Enhance looping in Java 5.0 with for/in
Find out what this convenient construct offers and when it's most applicable in your projects

Brett McLaughlin
November 30, 2004

The for/in loop -- often called either enhanced for or foreach is largely a convenience feature in Java 5.0. It doesn't really offer any new functionality, but certainly makes several routine coding tasks simpler. In this article, you'll learn about many of those, including using for/in to iterator over arrays and collections, as well as how it can help avoid unnecessary (or just plain annoying) typecasts. You'll also learn how for/in is implemented, glean details about the new Iterable interface, and even understand how to make your own custom objects usable with this new construct. Finally, you'll learn about the things that for/in can't do to make sure you understand when plain old for is the right choice.

Shorter might be better

It's one of the most basic axioms of the hardcore computer programmer: because shorter equates to less typing, shorter must therefore be intrinsically better. This philosophy gave birth to IDEs like vi, where commands like :wq! and 28G hold a wealth of meaning. It also led to some of the most cryptic code you'll ever come across -- variables like ar for Agile Runner (or is it Argyle, or Atomic Reactor... well, you get the idea).

Sometimes, in an effort to be short, programmers leave clarity somewhere back in the rearview mirror. That said, the backlash against short has resulted in code that is so verbose as to be painful. Variables named theAtomicReactorLocatedInPhiladelphia are just as annoying and unwieldy as those named ar. There must be a happy medium, right?

That happy medium, at least as far as I'm concerned, is rooted in finding convenient ways to do things, that aren't short for the the sake of being short. As a great example of just such a solution, Java 5.0 introduces a new version of the for loop, which I refer to as for/in. It's also called foreach, and sometimes enhanced for -- but these all refer to the same construct. Whatever you choose to call it, for/in makes for simpler code, as you'll see in the examples throughout this article.
Losing an Iterator
The most fundamental difference in using `for/in`, as opposed to "plain ol' for", is that you won't have to use a counter (usually called `i` or `count`) or `Iterator`. Check out Listing 1, which shows a `for` loop that uses an `Iterator`:

Listing 1. for loops, old-school style
```java
public void testForLoop(PrintStream out) throws IOException {
    List list = getList();  // initialize this list elsewhere
    for (Iterator i = list.iterator(); i.hasNext(); ) {
        Object listElement = i.next();
        out.println(listElement.toString);
        // Do something else with this list element
    }
}
```

Note: If you've been following my articles on the new features in Tiger (see Related topics), you know that I often thank O'Reilly Media, Inc., which allows me to publish code samples from my Tiger book in this article format. That means you get code that has gone through more testing, and more commenting, than I could otherwise provide to you.

If you're expecting a lengthy description of how to convert this code over to the new `for/in` loop, I'm about to disappoint you. Listing 2 shows the loop in Listing 1 using `for/in`, and it's remarkably similar. Take a look, and I'll explain what's going on in as much detail as I can muster (which will still barely take a paragraph).

Listing 2. Converting over to for/in
```java
public void testForInLoop(PrintStream out) throws IOException {
    List list = getList();  // initialize this list elsewhere
    for (Object listElement : list) {
        out.println(listElement.toString());
        // Do something else with this list element
    }
}
```

The basic syntax of the `for/in` loop is shown in Listing 3. This may look a little odd if you're not used to reading specifications, but it's pretty easy when you walk through it piece by piece.

Listing 3. Basic for/in loop structure
```java
for (declaration : expression)
    statement
```

How for/in got its name
Astute readers will note that the so-called `for/in` loop doesn't involve the word `in` at all. The naming comes from how this code is read. In Listing 2, you would say `for every Object in the variable named list, do ...`. Of course, the ellipses represents whatever that loop actually does. There are some variations in how you'll see this, but `for` and `in` are prominent in each representation.
**declaration** is just a variable, like **Object listElement**. This variable should be typed so that it's compatible with each item in the list, array, or collection being iterated over. In the case of Listing 2, **list** contains objects, so that's the type of **listElement**.

**expression** is ... well ... an expression. It should evaluate to something that can be iterated over (more on what that means under the hood later). For now, just ensure that **expression** evaluates to a collection or an array. This expression can be as simple as a variable (as shown in Listing 2) or a method call (like **getList()**), or a complex expression that involves boolean logic or a ternary operator. As long as it returns an array or collection, things will work fine.

**statement** is a placeholder for the contents of the loop, which operates on the variable defined in **declaration**; of course, this is a loop, so **statement** is applied to each item in the array or collection. And, with bracketing (**{** and **)**, you can use multiple statements.

The long and the short of it is that you create a variable, indicate the array or collection to iterate over, and then operate upon the variable you defined. There is no assignment of each item in the list, as **for/in** handles that for you. Of course, if this is still a little unclear, just keep reading -- there's a wealth of examples that will make this plenty clear.

Before moving on, though, I want to supply a few more specification-style illustrations of how **for/in** works. Listing 4 shows the **for/in** loop in action when a genericized type is supplied. This is actually what the statement looks like once the compiler converts it internally to a normal **for** loop.

Did you catch that? The compiler actually turns this shorter, more convenient **for/in** statement into a compiler-friendly **for** loop -- and you don't bear the brunt of the work. That's why I call this convenient, rather than simply short.

**Listing 4. for/in loop, in a transformed state, with an Iterable**

```java
for (Iterator<E> #i = (expression).iterator(); #i.hasNext(); ) {
    declaration = #i.next();
    statement
}
```

Listing 5 is another **for/in** after compiler transformation, this time without a genericized type. The same sort of thing is going on, although it's a little simpler. In either case, though, you can see that it's easily to mentally (and programmatically) convert a **for/in** statement to a normal **for** statement. Once you can do this conversion mentally, things get awfully easy.

**Listing 5. for/in loop, transformed, where there is no unparameterized type**

```java
for (Iterator #i = (expression).iterator(); #i.hasNext(); ) {
    declaration = #i.next();
    statement
}
```

**Working with arrays**

Now that you've got the basic semantics down, you're ready to move to some more concrete examples. You've already seen how **for/in** works with lists; it's just as easy to work with arrays.
Arrays, like collections, are assigned values (as shown in Listing 6), and then those values are routinely peeled off one by one, and operated upon.

**Listing 6. Simple array initialization**

```java
int[] int_array = new int[4];
String[] args = new String[10];
float[] float_array = new float[20];
```

In situations where you might use `for` and a counter or index variable, you can now use `for/in` (assuming you’re working with Tiger, of course). Listing 7 shows another simple example:

**Listing 7. Looping over an array with for/in is a piece of cake**

```java
public void testArrayLooping(PrintStream out) throws IOException {
    int[] primes = new int[] { 2, 3, 5, 7, 11, 13, 17, 19, 23, 29 };

    // Print the primes out using a for/in loop
    for (int n : primes) {
        out.println(n);
    }
}
```

There shouldn't be anything particularly revelatory here -- this is about as basic as it gets. Arrays are typed, and so you know exactly what variable type is going to be needed for each item in the array. This example creates the variable (named `n` in this case), and then operates upon that variable. Pretty easy, isn't it? I told you there's nothing complicated here.

It doesn't matter what types are in the array -- you just choose the right type for your *declaration*. In Listing 8, an array contains *Lists*. So you've actually got an array of collections here. Still, using `for/in` is simple as can be.

**Listing 8. It's also possible to loop over an object array with for/in**

```java
public void testObjectArrayLooping(PrintStream out) throws IOException {
    List[] list_array = new List[3];

    list_array[0] = getList();
    list_array[1] = getList();
    list_array[2] = getList();

    for (List l : list_array) {
        out.println(l.getClass().getName());
    }
}
```

You can even layer in your `for/in` loops, as shown in Listing 9:
Listing 9. There’s nothing wrong with using for/in inside of for/in!

```java
public void testObjectArrayLooping(PrintStream out) throws IOException {
    List[] list_array = new List[3];
    list_array[0] = getList();
    list_array[1] = getList();
    list_array[2] = getList();

    for (List l : list_array) {
        for (Object o : l) {
            out.println(o);
        }
    }
}
```

Working with collections

Once again, simplicity is the watchword. There’s simply nothing tricky or complicated about iterating over collections using for/in -- it works just like you’ve already seen, with both lists and arrays. Listing 10 shows an example of iterating over both a List and a Set, and has little in the way of surprise to offer. Still, walk through the code, and make sure you’re clear on what’s going on.

Listing 10. Lots of simple loops in this program demonstrate how to use for/in

```java
package com.oreilly.tiger.ch07;
import java.util.ArrayList;
import java.util.HashSet;
import java.util.List;
import java.util.Set;

public class ForInDemo {
    public static void main(String[] args) {
        // These are collections to iterate over below
        List wordlist = new ArrayList();
        Set wordset = new HashSet();

        // Basic loop, iterating over the elements of an array
        // The body of the loop is executed once for each element of args[].
        // Each time through, one element is assigned to the variable word.
        System.out.println("Assigning arguments to lists...");
        for (String word : args) {
            System.out.print(word + " ");
            wordlist.add(word);
            wordset.add(word);
        }
        System.out.println();

        // Iterate through the elements of the List now
        // Since lists have an order, these words should appear as above
        System.out.println("Printing words from wordlist " +
                ", (ordered, with duplicates)...");
        for (Object word : wordlist) {
            System.out.print((String)word + " ");
        }
        System.out.println();
    }
}
```
// Do the same for the Set. The loop looks the same but by virtue
// of using a Set, word order is lost, and duplicates are discarded.
System.out.println("Printing words from wordset " +
    "(unordered, no duplicates)...");
for (Object word : wordset) {
    System.out.print((String)word + " ");
}

Listing 11 shows the output of this program (with some data thrown in on the command line for
demonstration purposes):

Listing 11. The output does just what you would expect -- lots of printing!

run-ch07:
[echo] Running Chapter 7 examples from Java 5.0 Tiger: A Developer's Notebook
[echo] Running ForInDemo...
[java] Assigning arguments to lists...
[java] word1 word2 word3 word4 word1
[java] Printing words from wordList (ordered, with duplicates)...
[java] word1 word2 word3 word4 word1
[java] Printing words from wordset (unordered, no duplicates)...
[java] word4 word1 word3 word2

The pain of a typecast

So far, when dealing with collections, you've seen for/in used with a generic variable type, like
Object. This is nice, but doesn't really take advantage of another important Tiger feature: generics
(sometimes called parameterized types). I'll leave the details of generics for developerWorks'
upcoming tutorial on the subject, but generics make for/in even more powerful.

Remember that the declaration portion of a for/in statement creates a variable of the type of
each item in the collection being iterated over. In arrays, this was very specific, as arrays are
strongly typed -- an int[] can only contain ints -- and therefore there are no typecasts in the
loop. The same is possible when you use typed lists, via generics. Listing 12 shows a few simple
parameterized collections:

Listing 12. Adding a parameter to collection types means typecasting can be
avoided later on

List<String> wordlist = new ArrayList<String>();
Set<String> wordset = new HashSet<String>();

Now, your for/in loop can ditch plain old Object and be more specific. Listing 13 demonstrates
this:

Listing 13. When you know the types within the collection, your loop body can
be more type-specific

for (String word : wordlist) {
    System.out.print(word + " ");
}
As a more complete example, Listing 14 takes the program shown in Listing 10, and augments it with genericized lists and more specific for/in loops:

**Listing 14. Listing 10 can be rewritten to take advantage of generics**

```java
package com.oreilly.tiger.ch07;

import java.util.ArrayList;
import java.util.HashSet;
import java.util.List;
import java.util.Set;

public class ForInDemo {

    public static void main(String[] args) {

        // These are collections to iterate over below
        List<String> wordlist = new ArrayList<String>();
        Set<String> wordset = new HashSet<String>();

        // Basic loop, iterating over the elements of an array
        // The body of the loop is executed once for each element of args[].
        // Each time through, one element is assigned to the variable word.
        System.out.println("Assigning arguments to lists...");
        for (String word : args) {
            System.out.print(word + " ");
            wordlist.add(word);
            wordset.add(word);
        }
        System.out.println();

        // Iterate through the elements of the list now
        // Since lists have an order, these words should appear as above
        System.out.println("Printing words from wordlist " +
            "(ordered, with duplicates)...");
        for (String word : wordlist) {
            System.out.print((String)word + " ");
        }
        System.out.println();

        // Do the same for the Set. The loop looks the same but by virtue
        // of using a Set, word order is lost, and duplicates are discarded.
        System.out.println("Printing words from wordset " +
            "(unordered, no duplicates)...");
        for (String word : wordset) {
            System.out.print((String)word + " ");
        }
    }
}
```

Of course, the typecasting doesn't completely disappear in these cases. However, you’re offloading the work onto the compiler (which is more or less what generics does, for those who are interested in that sort of thing). At compile-time, all of these types will be checked, and you'll get any errors accordingly. And if someone else can do this work -- well, that's good for everyone, right?

**Who the heck is E?**

If you're a Java veteran, but new to Tiger, all the references to E may look odd to you here. These are all related to support for parameterized types (generics), allowing an
Integrating classes with for/in

So far, I've dealt only with iteration over Java's pre-packaged classes and types -- arrays, lists, maps, sets, and other collections. While that's a pretty good deal, the beauty of any programming language is the ability to define your own classes. Custom objects are the backbone of any large-scale application. This section deals with just that -- the concepts and steps involved in allowing your own objects to be used by the for/in construct.

A new interface

You should be familiar with the java.util.Iterator interface by now -- just in case you're not, Listing 15 shows this interface, as it appears in Tiger:

Listing 15. Iterator has long been a mainstay of the Java language

```java
package java.util;

public interface Iterator<E> {
    public boolean hasNext();
    public E next();
    public void remove();
}
```

To take advantage of for/in, though, you'll need to add another interface to your domain knowledge -- java.lang.Iterable. This interface is shown in Listing 16:

Listing 16. The Iterable interface is the cornerstone of the for/in construct

```java
package java.lang;

public interface Iterable<E> {
    public java.util.Iterator<E> iterator();
}
```

java.lang, not java.util

Take care to note that Iterable is in java.lang, not java.util. I haven't found any explicit documentation about why this is, but I personally suspect it's to avoid having to import the interface (java.lang is in the set of namespaces automatically imported for all Java code).

For your object or class to work with for/in, it needs to implement the Iterable interface. This leaves you with two basic scenarios:

- Extend an existing collection class that already implements Iterable (and therefore already supports for/in).
- Handle iteration manually, by defining your own implementation of Iterable.
Handling iteration manually

If at all possible, I strongly recommend you have your custom objects extend an existing collection. Things are dramatically simpler, and you get to avoid all sorts of messy detail. Listing 17 shows a class that does just this:

Listing 17. Extending an existing collection is an easy way to take advantage of for/in

```java
package com.oreilly.tiger.ch07;
import java.util.LinkedList;
import java.util.List;

public class GuitarManufacturerList extends LinkedList<String> {
    public GuitarManufacturerList() {
        super();
    }

    public boolean add(String manufacturer) {
        if (manufacturer.indexOf("Guitars") == -1) {
            return false;
        } else {
            super.add(manufacturer);
            return true;
        }
    }
}
```

Because `LinkedList` already works with `for/in`, you can use this new class with `for/in` with no special code. Listing 18 demonstrates this -- and how little work was required to make this possible:

Listing 18. The Iterable interface is the cornerstone of the for/in construct

```java
package com.oreilly.tiger.ch07;
import java.io.IOException;
import java.io.PrintStream;

public class CustomObjectTester {
    /** A custom object that extends List */
    private GuitarManufacturerList manufacturers;

    public CustomObjectTester() {
        this.manufacturers = new GuitarManufacturerList<String>();
    }

    public void testListExtension(PrintStream out) throws IOException {
        // Add some items for good measure
        manufacturers.add("Epiphone Guitars");
        manufacturers.add("Gibson Guitars");

        // Iterate with for/in
        for (String manufacturer : manufacturers) {
            out.println(manufacturer);
        }
    }
}
```
public static void main(String[] args) {
    try {
        CustomObjectTester tester = new CustomObjectTester();
        tester.testListExtension(System.out);
    } catch (Exception e) {
        e.printStackTrace();
    }
}

Handling iteration manually

In certain unusual cases -- and I had a hard time thinking of many, to be honest -- you may need to perform specific behavior when your custom objects are iterated over. In these (rather unfortunate) cases, you'll have to take things into your own hands. Listing 19 shows you how this works -- it's not as complex as it is a lot of work, so I'll let you walk through this code on your own. This class provides a wrapper over a text file that lists each line of a file as it is iterated over.

Listing 19. With some patience, you can implement Iterable on your own, providing custom behavior in looping

```java
package com.oreilly.tiger.ch07;

import java.util.Iterator;
import java.io.BufferedReader;
import java.io.FileReader;
import java.io.IOException;

/**
 * This class allows line-by-line iteration through a text file.
 *   The iterator's remove() method throws UnsupportedOperationException.
 *   The iterator wraps and rethrows IOExceptions as IllegalArgumentExceptions.
 */
public class TextFile implements Iterable<String> {

    // Used by the TextFileIterator below
    final String filename;

    public TextFile(String filename) {
        this.filename = filename;
    }

    // This is the one method of the Iterable interface
    public Iterator<String> iterator() {
        return new TextFileIterator();
    }

    // This non-static member class is the iterator implementation
    class TextFileIterator implements Iterator<String> {

        // The stream being read from
        BufferedReader in;

        // Return value of next call to next()
        String nextline;

        public TextFileIterator() {
            // Open the file and read and remember the first line
            // Peek ahead like this for the benefit of hasNext()
            try {
                in = new BufferedReader(new FileReader(filename));
                nextline = in.readLine();
            } catch (IOException e) {
                e.printStackTrace();
            }
        }
```

```
The bulk of the work here is in implementing an `Iterator`, which is then returned by the `iterator()` method. Everything else is pretty straightforward. As you can see, though, it takes a lot more to implement the `Iterable` interface manually than simply extending a class that does this work for you.

**What you can't do**

As with all good things -- and I do think that `for/in` is one of them -- there are limitations. Because of how `for/in` is setup, and specifically because it leaves out explicit usage of `Iterator`, there are things you just can't get done with this new construct.

**Positioning**

The most notable limitation is the inability to determine the position you're in within a list or array (or custom object). To refresh your memory, Listing 20 shows how a typical `for` loop might be
used -- note the usage of an index variable to not only move through the list, but also to indicate position:

**Listing 20. Using the iteration variable in a "normal" for loop**

```java
List<String> wordList = new LinkedList<String>();
for (int i=0; i<args.length; i++) {
    wordList.add("word "+ (i+1) + ": "+ args[i] + ":");
}
```

This isn't a fringe usage, either -- this is common programming. However, you can't do this very simple task with `for/in`, as Listing 21 illustrates:

**Listing 21. It's impossible to access position in a for/in loop**

```java
public void determineListPosition(PrintStream out, String[] args)
    throws IOException {
    List<String> wordList = new LinkedList<String>();

    // Here, it's easy to find position
    for (int i=0; i<args.length; i++) {
        wordList.add("word "+ (i+1) + ": "+ args[i] + ":");
    }

    // Here, it's possible to locate position
    for (String word : wordList) {
        out.println(word);
    }
}
```

Without any type of counter variable (or `Iterator`), you're simply out of luck here. If you need position, stick with "ordinary" `for`. Listing 22 shows another common usage of position -- working with strings:

**Listing 22. Another problem -- string concatenation**

```java
StringBuffer longList = new StringBuffer();
for (int i=0, len=wordList.size(); i < len; i++) {
    if (i < (len-1)) {
        longList.append(wordList.get(i)).append(", ");
    } else {
        longList.append(wordList.get(i));
    }
}
out.println(longList);
```

**Removing items**

Another limitation is in item removal. As Listing 23 shows, it's not possible to remove an item during list iteration:

**Listing 23. It's impossible to remove an item in a for/in loop**

```java
public void removeListItems(PrintStream out, String[] args)
```
In the big picture, these aren't limitations as much as they are guidelines on when to use `for` and when to use `for/in`. A subtle thing, perhaps, but worth noting.

The bottom line is that you will never find a case -- at least that I'm aware of -- where you need `for/in` instead of `for`. Instead, consider it a nice convenience function that can be brought in to make code a little clearer, a little more concise, all without getting into the terse code that gives the reader a headache.

In the big picture, these aren't limitations as much as they are guidelines on when to use `for` and when to use `for/in`. A subtle thing, perhaps, but worth noting.

The bottom line is that you will never find a case -- at least that I'm aware of -- where you need `for/in` instead of `for`. Instead, consider it a nice convenience function that can be brought in to make code a little clearer, a little more concise, all without getting into the terse code that gives the reader a headache.

```java
throws IOException {
    List<String> wordList = new LinkedList<String>();
    // Assign some words
    for (int i=0; i<args.length; i++) {
        wordList.add("word "+(i+1)+": "+args[i]);
    }
    // Remove all words with "1" in them. Impossible with for/in!
    for (Iterator i = wordList.iterator(); i.hasNext(); ) {
        String word = (String)i.next();
        if (word.indexOf("1") != -1) {
            i.remove();
        }
    }
    // You can print the words using for/in
    for (String word : wordList) {
        out.println(word);
    }
}
```
Related topics

- Download Tiger and try it out for yourself.
- The official J2SE 5.0 home page is a comprehensive resource you won't want to miss.
- For specifics on Tiger, see John Zukowski's Taming Tiger series, which offers short tips on the additions and changes in J2SE 5.0.
- Brett McLaughlin has also contributed some articles on new features in Tiger:
  - "Getting started with enumerated type"
- Java 1.5 Tiger: A Developer's Notebook (O'Reilly & Associates; 2004), by Brett McLaughlin and David Flanagan, covers almost all of Tiger's newest features -- including annotations -- in a code-centric, developer-friendly format.

© Copyright IBM Corporation 2004
Trademarks
(www.ibm.com/developerworks/ibm/trademarks/)