Squeezing the Last Byte and Last Ounce of Performance from Your MIDlets

Stephen Cheng

CEO
Innaworks
www.innaworks.com

TS-3418

Goal of This Talk

What You Will Learn

Pushing the size and performance limits on today's handsets

Agenda

Why Size and Performance Matters

Under the hood of a Java ME MIDlet

Optimization Strategy

Optimization Techniques

Demo

Why Size and Performance Matters

Adoption = Potential Market Size
x Value to User
x Marketing

Why Size and Performance Matters

Adoption = **Potential Market Size**
x Value to User
x Marketing

Volume Matters

Why Size and Performance Matters

Adoption = Potential Market Size
x **Value to User**
x Marketing

Perceived Quality Matters

Cost Matters

Constraints of Consumer Handsets

	JAR Size	Heap Memory
Nokia S40 v1 (3300 etc)	64kB	370kB
Nokia S40 v2 (6230 etc)	128kB	512kB
Sharp GX22	100kB	512kB
DoJa 2.5 (m420i)	30kB	1.5MB

15% Game Sales for Handsets < 64kB JAR Size

35% Game Sales for Handsets < 128kB JAR Size

Source: Nokia, Samsung, Bouygues Telecom, Innaworks Customer Study

Data Fee for Casual Users

	Fee/Month	Free Data	Per kB
Sprint PCS Casual	None	None	$0.02
Cingular Data Connect	$19.99	5MB	$0.008
T-Mobile Basic Plus	$20.00	Unlimited	N/A
Rogers Small Data Plan	C$25	3MB	C$0.01

Average Java ME Game in US Costs $3.99

Your 200kB FREE Application could cost US$4 to download on Sprint Network

Source: Cingular, T-Mobile, Sprint Websites and NetSize 2006 Report

Agenda

Why Size and Performance Matters

Under the hood of a Java ME MIDlet

Optimization Strategy

Optimization Techniques

Demo

What Is in a MIDlet JAR File?

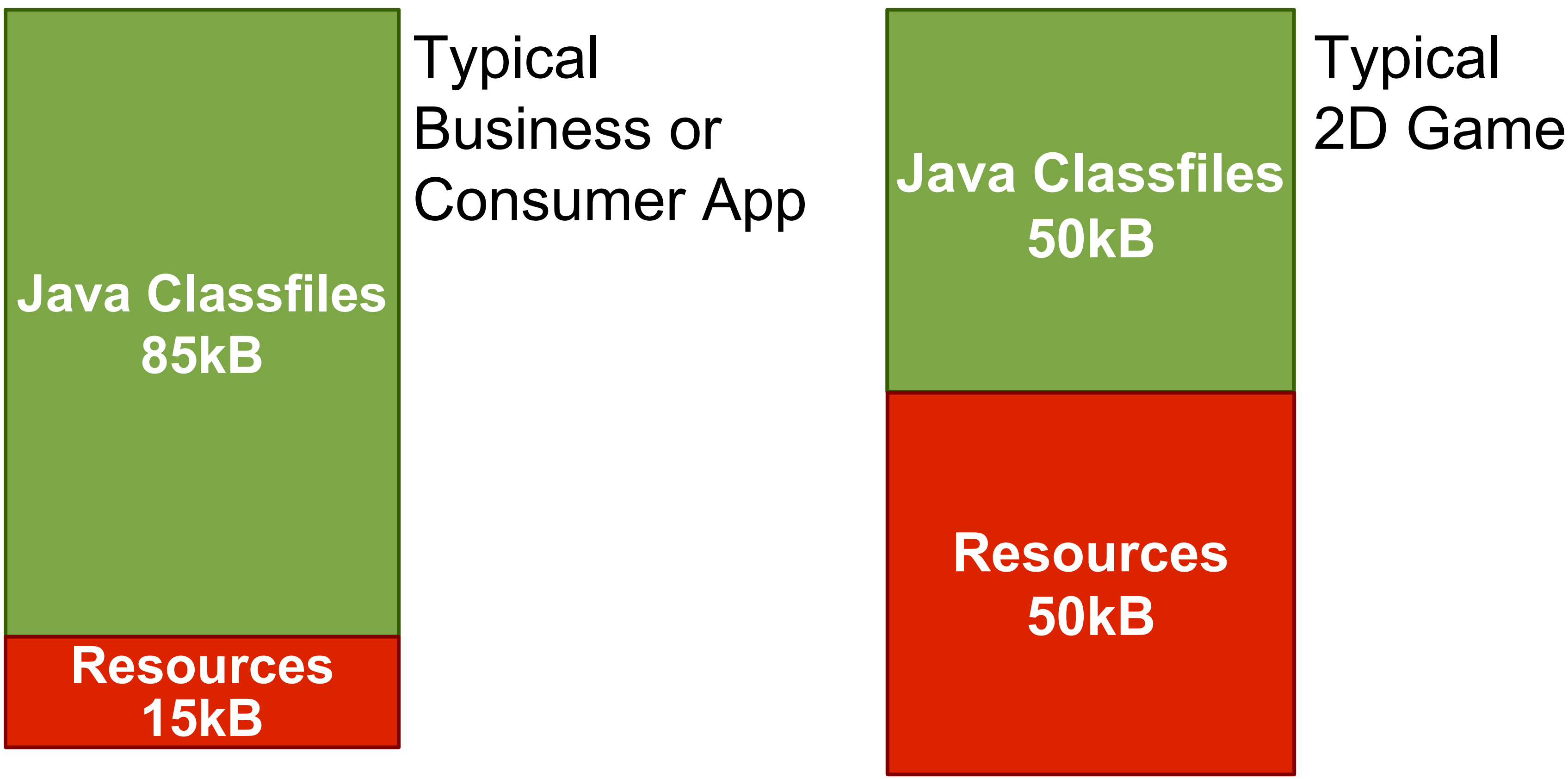
s60examples.jar\com\series60\examples\m3g\maze3d - ZIP archive, unp

Name	Size	Packed
..		
content		
ErrorScreen.class	1,610	750
Graphics3DProperties.class	2,258	1,017
Maze.class	5,379	2,795
MazeCanvas.class	14,342	7,046
MazeMIDlet.class	4,291	1,769
MenuList.class	2,455	1,236
Plane.class	2,114	1,054

- > 70 bytes JAR file overhead per file
- Compression does not work across files
- Overhead depends on path length

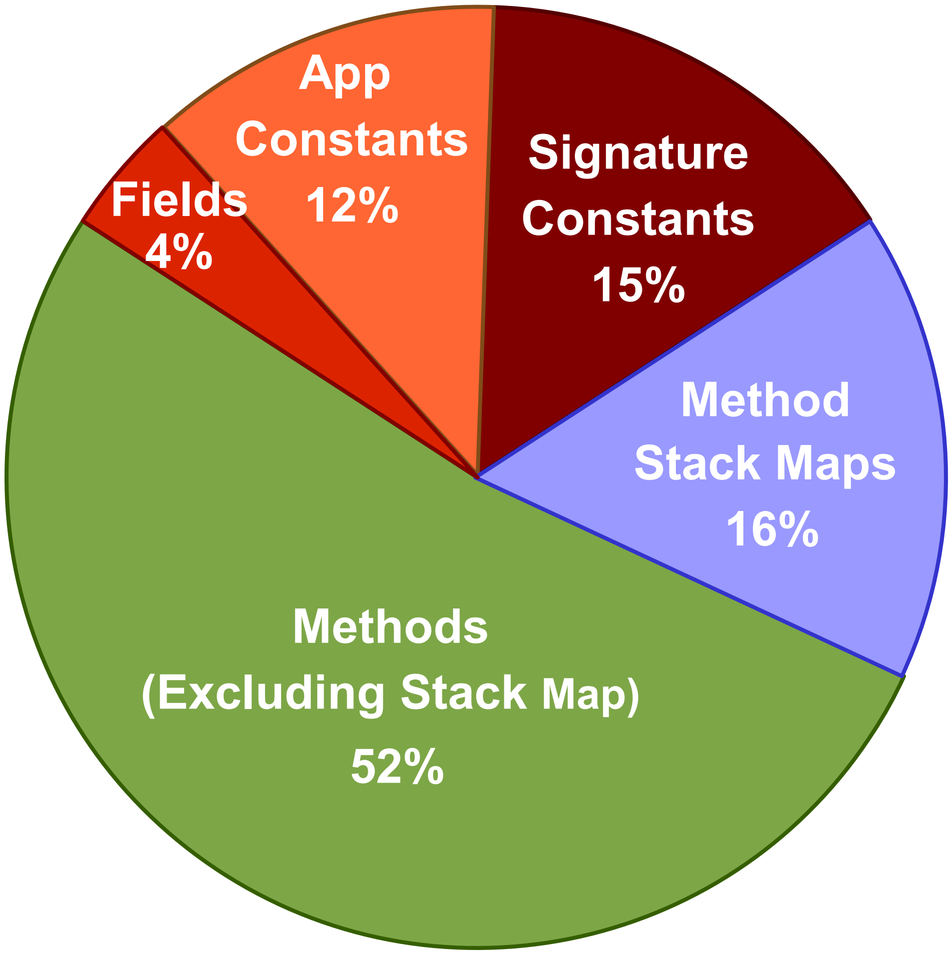
Source: Innaworks' Customer Study

Classfile versus Resource Files



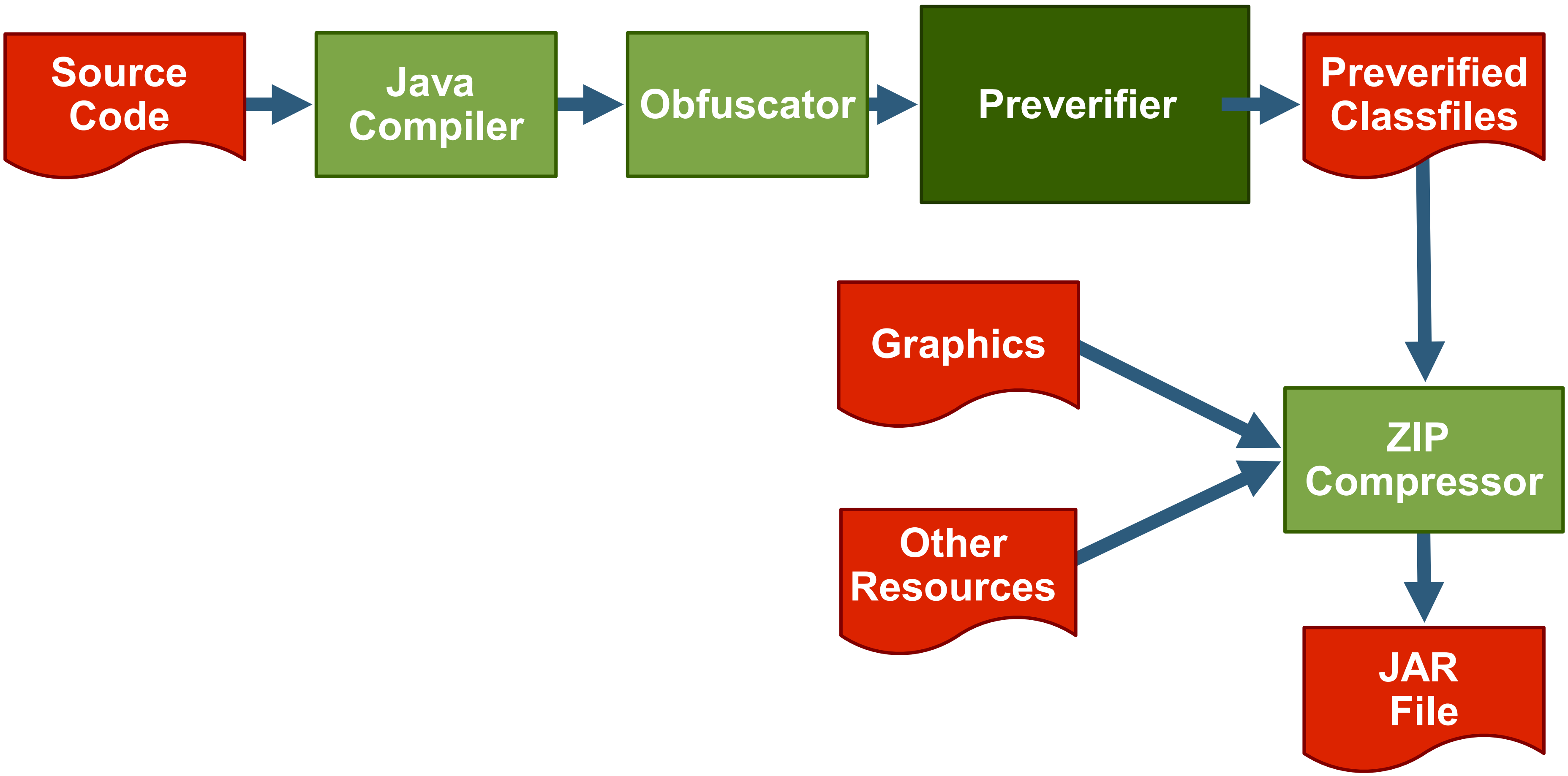
Source: Innaworks' Customer Study

Classfile Size Breakdown



Source: Innaworks' Customer Study

Java ME Toolchain



Stackmap

What Does the Preverifier Do?

- Preverifier inserts stackmap
  - Assists verification
  - Increases classfile size
  - Stackmap entries added at:
    - Control flow merge point
    - Exception handler

Stackmap

```
int speed = 10;
Monster[] monsters = getMonsters();

for (int i = 0; i < monsters.length; i++){
// This is a merge point - stackmap here
// Variable slot 1 = int (speed)
// Variable slot 2 = Monster[] (monster)
// Variable slot 3 = int (i)

   doSomethingToMonster(monsters[i]);
}

// This is a merge point - stackmap here
// Variable slot 1 = int (speed)
// Variable slot 2 = Monster[] (monster)
```

Java Technology Philosophy

Java SE Platform and Java EE Platform Philosophies:

- JVM™ performs optimization
  - Run-time profiling to identify hot code
  - Dynamic class loading
- Compiler generates mostly unoptimized code

Better than C++ Performance on Java SE/Java EE Platforms

Java Compiler

- Designed to work with J2SE/J2EE Java VMs
  - Generate “clean” code
- Almost no size or performance optimization
  - No method inlining
  - No redundancy elimination
  - No dead class elimination
  - No dead code elimination
  - No code layout optimization
  - Has String and StringBuffer optimization

Java ME Virtual Machines
Targeted to Handset Constraints

	KVM	CLDC Hotspot "Monty"
Memory Footprint	256kB	1MB
Bytecode Execution	Interpreter	Adaptive Single-Pass Compiler
Optimizations		Constant Folding, Constant Peeling, Loop Peeling, Method Inlining

Source: Sun Microsystems

Performance Bottleneck

- JVM performance
- I/O
  - Network
  - File
- UI
  - Graphics
  - Images

Agenda

Why Size and Performance Matters

Under the hood of a Java ME MIDlet

Optimization Strategy

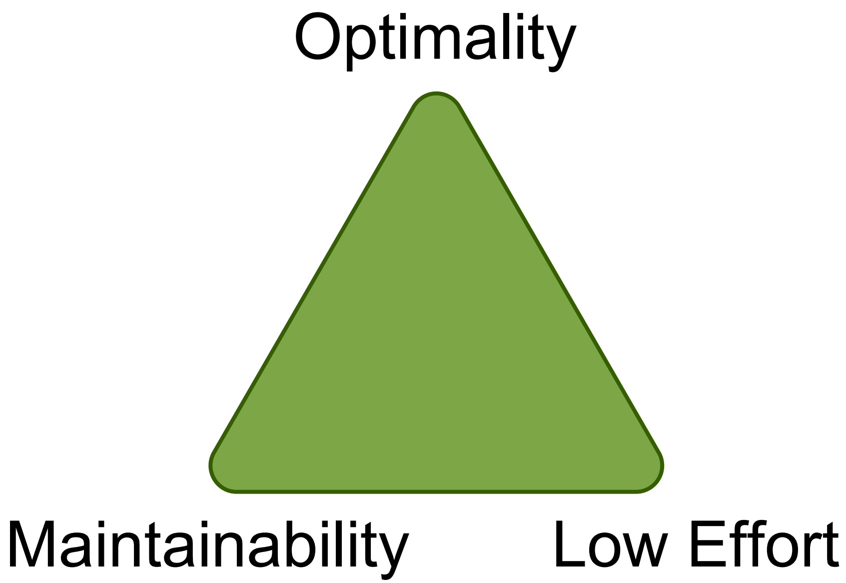
Optimization Techniques

Demo

What Are the Key Technical Problems?

- **JAR size**
- **Heap memory**
- **Performance**
- Handset bugs and quirks

Optimization Tradeoffs



Please Pick Any Two

Basic Optimization Rules

Rule #1

Be **Absolutely Clear** What Your Objectives Are

Basic Optimization Rules
Rule #2

80–20 Rule

Measure, Measure, and Measure

Basic Optimization Rules

Rule #3

Don't Do It

or

Automate the Mechanical Optimizations

Size Optimization

Most Optimizations Are Mechanical and Can Be "Automated"

Complete the Coding and Testing, then Refactor According to a Set of Strict Rules

Performance Optimization

Focus on the Architecture or Framework

Much Harder to Fix Later

Available Tools

- Obfuscator
- PNG optimizer
- ZIP compressor

Available Tools—Obfuscator

- Rename class, methods and fields
- Reduces the size and number of constant pool entries
- Example: Proguard

```
[1] UTF8: innaworks
[2] UTF8: ClassA
[3] UTF8: m
[4] Class: [1].[2]
[5] NameAndType: void [3](int);
[6] MethodRef: [1].[5]
```

▼

```
[1] UTF8: a
[2] Class:[1].[1]
[3] NameAndType: void [1](int);
[4] MethodRef: [1].[3]
```

Available Tools—PNG Optimizer

- Removes unnecessary information in PNG file
- Makes PNG data more compressible
- Example: PngCrush, AdvOpt

Available Tools—ZIP Compressor

- Standard JAR uses ZLIB deflate engine; up to 10% improvements with advance ZIP compressors
- Look out for operator restrictions
- Example: 7Zip, mBoosterZip

Agenda

Why Size and Performance Matters

Under the hood of a Java ME MIDlet

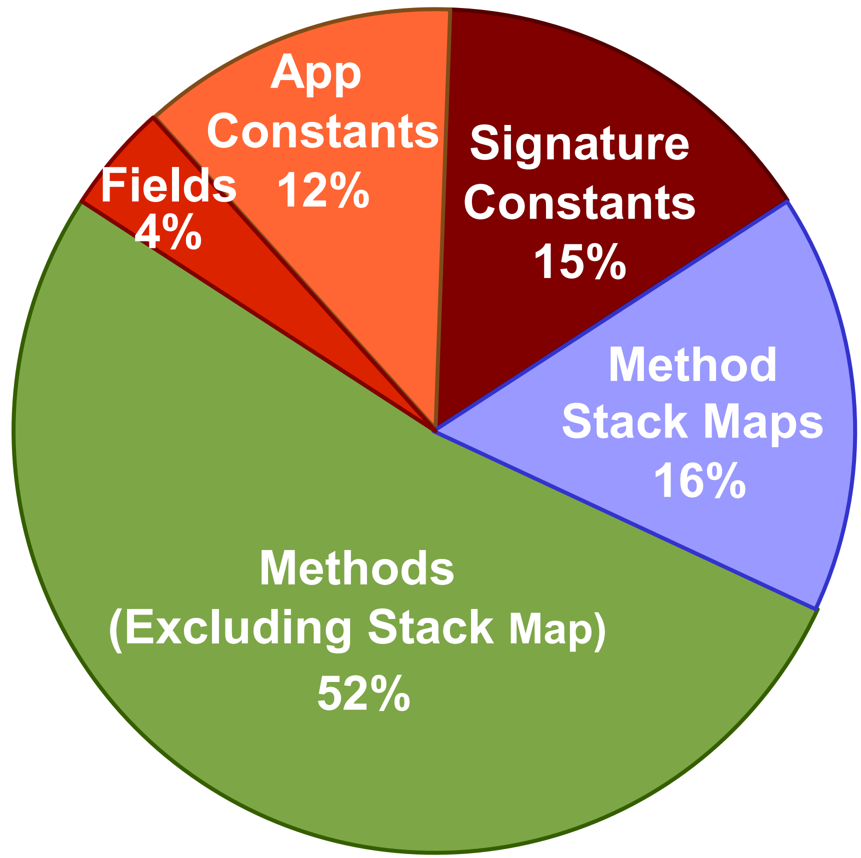
Optimization Strategy

Optimization Techniques

Demo

Where Should We Focus?

Typical Business or Consumer App



Source: Innaworks' Customer Study

Don’t Reinvent the Wheel

Make Use of Library API Whenever Possible

Merging Classes

Takes Two Classes and Combines Them

Reduces the ZIP overhead
Removes Java Class Overhead
Reduces Signature Constant Entries
Shares App Constant Entries
Increases Opportunities for Method Inlining

Merging Abstract Class with Concrete Class

Original:

```
abstract class AbstractSoundPlayer {
   String play(String soundFile) {...};
}

// Only class to extend AbstractSoundPlayer
class SamsungSoundPlayer extends
AbstractSoundPlayer {
   void play(String soundFile) {
      ...
   };
}
```

Merging Abstract Class with Concrete Class

Optimized:

```
class SamsungSoundPlayer {
   void play(String soundFile) {
      ...
   };
}
```

Merging Interface with Implementer

Original:

```
interface SoundPlayer {
  String play(String soundFile) {...};
}

// Only class to implement SoundPlayer
class SamsungSoundPlayer implements
SoundPlayer {
  void play(String soundFile) {
    ...
  };
}
```

Merging Interface with Implementer

Optimized:

```
class SamsungSoundPlayer {
  void play(String soundFile) {
    ...
  };
}
```

Merging Sibling Classes

Original:

```
abstract class AbstractMonster {
   abstract void doSomething();
   void runAway() {...};
   void drinkMore() {...};
}

class TimidMonster extends AbstractMonster {
   void doSomething() {runAway();}
}

class DrunkMonster extends AbstractMonster {
   void doSomething() {drinkMore();}
}
```

Merging Sibling Classes

Optimized:

```
// Combined the TimidMonster and
// DrunkMonster into one class
class CombinedMonster extends Monster {
   int monsterType;   // 0=TimidMonster,
                      // 1=DrunkMonster
   void doSomething() {
      switch (monsterType) {
         case 0: runAway(); break;
         case 1: drinkMore(); break;
      }
   }
}
```

Merging Classes

Very Powerful and Dangerous

- Look out for traps:
    - Instance of and casting
    - Arrays
    - Reflection
    - Class initialization order
- Can increase heap usage
- Maintainability and extensibility

Eliminating Local Variables

Combine Two Local Variables into One and Eliminate Temporary Local Variables

Reduces the Size of Stackmap Entries

Less Computation

Eliminating Temporary Variables

Original:

```
Pos myPos = getMyPos();
Pos monsterPos = getMonsterPos();
int dist = getDistance(myPos, monsterPos);
```

Smaller and faster:

```
dist = getDistance(getMyPos(),
          getMonsterPos());
```

Coalescing Local Variables

Original:

```
void someMethod() {
   int location = ...
   doSomeCalculation(location);
   // location is not used from here onwards

   int damage = ...
   if (damage > 10) { ... }
}
```

Coalescing Local Variables

Optimized:

```
void someMethod() {
   int mergedVar = ...
   doSomeCalculation(mergedVar);



   mergedVar = ...
   if (mergedVar > 10) { ... }
}
```

Method Inlining

Combine Two Methods Into One

Increases Opportunities for Intraprocedural Optimizations

Increases Opportunities for Eliminating Local Variables

Method Inlining

- How many places is the method called from?
- Is the call site a polymorphic call site?
- How big is the method?
- Is it called from the same class?

Method Inlining

Powerful and Works Well with Class Merging

Some Java VMs (e.g. HotSpot Based Java VMs) Impose Limits on Method Size to Compile to Native Code

Maintainability and Extensibility

Flattening 2D Arrays

Convert 2D Arrays to 1D Arrays

Less Array Bounds Checks

Less Dereferencing

Less array.length

Flattening 2D Arrays

Original:

```
boolean[][] enemyMap = new boolean[5][12];

// Check for any enemy next to us
// Assumes wrap around
if (enemyMap[myX+1][myY+1] ||
   enemyMap[myX-1][myY+1] ||
   enemyMap[myX+1][myY-1] ||
   enemyMap[myX-1][myY-1] } {
    . . .
}
```

Flattening 2D Arrays

Optimized:

```
boolean[] enemyMap = new boolean[5*12];

// Check for any enemy next to us
// Assumes wrap around
int myLoc = myX*12 + myY;
if (enemyMap[myLoc+1] ||
   enemyMap[myLoc-1] ||
   enemyMap[myLoc+12] ||
   enemyMap[myLoc-12] } {
   . . .
}
```

Array Initialization

- What code is generated by the Java compiler?

```
int[] map = {0, 1, 2, 3, …};
```

Array Initialization

- What code is generated by the Java compiler?

```
int[] map = {0, 1, 2, 3, …};
```

- Javac generated code is equivalent to:

```
map[0] = 0;
map[1] = 1;
map[2] = 2;
map[3] = 3;
 ...
```

Array Initialization

Optimized: Generate the Array at Run-Time

```
map = new int[100];
for (int i = 0; i < map.length; i++)
    map[i] = i;
```

Array Initialization

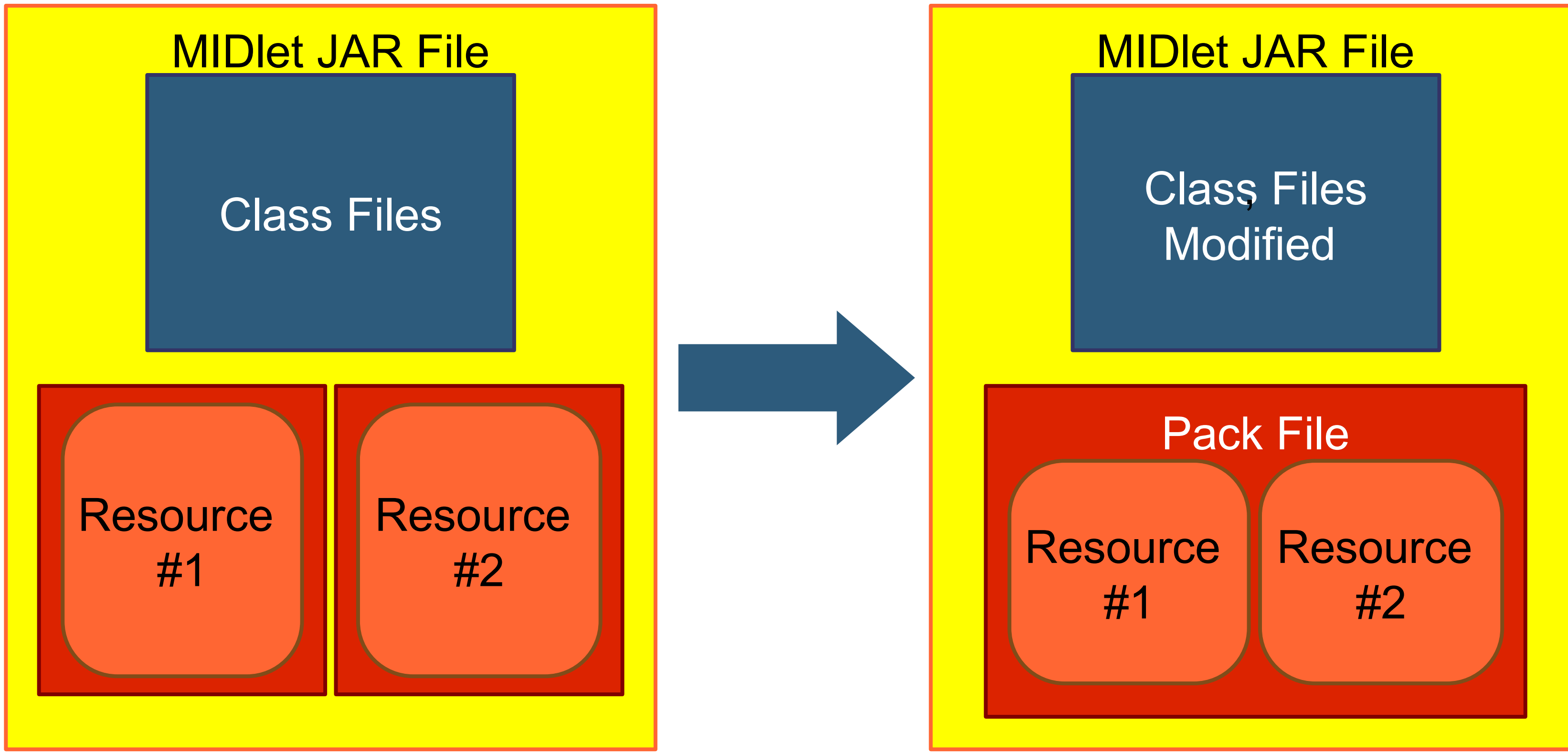
Optimized: Store the Array Data in a Resource

```
DataInputStream dis = new
DataInputStream("map.dat");

int len = dis.readInt();
int[] array = new int[len];
for (int i=0; i<len; i++)
    array[i] = dis.readInt();

dis.close();
```

Resource Packing



Resource Packing

```
public Image readImage(String file) {
   InputStream is = getResourceAsStream(pakfile);

   // Determine offset and filesize for file
   is.skip(offset);
   byte[] buffer = new byte[filesize];
   for (int i = 0; i < buffer.length; i++)
      buffer[i] = is.read();

   inputstream.close();
   return Image.createImage(buffer,0,buffer.length);
}
```

Resource Packing

Reduces the ZIP Overhead

Increases Compressability

Can Increase Heap Usage

Can Slow Resource File Access

Sharing Palette Across PNG Files

Improve Compressibility when Used in Conjunction with Resource Packing, By:

Reducing Palette of Each Subsequent PNG to 2 Bytes (Compressed)

Increasing Compressibility of Image Data

Optimization Summary

Tuned for **Minimum JAR Size**

	JAR Size	Heap Usage	Speed
Class Merging	▼ ▼ ▼ ▼	▲ ▲	
Eliminating Variables	▼		
Method Inlining	▼ ▼	▼	▲
Flattening 2D Arrays			▲ ▲
Array Initialization	▼ ▼	▲	▼
Resource Packing	▼ ▼ ▼	▲ ▲	▼
Sharing Palette	▼		

DEMO

Summary

- Size and performance matter especially for consumer applications
- 80–20 rule applies—focus on your effort where it counts
- Optimizations are highly interdependent
- Automate where possible

Q&A

Stephen Cheng

Squeezing the Last Byte and Last Ounce of Performance from Your MIDlets

Stephen Cheng

CEO
Innaworks
www.innaworks.com

TS-3418

Supplementary Slides

Your Mission Is

To develop a web browser:

- Runs on the widest varieties of Java™ Platform Micro Edition (Java ME) handsets
- Fits in 64kB
- Good user experience and performance
- Supports full HTML, stylesheets, and a wide range of graphics formats

Is it a Mission Impossible?

A Look at Opera Mini

	Opera for Windows	Opera Mini
Operating System	Windows	Java ME
Program Size	3.6MB	< 64kB
Processor	Pentium	N/A
RAM	16MB	< 205kB

Source: Opera Website

The Power of a Consumer Handset

	Nokia 7260	PC 1988
Processor	ARM-7 40MHz?	16MHz 386
RAM	Approx. 1MB	2MB
Screen Size	128 x 128	VGA
Graphics	CPU	CPU
Storage	4MB Flash	20MB Hard Disk

Source: Nokia Developer Website and mobileburns.com

The Power of a Consumer Handset

	Nokia 7260	PC 2006
Processor	ARM-7 40MHz?	2GHz AMD
RAM	Approx 1MB	512MB
Screen Size	128 x 128	XVGA
Graphics	CPU	Accelerated
Storage	4MB Flash	60GB Hard Disk

Source: Nokia Developer Website and mobileburns.com

Why Size and Performance Matters

Adoption = **Potential Market Size**
x Value to User
x Marketing

Volume Matters

- **Application feature set**
- **Addressable handsets**
- **Addressable carriers**
- **Emerging markets**

Why Size and Performance Matters

Adoption = Potential Market Size
x **Value to User**
x Marketing

Perceived Quality Matters

Cost Matters

- **Does the application satisfy needs?**
- **What is perceived quality?**
- **Does it feel polished and professional?**
- **What is the real cost of owning the app?**

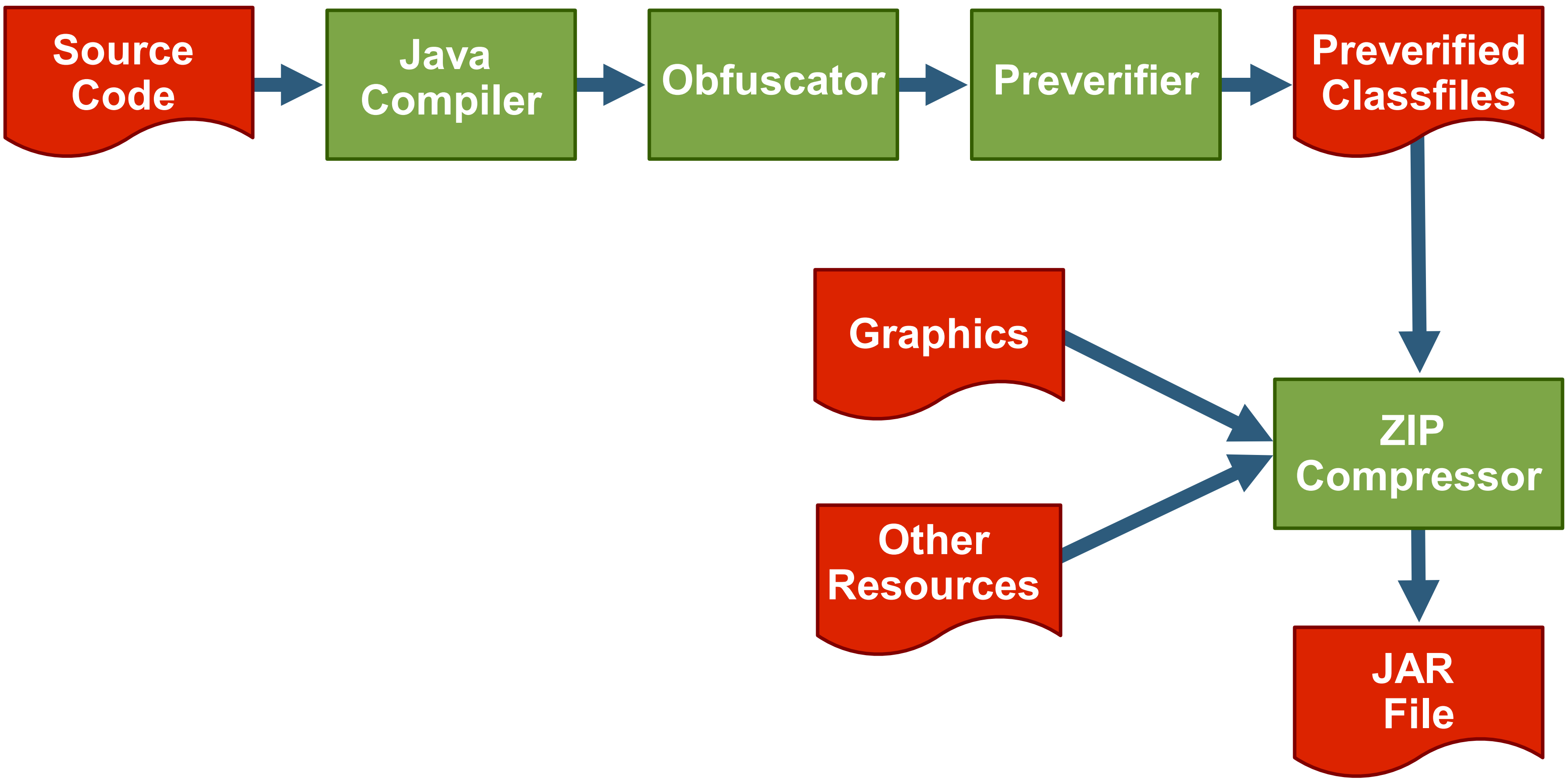
Signature Constants

What Are Signature Constants?

- Name of a referenced field, method or class

```
Constant Pool of innaworks.ClassA
   [1] UTF8: innaworks
   [2] UTF8: ClassA
   [3] UTF8: m
   [4] Class: [1].[2]
   [5] NameAndType: void [3](int);
   [6] MethodRef: [1].[5]
```

Java ME Toolchain



Method Inlining

- If JAR size is critical:
  - Always inline getters and setters
  - Always inline small methods
  - Inline methods that are called from a single non-polymorphic callsite
- If performance is critical:
  - Inline methods that are frequently called

Flattening 2D Arrays

Greatly Improve Performance for 2D Array Heavy Code

Array Initialization

Reduces the ZIP Overhead

Increases Compressability

Storing Array Data in Resource Slows Startup

Optimization Summary

Tuned for **JAR Size and Heap Usage**

	JAR Size	Heap Usage	Speed
Class Merging	▼ ▼	▼	
Eliminating Variables	▼		
Method Inlining	▼ ▼	▼	▲
Flattening 2D Arrays			▲ ▲
Array Initialization	▼ ▼	▲	▼
Resource Packing	▼ ▼	▲	▼
Sharing Palette	▼		