This no-nonsense, guide by bestselling Java authors Brett McLaughlin and David Flanagan jumps right into Tiger. Using the task-oriented format of this new series, you'll get complete practical coverage of generics, boxing and unboxing, varargs, enums, annotations, formatting, the for/in loop, concurrency, and more.
The Developer's Notebook Series

So, you've managed to pick this book up. Cool. Really, I'm excited about that! Of course, you may be wondering why these books have the oddlooking, college notebook sort of cover. I mean, this is O'Reilly, right? Where are the animals? And, really, do you need another series? Couldn't this just be a cookbook? How about a nutshell, or one of those cool hacks books that seems to be everywhere? The short answer is that a developer's notebook is none of those things—in fact, it's such an important idea that we came up with an entirely new look and feel, complete with cover, fonts, and even some notes in the margin. This is all a result of trying to get something into your hands you can actually use.

It's my strong belief that while the nineties were characterized by everyone wanting to learn everything (Why not? We all had six-figure incomes from dot-com companies), the new millennium is about information pain. People don't have time (or the income) to read through 600 page books, often learning 200 things, of which only about 4 apply to their current job. It would be much nicer to just sit near one of the ubercoders and look over his shoulder, wouldn't it? To ask the guys that are neck-deep in this stuff why they chose a particular method, how they performed this one tricky task, or how they avoided that threading issue when working with piped streams. The thinking has always been that books can't serve that particular need—they can inform, and let you decide, but ultimately a coder's mind was something that couldn't really be captured on a piece of paper.

This series says that assumption is patently wrong—and we aim to prove it.

A Developer's Notebook is just what it claims to be: the often-frantic scribbling and notes that a true-blue alpha geek mentally makes when working with a new language, API, or project. It's the no-nonsense code that solves problems, stripped of the time-consuming commentary that often serves more as a paperweight than an epiphany. It's code you've read and used not on what is nifty or might be fun to do when you've got some free time (which's the last time that happened?), but on what you need to simply "make it work." This is "just-do-it," folk. It's a lab. If you want a lot of concept, architecture, and UML diagrams, then happily provide them to you in our animal and nutshell books. If you want every answer to every problem under the sun, our omnibus cookbooks are killer. And if you are into arcane and often quirky uses of technology, hacks books simply rock. But if you're a coder, down to your core, and you just want to get on with it, then you want a Developer's Notebook. Coffee stains and all, this is from the mind of a developer to yours, barely even cleaned up enough for print. I hope you enjoy it...we sure had a good time writing them.

Notebooks Are...

Example-driven guides

As you'll see in Organization section, developer's notebooks are built entirely around example code. You'll see code on nearly every page, and it's code that does something—not trivial "Hello World!" programs that aren't worth more than the paper they're printed on.

Aimed at developers

Ever read a book that seems to be aimed at pointy-haired bosses, filled with buzzwords, and...
feels more like a marketing manifesto than a programming text? We have too—and these books are the antithesis of that. In fact, a good notebook is incomprehensible to someone who can't program (don't say we didn't warn you!), and that's just the way it's supposed to be. But for developers...it's as good as it gets.

Actually enjoyable to work through

Do you really have time to sit around reading something that isn't any fun? If you do, then maybe you're into thousand-page language references—but if you're like the rest of us, notebooks are a much better fit. Practical code samples, terse dialogue centered around practical examples, and even some humor here and there—these are the ingredients of a good developer's notebook.

About doing, not talking about doing

If you want to read a book late at night without a computer nearby, these books might not be that useful. The intent is that you're coding as you go along, knee deep in bytecode. For that reason, notebooks talk code, code, code. Fire up your editor before digging in.

Notebooks Aren't...

Lectures

We don't let just anyone write a developer's notebook—you've got to be a bona fide programmer, and preferably one who stays up all night coding. While full-time writers, academics, and theorists are great in some areas, these books are about programming in the trenches, so are filled with instruction, not lecture.

Filled with conceptual drawings and class hierarchies

This isn't a nutshell (there, we said it). You won't find 100-page indices with every method listed, and you won't see full-page UML diagrams with methods, inheritance trees, and flow charts. What you will find is page after page of source code. Are you starting to sense a recurring theme?

Long on explanation, light on application

It seems that many programming books these days have three, four, or more chapters before you even see any working code. I'm not sure who has authors convinced that it's good to keep a reader waiting this long, but it's not anybody working on this series. We believe that if you're not coding within ten pages, something's wrong. These books are also chock-full of practical application, taking you from an example in a book to putting things to work on your job, as quickly as possible.
Organization

Developer's Notebooks try to communicate different information than most books, and as a result, are organized differently. They do indeed have chapters, but that's about as far as the similarity between a notebook and a traditional programming book goes. First, you'll find that all the headings in each chapter are organized around a specific task. You'll note that we said task, not concept. That's one of the important things to get about these books—they are first and foremost about doing something. Each of these headings represents a single lab. A lab is just what it sounds like—steps to accomplish a specific goal. In fact, that's the first heading you'll see under each lab: "How do I do that?" This is the central question of each lab, and you'll find lots of down-and-dirty code and detail in these sections.

Some labs have some things not to do (ever played around with potassium in high school chemistry?), helping you avoid common pitfalls. Some labs give you a good reason for caring about the topic in the first place; we call this the "Why do I care?" section, for obvious reasons. For those times when code samples don't clearly communicate what's going on, you'll find a "What just happened" section. It's in these sections that you'll find concepts and theory—but even then, they are tightly focused on the task at hand, not explanation for the sake of page count. Finally, many labs offer alternatives, and address common questions about different approaches to similar problems. These are the "What about..." sections, which will help give each task some context within the programming big picture.

And one last thing—on many pages, you'll find notes scrawled in the margins of the page. These aren't for decoration; they contain tips, tricks, insights from the developers of a product, and sometimes even a little humor, just to keep you going. These notes represent part of our overall communication flow—getting you as close to reading the mind of the developer-author as we can. Hopefully they'll get you that much closer to feeling like you are indeed learning from a master.

And most of all, remember—these books are...

All Lab, No Lecture

—Brett McLaughlin, Series Creator
The first thing to realize is that proper use of a Queue implementation is to avoid the standard collection methods `add()` and `remove()`. Instead, you'll need to use `offer()` to add elements. Keep in mind that most queues have a fixed size. If you call `add()` on a full queue, an unchecked exception is thrown—which really isn't appropriate, as a queue being full is a normal condition, not an exceptional one. `offer()` simply returns `false` if an element cannot be added, which is more in line with standard queue usage.

In the same vein, `remove()` throws an exception if the queue is empty; a better choice is the new `poll()` method, which returns `null` if there is nothing in the queue. Both methods attempt to remove elements from the head of the queue. If you want the head without removing it, use `element()` or `peek()`. Example 1-2 shows these methods in action.

**Example 1-2. Using the Queue interface**

```java
package com.oreilly.tiger.ch01;

import java.io.IOException;
import java.io.PrintStream;
import java.util.LinkedList;
import java.util.Queue;

public class QueueTester {

    public Queue q;

    public QueueTester( ) {
        q = new LinkedList( );
    }

    public void testFIFO(PrintStream out) throws IOException {
        q.add("First");
        q.add("Second");
        q.add("Third");

        Object o;
        while (((o = q.poll( )) != null) { 
            out.println(o);
        }
```
2.2.2 What just happened?

As briefly mentioned in Using Type-Safe Lists, autoboxing helps when you want to stuff primitives into a collection. In this case, even though the Map is defined to take Integers, it's the int counter i that is used to create values. Without getting into the details covered in Chapter 4, Java autoboxes the int value of i into an Integer, behind the scenes, meeting the requirements of the squares Map.

2.3 Iterating Over Parameterized Types

Although the for/in loop provides a means of almost completely avoiding the java.util.Iterator class, that particular feature of Tiger isn't covered until Chapter 7. But until you get to that chapter (and probably occasionally after that), it's still useful to know how generic collection types affect Iterator. You'll need to perform an extra step to get the full power of generics.

2.3.1 How do I do that?

It would seem that once you've parameterized your collections, grabbing an Iterator and using it would be trivial:

```java
List<String> listOfStrings = new LinkedList<String>();
listOfStrings.add("Happy");
listOfStrings.add("Birthday");
listOfStrings.add("To");
listOfStrings.add("You");

for (Iterator i = listOfStrings.iterator(); i.hasNext(); ) {
    String s = i.next();
    out.println(s);
}
```

However, all is not well. Here's what the compiler spits back to you:

```
[javac] code\src\com\oreilly\tiger\ch02\GenericsTester.java:54:
    incompatible types
[javac] found    : java.lang.Object
[javac] required: java.lang.String
[javac]         String s = i.next();
[javac]             ^
```

Preview from Notesale.co.uk
Page 42 of 231
2.4 Accepting Parameterized Types as Arguments

So far, all of this parameterization has occurred in the same code block. However, that's unrealistic, and you'll quickly want to write methods that take advantage of parameterized types. This is where generics start to really become powerful. First, you need to understand how a method can tell the compiler that it only accepts a specific parameterization of a generic type.

2.4.1 How do I do that?

Just use the same syntax you've been using (and which should be getting oddly comfortable by this point) in your argument list:

```java
private void printListOfStrings(List<String> list, PrintStream out)
    throws IOException {

    for (Iterator<String> i = list.iterator(); i.hasNext(); ) {
        out.println(i.next());
    }
}
```

This allows your method body to act on that parameterization, avoiding class casts and the like. In this example, it's possible to parameterize the `Iterator` as well, because the compiler ensures that only `List<String>` is passed into the method. Any other `List` types are refused (at compiletime).

2.4.2 What about...

...trying to pass in a plain old `List`, without any parameterization, even if it has only `Strings` in it? This actually will work, with the caveat that you're left to your own devices in ensuring that the `List` has in it what it's supposed to. If not, you'll get more `ClassCastException` than you can shake a stick at, all at runtime. In either case, you'll get `lint` warnings, which are described in "Checking for Lint," later in this chapter.

2.5 Returning Parameterized Types

In addition to accepting parameterized types as arguments, methods in Tiger can return types that are parameterized.

2.5.1 How do I do that?

Remember the `getListOfStrings()` method, referred to in "Using Type-Safe Lists"? Here is the actual code for that method:

```java
private List getListOfStrings() {
    List list = new LinkedList();
```
effects of erasure, at compile-time (Example 2-1):

**Example 2-1. Breaking type safety with reflection**

```java
package com.oreilly.tiger.ch02;

import java.util.ArrayList;
import java.util.List;

public class BadIdea {

    private static List<Integer> ints = new ArrayList<Integer>();

    public static void fillList(List<Integer> list) {
        for (Integer i : list) {
            ints.add(i);
        }
    }

    public static void printList() {
        for (Integer i : ints) {
            System.out.println(i);
        }
    }

    public static void main(String[] args) {
        List<Integer> myInts = new ArrayList<Integer>();
        myInts.add(1);
        myInts.add(2);
        myInts.add(3);

        System.out.println("Filling list and printing in normal way...");
        fillList(myInts);
        printList();
    }
}
```
Chapter 3. Enumerated Types

NOTE

In this chapter:

- Creating an Enum
- Declaring Enums Inline
- Iterating Over Enums
- Switching on Enums
- Maps of Enums
- Sets of Enums
- Adding Methods to an Enum
- Implementing Interfaces with Enums
- Value-Specific Class Bodies
- Manually Defining an Enum
- Extending an Enum

In Java 1.4 and below, there were two basic ways to define new types: through classes and interfaces. For most object-oriented programming, this would seem to be enough. The problem is that there are still some very specific cases where neither is these is sufficient, most commonly when you need to define a finite set of allowed values for a specific data type. For instance, you might want a type called Grade that can only be assigned values of A, B, C, D, F, or Incomplete. Any other values are illegal for this type. This sort of construct is possible prior to Tiger, but it takes a lot of work, and there are still some significant problems.

Since we're good developers and try our best to avoid a lot of work whenever possible, Sun finally helped us out with the new enumerated type (generally referred to simply as an enum). This chapter deals with enums: how to create, use, and program with them.

3.1 Creating an Enum

Creating an enumerated type involves three basic components, at a minimum:

- The `enum` keyword
- A name for the new type
private String firstName;
private String lastName;
private Grade grade;

public Student(String firstName, String lastName) {
    this.firstName = firstName;
    this.lastName = lastName;
}

public void setFirstName(String firstName) {
    this.firstName = firstName;
}

public String getFirstName() {
    return firstName;
}

public void setLastName(String lastName) {
    this.lastName = lastName;
}

public String getLastName() {
    return lastName;
}

public String getFullName() {
    return new StringBuffer(firstName)
        .append(" ")
        .append(lastName)
        .toString();
}

Chapter 3. Enumerated Types
case D: // fall through to F

case F:
    outputText.append(" failed with a grade of ")
        .append(student1.getGrade().toString());
    break;

case INCOMPLETE:
    outputText.append(" did not complete the class.");
    break;
}

out.println(outputText.toString());

NOTE

This code assumes that student1 has already been created; this is taken care of in the test class, "GradeTester".

The argument to switch must be an enumerated value; in this case, the return type of getGrade() is Grade, which meets these requirements. However, there is another requirement that makes this code a little odd—did you catch it? Note the format of each case clause:

    case A:
    case B:
    case C:
    case D:
    case F:
    case INCOMPLETE:

See anything missing? How about the enum class identifier:

    case Grade.A:
    case Grade.B:
    case Grade.C:
    case Grade.D:
    case Grade.F:
With this initial work done (you could represent every guitar in a line this way), you can test a specific guitar for a specific features, using the bitwise AND operator:

   boolean hasAbRosette = (bourgeoisD150 & OldGuitarFeatures.IL_DIAMONDS) != 0;

Looking at the constants in OldGuitarFeatures, you should see that they are just another case of an enumerated type, and could be represented in Tiger as shown in Example 3-6.

Example 3-6. Representing guitar features in Tiger

   package com.oreilly.tiger.ch03;

   public enum GuitarFeatures {
      ROSEWOOD,  // back/sides
      MAHOGANY,  // back/sides
      ZIRICOTE,  // back/sides
      SPRUCE,    // top
      CEDAR,     // top
      AB_ROSETTE,  // abalone rosette
      AB_TOP_BORDER, // abalone top border
      IL_DIAMONDS, // diamond/square inlay
      IL_DOTS     // dots inlays
   }

NOTE

Be sure and continue to compile with the "-source 1.5" switch. Using the provided Ant scripts takes care of this, by the way.
SPRUCE(0), // top
CEDAR(0), // top

AB_ROSETTE(75), // abalone rosette
AB_TOP_BORDER(400), // abalone top border

IL_DIAMONDS(150), // diamond/square inlay
IL_DOTS(0); // dots inlays

/** The upcharge for the feature */
private float upcharge;

private GuitarFeatures(float upcharge) {
  this.upcharge = upcharge;
}

// Other method bodies

This compiles, but it just explicitly does what the compiler takes care of for you—making the constructor private. However, you cannot supply the standard `public` modifier:

```java
public GuitarFeatures(float upcharge) {
  this.upcharge = upcharge;
}
```

If you try this, you'll get a compiler error:

```
[javac] src\ch03\GuitarFeatures.java:21:
  modifier public not allowed here
[javac]   public GuitarFeatures(float upcharge) {
[javac]     ^
```

Preview from Notesale.co.uk
Page 86 of 231
You can now dispense with the manual conversions, and let the Java virtual machine (VM) handle conversion of primitives to object wrapper types:

```java
    Integer i = 0;
```

In the background, Java handles taking this primitive and turning it into a wrapper type. The same conversion happens with explicit primitive types:

```java
    int foo = 0;
    Integer integer = foo;
```

If you're not completely convinced of the value of this, try typing these statements into a pre-Tiger compiler, and watch in amazement as you get some rather ridiculous errors:

NOTE

The "compile-1.4" target compiles the examples from this chapter with the "-source 1.4" switch.

```
compile-1.4:

[echo] Compiling all Java files...
[javac] Compiling 1 source file to classes
[javac] src\com\oreilly\tiger\ch04\ConversionTester.java:6: incompatible types
[javac] found    : int
[javac] required: java.lang.Integer
[javac]           Integer i = 0;
[javac]                        ^
[javac] src\com\oreilly\tiger\ch04\ConversionTester.java:9: incompatible types
[javac] found    : int
[javac] required: java.lang.Integer
[javac]   Integer integer = foo;
[javac]                        ^
[javac] 2 errors
```

These errors "magically" disappear in Tiger when using the "-source 1.5" switch.

4.1.2 What just happened?

Behind the scenes, these primitive values are boxed. Boxing refers to the conversion from a primitive to its corresponding wrapper type: `Boolean, Byte, Short, Character, Integer, Long, Float, or Double`. Because this happens automatically, it's generally referred to as autoboxing.
the following code is legal:

```java
Integer i = null;
int j = i;
```

i is assigned null (which is legal), and then i is unboxed into j. However, null isn’t a legal value for a primitive, so this code throws a NullPointerException.

### 4.3 Incrementing and Decrementing Wrapper Types

When you begin to think about the implications of boxing and unboxing, you'll realize that they are far-reaching. Suddenly, every operation available to a primitive should be available to its wrapper-type counterpart, and vice versa. One of the immediate applications is the increment and decrement operations: ++ and --. Both of these operations now work for wrapper types.

#### 4.3.1 How do I do that?

Well, without much work, actually:

```java
Integer counter = 1;
while (true) {
    System.out.printf("Iteration %d\n", counter++);
    if (counter > 1000) break;
}
```

The variable counter is treated just as an int in this code.

#### 4.3.2 What just happened?

It’s worth noting that more happened here than perhaps meets the eye. Take this simple portion of the example code:

```
counter++
```

Remember that counter is an Integer. So the value in counter was first auto-unboxed into an int, as that’s the type required for the ++ operator.

This is actually an important point—the ++ operator has not been changed to work with object wrapper types—it’s only through autounboxing that this code works.

Once the value is unboxed, it is incremented. Then, the new value has to be stored back in counter, which requires a boxing operation. All this in a fraction of a second!
Example 5-2. Storing variable-length arguments as member variables

```java
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) {
```
I realize that I really haven’t told you how these work—that’s intentional. Now that you’ve seen each annotation in use, you should realize that each has an entirely different syntax. To get a handle on annotations, we’ll have to delve into just a bit of theory, and then there are labs going into each of the standard annotations in detail. Buckle up for a moment, let’s deal with some technical details, and then you’ll be ready for some more practical instruction.

6.1.2 What just happened?

First, you need to understand the difference between an annotation and an annotation type. Taking the last part first, an annotation type is a specific name of an annotation, along with any default values and related information. You just saw three annotation types: Override, Deprecated, and SuppressWarnings. An annotation, then, uses an annotation type to associate some piece of information with a Java program element (methods, classes, variables, etc.). So your code might only use one annotation type, like Override, and yet have ten or fifteen annotations (if it used Override ten or fifteen times).

Annotation types can have values as well—note that SuppressWarnings passed in the valued "unchecked", which would be used by the annotation type to process or store information. This passed-in value, along with any default values, all make up the annotation's members. This is where the name=value syntax comes in that I mentioned in the early part of this lab—it allows for an annotation's members to be set.

In addition to the three standard annotation types, there are three categories of annotations:

- **Marker annotations**
  
  Marker annotations are used with annotation types that define no members. They simply provide information contained within the name of the annotation itself. An annotation whose members all have default values can also be used as a marker, since no information must be passed in. The syntax of a marker is simply:
  
  ```
  @MarkerAnnotation
  ```

- **Single-value annotations**
  
  Single-value annotations have just a single member, named value. The format of a single-value annotation is:
  
  ```
  @SingleValueAnnotation("some value")
  ```

- **Full annotations**
  
  A full annotation really isn't a category, as much as it is an annotation type that uses the full range of annotation syntax. Here parentheses follow the annotation name, and all members are assigned values:
  
  ```
  @Reviews({ // Curly braces indicate an array of values
    @Review(grade=Review.Grade.EXCELLENT, reviewer="df"),
  })
  ```
source code of this book. Pull up those API docs, and navigate to the com.oreilly.tiger.ch06 package, and then the AnnotationTester class. Scroll down, if needed, to the calculateInterest() method—you should see something similar to Figure 6-1.

Figure 6-1. calculateInterest() without documentation of annotations

Nothing special, right? Right—but there's something missing. Remember the code for this method:

```java
@com.oreilly.tiger.ch06.InProgress
@GroupTODO(
    severity=GroupTODO.Severity.CRITICAL,
    item="Figure out the amount of interest per month",
    assignedTo="Brett McLaughlin",
    dateAssigned="04-26-2004"
)
public void calculateInterest(float amount, float rate) {
    // Need to finish this method later
}
```

While the source code contains some pretty important information, in the form of the InProgress and GroupTODO annotations, this information missing from the Javadoc.

To fix this, you need to add a @Documented meta-annotation to any annotation type that you want to appear in Javadoc. In this case, both InProgress and GroupTODO, as well as TODO, could use this addition. Example 6-12 shows an updated InProgress; you can add the same lines to the other annotation types.

Example 6-12. Adding documentation to InProgress
package com.oreilly.tiger.ch06;

import java.lang.annotation.Documented;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;

/**
 * Marker annotation to indicate that a method or class
 * is still in progress.
 */
@Documented
@Retention(RetentionPolicy.RUNTIME)
public @interface InProgress { }

I've also added the Retention meta-annotation to this class; anytime you use the Documented annotation, you should pair it with a retention policy of RetentionPolicy.RUNTIME. Make this same change to the other annotation types defined in com.oreilly.tiger.ch03.

Clean out your old Javadoc files. Now, recompile your classes and run the Javadoc generator again. This time, you'll get a bit different output—navigate again to com.oreilly.tiger.ch06, and then AnnotationTester, and finally to the calculateInterest() method. Figure 6-2 shows the same method, but this time the annotations show up in the Javadoc.

NOTE

"ant clean" will remove all compiled class files, and Javadoc files, allowing a clean generation of Javadoc.

Figure 6-2. Javadoc with annotations showing up
6.10.2 What about...

...interfaces? Annotations marked as Inherited only apply to subclasses, so implementing a method with annotations will not result in the annotations being inherited.

Additionally, if you override a method from a superclass, you don't inherit the specific method's annotations. Only the annotations on the superclass itself are inherited, and those by the subclass as a whole.

Yes, that's a mouthful. Stop, take three breaths, and read it again. It will start to make sense—I promise.

6.11 Reflecting on Annotations

So far, all the discussions on annotations have been around looking at them visually—either in source code or in Javadoc. However, there are enough code introspection tools these days that it's worth talking about using reflection to determine what annotations a class (or field, or method) has. The java.lang.reflect package has several additions that make this a piece of cake.

6.11.1 How do I do that?

The easiest way to check for an annotation is by using the isAnnotationPresent( ) method. This lets you specify the annotation to check for, and get a true/false result:

```java
public void testAnnotationPresent(PrintStream out) throws IOException {
    Class c = Super.class;
```
retention. So, one of the most valuable annotations, the one indicating deprecation, is undetectable through Java reflection.
for (Iterator i = list.iterator(); i.hasNext(); ) {
    Object listElement = i.next();
    out.println(listElement.toString());

    // Do something else with this list object
}

NOTE

All the unnamed samples in this chapter are in the class com.oreilly.tiger.ch07.ForInTester. Remember to compile using the "-source 1.5" switch with "javac".

This is a perfect example of Iterator simply being the means of getting at objects in the list, rather than providing real value to the loop. for/in allows this loop to be rewritten:

```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
    }
}
```

NOTE

This removal of Iterator has some consequences—see the later labs in the chapter for details.

Notice that the line `Object listElement = i.next();` from the first code sample has disappeared. This is the basis of what for/in does—it removes the Iterator from the process.

7.1.2 What just happened?

For those of you into specifications and language structure, the loop is structured like this:
List<String> wordlist = new ArrayList<String>();
Set<String> wordset = new HashSet<String>();

Then write your code normally, except substitute a type-specific variable instead of a generic Object and remove any typecasts:

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

Example 7-2 is a generics version of Example 7-1.

Example 7-2. Using the for/in loop with generics

package com.oreilly.tiger.ch07;

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

public class ForInGenericsDemo {

    public static void main(String[] args) {

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

        // We start with a basic loop 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 + ");
    }
This class doesn't do much in terms of customization—it does require that only `String` be allowed as a parameter (through the extends `LinkedList<String>` declaration), and that values passed into the `add()` method have "Guitars" as part of their value. This is a rather hackish way to ensure manufacturers are supplied, but it's useful for an illustration.

You can now use this class as shown in Example 7-6. The example creates a new instance of `GuitarManufacturerList`, seeds it with some sample data, and then uses for/in to iterate over it. With essentially no work on your part, you get the benefit of iteration from the superclass, `LinkedList`.

**Example 7-6. Iterating over GuitarManufacturerList**

```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();
    }
}

7.5.1.2 Handling iteration manually

In cases where you're not extending an existing collection class, you've got a little more work to do. Still, you'll usually find yourself borrowing at least some behavior from existing collection classes, and avoiding direct implementation of Iterator. Example 7-7 shows a simple text file reader that lists the lines of a file when iterated over.

NOTE

If you don't pass in "LinkedList <String>" here, and just use "LinkedList", you'll get compiler warnings indicating a possible type mismatch.

Example 7-7. Custom class that doesn't extend a collection

package com.oreilly.tiger.ch07;

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

NOTE

This code sample is from Java in a Nutshell, Fifth Edition (O'Reilly).

The interesting work is in the TextFileIterator class, which handles all the work of iteration. The first thing to notice is that this iteration is completely read-only—remove simply throws an UnsupportedOperationException. This is a perfectly legal and useful means of ensuring that programmers understand the use-case your custom classes are designed for. I'll leave you to work through the rest of the details; the source code is pretty self-explanatory.

7.6 Determining List Position and Variable Value

for/in, more than anything else, is about convenience. However, with that convenience comes a degree of lost flexibility. One such example is the inability to determine the position in a list that the for/in construct resides at. As your code executes within the for/in loop, there is no way to access the position in the list. Additionally, the list variable itself isn't accessible, making its access equally impossible.

7.6.1 How do I do that?

In short, you don't. Sometimes it's just as important to realize what you can't do as it is to learn what you can.

A common iteration technique is to use the loop variable, especially if it's numerical, in the loop body itself:
In particular, notice the second for loop, which uses remove() to yank any words in the List with "1" in the text. This depends on the usage of Iterator, which isn't available in for/in, so you're out of luck here.

7.7.2 What about...

...anything else that involves knowing where you are in the list? Nope— it's not possible. Here are just a few examples of other things you can't do with for/in:

1. Iterate backward through the elements of a List.

2. Use a single loop counter to access similarly numbered or positioned elements in two distinct arrays or collections.

3. Iterate through the elements of a List using calls to get() rather than calls to its iterator.

Watch out for this last one—it is a legitimate performance concern. For the java.util.ArrayList class, for example, looping with the size() and get() methods is measurably faster than using the list's Iterator. In many cases, the performance difference is negligible and irrelevant. But, when writing an inner loop or other performance-critical code, you might prefer to use a for loop with the get() method instead of a for/in loop.
Note that Arrays.sort() is a heavily overloaded method. This import directive imports the method by its name, and not any particular overloading of it.

```java
import static java.util.Arrays.sort;
```

You can now just code as shown here:

```java
sort(myObjectArray); // no need to use Arrays.sort()
```

This is a particularly nice improvement to the language, I think.

### 8.1.2 What about...

...static member types? A member type is a type defined within another class. For example, the `java.lang.Character` class defines a class within its body, called `Subset`. The formal name of this class, because it does not stand on its own, is `Character.Subset` (and it appears that way within the JavaDocs). You can import a member type, in Tiger and in previous versions, with a normal import statement:

```java
import java.lang.Character.Subset;
```

This is a little-known, little-used feature of Java that has nothing to do with Tiger. However, where Tiger does come in is adding the ability to do static imports. In the case of `Character.Subset`, not only is `Subset` a member type, it's a static member type—meaning that it could also be imported as seen here:

**NOTE**

Anonymous inner classes defined within a method are not members, and cannot be imported.

```java
import static java.lang.Character.Subset;
```

So, which is correct? Actually, both. However, you shouldn't have both in the same code—just one or the other. Otherwise, compilers will gripe about a name conflict (two identically named types). Personally, I like the clarity of `import static` for this sort of thing, as it reminds me that `Character.Subset` is static—but that's style more than any real best practice.

### 8.2 Using Wildcards in Static Imports

While importing a single member is a nice addition, there are times when you may want to import a lot of members. In these cases, you could easily fill a page with all the `import static` declarations you'd need. Luckily, wildcards work perfectly well with static imports.

#### 8.2.1 How do I do that?

Piece of cake...just use a wildcard, as you would with normal `import` statements:
import static java.lang.Math.*;

Now you can use expressions like the following in your code:

float foo = sqrt(abs(sin(bar)));

Again, nothing flashy here, but well worth knowing. It's possible to import anything declared as static into the Java namespace. However, you couldn't do something like this:

import static java.lang.System.out.println;

That's because while out is static, the method println() is not. Be careful to keep your static and non-static items straight.

NOTE

println() is an instance method of the PrintStream class.

8.2.2 What about...

...all the code legibility lost by doing this sort of thing? It's certainly possible to go crazy with imports, and lose all track of which methods belong to which class. Of course, this comes from someone who almost never uses wildcards in normal import statements:

import java.io.IOException;
import java.io.PrintWriter;
import javax.servlet.ServletException;
import javax.servlet.GenericServlet;
// etc.

Personally, I like it to be obvious what classes are used, as opposed to dropping an import java.io.* statement into code. But that's a stylistic decision, not a functional one, and now Tiger lets you keep whatever preference you choose.

As a best practice, though, I'd recommend you only use static imports if you were going to use a static member more than three times. In other words, there's value in the clarity of code that reads Math.sqrt() as opposed to just sqrt(), when that method is only used once in the entire program. However, if you're using the method fifty times, then it's just as clear to add a static import for the method and then use it without the prefix.

8.3 Importing Enumerated Type Values

The next natural thought in working with static imports is to use them in conjunction with another Tiger feature, enumerated values (detailed in Chapter 3). Since the compiler declares enumerated
Just like that, you're all set, and you get even more overloading. Now the compiler will actually decide, based on your arguments, which method, and even which class, to use to process your method call. Pretty cool!

NOTE

As cool as this is, it really increases the work that someone on the debugging or testing team has to do to figure out what's going on in your code.

8.4.2 What about...

...naming conflicts? As already stated, you'll get a compiler error if you import two types with the same name into your program. So the following lines would be illegal in a source listing:

```java
import java.util.Arrays;
import com.oreilly.tiger.ch08.Arrays;
```

You'd get the following error:

```
[javac] src\ch08\SortImporter.java:4:
java.util.Arrays is already defined in a single-type import
[javac] import com.oreilly.tiger.ch08.Arrays;
[javac] ^
```

When it comes to naming conflicts among imported static methods, though, things are a little more obscure. I've created a simple class shown in Example 8-3, that defines a method of `sort()` with the exact same arguments as one of the `sort()` methods in `java.util.Arrays`.

**Example 8-3. Setting up a namespace conflict**

```java
package com.oreilly.tiger.ch08;

public class Arrays {

    public static void sort(float[] a) {
        // Do nothing
        // This is just used to illustrate some naming conflicts
    }
}
```

Now, add you can create a naming conflict with the following lines in another class:
package com.oreilly.tiger.ch08;

import static java.lang.System.err;
import static java.lang.System.out;
import java.io.IOException;
import java.io.PrintStream;

public class StaticImporter {

    public static void writeError(PrintStream err, String msg) throws IOException {
        // Note that err in the parameter list overshadows the imported err
        err.println(msg);
    }

    public static void main(String[] args) {
        if (args.length < 2) {
            err.println("Incorrect usage: java com.oreilly.tiger.ch08 [arg1] [arg2]");
            return;
        }
        out.println("Good morning, " + args[0]);
        out.println("Have a " + args[1] + " day!");

        try {
            writeError(System.out, "Error occurred.");
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
public void testQueue(PrintStream out) throws IOException {
    BlockingQueue queue = new LinkedBlockingQueue(10);
    Producer p = new Producer(queue, out);
    Consumer c1 = new Consumer("Consumer 1", queue, out);
    Consumer c2 = new Consumer("Consumer 2", queue, out);
    Consumer c3 = new Consumer("Consumer 3", queue, out);
    Consumer c4 = new Consumer("Consumer 4", queue, out);

    p.start(); c1.start(); c2.start(); c3.start(); c4.start();
    while (true) {
        // hang out for a while
    }
}

NOTE

You can test out this method by running "ant runch10". It will run forever, though, so you'll have to break out of the program.

You'll see tons of output, as the producer fills the queue and the consumers grab information out of it. What's cool, though, is that the processing cycles through the four consumers, in order:

[java] Consumer 1 processing object:
[java] 'Inserted at Tue May 04 08:43:50 GMT-06:00 2004'
[java] Consumer 2 processing object:
[java] 'Inserted at Tue May 04 08:43:50 GMT-06:00 2004'
[java] Consumer 3 processing object:
[java] 'Inserted at Tue May 04 08:43:50 GMT-06:00 2004'
[java] Consumer 4 processing object:
[java] 'Inserted at Tue May 04 08:43:50 GMT-06:00 2004'
[java] Consumer 1 processing object:
[java] 'Inserted at Tue May 04 08:43:50 GMT-06:00 2004'
[java] Consumer 2 processing object:
[java] 'Inserted at Tue May 04 08:43:50 GMT-06:00 2004'
[java] Consumer 3 processing object:
Fixed thread executor

Obtained with `Executors.newFixedThreadPool(int poolSize)`, this creates an executor with the specified number of threads to run the tasks with which you supply it.

Cached thread executor

This executor, obtained with `Executors.newCachedThreadPool()`, will use as many threads as it needs to run the objects in its queue. It will reuse threads as they become available, as well as create new threads.

Scheduled thread executor

This executor is obtained with `Executors.newScheduledThreadPool()`, or `Executors.newSingleThreadScheduledExecutor()`, and is detailed in Executing Tasks Without an ExecutorService.

So what happens when you have just one little Callable object that you want to execute, and you don't need the overhead of ExectorService? Well, it seems those Sun guys thought of everything (except perhaps open-sourcing Java)—you use FutureTask.

10.5.2 How do I do that?

`java.util.concurrent.FutureTask` can be wrapped around Callable objects, allowing them to behave like a Future implementation returned from `ExecutorService.submit()`. The syntax is similar as well:

```java
FutureTask<BigInteger> task =
    new FutureTask<>(new RandomPrimeSearch(512));

new Thread(task).start();

BigInteger result = task.get();
```

The methods available to `FutureTask` are similar to `Future`, so I'll leave it to you to check out the Javadoc. With the details from Using Callable Objects, you shouldn't have any problems.

10.5.3 What about...

...good old Runnable? Fortunately, plain old Thread and Runnable didn't get left out of the mix. You can wrap a FutureTask around a Runnable object just as easily as you can around a Callable object, and the same functionality applies. However, since Runnable's `run()` method doesn't return a result, the constructor is a bit different:

```java
FutureTask<String> task =
    new FutureTask<String>(new MyRunnableObject, "Success!");
```
try {
    BigInteger bigger = (prime1.get().multiply(prime2.get())).
    multiply(prime3.get());
    out.println(bigger);
} catch (InterruptedException e) {
    e.printStackTrace(out);
} catch (ExecutionException e) {
    e.printStackTrace(out);
}

The other methods are typically useful when you have more of a daemon and need to cleanly cancel or shut down tasks yet to be executed.

NOTE

You don't need to call isDone() and then call get(); get() will wait for a completed result before returning.

10.8 Executing Tasks Without an ExecutorService

So what happens when you just have one little Callable object that you want to execute, and you don't need the overhead of ExecutorService? Well, it seems those Sun guys thought of everything (except perhaps open sourcing java) — you use FutureTask.

10.8.1 How do I do that?

java.util.concurrent.FutureTask can be wrapped around Callable objects, allowing them to behave like a Future implementation returned from ExecutorService.submit(). The syntax is similar as well:

```java
FutureTask<BigInteger> task =
    new FutureTask<BigInteger>(new RandomPrimeSearch(512));

new Thread(task).start();

BigInteger result = task.get();
```

The methods available to FutureTask are similar to Future, so I'll leave it to you to check out the Javadoc. With the details from Using Callable Objects, you shouldn't have any problems.
types, revolving around Boolean, Long, Integer, and object references. This allows you to perform atomic operations on these types, using AtomicBoolean, AtomicLong, AtomicInteger, and AtomicReference, respectively.

Each type provides a get() and set() method, which do what you would expect (get and set the type’s value, using an atomic operation). They also offer getAndSet(), which sets the value, returning the previous value, as well as compareAndSet(), which checks the value, and if it matches the supplied value, sets it to a new value. Additionally, AtomicInteger and AtomicLong provide for atomic versions of ++ and --, through variations on decrement() and increment() methods. For example, decrementAndGet() decrements the value of the atomic type, and returns the update value; getAndIncrement() returns the current value, and then increments it in the type.

Here are several different ways to write a thread-safe counter, lifted straight out of Java in a Nutshell, Fifth Edition (O'Reilly):

NOTE

There are some variations on these types, with additional features, that you can check out in the Tiger Javadocs.

```java
// Rely on locking to prevent concurrent access
int count1 = 0;
public synchronized int count1() {
    return count1++;
}

// Rely on the atomic operations to prevent concurrent access
AtomicInteger count2 = new AtomicInteger(0);
public int count2() {
    return count2.getAndIncrement();
}

// Optimistic locking -- compare the result, to minimize overhead,
// and only correct if needed
AtomicInteger count3 = new AtomicInteger(0);
public int count3() {
    int result;
    do {
        result = count3.get();
    } while (!count3.compareAndSet(result, result+1));
    return result;
}
```
// Signals when the node this is connected to changes
Condition linkChanged;

public LinkList(E value) {
    this.value = value;
    rest = null;
    lock = new ReentrantLock();
    valueChanged = lock.newCondition();
    linkChanged = lock.newCondition();
}

public void setValue(E value) {
    lock.lock();
    try {
        this.value = value;
        // Let waiting threads that the value has changed
        valueChanged.signalAll();
    } finally {
        lock.unlock();
    }
}

public void executeOnValue(E desiredValue, Runnable task)
    throws InterruptedException {
    lock.lock();
    try {
        // Checks the value against the desired value
        while (!value.equals(desiredValue)) {
            // This will wait until the value changes
            // Signals when the node this is connected to changes
            Condition linkChanged;

            public LinkList(E value) {
                this.value = value;
                rest = null;
                lock = new ReentrantLock();
                valueChanged = lock.newCondition();
                linkChanged = lock.newCondition();
            }

            public void setValue(E value) {
                lock.lock();
                try {
                    this.value = value;
                    // Let waiting threads that the value has changed
                    valueChanged.signalAll();
                } finally {
                    lock.unlock();
                }
            }

            public void executeOnValue(E desiredValue, Runnable task)
                throws InterruptedException {
                lock.lock();
                try {
                    // Checks the value against the desired value
                    while (!value.equals(desiredValue)) {
                        // This will wait until the value changes
                    }
                } finally {
                    lock.unlock();
                }
            }
        }
    } finally {
        lock.unlock();
    }
}