preface

> *"All undetectable errors will be treated as if no error occurred."*
> —*Technical reference from a major*
> *computing company (attributed)*

> *"The goal of Computer Science is to build something that will last—*
> *at least until we've finished building it."*
> —*Anonymous*

> *"As soon as we started programming, we found to our surprise that it wasn't*
> *as easy to get programs right as we had thought. Debugging had to be*
> *discovered. I can remember the exact instant when I realized that a large part*
> *of my life from then on was going to be spent in finding mistakes in my own*
> *programs."*
> —*Maurice Wilkes (attributed)*

Wouldn't it be a wonderful world if everything always went as you planned? Wouldn't it be great if every project you started was a total success, if every hiking trip was bright and sunny, if you won every contest that you entered? Wouldn't it be incredible if you were instantly a master at everything you tried? Whether you decided to build a new deck for your house, a catamaran, or a space shuttle, everything was always guaranteed to be an instant success. What a great world it would be....

In the meantime, we have to live in the real world. And in *this* world, things sometimes go wrong. Not always, but often enough that it makes sense to have a fallback plan sometimes. You start to build your deck, but you keep the phone number for a carpenter nearby just in case. When you go on a hike, you hope for good weather, but you bring along a poncho in case it rains. In extreme cases you might even bring:

- A poncho (for rain)
- A parka (for cold)
- Suntan lotion (for sun)
- Water
- Insect repellent
- A first aid kit
- A map
- A compass
- Mountain climbing rope

- Emergency flares
- Sled dog team
- Oxygen equipment

OK, the last few items on the list might be a *little* far fetched, but some of us like to be prepared. It's the same way for software—we want to design our code to plan for problems. Rather than assuming that everything will run perfectly every time, we should work out what can go wrong, and develop an approach to address potential problems.

How should your applications respond if there's a problem in code? Well, it depends. You don't always respond the same way when things don't go as planned in everyday life—why would you want your code to behave that way? Think about it: what would happen if you just smiled and laughed every time something went wrong? People would walk all over you. On the other hand, if you yelled and threw heavy objects every time something didn't go as planned, people might accuse you of overreacting. Clearly, we need to have some kind of strategy for our software to help us decide when to ignore problems and when to throw the heavy objects.

It's entirely appropriate for software systems to follow different strategies depending on the category or type of error. Usually, you need to take different approaches to fix the problems that can occur; it's rare to find a "one size fits all" solution for the code you write. In some cases, you may be able to ignore the error case. In other cases, you may decide you have to take action. In any event, establishing a plan for handling errors is one of the hallmarks of a well-written application. With a good exception handling strategy, you can't guarantee that everything will always go according to plan … but you have a "plan B" (and possibly C, D and E) in case something goes wrong.

Why It's Important to Know This Stuff

All right, why did I write this book? I've taught a lot of people about Java over the years, and I've noticed that many of my students never really had a good reference on how to deal effectively with errors in their Java code. For many years, you were lucky to find a Java book with a full chapter on the subject. As a result, a generation of Java developers has had to manage exception handling in their code as best they could. In a sense, I can understand why the subject has been so neglected. Dealing with error scenarios in code is a lot like going to the dentist. We know that it's something we have to do, but we don't particularly look forward to it.

At the same time, it is a vitally important topic. The way in which we handle errors has a direct, bottom-line impact on whether our code will run or not. It directly affects vital aspects of software such as maintainability. And it *definitely* makes a big difference in how easy it is to test and debug our code. Who hasn't stayed up late working on a project, desperately trying to make sense out of someone else's code? As the hours

stretch slowly outward into morning, who hasn't felt a twinge of helpless panic when faced with code that makes about as much sense as the programmer who wrote it after a dozen cups of coffee and two sleepless nights? Who hasn't felt the rush of terror when faced with a production problem, aware that an application's source code is hiding (yes, hiding) erroneous assumptions, faulty logic and mishandled errors somewhere deep inside? You can directly trace a lot of these problems to mistakes in dealing with exceptions and errors in code.

Of course, exception and error handling isn't the *only* thing responsible for errors and maintainability problems in code. But it *does* have a major impact in more applications that you might think. There have been many code examples, even in well-designed systems and books on "correct" Java coding, where you can find programming gems like this one:

```
try{
  operationA();
  component.operateB();
}
catch (Exception e){}
```

The problem, of course, is that it's easy to write code like the example above. It's a quick, simple way to get a class to compile. If you've never read much about exceptions, it might seem OK. If you've never had to maintain the code, it might seem like a pretty good idea. To developers who work in production support, it usually means pure terror.

Robust Java is intended to teach you what you need to know about exceptions, errors and handling—to help you learn to write truly robust, maintainable code. I spent the last year and a half of my life working on this project because I wanted to give developers a good, practical reference about how to deal with problems that can occur in their code. This book should be a good, practical resource for you that will lead to a better understanding of the topic, and ultimately to better practices in the software development community.

Part of this book is based on writing effective code, and part is based on showing you how to apply that knowledge. It takes an interesting mix of skills to write truly robust software systems. To be effective, to write really robust Java code, you need a good foundation understanding of what exceptions are and how they work. You need to understand how to write effective exception handling code. You need some grounding in software design, so that you can structure your code to effectively deal with exceptions and errors. You need to know a little about the APIs, to understand how and why code can fail. Finally, you need exposure to topics like architectural decisions and design patterns, so that you can develop an application that will stand the test of time. Ultimately, all of these skills come in handy as you develop a software system.

This book is not:

- A catalog of Java exceptions and errors
- A substitute for thought and reasoning
- A methodology or silver bullet

This book isn't meant to document all of the exceptions thrown by every method in the Java APIs. If I had tried to do that, this book would be longer than the collected works of Isaac Asimov. Also, I'd be so old by the time I was done that I would have forgotten what I was writing about. It's much more useful to talk about ways that code can fail, and to provide an overview of common errors that can occur in an API or application.

I'm a great believer in the benefits of applied reasoning, especially in a field like ours. This book contains a number of best practices. Some of them are widely documented. I've seen others mentioned in e-mail discussions or documented as lessons learned in development projects. However, keep in mind that any best practice is ultimately a suggestion. Suggestions have to be understood and intelligently applied, or they have little practical benefit. Ultimately, the best "best practices" won't do you much good if you apply them indiscriminately. This means you should think about how to apply the practices to your own projects, and do so when it makes good sense to you as a developer.

The approaches described in this book are not silver bullet solutions to all your problems. There is not such thing as a silver bullet in our business. Never has been, never will be. This book can make your code more solid, more maintainable, easier to test, easier to debug. But I know of nothing in this universe that will make all of the problems in a software project go away.

How this Book Is Organized

I believe this topic really depends on a full spectrum of software development skills. The act of creating a really robust software system depends on skills in development, design and architecture. Because of this, I wrote *Robust Java* in three parts.

Part 1 focuses on the mechanics of exceptions and exception handling in Java—the "nuts and bolts" of dealing with errors in code. This section gives you a good idea of what exceptions really are, how they are produced, and how they can be used within a system. It also describes some best practices for developers, and presents general APIs and techniques used in almost every type of exception handling. This is general material that relates to any Java developer. The topics are intended to give the reader a well-rounded view of how exceptions fit into the Java, acting as a foundation for the next two parts of the book.

Part 2 focuses on design concepts associated with exception handling. It discusses how to incorporate exception handling into your software design, and introduces the

concept of "failure mode analysis." We spend a lot of time in this chapter looking at commonly-used Java APIs. This section provides a perspective on what exceptions are produced in a given API and why. Since exception handling tends to be so vital in complex multi-tier systems, this section provides a special focus on Java's distributed APIs and the J2EE architectural framework.

Part 3 covers the topics related to using exceptions, errors and handling effectively over the full lifecycle of software development. It covers software architecture, design patterns, testing and debugging. The intent is to carry the topic beyond the basic discussion about exception handling to how to produce well-architected systems, systems that are effectively maintainable over the long term. Part of this section focuses on possible design patterns that tend to be well-suited for exception handling frameworks. Part examines the impact of exception handling strategies on the overall system architecture. And part of this section discusses the closely-related tasks of testing and debugging.

To Find Out More

Education is a constant process. If I've learned anything from seven years as a technical instructor, it's that lesson. It's helpful to read the stuff that's in this book, but it's also important to try out code, read articles and have discussions. Software developers, like most professionals in engineering, tend to follow a "learn by doing" model at least some of the time. It's usually important to have a number of follow-on sources for information, questions and so on.

There are a bunch of good additional resources for this book. If you want to look at related material, take a look at the bibliography. I've also set up a Web site for *Robust Java*. You can find it at:

```
http://talk-about-tech/robustjava
```

The site provides discussion forums, code examples and additional topics of interest for the book. Take a look, let me know what you think and what you'd like to see. I hope to see you there!
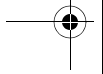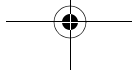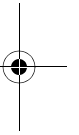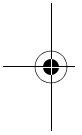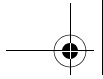
acknowledgments

There's something about the end of a writing project that makes a person get misty eyed and philosophical. Sitting in my favorite coffee shop, I can't help but think back a couple of years to when I started to write this book. When I co-authored *Applied Java Patterns*, I promised myself that I'd learn from the experience. The *next* time, I said I would carefully budget my time. I would work out a detailed outline and stick to it throughout the project. I would produce regular, incremental deliverables. Naturally, I broke all those promises. It's taken me about six months longer than I'd planned, and the book is much longer than I'd originally imagined.

I'm constantly amazed that it's even possible to transform a tangled mass of ideas, musings and code examples into a polished, coherent book. Maybe one person can do it on his own, but I doubt it. I know that I certainly don't have the patience for the job. If I didn't have the ongoing support of a wonderfully talented team, I would have given up long ago and spent all my time playing "Ratchet and Clank." So, to the people who have made *Robust Java* possible, you have my heartfelt thanks.

For Greg Doench, editorial wonder worker:
In the technical publishing business, editors are the ultimate champions of a book. They fight long and hard to get the advertising, editing and production resources that a book needs if it's going to succeed. I'm extremely grateful for all Greg's help and support in assisting me to navigate the waters of the publishing industry and make this book a reality.

For Brian O'Donnell, Mike Ernest and Natalie Levi, my talented technical reviewers:
Thank you so much for your honest, objective feedback. In this business, it's always good if a technical book actually makes sense. Thank you for helping me find a way to translate my thoughts into something resembling normal human speech.

For Mary Sudul, Robin Carroll, and Carol Cramer of the prolific Prentice-Hall production department:
My sincere thanks for your help and support. I appreciate all your hard work on a tight schedule; you've helped us turn a manuscript into something we can all be proud of.

For Wayne and Peggy Kovsky, organizers of the Colorado Software Summit:
Thanks for your ongoing support over the years. In my opinion, you've got one of the best, most informative Java conferences in the world. Thanks also for your interest in this work. Several of the chapters in this book are based around topics that I first
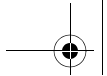
presented at the conference. For details on the Summit and its great technical presentations, take a look at **http://www.softwaresummit.com**.

Since this book covers a lot of material, I'd like to give special thanks to the following people for their inspiration, help and advice on specific parts of this book:

J. Keith Ratliff and Simon Roberts, for their inspiring work on J2EE architectures for Sun's SL425 course.

Bryan Basham, for his substantial insight into J2EE Web tier technologies.

Kathy Sierra, for most of the inspiration on EJB technologies.

I'd like to thank my fellow instructors at Sun Microsystems and our partner organizations. You're an amazing bunch of folks. It's a great honor to be a part of such a talented, dedicated and dynamic team. Thanks also to my students. Over the years, you've helped me broaden my own horizons through questions and discussions. I hope to be able to pass along some of the things we've learned to others.

Finally, to all the coffee shops of the world, you have my thanks. I would have slept most of my life away if it weren't been for you, and this book would probably still be unfinished. As it is, I'm hyper but happy.