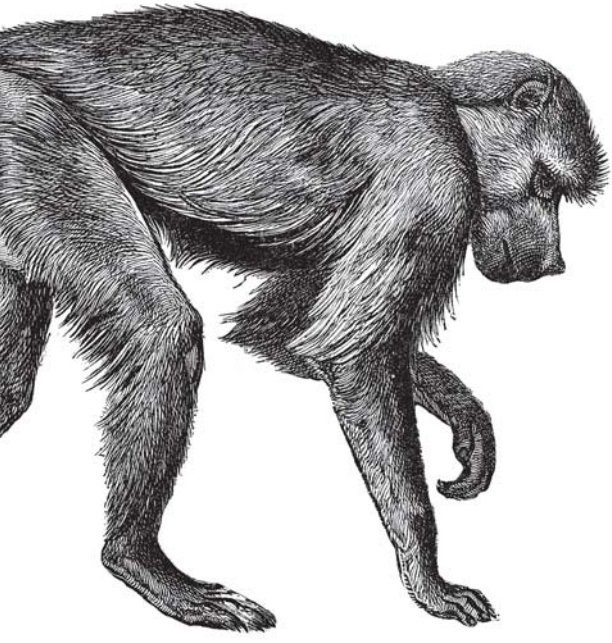
Quick Lookup and Advice

JUnit

Pocket Guide



O'REILLY®

Kent Beck

Java

O'REILLY®

JUnit Pocket Guide

JUnit is an award-winning framework to help programmers write and run automated tests. Basic use of JUnit encourages programmers to write code with fewer defects, improved design, and increased maintainability. More advanced users of JUnit can write their own, special-purpose, testing tools that integrate with JUnit.

The *JUnit Pocket Guide* includes:

- Comprehensive API documentation for JUnit
- The rationale for developer testing
- Test-first programming
- Stubbing
- JUnit and Ant
- JUnit and IDEs: Eclipse, JBuilder, and IntelliJ
- The history of JUnit

Written by Kent Beck, one of the original developers (along with Erich Gamma) of JUnit, the *JUnit Pocket Guide* will help you learn to test better as you develop.

ISBN 0-596-00743-4

US $ 9.95
CAN $14.95

90000

9 780596 007430

6 36920 00743 2

Visit O'Reilly on the Web at *www.oreilly.com*

JUnit
Pocket Guide

Kent Beck

O'REILLY®
Beijing · Cambridge · Farnham · Köln · Paris · Sebastopol · Taipei · Tokyo

JUnit Pocket Guide
by Kent Beck

Copyright © 2004 O'Reilly Media, Inc. All rights reserved.
Printed in the United States of America.

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (*safari.oreilly.com*). For more information, contact our corporate/institutional sales department: (800) 998-9938 or *corporate@oreilly.com*.

Editor:	Mike Hendrickson
Production Editor:	Darren Kelly
Cover Designer:	Ellie Volckhausen
Interior Designer:	Melanie Wang

Printing History:
September 2004: First Edition.

Nutshell Handbook, the Nutshell Handbook logo, and the O'Reilly logo are registered trademarks of O'Reilly Media, Inc. The *Pocket Guide* series designations, *JUnit Pocket Guide*, the image of a yellow baboon, and related trade dress are trademarks of O'Reilly Media, Inc.

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and O'Reilly Media, Inc. was aware of a trademark claim, the designations have been printed in caps or initial caps.

While every precaution has been taken in the preparation of this book, the publisher and author assume no responsibility for errors or omissions, or for damages resulting from the use of the information contained herein.

0-596-00743-4
[C] [12/04]

Contents

JUnit Pocket Guide

Automating Tests

Nearly every programmer tests his code. Testing with JUnit isn't a totally different activity from what you're doing right now. It's a different way of doing what you're already doing. The difference is between *testing*, that is checking that your program behaves as expected, and *having a battery of tests*, little programs that automatically check to ensure that your program behaves as expected. In this chapter we'll go from typical `println( )`-based testing code to a fully automated test.

Let's begin writing our first automated test. Imagine that we have been asked to test Java's built-in `ArrayList`. One bit of functionality to test is the method `length ( )`. When we create a new list it should have a length of `0`. After we add an element the length should be `1`. Imagine a little code snippet to test this:

```
List fixture= new ArrayList( );
// fixture should be empty
Object element= new Object( );
fixture.add(element);
// fixture should have one element
```

TIP

How do you pick good data values for tests? My habit is to pick a few representative values and to choose values that result in easy-to-check results. If I'm testing a currency converter, for example, an exchange rate of 2:1 is just as valuable as a programming tool as a "realistic" rate of 1.32471:1. For collections, like the one here, no elements and one element may be sufficient. Use induction to simplify testing. If "zero and one element work" implies "any number of elements work," use zero and one as your inputs.

A really simple way to check whether we are getting the results we expect is to print the size of the list before and after adding the element. If we get `0` and then `1`, `ArrayList` is behaving as expected.

```
List fixture= new ArrayList( );
System.out.println(fixture.size( ));
Object element= new Object( );
fixture.add(element);
System.out.println(fixture.size( ));
```

Now, we would like to move from tests that require manual interpretation to tests that can run automatically. We can take one step by comparing the actual values and our expected values and just printing out booleans. Then, we can look at the printed output and make sure all the lines are `true`. If we ever see a `false`, we know something is wrong.

```
List fixture= new ArrayList( );
System.out.println(fixture.size( ) == 0);
Object element= new Object( );
fixture.add(element);
System.out.println(fixture.size( ) == 1);
```

Checking booleans is the kind of boring, tedious, error-prone, rote work that is easier for computers to handle than humans. Suppose we create a function that takes a boolean as its input and throws an exception only if the boolean is false.

```
void assertTrue(boolean condition) throws Exception {
    if (! condition)
        throw new Exception("Assertion failed");
}
```

Now our test output gets simpler. Nothing gets printed if the test succeeds. If we see an unhandled exception, we know something has gone wrong.

```
List fixture= new ArrayList( );
assertTrue(fixture.size( ) == 0);
Object element= new Object( );
fixture.add(element);
assertTrue(fixture.size( ) == 1);
```

This test is completely automated. Instead of just *testing* as we did with our first version, with this version we have an *automated test*.

We only have one test. When we go to write `remove( )` we'll need another test. Then another for `iterator( )`. And another and another. We could write the infrastructure for all these tests from scratch. However, we would like to write the infrastructure once and then only need to write the unique parts of each test. JUnit is just such an infrastructure. Before diving into more details of writing tests, let's ask the question: why should developers write automated tests?

TIP

Installing JUnit

Download the latest `junit.jar` file from `junit.org` or use the one distributed with your IDE. Place it somewhere on your classpath. That should be all you need to do to begin using JUnit. If you have problems, go to `www.junit.org`.

Why Test?

This book shows you how to write automated tests using JUnit. JUnit is a framework that automates the tedious, repetitive parts of writing tests.