

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**

Copyright © 2012 by Godfrey Nolan

This work is subject to copyright. All rights are reserved by the Publisher, whether the whole or part of the material is concerned, specifically the rights of translation, reprinting, reuse of illustrations, recitation, broadcasting, reproduction on microfilms or in any other physical way, and transmission of information storage and retrieval, electronic adaptation, computer software, or by similar or dissimilar methodology now known or hereafter developed. Exempted from this legal reservation are brief excerpts in connection with reviews or scholarly analysis or material supplied specifically for the purpose of being entered and executed on a computer system, for exclusive use by the purchaser of the work. Duplication of this publication or parts thereof is permitted only under the provisions of the Copyright Law of the Publisher's location, in its current version, and permission for use must always be obtained from Springer. Permissions for use may be obtained through RightsLink at the Copyright Clearance Center. Violations are liable to prosecution under the respective Copyright Law.

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

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

Trademarked names, logos, and images may appear in this book. Rather than use a trademark symbol with every occurrence of a trademarked name, logo, or image we use the names, logos, and images only in an editorial fashion and to the benefit of the trademark owner, with no intention of infringing of the trademark.

The images of the Android Robot (01 / Android Robot) are reproduced from work created and shared by Google and used according to terms described in the Creative Commons 3.0 Attribution License. Android and all Android and Google-based marks are trademarks or registered trademarks of Google, Inc., in the U.S. and other countries. Apress Media, L.L.C. is not affiliated with Google, Inc., and this book was written without endorsement from Google, Inc.

The use in this publication of trade names, trademarks, service marks, and similar terms, even if they are not identified as such, is not to be taken as an expression of opinion as to whether or not they are subject to proprietary rights.

While the advice and information in this book are believed to be true and accurate at the date of publication, neither the authors nor the editors nor the publisher can accept any legal responsibility for any errors or omissions that may be made. The publisher makes no warranty, express or implied, with respect to the material contained herein.

President and Publisher: Paul Manning

Lead Editor: James Markham

Technical Reviewer: Martin Larochelle

Editorial Board: Steve Anglin, Ewan Buckingham, Gary Cornell, Louise Corrigan, Morgan Ertel, Jonathan Gennick,

Jonathan Hassell, Robert Hutchinson, Michelle Lowman, James Markham, Matthew Moodie, Jeff Olson, Jeffrey

Pepper, Douglas Pundick, Ben Renow-Clarke, Dominic Shakeshaft, Gwenan Spearin, Matt Wade, Tom Welsh

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 Spring Street, 6th Floor, New York, NY 10013. Phone 1-800-SPRINGER, fax (201) 348-4505, e-mail [orders-ny@springer-sbm.com](mailto:orders-ny@springer-sbm.com), or visit [www.springeronline.com](http://www.springeronline.com).

For information on translations, please e-mail [rights@apress.com](mailto:rights@apress.com), or visit [www.apress.com](http://www.apress.com).

Apress and friends of ED books may be purchased in bulk for academic, corporate, or promotional use. eBook versions and licenses are also available for most titles. For more information, refer to our Special Bulk Sales—eBook Licensing web page at [www.apress.com/bulk-sales](http://www.apress.com/bulk-sales).

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

- **About the Author**
- **About the Technical Reviewer**
- **Acknowledgments**
- **Preface**
- **Chapter 1: Laying the Groundwork**
- **Chapter 2: Ghost in the Machine**
- **Chapter 3: Inside the DEX File**
- **Chapter 4: Tools of the Trade**
- **Chapter 5: Decompiler Design**
- **Chapter 6: Decompiler Implementation**
- **Chapter 7: Hear No Evil, See No Evil: A Case Study**
- **Appendix A: Opcode Tables**
- **Index**

---

# Contents

- **About the Author**
- **About the Technical Reviewer**
- **Acknowledgments**
- **Preface**
- **Chapter 1: Laying the Groundwork**

## **Compilers and Decompilers**

### **Virtual Machine Decompilers**

#### **Why Java with Android?**

#### **Why Android?**

#### **History of Decompilers**

Reviewing Interpreted Languages More Closely: Visual Basic

Hanpeter van Vliet and Mocha

#### **Legal Issues to Consider When Decompiling**

Protection Laws

The Legal Big Picture

#### **Moral Issues**

#### **Protecting Yourself**

#### **Summary**

- **Chapter 2: Ghost in the Machine**

### **The JVM: An Exploitable Design**

#### **Simple Stack Machine**

Heap

Program Counter Registers

Method Area

JVM Stack

#### **Inside a Class File**

Magic Number

Minor and Major Versions

Constant-Pool Count

Constant Pool

Access Flags

---

The this Class and the Superclass

Interfaces and Interface Count

Fields and Field Count

Methods and Method Count

Attributes and Attributes Count

## Summary

### ■ Chapter 3: Inside the DEX File

#### Ghost in the Machine, Part Deux

Converting Casting.class

#### Breaking the DEX File into Its Constituent Parts

The Header Section

The string\_ids Section

The type\_ids Section

The proto\_ids Section

The field\_ids Section

The method\_ids Section

The class\_defs Section

The data Section

## Summary

### ■ Chapter 4: Tools of the Trade

#### Downloading the APK

Backing Up the APK

Forums

Platform Tools

#### Decompiling an APK

What's in an APK File?

Random APK Issues

#### Disassemblers

Hex Editors

dx and dexdump

dedexer

baksmali

#### Decompilers

Mocha

Jad



JD-GUI

dex2jar

undx

apktool

## **Protecting Your Source**

Writing Two Versions of the Android App

Obfuscation

## **Summary**

## **■ Chapter 5: Decompiler Design**

### **Theory Behind the Design**

#### **Defining the Problem**

#### **(De)Compiler Tools**

Lex and Yacc

JLex and CUP Example

ANTLR

#### **Strategy: Deciding on your Parser Design**

Choice One

Choice Two

Choice Three

Choice Four

#### **Parser Design**

## **Summary**

## **■ Chapter 6: Decompiler Implementation**

### **DexToXML**

Parsing the dex.log Output

### **DexToSource**

#### **Example 1: Casting.java**

Bytecode Analysis

Parser

Java

#### **Example 2: Hello World**

Bytecode Analysis

Parser

Java

#### **Example 3: if Statement**

Bytecode Analysis

Parser

Java

---

## **Refactoring**

### **Summary**

#### ■ **Chapter 7: Hear No Evil, See No Evil: A Case Study**

### **Obfuscation Case Study**

### **Myths**

### **Solution 1: ProGuard**

SDK Output

Double-Checking Your Work

Configuration

Debugging

### **Solution 2: DashO**

Output

### **Reviewing the Case Study**

### **Summary**

#### ■ **Appendix A: Opcode Tables**

#### ■ **Index**

---

# About the Author



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

---

# About the Technical Reviewer



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

---

# Acknowledgments

Thanks to my technical reviewer, Martin Larochelle, for all the suggestions and support. Book writing can be like pulling teeth, so it's always easier when the reviewer comments are logical and nudge the 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 for helping me get the book accepted in the first place. I hope your other authors aren't as difficult to work with as I.

Thanks to Rory and Dayna, my son and daughter, for making me laugh as much as you do. Thanks to Nancy, my wife, for putting up with the endless hours spent writing when I should have been spending 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 an esoteric book for people interested in decompilation rather than anything approaching a general programming audience.

When I began writing the book way back in 1998, there were lots of applets on websites, and I thought that someone could download your hard work and reverse-engineer it into Java source code. This was a frightening thought for many. But applets went the same way as dial-up, and I suspect that many 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 class files was to first hack into your web server and download them from there. If they'd accomplished 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 desktop application, and other Swing applications—for a decade or more Java code primarily lived on the server. Little or nothing was on the client browser, and zero access to class files meant zero problems with decompilation. But by an odd twist of fate, this has all changed with the Android platform: your Android apps live on your mobile device and can be easily downloaded and reverse-engineered by someone with very limited programming knowledge.

An Android app is downloaded to your device as an APK file that includes all the images and resources along with the code, which is stored in a single `classes.dex` file. This is a very different 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 original source.

Decompilation is the process that transforms machine-readable code into a human-readable format. When an executable or a Java class file or a DLL is decompiled, you don't quite get the original format; instead, you get a type of pseudo source code, which is often incomplete and almost 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 reason, the ability to decompile Android APKs has been largely ignored, even though it's relatively easy for anyone with the appropriate mindset to decompile an APK back into Java code. This book redresses the balance by looking at what tools and tricks of the trade are currently being employed by people who are trying to recover source code and those who are trying to protect it using, for example, obfuscation.

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

- 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 complete opposite of a standard textbook where the author teaches you how to translate ideas and concepts into code. You're interested in turning the partially compiled Android opcodes back into source code so you can see what the original programmer was thinking. I don't cover the language structure in depth except where it relates to opcodes and the DVM. All emphasis is on low-level virtual machine design 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 in the DEX file and subsequently executed by the DVM. You also look at the theory and practice of decompilation and obfuscation. I present some of the decompiler's tricks of the trade and explain how to unravel the most awkward APK. You learn about the different ways people try to protect their source code; when appropriate, I expose any flaws or underlying problems with the techniques so 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 and obfuscator. You build an extendable Android bytecode decompiler. Although the Java virtual machine (JVM) design is fixed, the language isn't. Many of the early decompilers couldn't handle Java constructs that appeared in the JDK 1.1, such as inner classes. So if new constructs appear in `classes.dex`, you'll be equipped to handle them.

## Laying the Groundwork

To begin, in this chapter I introduce you to the problem with decompilers and why virtual machines and the Android platform in particular are at such risk. You learn about the history of decompilers; you may be surprised that they've been around almost as long as computers. And because this can be such an emotive topic, I take some time to discuss the legal and moral issues behind decompilation. Finally, you're introduced to some of the 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 its nearest equivalent, Assembler. Fortunately, people realized pretty early in the development of computing technology that humans weren't cut out to program in machine code. Computer languages such as Fortran, COBOL, C, VB, and, more recently, Java and C# were developed to allow us to put our ideas in a human-friendly format that can then be converted into a format a computer chip can understand.

At its most basic, it's the compiler's job to translate this textual representation or source code into a series of 0s and 1s or machine code that the computer can interpret as actions or steps you want it to perform. It does this using a series of pattern-matching rules. A lexical analyzer tokenizes the source code—and any mistakes or words that aren't in the compiler's lexicon are rejected. These tokens are then passed to the language parser, which matches one or more tokens to a series of rules and translates the tokens into intermediate code (VB.NET, C#, Pascal, or Java) or sometimes straight into 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 has always been a specialized and sometimes complicated area of computing. Modern advances mean things are going to get even more complicated, especially in the virtual machine domain. In part, the drive comes from Java and .NET. Just-in-time (JIT) compilers have tried to close the gap between Java and C++ execution times by optimizing the execution of Java bytecode. This seems like an impossible task, because Java bytecode is, after all, interpreted, whereas C++ is compiled. But JIT compilation technology is making significant advances and also making Java compilers and virtual machines much more complicated beasts.

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



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

---

Decompilers (no big surprise here) translate the machine code or intermediate code back into source code. In other words, the whole compiling process is reversed. Machine code is tokenized in some way and parsed or translated back into source code. This transformation rarely results in the original 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 into Java source is like translating German (`classes.dex`) into French (Java class file) and then into English (Java source). Along the way, bits of information are lost in translation. Java source code is designed for humans and not computers, and often some steps are redundant or can be performed more quickly in a slightly different order. Because of these lost elements, few (if any) decompilations result in the original source.

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

## Virtual Machine Decompilers

There have been several notable attempts to decompile machine code. Cristina Cifuentes' dcc and 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 not impossible) task to recover the original code.

In a virtual machine, the code has simply passed through a preprocessor, and the decompiler's job is to reverse the preprocessing stages of compilation. This makes interpreted code much, much easier to decompile. Sure, there are no comments and, worse still, there is no specification, but then again there 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 Android apps are written in Java—I cover HTML5 apps too. But Java and Android are joined at the hip, so I 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 such, it's a very small-stack machine that pushes and pops its instructions on and off a stack using a limited instruction set. This makes the instructions very easy to understand with relatively little practice. Because compilation is now a two-stage process, the JVM also requires the compiler to pass a lot of information, such as variable and method names, that wouldn't otherwise be available. These names 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, the 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 backward compatibility and all current Java code would have to be recompiled. And although this has happened before in the Microsoft world with different versions of VB, many companies other than Oracle have developed virtual machines.

What makes this situation even more interesting is that companies that want to Java-enable their operating system or browser usually create their own JVMs. Oracle is only responsible for the JVM specification. This situation has progressed so far that any fundamental changes to the JVM specification would have to be backward compatible. Modifying the JVM to prevent decompilation would require significant surgery and would in all probability break this backward compatibility, thus 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 each release. And although the first crop of decompilers, such as Mocha, dramatically failed when inner classes were introduced in the JDK 1.1, the current favorite JD-GUI is more than capable of handling 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 the 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 disassembler that ships with the JDK. You can decompile Java so easily because—as you see later in the book—the JVM is a simple stack machine with no registers and a limited number of high-level instructions and 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 this book is to show you how to take this information and reverse-engineer it into source code. I'll also 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 side with little or no code running on the client. This changed with the introduction of Google's Android operating system. Android apps, whether they're written in Java or HTML5/CSS, are client-side applications in the form of APKs. These APKs are then executed on the Dalvik virtual machine.

(DVM).

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

Because the APKs live on the phone, they can be easily downloaded to a PC or Mac and then decompiled. You can use lots of different tools and techniques to gain access to an APK, and there are many decompilers, which I cover later in the book. But the easiest way to get at the source is to copy the APK onto the phone's SD card using any of the file-manager tools available in the marketplace, such as ASTRO File Manager. Once the SD card is plugged into your PC or Mac, it can then be 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 by default. For the moment (until this issue achieves a higher profile), the code is unlikely to have been protected using obfuscation, so there's a good chance the code can be completely decompiled back 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. The usernames and passwords that allow the app to connect to these backend systems are often hard-coded in the Android app. So, if you haven't protected your code and you leave the keys to your backend system in your app, you're running the risk of someone compromising your database and gaining 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 app to get it to talk to a different backend system, and use it as a means of harvesting usernames and passwords. This information can then be used at a later stage to gain access to private data using the real Android app.

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

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

In [Chapter 4](#), you learn more about why Android is at risk from decompilation, but for the moment 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 been converted to the DEX format. As you can see, it's similar information but in a new format. [Chapter 1](#) 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 almost every compiler, there has been a decompiler. Let's take a moment to talk about their history so you can see how and why decompilers were created so quickly for the JVM and, to a lesser extent, the

DVM.

---

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

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

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

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

At a much more basic level, hexadecimal dumps of PC machine code give programmers extra insight into how something was achieved and have been used to break artificial restrictions placed on software. For example, magazine CDs containing time-bombed or restricted copies of games and other utilities were often patched to change demonstration copies into full versions of the software; this was 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 appropriate source-code fragments. Pirate software is a huge problem for the software industry, and disassembling the code is just one technique employed by professional and amateur bootleggers. Hence the downfall of many an arcane copy-protection technique. But these are primitive tools and techniques, and it would probably be quicker to write the code from scratch rather than to re-create the source code from Assembler.

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

It's likely that the first real Java decompiler was written in IBM and not by Hanpeter van Vliet, author of Mocha. Daniel Ford's white paper "Jive: A Java Decompiler" (May 1996) appears in IBM 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 such as Hex-Ray's IDA have also begun to appear. Fortunately for the likes of Microsoft, decompiling Office using dcc or Hex-Rays would create so much code that it's about as user friendly as debug or a hexadecimal dump. Most modern commercial software's source code is so huge that it becomes unintelligible without the design documents and lots of source-code comments. Let's face it: many people's C++ code is hard enough to read six months after they wrote it. How easy would it be for

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 VB were interpreted by its runtime module `vbrun.dll` in a fashion somewhat similar to Java and the JVM. Like a Java class file, the source code for a VB program is bundled within the binary. Bizarrely, 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 Pascal and originated in the P-System ([www.threedee.com/jcm/psystem/](http://www.threedee.com/jcm/psystem/)). And before you say anything, yes, Pascal and all its derivatives are just as vulnerable to decompilation—that includes early versions of Microsoft's C compiler, so nobody feels left out. The p-codes aren't dissimilar to bytcodes and are essentially VB opcodes that are interpreted by `vbrun.dll` at run time. If you've ever wondered why you needed to include `vbrun300.dll` with VB executables, now you know. You have to include `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—perhaps the most famous VB decompiler. At one time, VB had a culture of decompilers and obfuscators (deprotection tools, as they're called in VB). But as VB moved to compiled rather than interpreted code, the number of decompilers decreased dramatically. DoDi provides VBGuard for free on his site, and programs such as Decompiler Defeater, Protect, Overwrite, Shield, and VBShield are available from 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. Not surprisingly, many decompilers and obfuscators are again appearing in the .NET world, such as the 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 Vliet wrote the first public-domain decompiler, Mocha, while recovering from a cancer operation in the Netherlands in 1996. He also wrote an obfuscator called Crema that attempted to protect an applet's source code. If Mocha was the UZI machine gun, then Crema was the bulletproof jacket. In a novel 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's web site, especially after it was featured in a CNET article. Because of the controversy, Hanpeter took the very honorable step of removing Mocha from his web site. He then allowed visitors to his site to vote about whether Mocha should once again be made available. The vote was ten to one in favor of 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 Techniques article on this subject, I learned from his wife, Ingrid, that Hanpeter's throat cancer finally got him 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 death, with all proceeds going to Ingrid. Some early versions of JBuilder shipped with an obfuscator, which was probably Crema. It attempted to protect Java code from decompilation by replacing ASCII variable names with control characters.



## Legal Issues to Consider When Decompiling

Before you start building your own decompiler, let's take this opportunity to consider the legal implications of decompiling someone else's code for your own enjoyment or benefit. Just because Java has taken decompiling technology out of some very serious propeller-head territory and into more mainstream computing doesn't make it any less likely that you or your company will be sued. It 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 to decompiling software. Companies can use a number of legal mechanisms to stop you from 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 law, copyright law, anti-reverse-engineering clauses in shrinkwrap licenses, as well as a number of laws such as the Digital Millennium Copyright Act (DMCA) may all be used against you. Different laws may apply in different countries or states: for example, the "no reverse engineering clause" software license is a null and void clause in the European Union (EU). But the basic concepts are the same: if you decompile a program for the purpose of cloning the code into another competitive product, and you're probably breaking the law. The secret is that you shouldn't be standing, kneeling, or pressing down very hard on the legitimate rights (the copyright) of the original author. That's not to say it's *never* OK to decompile. There are certain limited conditions under which the law favors decompilation through reverse engineering through a concept known as *fair use*. From almost the dawn of time, and certainly from the beginning of the Industrial Age, many of humankind's greatest inventions have come from individuals who created something special while Standing on the Shoulders of Giants. For example, the invention of the steam train and the light bulb were relatively modest incremental steps in technology. The underlying concepts were provided by other people, and it was up to someone like George Stephenson or Thomas Edison to create the final object. (You can see an excellent example of Stephenson's debt to many other inventors such as James Watt at [www.usgennet.org/usa/topic/steam/Early/Time.html](http://www.usgennet.org/usa/topic/steam/Early/Time.html)). This is one of the reasons patents appeared: to allow people to build on other creations while still giving the original inventors some compensation for their initial ideas for period of, say, 20 years.

## Patents



---

## sample content of Decompiling Android

- [Walden on Wheels: On the Open Road from Debt to Freedom for free](#)
- [download Wards of Faerie \(Shannara: The Dark Legacy of Shannara, Book 1\) for free](#)
- [MemÃ³ria de Elefante book](#)
- [read Vera Brittain and the First World War: The Story of Testament of Youth pdf, azw \(kindle\), epub](#)
- [click Records of the Grand Historian](#)
- [read Anne of the Island \(Anne of Green Gables series, Book 3\)](#)
  
- <http://thewun.org/?library/Walden-on-Wheels--On-the-Open-Road-from-Debt-to-Freedom.pdf>
- <http://redbuffalodesign.com/ebooks/Wards-of-Faerie--Shannara--The-Dark-Legacy-of-Shannara--Book-1-.pdf>
- <http://yachtwebsitedemo.com/books/Mem--ria-de-Elefante.pdf>
- <http://serazard.com/lib/Better-Made-At-Home--Salty--Sweet--and-Satisfying-Snacks-and-Pantry-Staples-You-Can-Make-Yourself.pdf>
- <http://fitnessfatale.com/freebooks/Guardians--The-Turn--The-Guardians-Series--Book-3-.pdf>
- <http://toko-gumilar.com/books/Lost-and-Found--The-Taken-Trilogy--Book-1-.pdf>