Notebook

[echo] Running VarargsTester...
[java] [Ljava.lang.String;@c44b88
```

While this may look like gibberish to you, it's probably what reflection-based or other introspective code wants to take a look at.