for/inでJava 5.0 のループを拡張

この便利な構成体が何をもたらすか、そしてプロジェクトのどのような状況に最も実用的であるかを考察する

Brett D. McLaughlin, Sr.
Author and Editor
O'Reilly Media, Inc.

2004年 11月 30日

しばしばenhanced forまたはforeachと呼ばれるfor/inループは、Java 5.0の多分に便利な機能です。それは特に新規の機能を追加したりはしませんが、確かに幾つかのありきたりなコーディング作業をより簡単にします。この記事では、それがどのようにして不必要な（またはただ単にうっとうしいだけの）型変換(typecasts)を回避するかのみならず、for/inイテレーターを配列とコレクションに対して使用する方法を含め、多くを学ぶことにします。どのようにしてfor/inが実装され、新規のIterableインターフェースに関する詳細を探り出し、そしてさらには使用可能な独自のカスタムのオブジェクトをどのようにして新規の構成体で作成するかを理解できるところまで学習します。最後に、何のひねりもない昔ながらのforが正しい選択肢である状況を理解できるようにするために、for/inに対処『できない』ものについても学ぶことになります。

より短いほうがいい

ハードコアなコンピューター・プログラマーにとって、これは最も原理的な原則です。より短いほうがより少ない型定義につながりますので、より短いほうが本質的によりよいはずなのです。この哲学は、（:wq! や28Gのようなコマンドが豊富な意味を持つ）viのようなIDEの誕生をうながしました。それは遭遇するものの中でも最も謎めいたコード（Agile Runnerを意味するべきであり、ArgyleやAtomic Reactor（原子炉）とも間違って解釈されてしまう、arのような変数）と言う有難くもない結果をもたらしました。

時には、省略の度が過ぎるがために、プログラマーは明確さを（自動車の運転にたとえればバックミラーに映るほどに）『置いてけぼり』にします。それはそれとして、短さへの反動として、痛いほどに冗長なコードを生み出す結果をも招きました。theAtomicReactorLocatedInPhiladelphia と言う名の変数は、arと呼ばれるものと同様にうっとうしくて手にあまります。ほどほどなものがあってもよろしいと思いませんでしょうか？

ここで解るかぎりでは、その『ほどほど』は、（短いから短いのではない）便利な方法を探すことに根源を成します。そのような解決法の良い例として、Java 5.0 はfor ループの新規バージョンを導入し、それはfor/inとして参照されます。これはforeachとも、そして時にはenhanced forとも

呼ばれますが、どちらとも同一の構成体を参照します。それをどう呼ぶにせよ、この記事の用例に見られるとおり、`for/in`は簡素なコード作成に役立ちます。

Iteratorの喪失

`for/in`または「昔ながらのfor」を使用する場合の最も根源的な相違点は、前者を使うのであれば（通常`i`または`count`と呼ばれる）カウンターまたは`Iterator`を必要としないことです。`Iterator`を使う`for`ループを示すリスト1を見て下さい。

リスト1. 古典的なスタイルのforループ

```
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
  }
}
```

注釈: もしもTiger の新機能に関する著者の記事（参考文献を参照）に目を通しているのであれば、（この記事のフォーマットと言うかたちで著者のTiger関連の本からコード・サンプルを発行することを可能にしてくれる）O'Reilly Media, Inc.に感謝しているのはご存知のことでしょう。そのおかげで、（それ無しで提供できるよりも）多くのテストそしてコメントを経験したコードをここにて使えるのです。

もしもこのコードを新規の`for/in`ループに変換する方法の説明が長くなると予測しているのでしたら、おあいにくさまです。リスト2にて、`for/in`をリスト1のループに採用した場合を示しますが、どちらとも驚くほどに互いにそっくりです。可能な限り詳細に渡り、何が起きているのかを説明しますので、とにかく注目してみてください。（もっとも、どんなに詳しく説明しても1パラグラフがやっとでしょうが。）

リスト2. for/inへ変換

```
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
  }
}
```

リスト3にて`for/in`ループの基本構文を示します。仕様を読むのに慣れていないのであればこれは少々奇妙に映るかもしれませんが、ひとつずつ丁寧に理解すれば、これは結構簡単です。

リスト3. 基本的なfor/inループの構造

```
for (declaration :expression)
statement
```

for/inの名前の由来

`for/in`ループが『in』と言う単語と全く関与しないことに、注意深い読者は気付いていることでしょう。この名前は、コードの読み方に由来します。リスト2では、「listと言う名の変数の中（in）にあるそれぞれのObjectにおいて（for）……」と言うでしょう。もちろんですが、省略記号はそのループが何をするかを表記します。これを観察する方法には様々あり

ますが、何にせよ『において（for）』と『中（in）』はそれぞれの表記にて浮き彫りにされています。

declarationは（`Object listElement`のような）ただの変数です。この変数は、繰り返されるリスト、配列、またはコレクションにあるそれぞれの項目との互換性を持つように型定義されるべきです。リスト2の場合、`list`はオブジェクトを含みますので、それが`listElement`の型です。

expressionとは、つまり、式（expression）なのです。それは反復される何かを評価すべきです（それが水面下で何を意味するかを後に説明します）。ここでは、expressionがコレクションまたは配列を評価することを保証するのみです。この式は（リスト2にて示されるとおり）変数や（`getList()`のような）メソッド呼び出しみたいに簡素になると思えば、ブール(boolean)のロジックや三項演算子(ternary operator)と 関与する複雑な式にもなり得ます。それが配列またはコレクションを戻すかぎり、全てがうまく行きます。

statement（ステートメント）とはdeclaration（宣言）にて定義された変数に働きかけるループのコンテントのためのプレースホルダーです。当然ですがこれはループですので、statementは配列またはコレクションにあるそれぞれの項目に応用されます。括弧（`{`と`}`）で囲めば、複数のステートメントを使えるようになります。

要点をまとめて言えば、変数を作成し、繰り返す配列またはコレクションを指示し、そしてそれから定義した変数に式を働きかけさせるだけの話です。リスト内のそれぞれの項目には何も割り当てられないのは、`for/in`がそれを担ってくれるからです。もちろん、これがまだ不明瞭なのであれば、とにかく記事を読み続けてください。豊富な用例がかなり明確にしてくれますでしょう。

とはいえ、次に進む前に、`for/in`がどのようにして機能するかを示す仕様のような説明を提供したいものです。汎用化された型が供給された場合の、実行中の`for/in`ループをリスト4は示します。コンパイラーが一度それを通常の`for`ループに変換すれば、実際にステートメントはこのように映ります。

解りましたでしょうか？コンパイラーはこのより短くより便利な`for/in`ステートメントをコンパイラーに優しい`for`ループに変換します。作業には付き物の過酷な状況とも遭遇しなくて済みます。だからこそ、これはただ単に短いだけではなく、便利だと言えるのです。

リスト4. Iterable付きであり、変換された状態にあるfor/inループ

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

リスト5は別のコンパイラー変換後の`for/in`であり、今回は汎用型(genericized type)無しです。少々簡素ですが、同様のことが進行しています。しかし、いずれにせよ、`for/in`ステートメントを通常の`for`ステートメントに頭の中で（そしてプログラム上で）変換するのは簡単です。頭の中でこの変換を出来るようになれば、あとは嘘みたいに簡単です。

リスト5. 非パラメーター化型(unparameterized type)が無い状態であり、変換されたfor/inループ

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

配列との処理

ここで基本的なセマンティックスを押さえましたので、より具体的な例に取り組む準備ができていることでしょう。リストに対して`for/in`がどのように機能するかをご存知のことだと思います。それは配列に取り掛かるのと同様に簡単です。コレクション同様に、配列は（リスト6にて示されるとおり）値を割り当てられ、それらの値は1つずつおきまりの作業として剥ぎ取られて演算を働きかけられます。

リスト6. 単純な配列の初期化

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

インデックス変数（またはカウンター）そして`for`を使う状況では、（当然、Tigerとともに作業をしていると言う想定のもとで）`for/in`を使うことができます。リスト7はまた別の簡素な例を示します。

リスト7. for/inを使って配列でループするのは楽勝

```
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);
  }
}
```

ここでは特に啓示的なものは無いはずです。これ以上に基本に忠実なものはあり得ません。配列は型定義されていますので、配列内のそれぞれの項目にてどのような変数型が必要とされているかを知っているはずです。この例では（このケースでは`n`と名付けられている）変数を作成し、その変数に演算が作用します。言ったとおりここには特に複雑なものはなく、結構簡単です。

どのような型が配列にあるかは関係ありません。宣言（declaration）に対して正しい型を選ぶだけのことです。リスト8では、配列は`List`を含みます。つまり、実際にコレクションの配列をここにて手に入れたことになります。それでも、`for/in`を使用する方が可能な限り簡素なのです。

リスト8. for/inを使ってオブジェクト配列にてループさせるのも可能

```
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());
  }
}
```

リスト9にて示されるとおり、`for/in`ループにレイヤーを入れることも可能です。

リスト9. for/inの中でfor/inを使うことに何の問題もありません

```
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);
    }
  }
}
```

コレクションとの処理

何度も述べますが、合言葉は『簡素』です。`for/in` を使ってコレクションにてイテレーションを行なうことはトリッキーでも複雑でもありません。リストにせよ配列にせよ、これまでに見てきたとおりです。リスト10は`List` と`Set`での繰り返しの例であり、意外なものが待ち受けています。それでも、コードを実際にチェックして、何が起きているかをはっきりと把握してみてください。

リスト10. このプログラム内にある多数の簡素なプログラムが、どのようにしてfor/inを使用するかを実演

```
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 + " ");
    }
  }
}
```

リスト11はこのプログラムの出力を示しています（実演のためにコマンド行にデータを投げ掛けています。）

リスト11. 予想どおり多くの出力をともないます

```
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
```

型変換(typecast)の痛み

ここまでのところ、コレクションに対処するには、`Object`のような汎用的な変数型とともに使われる`for/in`に注目しました。これはこれで結構なのですが、Tigerにある別の重要な機能（時としてパラメーター化型(parameterized types)と呼ばれる汎用型）を活用しているとは言えません。汎用型に関する詳細については今後発行される予定であるdeveloperWorksが網羅するこの論題のチュートリアルに任せますが、ここで言えるのは汎用型が`for/in`をより強力なものにしてくれると言うことです。

`for/in`ステートメントにあるdeclaration（宣言）の部分が、繰り返されるコレクションにあるそれぞれの項目の型の変数を作成することを忘れないでください。配列においてこれはかなり明確であり、配列は強く型定義されている（`int[]`はint（整数）のみを含めます）ために、ループには型変換はありません。型定義されたリストを汎用型で使用しても同じことができます。リスト12はいくつかの簡素なパラメーター化されたコレクションを示します。

リスト12. コレクション型へのパラメーターの追加は型変換の回避を意味します

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

これで、`for/in`は古くからある`Object` を捨ててより明確になれます（リスト13参照）

リスト13. コレクション内の型が解れば、ループ本体はより型に対して明確になります

```
for (String word : wordlist) {
  System.out.print(word + " ");
}
```

汎用化されたリストとより明確な`for/in`ループでリスト10にあるプログラムを強化して、より完全な例をリスト14として示します。

リスト14. 汎用型を活用するためにリスト10を書き直すこともできます

```
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 + " ");
    }
  }
}
```

もちろん、このケースにて型変換が完全に消えるわけではありません。しかしながら、コンパイラーに処理を移しているのです（興味のある人が興味をいだくことですが、多かれ少なかれそれは汎用型が実行することです）。コンパイル時間にて、これらの型は全てチェックされ、それに応じてエラーが生じます。誰かにこの処理を任せられるのでしたら、それはそれでよいことです。

Eって何様？

tutorial on generics from developerWorks. Javaの熟達者でありながらもTigerの初心者でしたら、ここにあるEへの参照はどれもが不思議に映ることでしょう。これらは全てパラメーター化型（汎用型）へのサポートと関連し、`Iterator` が型定義されたコレクションとともに機能するようにします。例えば、`Iterator<String>` はこのインターフェースの新バージョンとともに機能します。

クラスをfor/inに統合

これまでに、Javaのパッケージ化されたクラスと型（配列、リスト、マップ、セット、そして他のコレクション）のイテレーションのみに対処してきました。それはかなりお得な感じがするのですが、（どのプログラム言語にも見受けられる）美点は独自のクラスを定義できるところにあり

ます。カスタム・オブジェクトはどの大規模アプリケーションのバックボーンをも成します。この章では、独自のオブジェクトが`for/in`構成体に使われるようにするのに関与する概念と手順について説明します。

新規のインターフェース

この時点で既に`java.util.Iterator` インターフェースに慣れ親しんでいるはずです。もしもそうでなければ、そのインターフェースがTigerにてどのように見えるかをリスト15にて示します。

リスト15. Iterator は長年にわたりJava言語の大黒柱を担ってきました

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

`for/in`をより深く活用するには、ドメインに関する知識にまた別のインターフェース（ `java.lang.Iterable` ）を追加すべきです（ リスト16を参照 ）

リスト16. Iterable インターフェースはfor/in構成体の基礎を成します

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

java.lang, not java.util

`Iterable` が`java.util`ではなく`java.lang`にあることに気を付けてください。これがなぜそうなのかに関する明確な文書をまだ見付けていませんが、インターフェースのインポートの必要性を回避するためだからだと思われます。（ 全てのJavaコードに自動的にインポートされたネーム・スペースのセットに、`java.lang` があります。 ）

`for/in` とともにオブジェクトまたはクラスが機能するようにするには、それは`Iterable`インターフェースを実装すべきです。これは2つの基本的なシナリオにつながります。

- `Iterable` をすでに実装している（ つまり、すでに`for/in`をサポートする ）現存するコレクション・クラスを拡張
- 独自の`Iterable`の実装を定義することにより、手動でイテレーションを取り扱う

手動でイテレーションを取り扱う

できることでしたら、カスタム・オブジェクトが現存のコレクションを拡張するようにすることをおすすめします。事は劇的に簡単になり、あらゆる厄介な詳細を回避できます。リスト17はまさにそれをするクラスを示します。

リスト17. 現存するコレクションの拡張は、簡単なfor/inの活用術

```
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;
    }
  }
}
```

`LinkedList` は`for/in`とともにすでに機能しますので、特殊なコード無しでこの新規のクラスを`for/in`とともに使えます。リスト18はその実例を示し、それを実現するのにいかに少ない手間しか要さないかを表わします。

リスト18. Iterable インターフェースはfor/in構成体の基礎を成します

```
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();
    }
  }
}
```

手動でイテレーションを取り扱う

（実を言えばあまり思い付かないのですが）特殊なケースでは、（カスタム・オブジェクトが反復されているのであれば）とある特定の振る舞いを実行すべきかも知れません。これらの（幾分に不幸な）状況では、全てを自身の手でやらなくてはなりません。リスト19にてそれがどのように機能するかを示します（これは複雑な作業と言うよりも重労働です）ので、自身でコードを観察することをおすすめします。反復される場合においてファイルのそれぞれの行をリスト化するテキスト・ファイルへのラッパーを、このクラスは提供します。

リスト19. 忍耐力があればIterable を自身の力で実装し、ループでのカスタムの振る舞いを提供できます

```
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 UnsupportedOperatorException.
 *   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) {
        throw new IllegalArgumentException(e);
      }
    }
    // If the next line is non-null, then we have a next line
    public boolean hasNext() {
      return nextline != null;
    }
    // Return the next line, but first read the line that follows it
    public String next() {
      try {
        String result = nextline;
        // If we haven't reached EOF yet...
        if (nextline != null) {
          nextline = in.readLine();    // Read another line
          if (nextline == null)
            in.close();                // And close on EOF
        }
        // Return the line we read last time through
        return result;
        } catch (IOException e) {
        throw new IllegalArgumentException(e);
      }
    }
    // The file is read-only; we don't allow lines to be removed
    public void remove() {
      throw new UnsupportedOperationException();
    }
  }
  public static void main(String[] args) {
    String filename = "TextFile.java";
```

```
    if (args.length > 0)
      filename = args[0];
      for (String line : new TextFile(filename))
      System.out.println(line);
  }
}
```

ここにある作業の中心を成すのは、`iterator()` メソッドにより戻される`Iterator`の実装です。他はかなり単刀直入です。このとおり、作業を任せられるクラスをただ単に拡張するのにくらべ、`Iterable` インターフェースを手動で実装するのには手間がかかります。

for/inに出来ないこと

どのようによいもの（`for/in`もそのひとつだと思います）にでも制限があります。`for/in`の設定方法（特に`Iterator`の明確な用途を除外すること）から、この新規の構成体ではどうにも出来ないことがあります。

位置決め

最も顕著な制限はカスタム・オブジェクトまたは配列のリストの中での位置を決定する能力の欠落です。復習のために、典型的な`for`ループがどのように使用されるかをリスト20にて示します。リストの中を移動するだけではなく位置を指示するためのインデックス変数の用途に注目してください。

リスト20. 『通常』のforループにてイテレーションの変数（i）を使用

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

これは二次的なおまけでも何でもなく、よくあるプログラミングです。とはいえ、リスト21にて示されるとおり、`for/in`でこのとても簡単なタスクを実行することができません。

リスト21. for/inループにて位置にアクセスするのは不可能

```
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'snot possible to locate position
    for (String word : wordList) {
      out.println(word);
    }
}
```

どのような種類のカウンターの変数（または`Iterator`）もなければ、それが運の尽きです。位置の把握が必要なのであれば、『通常』の`for`に固執してください。位置のまた別の使用法（ストリングとの作業）をリスト22に示します。

リスト22. また別の問題---その名はストリング連結

```
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);
```

項目を除去

また別の制限は項目の除去にあります。リスト23にて示されるとおり、リストのイテレーションの途中で項目を除去するのは不可能です。

リスト23. for/inループにて項目を除去するのは不可能

```
public void removeListItems(PrintStream out, String[] args)
    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);
    }
}
```

総括的な展望では、どのような状況で`for`または`for/in`を使うかを示すガイドラインとしてこれらを捉えれば、それは制限とは言わないでしょう。取るに足らないことかも知れませんが、触れておく価値はあると思います。

要するに、（ここで解っている限りでは）`for/in`を『必要』とする状況には出くわしません。その代わりに、（頭痛の種となるほどに素っ気無いコードと関与せずして）コードを少しでもより明瞭にしてより簡潔にしたい場合に導入できる、使いやすくて利便性に富んだ機能として解釈することができます。

著者について

Brett D. McLaughlin, Sr.

Brett McLaughlin氏は、Logo (小さな三角形を覚えていますか?) の時代からコンピューターの仕事をしています。現在の専門は、JavaおよびJava関連のテクノロジーを使ったアプリケーション・インフラストラクチャーの構築です。ここ数年は、Nextel Communications and Allegiance Telecom, Inc. でこれらのインフラストラクチャーの実装に携わっています。Brett氏は、Javaサーブレットを使ってWebアプリケーション開発のための再利用可能なコンポーネント・アーキテクチャーを構築するJava Apacheプロジェクトturbineの共同設立者の1人です。同氏はまた、オープン・ソースのEJBアプリケーション・サーバーであるEJBossプロジェクトと、オープン・ソースのXML Web公開エンジンであるCocoonにも貢献しています。