Varargs

In past releases, a method that took an arbitrary number of values required you to create an array and put the values into the array prior to invoking the method. For example, here is how one used the `MessageFormat` class to format a message:

```
Object[] arguments = {
    new Integer(7),
    new Date(),
    "a disturbance in the Force"
};

String result = MessageFormat.format(
    "At {1,time} on {1,date}, there was {2} on planet "
     + "{0,number,integer}.", arguments);
```

It is still true that multiple arguments must be passed in an array, but the varargs feature automates and hides the process. Furthermore, it is upward compatible with preexisting APIs. So, for example, the `MessageFormat.format` method now has this declaration:

```
    public static String format(String pattern,
                                Object... arguments);
```

The three periods after the final parameter's type indicate that the final argument may be passed as an array *or* as a sequence of arguments. Varargs can be used *only* in the final argument position. Given the new varargs declaration for `MessageFormat.format`, the above invocation may be replaced by the following shorter and sweeter invocation:

```
String result = MessageFormat.format(
    "At {1,time} on {1,date}, there was {2} on planet "
    + "{0,number,integer}.",
    7, new Date(), "a disturbance in the Force");
```

There is a strong synergy between autoboxing and varargs, which is illustrated in the following program using reflection:

```
// Simple test framework
public class Test {
    public static void main(String[] args) {
        int passed = 0;
        int failed = 0;
        for (String className : args) {
            try {
                Class c = Class.forName(className);
                c.getMethod("test").invoke(c.newInstance());
                passed++;
            } catch (Exception ex) {
                System.out.printf("%s failed: %s%n", className, ex);
                failed++;
            }
        }
        System.out.printf("passed=%d; failed=%d%n", passed, failed);
    }
}
```

This little program is a complete, if minimal, test framework. It takes a list of class names on the command line. For each class name, it instantiates the class using its parameterless constructor and invokes a parameterless method called test. If the instantiation or invocation throws an exception, the test is deemed to have failed. The program prints each failure, followed by a summary of the test results. The reflective instantiation and invocation no longer require explicit array creation, because the `getMethod` and `invoke` methods accept a variable argument list. The program also uses the new `printf` facility, which relies on varargs. The program reads much more naturally than it would without varargs.

So when should you use varargs? As a client, you should take advantage of them whenever the API offers them. Important uses in core APIs include reflection, message formatting, and the new `printf` facility. As an API designer, you should use them sparingly, only when the benefit is truly compelling. Generally speaking, you should not overload a varargs method, or it will be difficult for programmers to figure out which overloading gets called.