

ProGuard



Optimizer and Obfuscator for Android

Eric Lafortune
Saiko

Eric Lafortune

- 1991 – 1996 K.U.Leuven (Belgium), Phd Eng CompSci
- 1996 – 1999 Cornell University (Ithaca, NY)
- 1999 – 2011 Java GIS
- 2012 Founder Saikoa

Maybe more importantly:

- 1982 TMS-9900 processor
- 1995 ARM2/ARM3 processor
- 2001 Java bytecode
- 2010 Dalvik bytecode

ProGuard

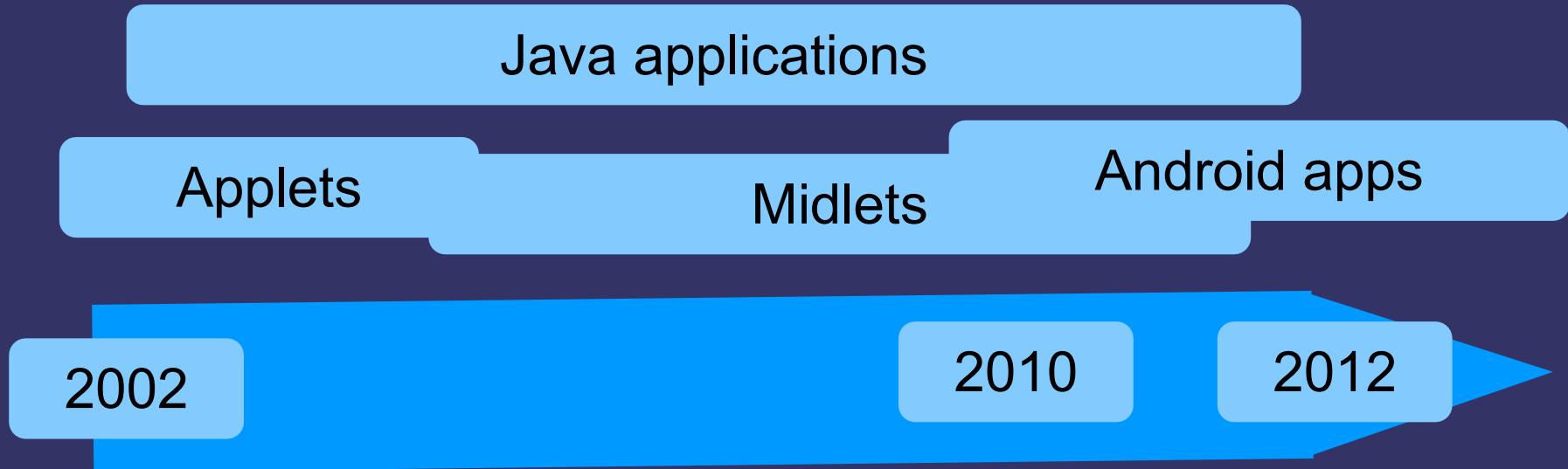
Open source

Generic

Shrinker
Optimizer
Obfuscator

For Java bytecode

ProGuard history



- May 2002 First download!
- Sep 2010 Recommended for protecting LVL
- Dec 2010 Part of Android SDK
- Jan 2012 Startup Saikoa

Why use ProGuard?

- Application size
- Performance
- Remove logging, debugging, testing code
- Battery life
- Protection

Application size

	classes.dex size			.apk size		
	Without ProGuard	With ProGuard	Reduction	Without ProGuard	With ProGuard	Reduction
ApiDemos	716 K	482 K	33%	2.6 M	2.5 M	4%
ApiDemos in Scala*	~6 M	542 K	~90%	~8 M	2.5 M	~70%

* [Stéphane Micheloud, <http://lampwww.epfl.ch/~michelou/android/library-code-shrinking.html>]

Performance: CaffeineMark

Without ProGuard

Sieve score = 6833
Loop score = 14831
Logic score = 19038
String score = 7694
Float score = 6425
Method score = 4850
Overall score = 8794

With ProGuard

Sieve score = 6666
Loop score = 15473
Logic score = 47840
String score = 7717
Float score = 6488
Method score = 5229
Overall score = 10436

Improvement: 18%

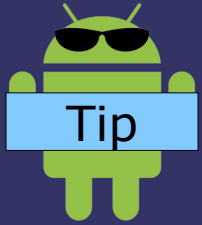
[Acer Iconia Tab A500, nVidia Tegra 2, 1.0 GHz, Android 3.2.1]

Battery life

Extreme example:

“5 x better battery life,
by removing verbose logging code
in a background service”

(but don't count on it)



How to enable ProGuard?

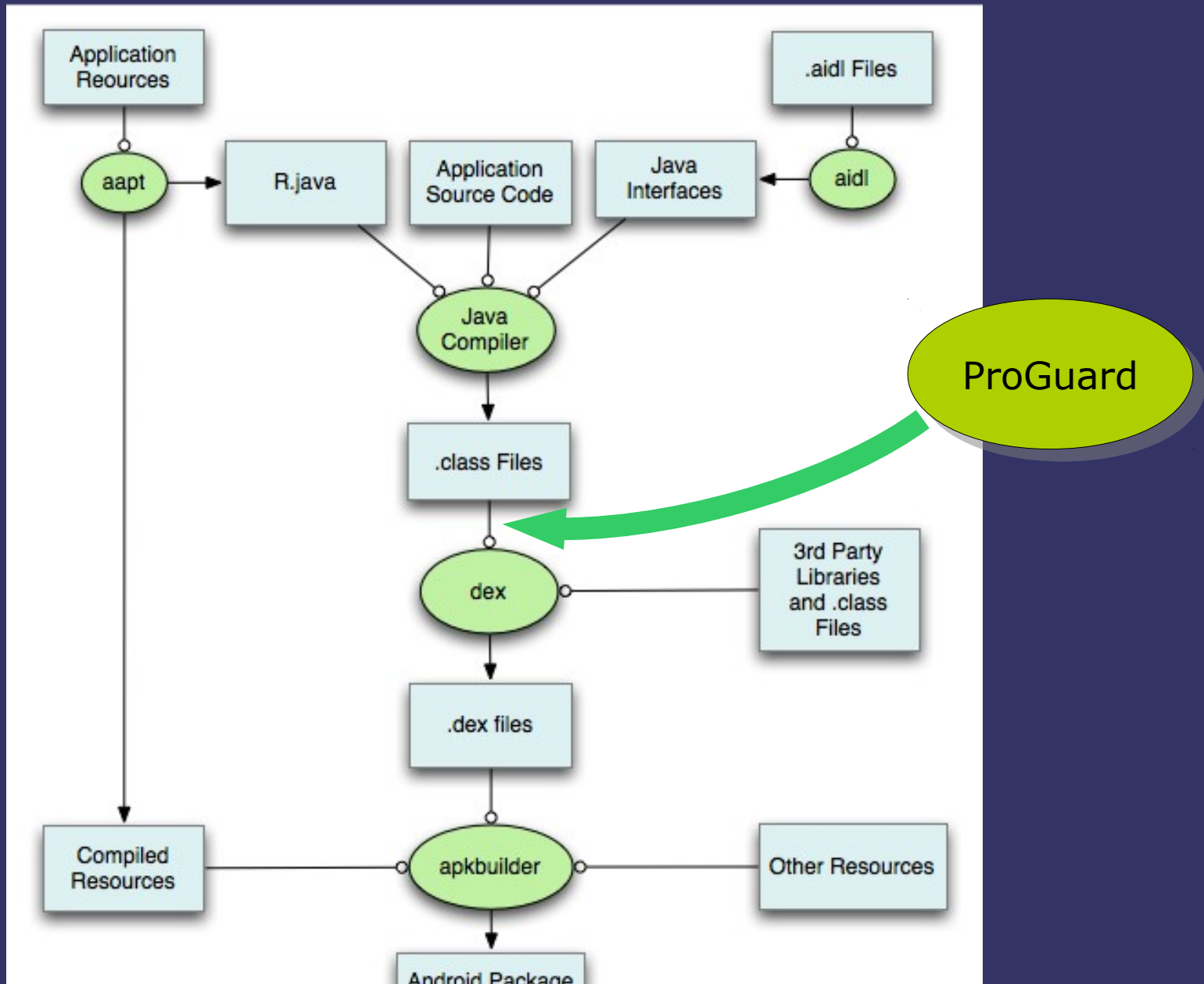
project.properties:

```
# To enable ProGuard to shrink and obfuscate  
your code, uncomment this  
#proguard.config=  
    ${sdk.dir}/tools/proguard/proguard-android.txt:  
    proguard-project.txt
```

→ only applied when building release versions

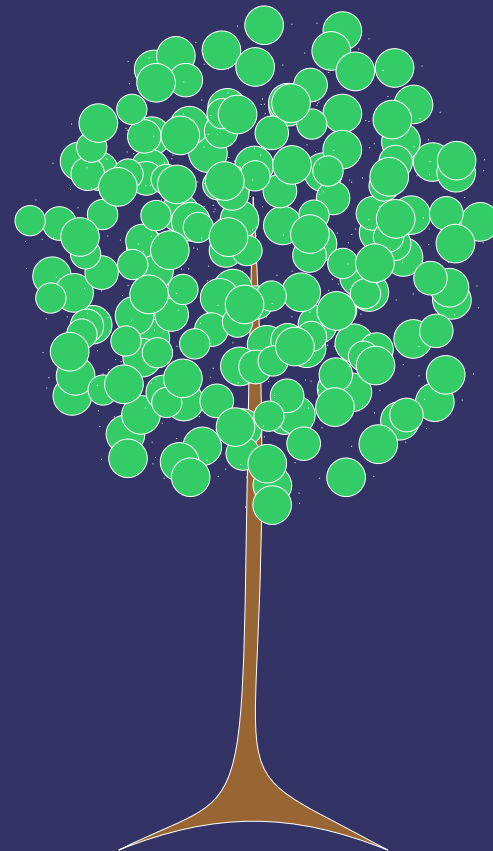
Build process

[Android documentation]



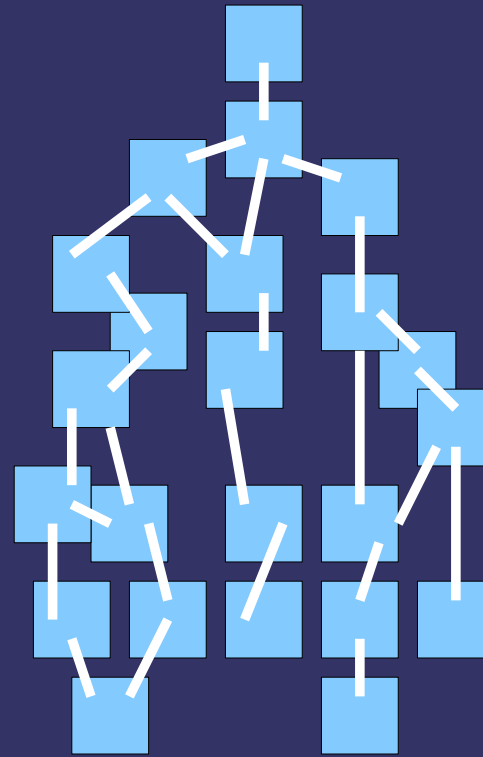
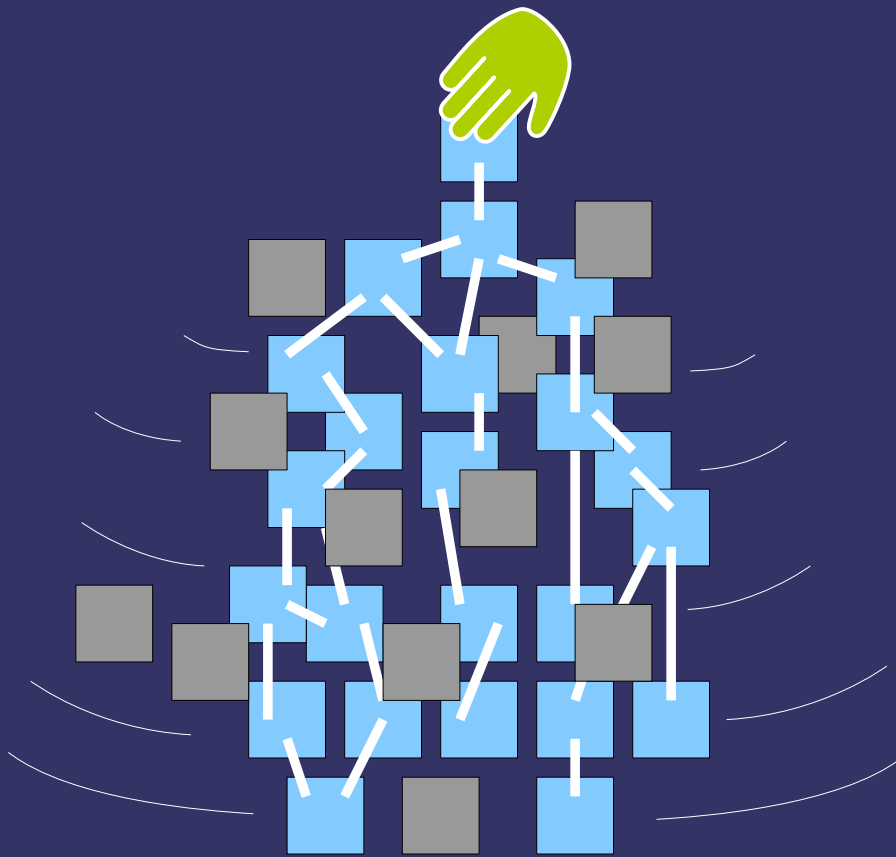
Shrinking

Also called treeshaking, minimizing, shrouding



Shrinking

- Classes, fields, methods



Entry points

- 1) Activities, applications, services, fragments,...
- provided automatically by Android build process

```
-keep public class * extends android.app.Activity
-keep public class * extends android.app.Application
-keep public class * extends android.app.Service
...
```



Entry points

2) Introspection, e.g. Google License Verification Library
→ must be specified in proguard-project.txt

```
-keep public interface  
    com.android.vending.licensing.ILicensingService
```

Entry points

More introspection, e.g. Google vending library

→ must be specified in proguard-project.txt

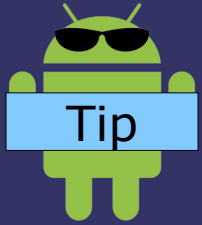
```
-keepclassmembers public class  
    com.google.android.vending.expansion.downloader.impl.DownloadsDB$* {  
        public static final java.lang.String[][] SCHEMA;  
        public static final java.lang.String     TABLE_NAME;  
    }
```

Entry points

More introspection, e.g. Guice, RoboGuice

→ must be specified in proguard-project.txt:

```
-keepclassmembers class * {  
    @javax.inject.**          <fields>;  
    @com.google.inject.**     <fields>;  
    @roboguice.**             <fields>;  
    @roboguice.event.Observes <methods>;  
}
```

Notes and warnings

“Closed-world assumption”

```
Warning: twitter4j.internal.logging.Log4JLoggerFactory:  
    can't find referenced class org.apache.log4j.Logger  
Warning: twitter4j.internal.logging.SLF4JLoggerFactory:  
    can't find referenced class org.slf4j.LoggerFactory  
...
```

```
Warning: com.dropbox.client2.DropboxAPI:  
    can't find referenced class org.json.simple.JSONArray
```

→ if debug build works fine,
then ok to ignore in proguard-project.txt:

```
-dontwarn twitter4j.internal.logging.**  
-dontwarn com.dropbox.client2.**
```

Optimization

At the bytecode instruction level:

- Dead code elimination
- Constant propagation
- Method inlining
- Class merging
- Remove logging code
- Peephole optimizations
- Devirtualization
- ...

Dead code elimination

```
boolean debug = false;  
...  
if (debug) Log.v(".....");  
...
```

Note: showing equivalent source code instead of bytecode

Dead code elimination

```
boolean debug = false;  
...  
  
if (debug) Log.v(".....");  
...
```

Note: showing equivalent source code instead of bytecode

Constant propagation

Inside methods:

```
int f1 = 6;  
...  
int f2 = 7;  
...  
int answer = f1 * f2;
```

Constant propagation

Inside methods:

```
int f1 = 6;  
...  
int f2 = 7;  
...  
int answer = f1 * f2;
```

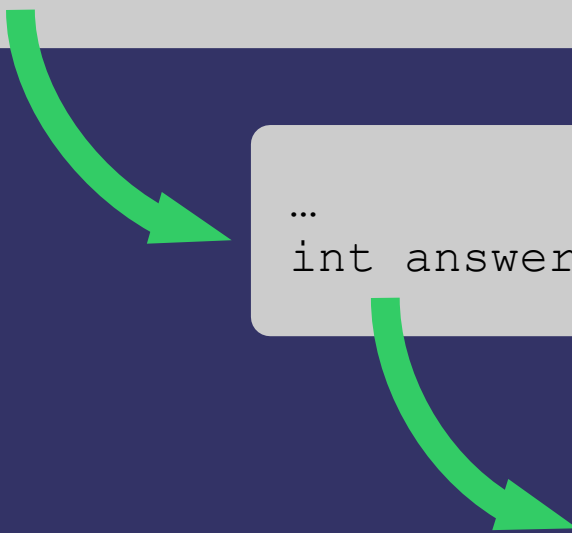


```
...  
int answer = 6 * 7;
```

Constant propagation

Inside methods:

```
int f1 = 6;  
...  
int f2 = 7;  
...  
int answer = f1 * f2;
```



```
...  
int answer = 6 * 7;
```

```
...  
int answer = 42;
```

Constant propagation

Across methods:

```
int answer = computeAnswer(6, 7);
```

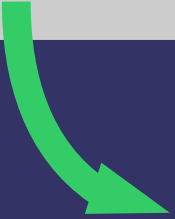
```
int computeAnswer(int f1, int f2) {  
    return f1 * f2;  
}
```


Constant propagation

Across methods:

```
int answer = computeAnswer(6, 7);
```

```
int computeAnswer(int f1, int f2) {  
    return f1 * f2;  
}
```



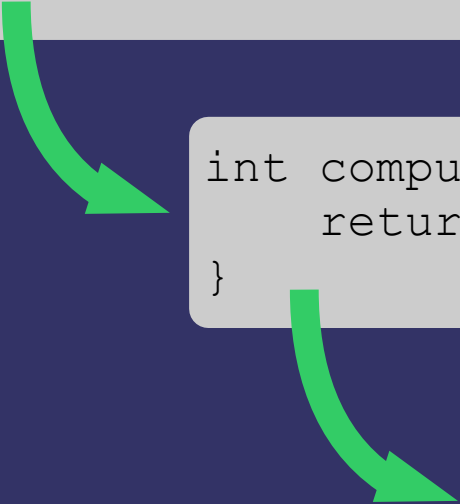
```
int computeAnswer(int f1, int f2) {  
    return 6 * 7;  
}
```

Constant propagation

Across methods:

```
int answer = computeAnswer(6, 7);
```

```
int computeAnswer(int f1, int f2) {  
    return f1 * f2;  
}
```



```
int computeAnswer(int f1, int f2) {  
    return 6 * 7;  
}
```

```
int computeAnswer() {  
    return 42;  
}
```

Constant propagation

Across methods:

```
int answer = computeAnswer(6, 7);
```

```
int computeAnswer(int f1, int f2) {  
    return f1 * f2;  
}
```

```
int computeAnswer(int f1, int f2) {  
    return 6 * 7;  
}
```

```
int computeAnswer() {  
    return 42;  
}
```

```
int answer = 42;
```



Method inlining

Short methods (or methods that are only invoked once):

```
int answer = image.getPixel(i, j);
```

```
int getPixel(int x, int y) {  
    return array[y * width + x];  
}
```

Method inlining

Short methods (or methods that are only invoked once):

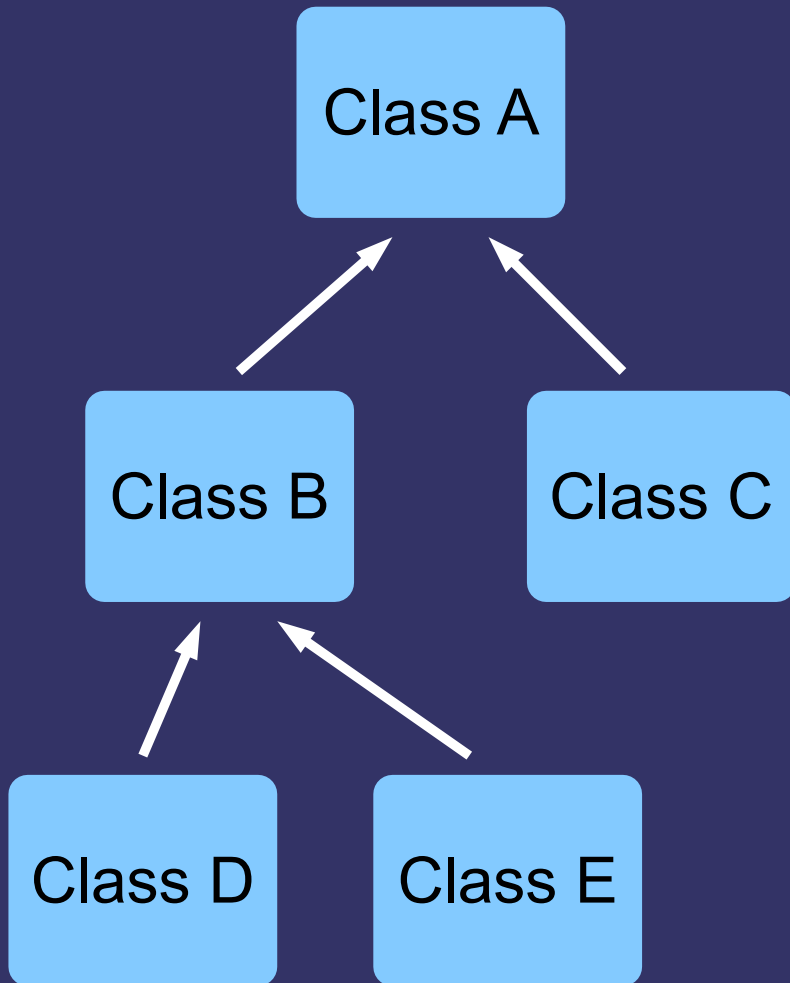
```
int answer = image.getPixel(i, j);
```

```
int getPixel(int x, int y) {  
    return array[y * width + x];  
}
```

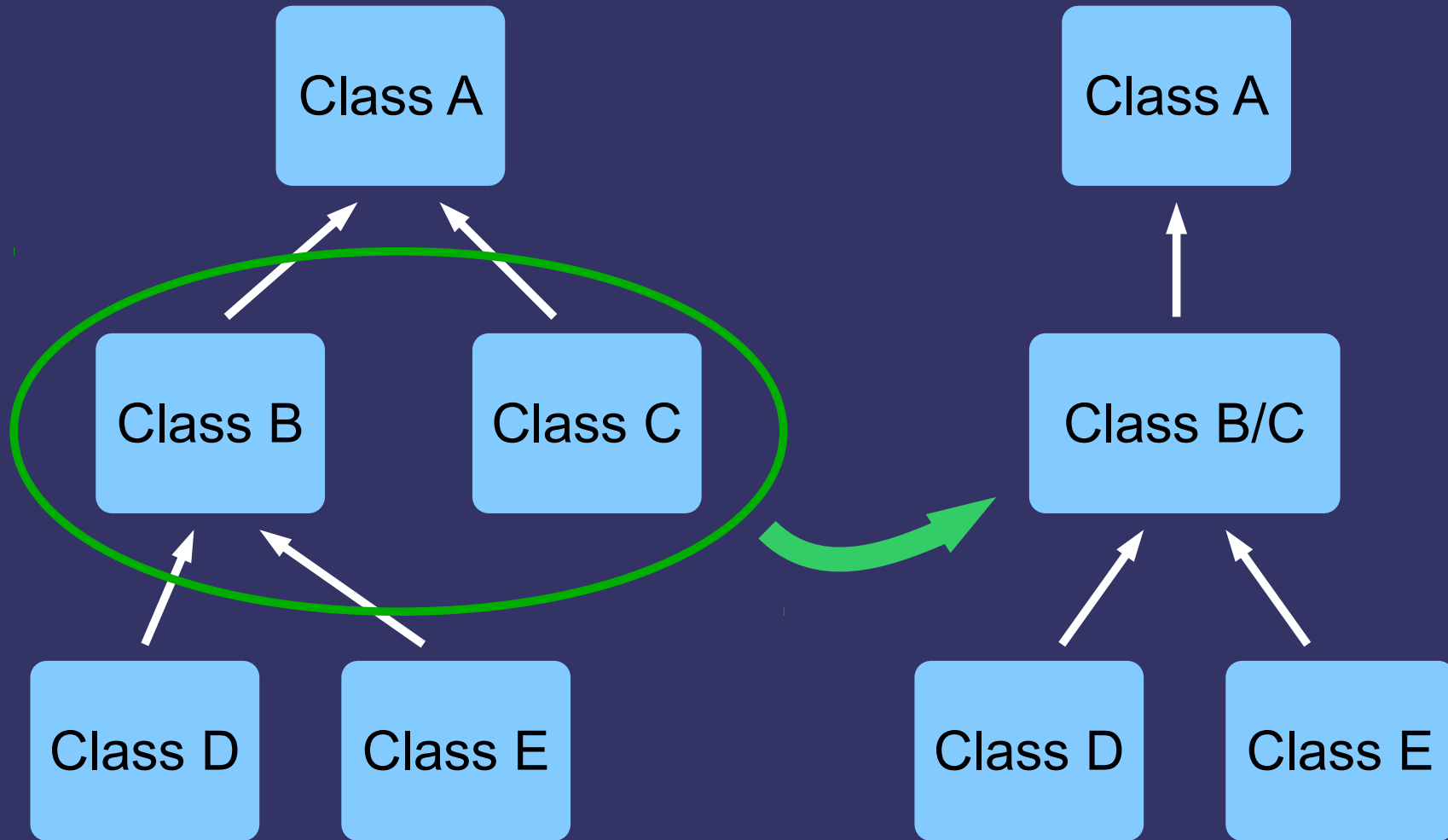
```
int answer = image.array[j * image.width + i];
```



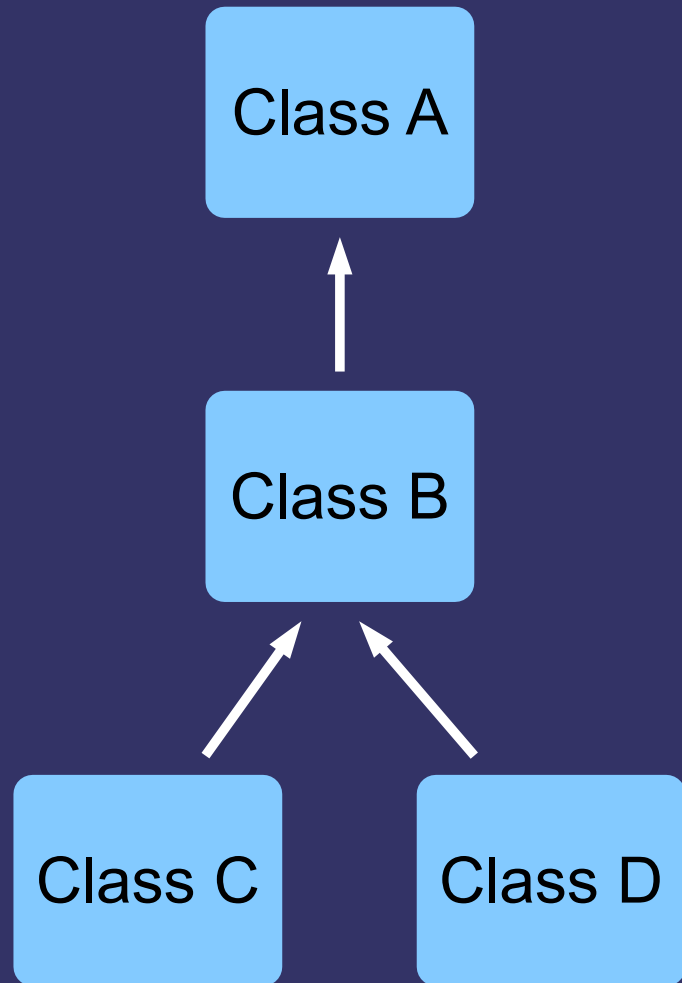
Class merging: horizontally



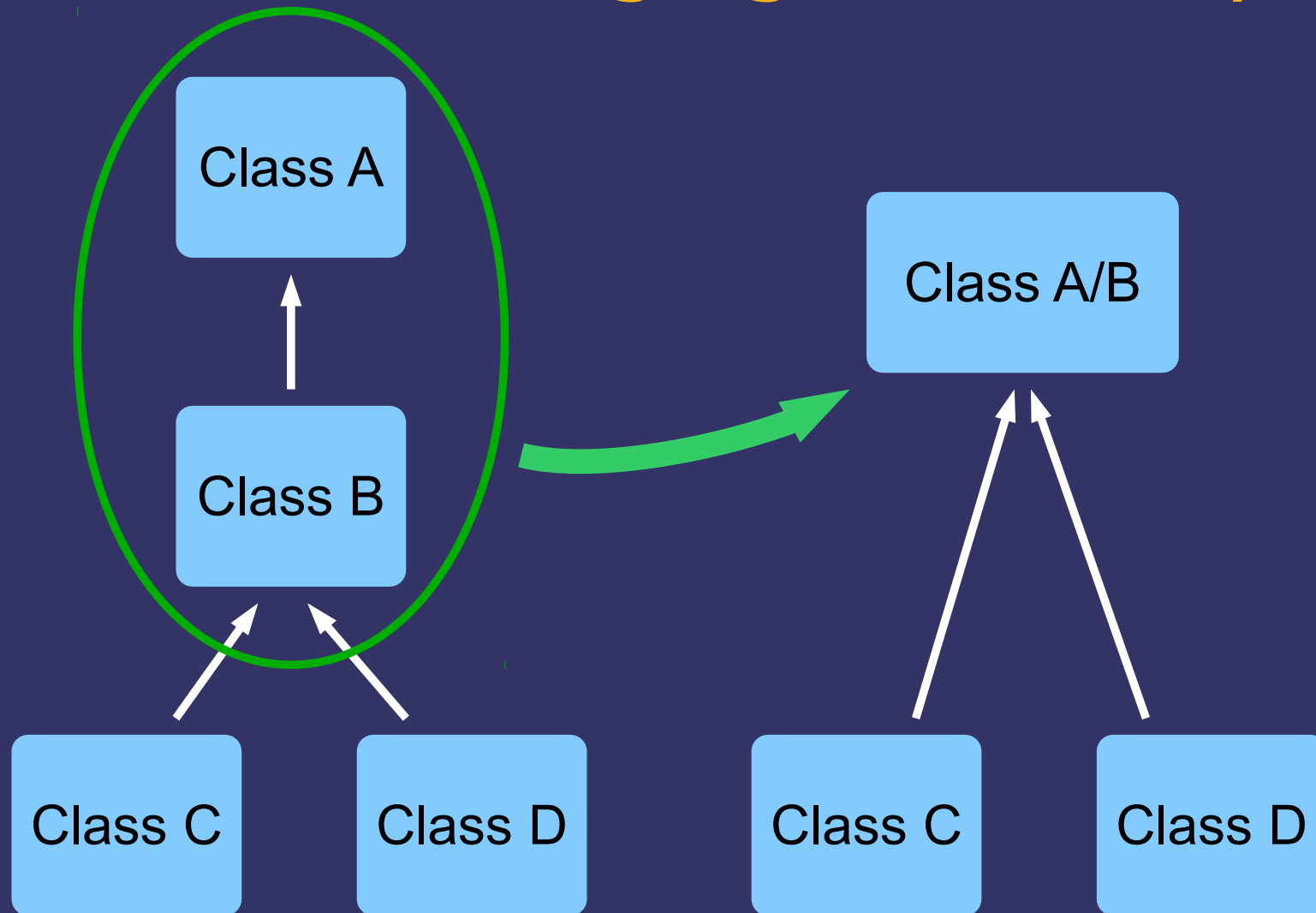
Class merging: horizontally



Class merging: vertically



Class merging: vertically



Tail recursion simplification


```
int answer = computeAnswer(1, 2, 3, 7);
```

```
int computeAnswer(int f1, int f2, int f3, int f4) {  
    if (f2 == 1 && f3 == 1 && f4 == 1) {  
        return f1;  
    } else {  
        return computeAnswer(f1 * f2, f3, f4, 1);  
    }  
}
```

Tail recursion simplification

```
int answer = computeAnswer(1, 2, 3, 7);
```

```
int computeAnswer(int f1, int f2, int f3, int f4) {  
    if (f2 == 1 && f3 == 1 && f4 == 1) {  
        return f1;  
    } else {  
        return computeAnswer(f1 * f2, f3, f4, 1);  
    }  
}
```



```
int computeAnswer(int f1, int f2, int f3, int f4) {  
    do {  
        if (f2 == 1 && f3 == 1 && f4 == 1) {  
            return f1;  
        } else {  
            f1 = f1 * f2; f2 = f3, f3 = f4, f4 = 1;  
        }  
    } while (true);  
}
```

Tail recursion simplification

```
int answer = computeAnswer(1, 2, 3, 7);
```

```
int answer = 42;
```

```
int computeAnswer(int f1, int f2, int f3, int f4) {  
    if (f2 == 1 && f3 == 1 && f4 == 1) {  
        return f1;  
    } else {  
        return computeAnswer(f1 * f2, f3, f4, 1);  
    }  
}
```

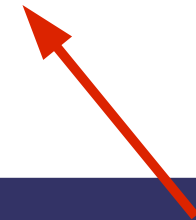
```
int computeAnswer(int f1, int f2, int f3, int f4) {  
    do {  
        if (f2 == 1 && f3 == 1 && f4 == 1) {  
            return f1;  
        } else {  
            f1 = f1 * f2; f2 = f3, f3 = f4, f4 = 1;  
        }  
    } while (true);  
}
```



How to enable optimization?

project.properties:

```
# To enable ProGuard to shrink and obfuscate  
your code, uncomment this  
proguard.config=  
    ${sdk.dir}/tools/proguard/proguard-android-optimize.txt:  
    proguard-project.txt
```



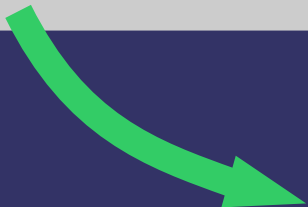
Obfuscation

Traditional name obfuscation:

- Rename identifiers:
class/field/method names
- Remove debug information:
line numbers, local variable names,...

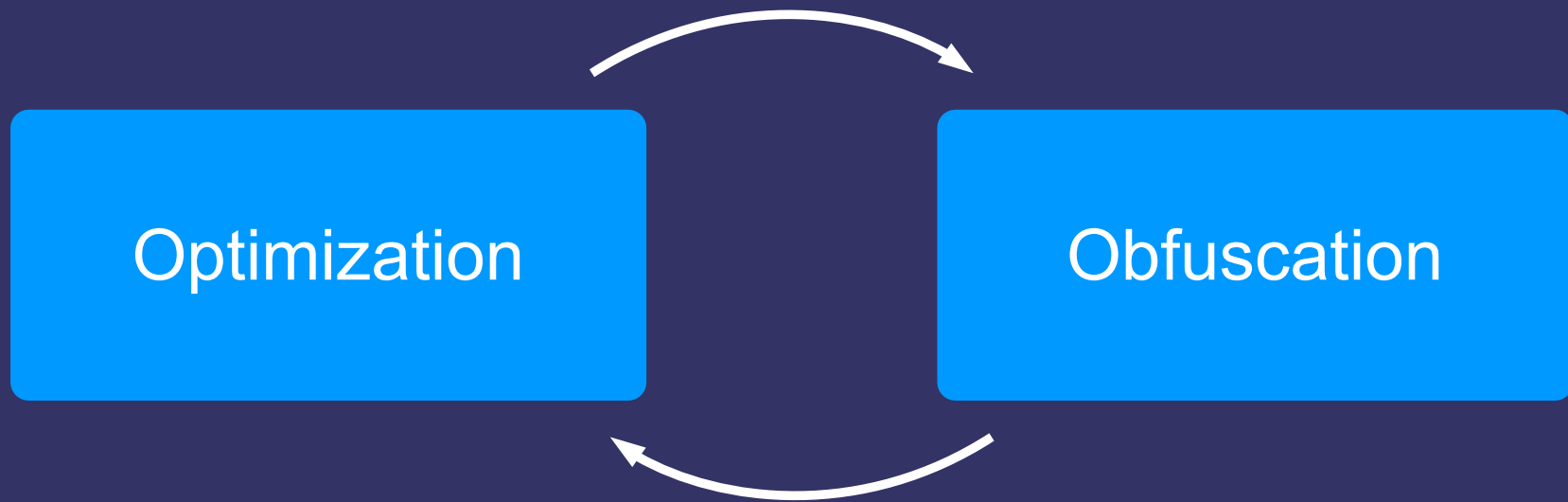
Obfuscation

```
public class MyComputationClass {  
    private MySettings settings;  
    private MyAlgorithm algorithm;  
    private int answer;  
  
    public int computeAnswer(int input) {  
        ...  
        return answer;  
    }  
}
```



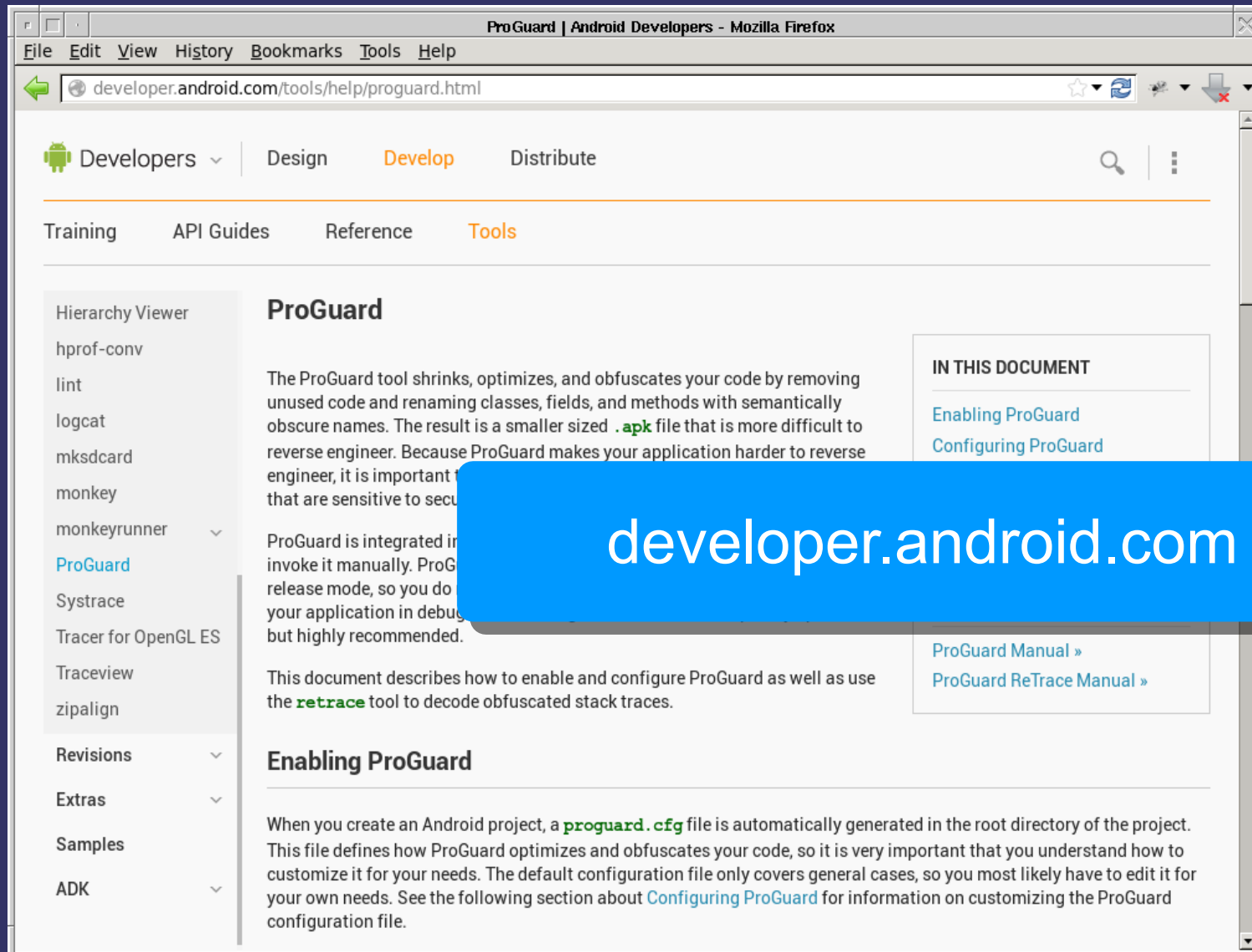
```
public class a {  
    private b a;  
    private c b;  
    private int c;  
  
    public int a(int a) {  
        ...  
        return c;  
    }  
}
```


Complementary steps



Irreversibly remove information

ProGuard guide – Android SDK



The screenshot shows a web browser window displaying the ProGuard guide on the Android Developers website. The browser's address bar shows the URL `developer.android.com/tools/help/proguard.html`. The page features a navigation menu with categories like Design, Develop, and Distribute, and a sidebar with links to various tools including Hierarchy Viewer, lint, logcat, and ProGuard. The main content area is titled "ProGuard" and contains introductory text about the tool's purpose. A blue overlay with the text "developer.android.com" is positioned over the middle of the page.

ProGuard | Android Developers - Mozilla Firefox

File Edit View History Bookmarks Tools Help

developer.android.com/tools/help/proguard.html

Developers ▾ Design Develop Distribute

Training API Guides Reference Tools

Hierarchy Viewer
hprof-conv
lint
logcat
mkshcard
monkey
monkeyrunner ▾
ProGuard
Systrace
Tracer for OpenGL ES
Traceview
zipalign

Revisions ▾
Extras ▾
Samples
ADK ▾

ProGuard

The ProGuard tool shrinks, optimizes, and obfuscates your code by removing unused code and renaming classes, fields, and methods with semantically obscure names. The result is a smaller sized **.apk** file that is more difficult to reverse engineer. Because ProGuard makes your application harder to reverse engineer, it is important that you understand how to use it properly, especially for applications that are sensitive to security.

ProGuard is integrated into the Android build system, so you can invoke it manually. ProGuard is enabled by default in release mode, so you do not need to enable it. However, you should always use ProGuard when debugging your application in debug mode, as it is highly recommended.

This document describes how to enable and configure ProGuard as well as use the **retrace** tool to decode obfuscated stack traces.

Enabling ProGuard

When you create an Android project, a **proguard.cfg** file is automatically generated in the root directory of the project. This file defines how ProGuard optimizes and obfuscates your code, so it is very important that you understand how to customize it for your needs. The default configuration file only covers general cases, so you most likely have to edit it for your own needs. See the following section about [Configuring ProGuard](#) for information on customizing the ProGuard configuration file.

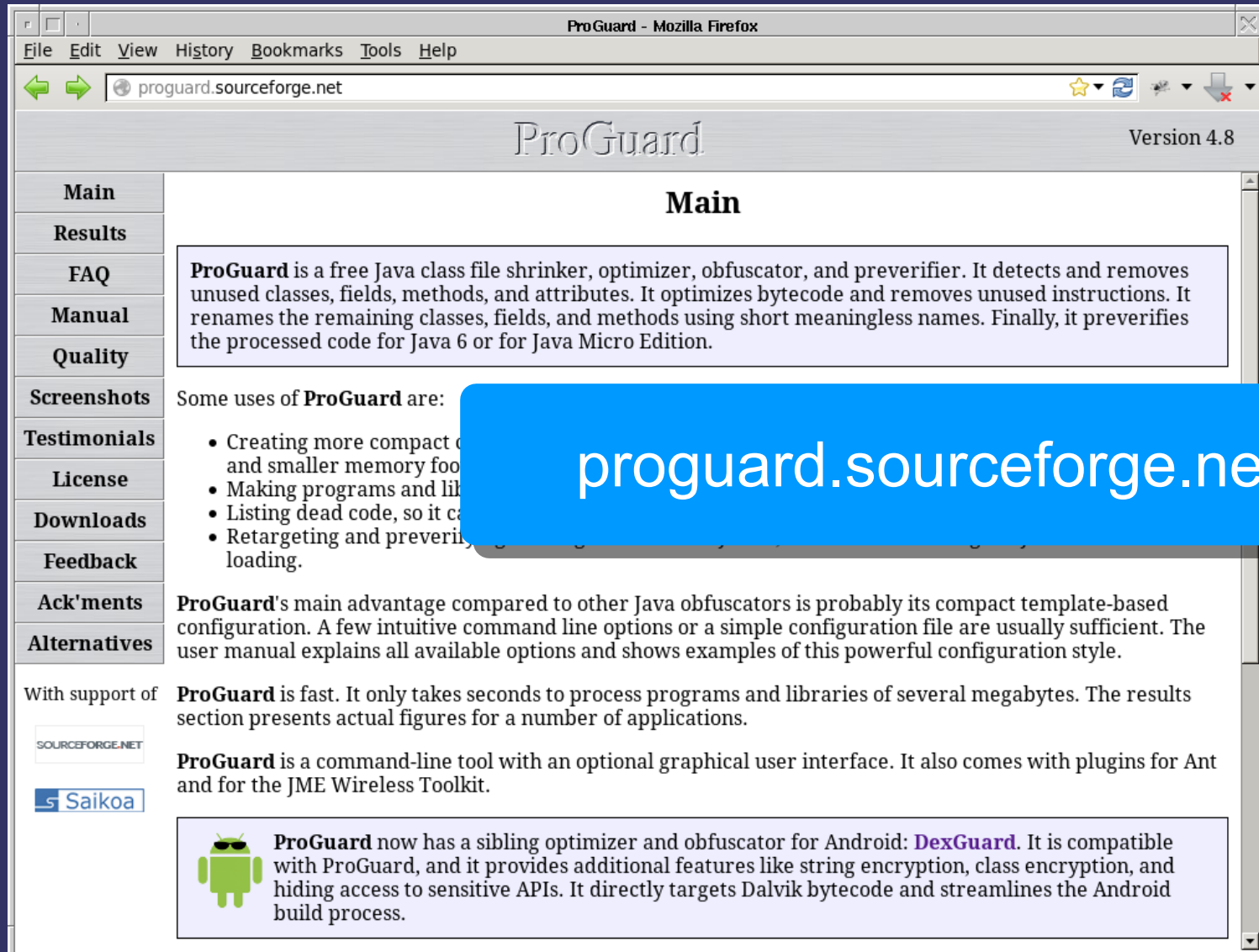
IN THIS DOCUMENT

- [Enabling ProGuard](#)
- [Configuring ProGuard](#)

[ProGuard Manual »](#)
[ProGuard ReTrace Manual »](#)

developer.android.com

ProGuard website



ProGuard manual

ProGuard - Mozilla Firefox

File Edit View History Bookmarks Tools Help

proguard.sourceforge.net/#manual/introduction.html

ProGuard Version 4.8

<< Main menu

ProGuard Manual

Introduction

Usage

Limitations

Examples

Troubleshooting

Ref Card

GUI

Ant Task

JME WTK

ReTrace Manual

Introduction

Usage

Examples

With support of

SOURCEFORGE.NET

Introduction

ProGuard is a Java class file shrinker, optimizer, obfuscator, and preverifier. The shrinking step detects and removes unused classes, fields, methods, and attributes. The optimization step analyzes and optimizes the bytecode of the methods. The obfuscation step renames the remaining classes, fields, and methods using short meaningless names. These first steps make the code base smaller, more efficient, and harder to reverse-engineer. The final preverification step adds preverification information to the classes, which is required for Java Micro Edition and for Java 2 SE 5.0.

Each of these steps is optional and can be disabled. The final preverification step is required to preverify class files for Java 2 SE 5.0.

```
graph LR; Input[Input jars] -- shrink --> Shrunk[Shrunk code]; Shrunk -- optimize --> Optim[Optim. code]; Optim -- obfuscate --> Obfusc[Obfusc. code]; Obfusc -- preverify --> Output[Output jars]; Library[Library jars] -.->|unchanged| LibraryOut[Library jars];
```

ProGuard first reads the **input jars** (or wars, ears, zips, or directories). It then subsequently shrinks, optimizes, obfuscates, and preverifies them. You can optionally let ProGuard perform multiple optimization passes. ProGuard writes the processed results to one or more **output jars** (or wars, ears, zips, or directories). The input may contain resource files, whose names and contents can optionally be updated to reflect the obfuscated class names.

ProGuard requires the **library jars** (or wars, ears, zips, or directories) of the input jars to be specified. These are essentially the libraries that you would need for compiling the code. ProGuard uses them to reconstruct

Startup: Saikoa

- Open source: ProGuard
- Services: Professional ProGuard support
- Product: DexGuard

ProGuard - DexGuard

Compatible

Open source

Closed source

Generic

Specialized

Shrinker
Optimizer
Obfuscator

Shrinker
Optimizer
Obfuscator
Protector

For Java bytecode

For Android

Motivations for hacking/cracking

- Anti-malware research
- Reverse-engineering protocols, formats,...
- Fun
- Translation
- Game cheating
- Software piracy
- Remove ads
- Different ads
- Different market
- Extortion
- Extract assets
- Extract API keys
- Insert malware (SMS,...)
- ...

Solutions?

- Ignore it
- Different business model (open source, service)
- Regular updates
- Lock down device
- Server
- Remove motivations
- Obfuscation, application protection

More application protection

Nothing is unbreakable, but you can raise the bar:

- Reflection
- String encryption
- Class encryption
- Tamper detection
- Debug detection
- Emulator detection
- ...

→ Automatically applied by DexGuard

Saikoa website

Saikoa
Applied Compiler Technology

Home DexGuard ProGuard ProGuard Support Contact My Account

DexGuard

 DexGuard is our specialized optimizer and obfuscator for Android. It helps you create Android apps that are faster, more compact, and more difficult to crack. It automates the recommended application protection techniques for Android and directly targets the Dalvik virtual machine.

Optimize and protect your apps

DexGuard delivers all the necessary integrated features:

- ✓ **Optimize and obfuscate.** DexGuard extends ProGuard with configuration and processing. It goes further by directly producing optimized code.
- ✓ **Encrypt strings.** Thwart hacking through string literals. Strings should become invisible in your code. DexGuard doesn't have to burden your source code and your development process.
- ✓ **Encrypt entire classes.** Hide important code like license checks or paid downloads, by specifying entire classes that should be encrypted. DexGuard does this transparently and effectively.
- ✓ **Hide access to sensitive APIs.** Further harden your code, by letting DexGuard insert reflection to access sensitive APIs, like the standard Android APIs for signature validation or cryptographic operations.
- ✓ **Add tamper detection.** Let your application react accordingly if a hacker has tried to

Open World Forum

We've been invited to speak on the Android track of the *Open World Forum* in Paris, France, Oct 11th-13th. I'll be talking about my technical and practical experiences with a successful open-source project like ProGuard. If you're at the

Many of our customers place great value on software updates and upgrades, so we're happy to announce that we're extending the standard support period for DexGuard from 6 months to 1 year. We want DexGuard to offer solid value — right now, but also in the

www.saikoa.com

Questions?

ProGuard

Open source

Shrinking
Optimization
Obfuscation

Saiko

Java bytecode

Dalvik bytecode

DexGuard