Discover how Android apps are compiled and built to secure your data and build better apps

Decompiling Android

Godfrey Nolan

apress®

Decompiling Android

Godfrey Nolan

Apress

ISBN-13 (pbk): 978-1-4302-4248-2

ISBN-13 (electronic): 978-1-4302-4249-9

President and Publisher: Paul Manning
Lead Editor: James Markham
Technical Reviewer: Martin Larochelle

Coordinating Editor: Corbin Collins
Copy Editor: Tiffany Taylor

Compositor: Bytheway Publishing Services
Indexer: SPi Global
Artist: SPi Global
Cover Designer: Anna Ishchenko

Distributed to the book trade worldwide by Springer Science+Business Media New York, 233 Sprin Street, 6th Floor, New York, NY 10013. Phone 1-800-SPRINGER, fax (201) 348-4505, e-ma orders-ny@springer-sbm.com, or visit www.springeronline.com.

For information on translations, please e-mail rights@apress.com, or visit www.apress.com.

Apress and friends of ED books may be purchased in bulk for academic, corporate, or promotion use. eBook versions and licenses are also available for most titles. For more information, referen our Special Bulk Sales–eBook Licensing web page at www.apress.com/bulk-sales.

Any source code or other supplementary materials referenced by the author in this text is available readers at www.apress.com. For detailed information about how to locate your book's source code, g to www.apress.com/source-code.

For Nancy, who was there when I wrote my first published article, gave my first talk at a conference and wrote my first book, and is still here for my second. Here's to the next one.

–Godfrey Nolan

Contents at a Glance

Contents

About the Author

Godfrey Nolan is the founder and president of RIIS LLC in Southfield, MI. H has over 20 years of experience running software development teams. Original from Dublin, Ireland, he has a degree in mechanical engineering from Universi College Dublin and a masters in computer science from the University of the We of England. He is also the author of Decompiling Java, published by Apress 2004.

About the Technical Reviewer

Martin Larochelle has more than 10 years of experience in softwa development in project leader and architect roles. Currently, Mart works at Macadamian as a solutions architect, planning and supportir projects. His current focus is on mobile app development for Android ar other platforms. Martin's background is in C++ and VoIP development o soft clients, hard phones, and SIP servers.

Acknowledgments

Thanks to my technical reviewer, Martin Larochelle, for all the suggestions and support. Book writin can be like pulling teeth, so it’s always easier when the reviewer comments are logical and nudge th author in the right direction. I still have some teeth left—no hair, but some teeth.

Thanks to the Apress staff: Corbin Collins and James Markham for all the help and Steve Anglin fo helping me get the book accepted in the first place. I hope your other authors aren’t as difficult work with as I.

Thanks to Rory and Dayna, my son and daughter, for making me laugh as much as you do. Thanks Nancy, my wife, for putting up with the endless hours spent writing when I should have been spendin them with you.

Thanks to all the staff at RIIS who had to put up with my book deadlines more than most.

Preface

Decompiling Java was originally published in 2004 and, for a number of reasons, became more of a esoteric book for people interested in decompilation rather than anything approaching a gener programming audience.

When I began writing the book way back in 1998, there were lots of applets on websites, and th thought that someone could download your hard work and reverse-engineer it into Java source coo was a frightening thought for many. But applets went the same way as dial-up, and I suspect that man readers of this book have never seen an applet on a web page.

After the book came out, I realized that the only way someone could decompile your Java cla files was to first hack into your web server and download them from there. If they'd accomplishe that, you had far more to worry about than people decompiling your code.

With some notable exceptions—applications such as Corel's Java for Office that ran as a deskto application, and other Swing applications—for a decade or more Java code primarily lived on th server. Little or nothing was on the client browser, and zero access to class files meant zero problem with decompilation. But by an odd twist of fate, this has all changed with the Android platform: you Android apps live on your mobile device and can be easily downloaded and reverse-engineered b someone with very limited programming knowledge.

An Android app is downloaded to your device as an APK file that includes all the images an resources along with the code, which is stored in a single `classes.dex` file. This is a very differe format from the Java class file and is designed to run on the Android Dalvik virtual machine (DVM But it can be easily transformed back into Java class files and decompiled back into the origin source.

Decompilation is the process that transforms machine-readable code into a human-readab format. When an executable or a Java class file or a DLL is decompiled, you don't quite get th original format; instead, you get a type of pseudo source code, which is often incomplete and almo always without the comments. But, often, it's more than enough to understand the original code.

Decompiling Android addresses an unmet need in the programming community. For some reaso the ability to decompile Android APKs has been largely ignored, even though it's relatively easy fo anyone with the appropriate mindset to decompile an APK back into Java code. This book redresse the balance by looking at what tools and tricks of the trade are currently being employed by peop who are trying to recover source code and those who are trying to protect it using, for exampl obfuscation.

This book is for those who want to learn Android programming by decompilation, those wh simply want to learn how to decompile Android apps into source code, those who want to protect the Android code, and, finally, those who want to get a better understanding of `.dex` bytecodes and th DVM by building a `.dex` decompiler.

This book takes your understanding of decompilers and obfuscators to the next level by

- Exploring Java bytecodes and opcodes in an approachable but detailed manner
- Examining the structure of DEX files and opcodes and explaining how it differs from the

Java class file

- Using examples to show you how to decompile an Android APK file
- Giving simple strategies to show you how to protect your code
- Showing you what it takes to build your own decompiler and obfuscator

Decompiling Android isn't a normal Android programming book. In fact, it's the comple opposite of a standard textbook where the author teaches you how to translate ideas and concepts in code. You're interested in turning the partially compiled Android opcodes back into source code s you can see what the original programmer was thinking. I don't cover the language structure in dept except where it relates to opcodes and the DVM. All emphasis is on low-level virtual machine desig rather than on the language syntax.

The first part of this book unravels the APK format and shows you how your Java code is stored the DEX file and subsequently executed by the DVM. You also look at the theory and practice o decompilation and obfuscation. I present some of the decompiler's tricks of the trade and explain ho to unravel the most awkward APK. You learn about the different ways people try to protect the source code; when appropriate, I expose any flaws or underlying problems with the techniques s you're suitably informed before you use any source code protection tools.

The second part of this book primarily focuses on how to write your own Android decompiler an obfuscator. You build an extendable Android bytecode decompiler. Although the Java virtual machin (JVM) design is fixed, the language isn't. Many of the early decompilers couldn't handle Jav constructs that appeared in the JDK 1.1, such as inner classes. So if new constructs appear `classes.dex`, you'll be equipped to handle them.

Chapter

Laying the Groundwork

To begin, in this chapter I introduce you to the problem with decompilers and why virtual machine and the Android platform in particular are at such risk. You learn about the history of decompilers; may surprise you that they've been around almost as long as computers. And because this can be suc an emotive topic, I take some time to discuss the legal and moral issues behind decompilatio Finally, you're introduced to some of options open to you if you want to protect your code.

Compilers and Decompilers

Computer languages were developed because most normal people can't work in machine code or i nearest equivalent, Assembler. Fortunately, people realized pretty early in the development c computing technology that humans weren't cut out to program in machine code. Computer languag such as Fortran, COBOL, C, VB, and, more recently, Java and C# were developed to allow us to p our ideas in a human-friendly format that can then be converted into a format a computer chip ca understand.

At its most basic, it's the compiler's job to translate this textual representation or source code into series of 0s and 1s or machine code that the computer can interpret as actions or steps you want it perform. It does this using a series of pattern-matching rules. A lexical analyzer tokenizes the sourc code—and any mistakes or words that aren't in the compiler's lexicon are rejected. These tokens a then passed to the language parser, which matches one or more tokens to a series of rules an translates the tokens into intermediate code (VB.NET, C#, Pascal, or Java) or sometimes straight in machine code (Objective-C, C++, or Fortran). Any source code that doesn't match a compiler's rule is rejected, and the compilation fails.

Now you know what a compiler does, but I've only scratched the surface. Compiler technology h always been a specialized and sometimes complicated area of computing. Modern advances mea things are going to get even more complicated, especially in the virtual machine domain. In part, th drive comes from Java and .NET. Just in time (JIT) compilers have tried to close the gap between Jav and C++ execution times by optimizing the execution of Java bytecode. This seems like an impossib task, because Java bytecode is, after all, interpreted, whereas C++ is compiled. But JIT compil technology is making significant advances and also making Java compilers and virtual machines muc more complicated beasts.

Most compilers do a lot of preprocessing and post-processing. The preprocessor readies the sourc code for the lexical analysis by stripping out all unnecessary information, such as the programmer comments, and adding any standard or included header files or packages. A typical post-processo

stage is code optimization, where the compiler parses or scans the code, reorders it, and removes ar redundancies to increase the efficiency and speed of your code.

Decompilers (no big surprise here) translate the machine code or intermediate code back into sourc code. In other words, the whole compiling process is reversed. Machine code is tokenized in some wa and parsed or translated back into source code. This transformation rarely results in the origin source code, though, because information is lost in the preprocessing and post-processing stages.

Consider an analogy with human languages: decompiling an Android package file (APK) back in Java source is like translating German (`classes.dex`) into French (Java class file) and then in English (Java source). Along they way, bits of information are lost in translation. Java source code designed for humans and not computers, and often some steps are redundant or can be performed mo quickly in a slightly different order. Because of these lost elements, few (if any) decompilations resu in the original source.

A number of decompilers are currently available, but they aren't well publicized. Decompilers disassemblers are available for Clipper (Valkyrie), FoxPro (ReFox and Defox), Pascal, C (dc decomp, Hex-Rays), Objective-C (Hex-Rays), Ada, and, of course, Java. Even the Newton, loved b *Doonesbury* aficionados everywhere, isn't safe. Not surprisingly, decompilers are much mo common for interpreted languages such as VB, Pascal, and Java because of the larger amounts information being passed around.

Virtual Machine Decompilers

There have been several notable attempts to decompile machine code. Cristina Cifuentes' dcc ar more recently the Hex-Ray's IDA decompiler are just a couple of examples. However, at the machine code level, the data and instructions are comingled, and it's a much more difficult (but n impossible) task to recover the original code.

In a virtual machine, the code has simply passed through a preprocessor, and the decompiler's job to reverse the preprocessing stages of compilation. This makes interpreted code much, much easier decompile. Sure, there are no comments and, worse still, there is no specification, but then again the are no R&D costs.

Why Java with Android?

Before I talk about "Why Android?" I first need to ask, "Why Java?" That's not to say all Androi apps are written in Java—I cover HTML5 apps too. But Java and Android are joined at the hip, so can't really discuss one without the other.

The original Java virtual machine (JVM) was designed to be run on a TV cable set-top box. As suc it's a very small-stack machine that pushes and pops its instructions on and off a stack using a limite instruction set. This makes the instructions very easy to understand with relatively little practic Because compilation is now a two-stage process, the JVM also requires the compiler to pass a lot information, such as variable and method names, that wouldn't otherwise be available. These nam can be almost as helpful as comments when you're trying to understand decompiled source code.

The current design of the JVM is independent of the Java Development Kit (JDK). In other words, th language and libraries may change, but the JVM and the opcodes are fixed. This means that if Java

prone to decompilation now, it's always likely to be prone to decompilation. In many cases, as you' see, ~~decompiling a Java class is as easy as running a simple DOS or UNIX command.~~

In the future, the JVM may very well be changed to stop decompilation, but this would break ar backward compatibility and all current Java code would have to be recompiled. And although this ha happened before in the Microsoft world with different versions of VB, many companies other tha Oracle have developed virtual machines.

What makes this situation even more interesting is that companies that want to Java-enable the operating system or browser usually create their own JVMs. Oracle is only responsible for the JV specification. This situation has progressed so far that any fundamental changes to the JV specification would have to be backward compatible. Modifying the JVM to prevent decompilatio would require significant surgery and would in all probability break this backward compatibility, thu ensuring that Java classes will decompile for the foreseeable future.

There are no such compatibility restrictions on the JDK, and more functionality is added with eac release. And although the first crop of decompilers, such as Mocha, dramatically failed when inn classes were introduced in the JDK 1.1, the current favorite JD-GUI is more than capable of handlin inner classes or later additions to the Java language, such as generics.

You learn a lot more about why Java is at risk from decompilation in the next chapter, but for th moment here are seven reasons why Java is vulnerable:

- For portability, Java code is partially compiled and then interpreted by the JVM.
- Java's compiled classes contain a lot of symbolic information for the JVM.
- Due to backward-compatibility issues, the JVM's design isn't likely to change.
- There are few instructions or opcodes in the JVM.
- The JVM is a simple stack machine.
- Standard applications have no real protection against decompilation.
- Java applications are automatically compiled into smaller modular classes.

Let's begin with a simple class-file example, shown in Listing 1-1.

Listing 1-1. *Simple Java Source Code Example*

```
public class Casting {
 public static void main(String args[]){
 for(char c=0; c < 128; c++) {
     System.out.println("ascii " + (int)c + " character "+ c);
 }
 }
}
```

Listing 1-2 shows the output for the class file in Listing 1-1 using javap, Java's class-file disassembl that ships with the JDK. You can decompile Java so easily because—as you see later in the book—th JVM is a simple stack machine with no registers and a limited number of high-level instructions opcodes.

Listing 1-2. *Javap Output*

```
Compiled from Casting.java
```

```
public synchronized class Casting extends java.lang.Object
 /* ACC_SUPER bit set */
{
 public static void main(java.lang.String[]);
/* Stack=4, Locals=2, Args_size=1 */
 public Casting();
/* Stack=1, Locals=1, Args_size=1 */
}

Method void main(java.lang.String[])
 0 iconst_0
 1 istore_1
 2 goto 41
 5 getstatic #12 <Field java.io.PrintStream out>
 8 new #6 <Class java.lang.StringBuffer>
 11 dup
 12 ldc #2 <String "ascii ">
 14 invokespecial #9 <Method
java.lang.StringBuffer(java.lang.String)>
 17 iload_1
 18 invokevirtual #10 <Method java.lang.StringBuffer append(char)>
 21 ldc #1 <String " character ">
 23 invokevirtual #11 <Method java.lang.StringBuffer
append(java.lang.String)>
 26 iload_1
 27 invokevirtual #10 <Method java.lang.StringBuffer append(char)>
 30 invokevirtual #14 <Method java.lang.String toString()>
 33 invokevirtual #13 <Method void println(java.lang.String)>
 36 iload_1
 37 iconst_1
 38 iadd
 39 i2c
 40 istore_1
 41 iload_1
 42 sipush 128
 45 if_icmplt 5
 48 return

Method Casting()
 0 aload_0
 1 invokespecial #8 <Method java.lang.Object()>
 4 return<
```

It should be obvious that a class file contains a lot of the source-code information. My aim in th book is to show you how to take this information and reverse-engineer it into source code. I'll als show you what steps you can take to protect the information.

Why Android?

Until now, with the exception of applets and Java Swing apps, Java code has typically been server sid with little or no code running on the client. This changed with the introduction of Google's Androi operating system. Android apps, whether they're written in Java or HTML5/CSS, are client-sid applications in the form of APKs. These APKs are then executed on the Dalvik virtual machin

(DVM).

The DVM differs from the JVM in a number of ways. First, it's a register-based machine, unlike th stack-based JVM. And instead of multiple class files bundled into a jar file, the DVM uses a sing Dalvik executable (DEX) file with a different structure and opcodes. On the surface, it would appe to be much harder to decompile an APK. However, someone has already done all the hard work fo you: a tool called dex2jar allows you to convert the DEX file back into a jar file, which then can b decompiled back into Java source.

Because the APKs live on the phone, they can be easily downloaded to a PC or Mac and the decompiled. You can use lots of different tools and techniques to gain access to an APK, and there ar many decompilers, which I cover later in the book. But the easiest way to get at the source is to cop the APK onto the phone's SD card using any of the file-manager tools available in the marketplac such as ASTRO File Manager. Once the SD card is plugged into your PC or Mac, it can then b decompiled using dex2jar followed by your favorite decompiler, such as JD-GUI.

Google has made it very easy to add ProGuard to your builds, but obfuscation doesn't happen b default. For the moment (until this issue achieves a higher profile), the code is unlikely to have bee protected using obfuscation, so there's a good chance the code can be completely decompiled bac into source. ProGuard is also not 100% effective as an obfuscation tool, as you see in Chapter 4 and 7

Many Android apps talk to backend systems via web services. They look for items in a database, complete a purchase, or add data to a payroll system, or upload documents to a file server. Th usernames and passwords that allow the app to connect to these backend systems are often hard-code in the Android app. So, if you haven't protected your code and you leave the keys to your backen system in your app, you're running the risk of someone compromising your database and gainin access to systems that they should not be accessing.

It's less likely, but entirely possible, that someone has access to the source and can recompile the ap to get it to talk to a different backend system, and use it as a means of harvesting usernames ar passwords. This information can then be used at a later stage to gain access to private data using th real Android app.

This book explains how to hide your information from these prying eyes and raise the bar so it takes lot more than basic knowledge to find the keys to your backend servers or locate the credit-ca information stored on your phone.

It's also very important to protect your Android app before releasing it into the marketplace. Sever web sites and forums share APKs, so even if you protect your app by releasing an updated version, th original unprotected APK may still be out there on phones and forums. Your web-service APIs mus also be updated at the same time, forcing users to update their app and leading to a bad us experience and potential loss of customers.

In Chapter 4, you learn more about why Android is at risk from decompilation, but for the mome here is a list of reasons why Android apps are vulnerable:

- There are multiple easy ways to gain access to Android APKs.
- It's simple to translate an APK to a Java jar file for subsequent decompilation.
- As yet, almost nobody is using obfuscation or any form of protection.
- Once the APK is released, it's very hard to remove access.
- One-click decompilation is possible, using tools such as apktool.

- APKs are shared on hacker forums.

Listing 1-3 shows the dexdump output of the `Casting.java` file from Listing 1-1 after it has bee converted to the DEX format. As you can see, it's similar information but in a new format. Chapter looks at the differences in greater detail.

Listing 1-3. *Dexdump Output*

```
Class #0            -
 Class descriptor  : 'LCasting;'
 Access flags      : 0x0001 (PUBLIC)
 Superclass        : 'Ljava/lang/Object;'
 Interfaces        -
 Static fields -
 Instance fields -
 Direct methods -
 #0              : (in LCasting;)
  name          : '<init>'
  type          : '()V'
  access        : 0x10001 (PUBLIC CONSTRUCTOR)
  code          -
  registers     : 1
  ins           : 1
  outs          : 1
  insns size    : 4 16-bit code units
  catches       : (none)
  positions     :
    0x0000 line=1
  locals        :
    0x0000 - 0x0004 reg=0 this LCasting;
 #1              : (in LCasting;)
  name          : 'main'
  type          : '([Ljava/lang/String;)V'
  access        : 0x0009 (PUBLIC STATIC)
  code          -
  registers     : 5
  ins           : 1
  outs          : 2
  insns size    : 44 16-bit code units
  catches       : (none)
  positions     :
    0x0000 line=3
    0x0005 line=4
    0x0027 line=3
    0x002b line=6
  locals        :
 Virtual methods -
 source_file_idx : 3 (Casting.java)
```

History of Decompilers

Very little has been written about the history of decompilers, which is surprising because for almo every compiler, there has been a decompiler. Let's take a moment to talk about their history so yo can see how and why decompilers were created so quickly for the JVM and, to a lesser extent, th

DVM.

Since before the dawn of the humble PC—scratch that, since before the dawn of COBOL, decompile have been around in one form or another. You can go all the way back to ALGOL to find the earlies example of a decompiler. Joel Donnelly and Herman Englander wrote D-Neliac at the U.S. Nav Electronic Labs (NEL) laboratories as early as 1960. Its primary function was to convert non-Neli compiled programs into Neliac-compatible binaries. (Neliac was an ALGOL-type language and stan for Navy Electronics Laboratory International ALGOL Compiler.)

Over the years there have been other decompilers for COBOL, Ada, Fortran, and many other esoter as well as mainstream languages running on IBM mainframes, PDP-11s, and UNIVACs, amon others. Probably the main reason for these early developments was to translate software or conve binaries to run on different hardware.

More recently, reverse-engineering to circumvent the Y2K problem became the acceptable face decompilation—converting legacy code to get around Y2K often required disassembly or fu decompilation. But reverse engineering is a huge growth area and didn't disappear after the turn of t millennium. Problems caused by the Dow Jones hitting the 10,000 mark and the introduction of t Euro have caused financial programs to fall over.

Reverse-engineering techniques are also used to analyze old code, which typically has thousands incremental changes, in order to remove redundancies and convert these legacy systems into muc more efficient animals.

At a much more basic level, hexadecimal dumps of PC machine code give programmers extra insig into how something was achieved and have been used to break artificial restrictions placed software. For example, magazine CDs containing time-bombed or restricted copies of games and oth utilities were often patched to change demonstration copies into full versions of the software; this w often accomplished with primitive disassemblers such as the DOS's debug program.

Anyone well versed in Assembler can learn to quickly spot patterns in code and bypass the appropria source-code fragments. Pirate software is a huge problem for the software industry, and disassemblin the code is just one technique employed by professional and amateur bootleggers. Hence the downfa of many an arcane copy-protection technique. But these are primitive tools and techniques, and would probably be quicker to write the code from scratch rather than to re-create the source code fro Assembler.

For many years, traditional software companies have also been involved in reverse-engineerin software. New techniques are studied and copied all over the world by the competition using revers engineering and decompilation tools. Generally, these are in-house decompilers that aren't for publ consumption.

It's likely that the first real Java decompiler was written in IBM and not by Hanpeter van Vliet, auth of Mocha. Daniel Ford's white paper "Jive: A Java Decompiler" (May 1996) appears in IB Research's search engines; this beats Mocha, which wasn't announced until the following July.

Academic decompilers such as dcc are available in the public domain. Commercial decompilers su as Hex-Ray's IDA have also begun to appear. Fortunately for the likes of Microsoft, decompilin Office using dcc or Hex-Rays would create so much code that it's about as user friendly as debug or hexadecimal dump. Most modern commercial software's source code is so huge that it becom unintelligible without the design documents and lots of source-code comments. Let's face it: ma people's C++ code is hard enough to read six months after they wrote it. How easy would it be f

someone else to decipher without help C code that came from compiled C++ code?

Reviewing Interpreted Languages More Closely: Visual Basic

Let's look at VB as an example of an earlier version of interpreted language. Early versions of V were interpreted by its runtime module `vbrun.dll` in a fashion somewhat similar to Java and t JVM. Like a Java class file, the source code for a VB program is bundled within the binary. Bizarrel VB3 retains more information than Java—even the programmer comments are included.

The original versions of VB generated an intermediate pseudocode called *p-code*, which was in Pasc and originated in the P-System (`www.threedee.com/jcm/psystem/`). And before you say anythin yes, Pascal and all its derivatives are just as vulnerable to decompilation—that includes early versio of Microsoft's C compiler, so nobody feels left out. The p-codes aren't dissimilar to bytecodes and a essentially VB opcodes that are interpreted by `vbrun.dll` at run time. If you've ever wondered wh you needed to include `vbrun300.dll` with VB executables, now you know. You have to includ `vbrun.dll` so it can interpret the p-code and execute your program.

Doctor H. P. Diettrich, who is from Germany, is the author of the eponymously titled DoDi—perhap the most famous VB decompiler. At one time, VB had a culture of decompilers and obfuscators (protection tools, as they're called in VB). But as VB moved to compiled rather than interpreted cod the number of decompilers decreased dramatically. DoDi provides VBGuard for free on his site, a programs such as Decompiler Defeater, Protect, Overwrite, Shield, and VBShield are available fro other sources. But they too all but disappeared with VB5 and VB6.

That was of course before .NET, which has come full circle: VB is once again interpreted. N surprisingly, many decompilers and obfuscators are again appearing in the .NET world, such as t ILSpy and Reflector decompilers as well as Demeanor and Dotfuscator obfuscators.

Hanpeter van Vliet and Mocha

Oddly enough for a technical subject, this book also has a very human element. Hanpeter van Vli wrote the first public-domain decompiler, Mocha, while recovering from a cancer operation in t Netherlands in 1996. He also wrote an obfuscator called Crema that attempted to protect an applet source code. If Mocha was the UZI machine gun, then Crema was the bulletproof jacket. In a no classic Internet marketing strategy, Mocha was free, whereas there was a small charge for Crema.

The beta version of Mocha caused a huge controversy when it was first made available on Hanpeter web site, especially after it was featured in a CNET article. Because of the controversy, Hanpeter to the very honorable step of removing Mocha from his web site. He then allowed visitor's to his site vote about whether Mocha should once again be made available. The vote was ten to one in favor Mocha, and soon after it reappeared on Hanpeter's web site.

However, Mocha never made it out of Beta. And while doing some research for a Web Technique article on this subject, I learned from his wife, Ingrid, that Hanpeter's throat cancer finally got hi and he died at the age of 34 on New Year's Eve 1996.

The source code for both Crema and Mocha were sold to Borland shortly before Hanpeter's deat with all proceeds going to Ingrid. Some early versions of JBuilder shipped with an obfuscator, whic was probably Crema. It attempted to protect Java code from decompilation by replacing ASC variable names with control characters.

I talk more about the host of other Java decompilers and obfuscators later in the book.

Legal Issues to Consider When Decompiling

Before you start building your own decompiler, let's take this opportunity to consider the leg implications of decompiling someone else's code for your own enjoyment or benefit. Just becau Java has taken decompiling technology out of some very serious propeller-head territory and in more mainstream computing doesn't make it any less likely that you or your company will be sued. may make it more fun, but you really should be careful.

As a small set of ground rules, try the following:

- Don't decompile an APK, recompile it, and then pass it off as your own.
- Don't even think of trying to sell a recompiled APK to any third parties.
- Try not to decompile an APK or application that comes with a license agreement that expressly forbids decompiling or reverse-engineering the code.
- Don't decompile an APK to remove any protection mechanisms and then recompile it for your own personal use.

Protection Laws

Over the past few years, big business has tilted the law firmly in its favor when it comes decompiling software. Companies can use a number of legal mechanisms to stop you fro decompiling their software; you would have little or no legal defense if you ever had to appear in court of law because a company discovered that you had decompiled its programs. Patent la copyright law, anti-reverse-engineering clauses in shrinkwrap licenses, as well as a number of lav such as the Digital Millennium Copyright Act (DMCA) may all be used against you. Different lav may apply in different countries or states: for example, the "no reverse engineering clause" softwa license is a null and void clause in the European Union (EU). But the basic concepts are the sam decompile a program for the purpose of cloning the code into another competitive product, and you' probably breaking the law.The secret is that you shouldn't be standing, kneeling, or pressing dow very hard on the legitimate rights (the copyright) of the original author. That's not to say it's *never* to decompile. There are certain limited conditions under which the law favors decompilation reverse engineering through a concept known as *fair use*. From almost the dawn of time, and certain from the beginning of the Industrial Age, many of humankind's greatest inventions have come fro individuals who created something special while Standing on the Shoulders of Giants. For exampl the invention of the steam train and the light bulb were relatively modest incremental steps technology. The underlying concepts were provided by other people, and it was up to someone li George Stephenson or Thomas Edison to create the final object. (You can see an excellent example Stephenson's debt to many other inventors such as James Watt www.usgennet.org/usa/topic/steam/Early/Time.html). This is one of the reasons paten appeared: to allow people to build on other creations while still giving the original inventors som compensation for their initial ideas for period of, say, 20 years.

Patents