



JavaOne

Optimizing Midlets for Size and Performance

Simon Robinson

Innaworks

www.innaworks.com

TS-5109

Goal of This Talk

Pushing the size and performance of Java™ Platform, Micro Edition (Java ME platform) applications on **today's handsets**

Agenda

Why size and performance matters

Under the bonnet of a Java ME
Platform MIDlet

Optimization strategy

Optimization techniques

Optimizing for Jazelle DBX and
Java HotSpot™ technology

Case study

Agenda

Why size and performance matters

Under the bonnet of a Java ME
Platform MIDlet

Optimization strategy

Optimization techniques

Optimizing for Jazelle DBX and
Java HotSpot™ technology

Case study

Why Size and Performance Matters

Adoption = Potential market size
× How much fun
× Marketing

Why Size and Performance Matters

$$\text{Adoption} = \text{Potential market size} \\ \times \text{How much fun} \\ \times \text{Marketing}$$

Handset coverage matters

Why Size and Performance Matters

$$\text{Adoption} = \text{Potential market size} \\ \times \text{How much fun} \\ \times \text{Marketing}$$

How fun is your game?
Perceived quality matters

Constraints of Consumer Handsets

	JAR size	Heap memory
Nokia S40 v1 (3300, etc.)	64 kB	370 kB
Nokia S40 v2 (6230, etc.)	128 kB	512 kB
Sharp GX22	100 kB	512 kB
DoJa 2.5 (m420i)	30 B	1.5 MB

15% game sales for handsets < 64 kB Java Archive (JAR) file size

35% game sales for handsets < 128 kB JAR file size

Agenda

Why size and performance matters

**Under the bonnet of a Java ME
Platform MIDlet**

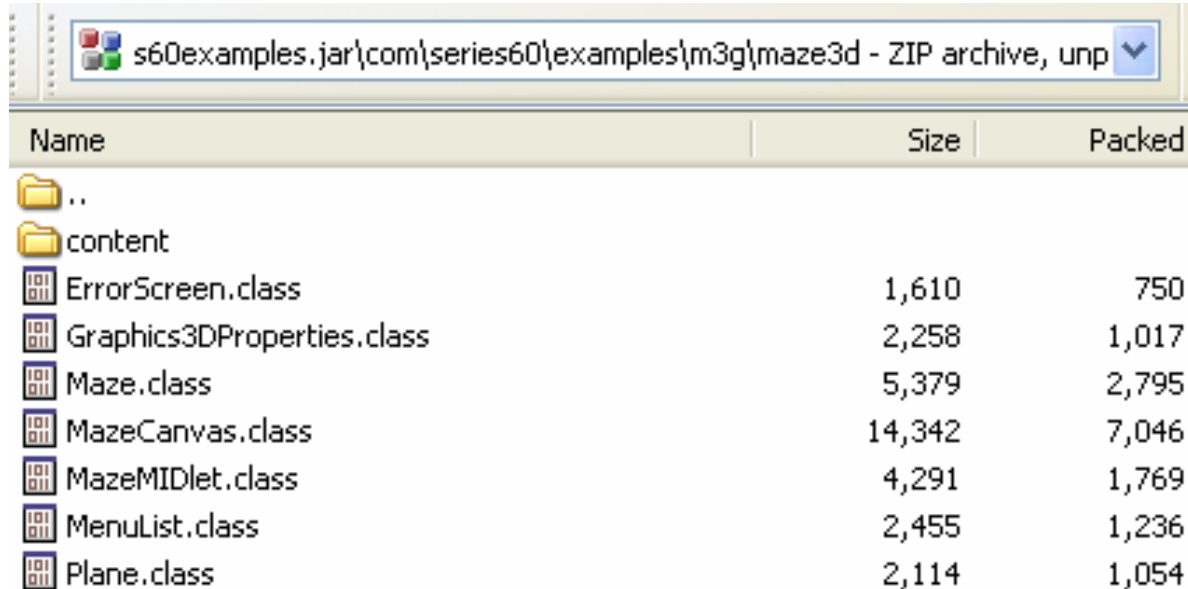
Optimization strategy

Optimization techniques

Optimizing for Jazelle DBX and
Java HotSpot technology

Case study

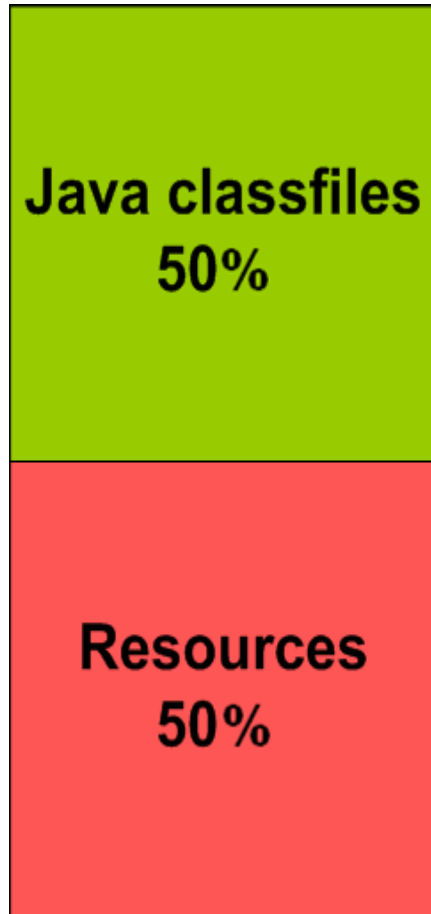
What Is in a MIDlet JAR File?



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

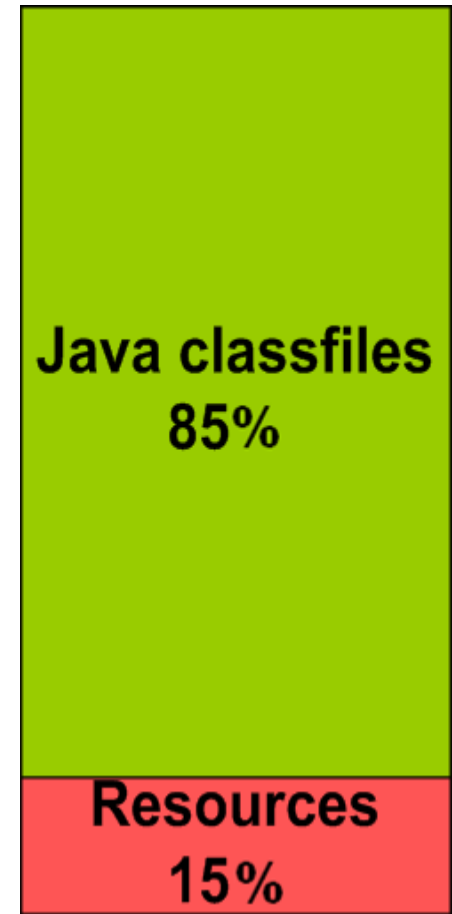
- 70 bytes JAR file overhead per file
- Compression does not work across files
- Overhead depends on **path** length

Classfile vs. Resource Files



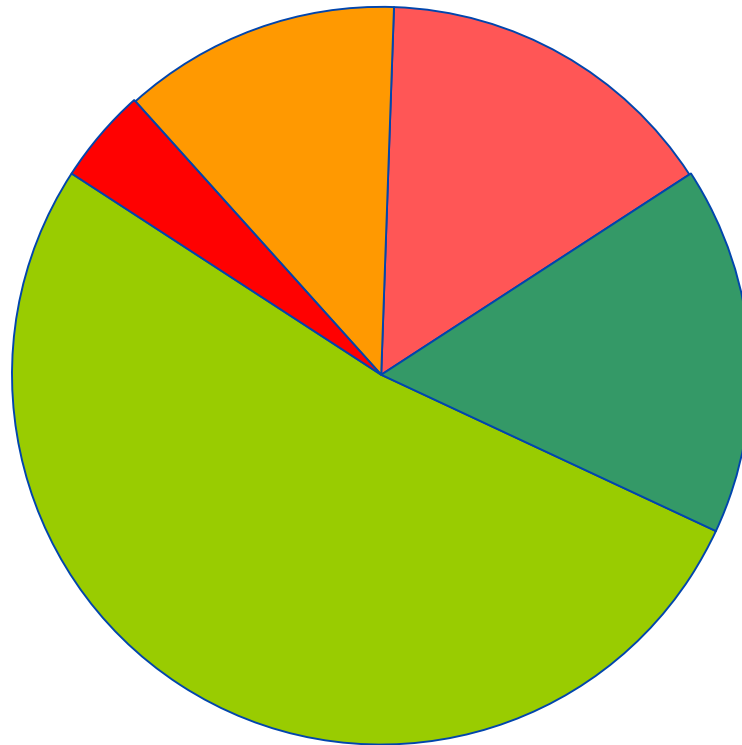
Typical 2D
Game

Typical Business
or Consumer
App



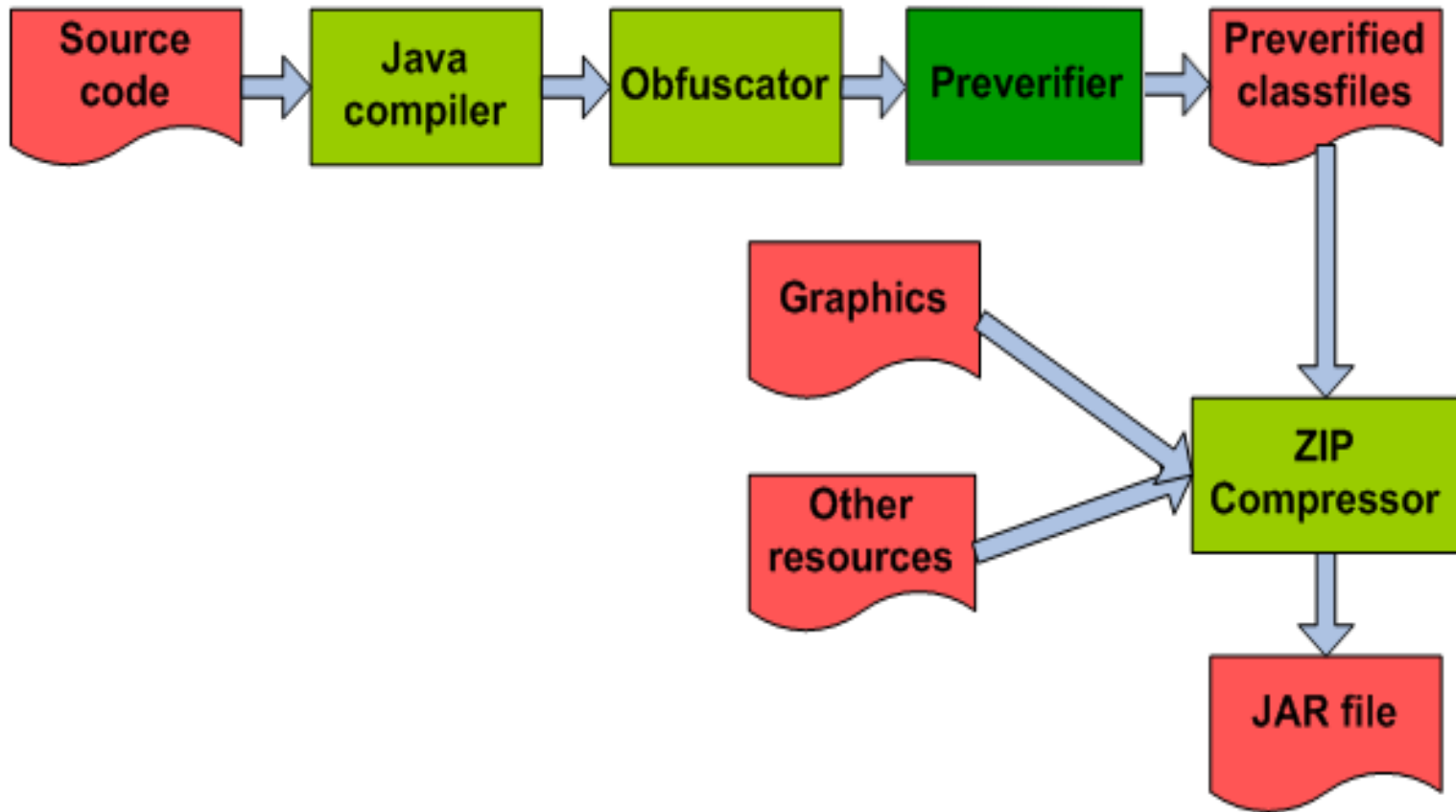
Source: Innaworks' customer study

Classfile Size Breakdown



Source: Innaworks' customer study

Java ME Platform 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 Compiler

- Designed to work with Java Platform, Standard Edition (Java SE platform)/Java Platform, Enterprise Edition (Java EE platform) JVM™ machines
 - 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 Platform Virtual Machines

Targeted to handset constraints

	KVM	CLDC “Hotspot Monty”
Memory footprint	256 kB	1 MB
Bytecode execution	Interpreter	Adaptive Single-pass compiler
Optimizations		Constant folding Constant peeling Loop peeling

Source: Sun Microsystems

Performance Bottleneck

- Virtual Machine for the Java platform (JVM™ machine) performance
- I/O
 - Network
 - File
- UI
 - Graphics

The terms “Java Virtual Machine” and “JVM” mean a Virtual Machine for the Java™ platform.

Agenda

Why size and performance matters

Under the bonnet of a Java ME
Platform MIDlet

Optimization strategy

Optimization techniques

Optimizing for Jazelle DBX and
Java HotSpot technology

Case study

What Are the Key Technical Problems?

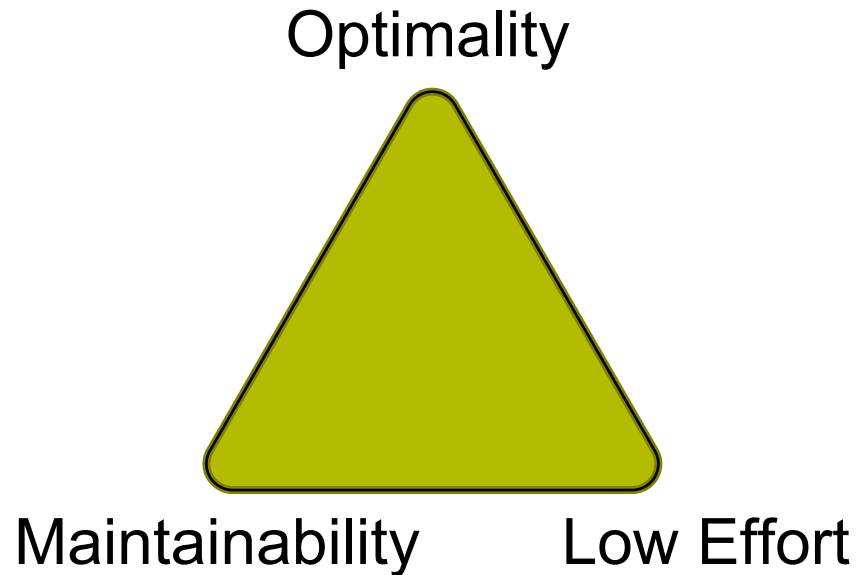
JAR file size

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 apply recipes
- Affects maintainability

Performance Optimization

- Focus on the architecture or framework
- Need to understand the characteristics of target handsets
- Much harder to fix later

Available Tools—Obfuscator

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

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



```
[1] UTF8: a
[2] Class: [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 file 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 bonnet of a Java ME
Platform MIDlet

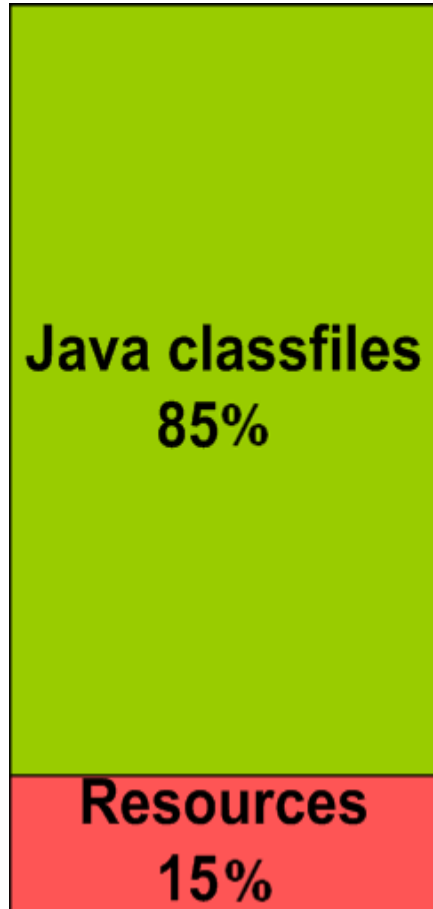
Optimization strategy

Optimization techniques

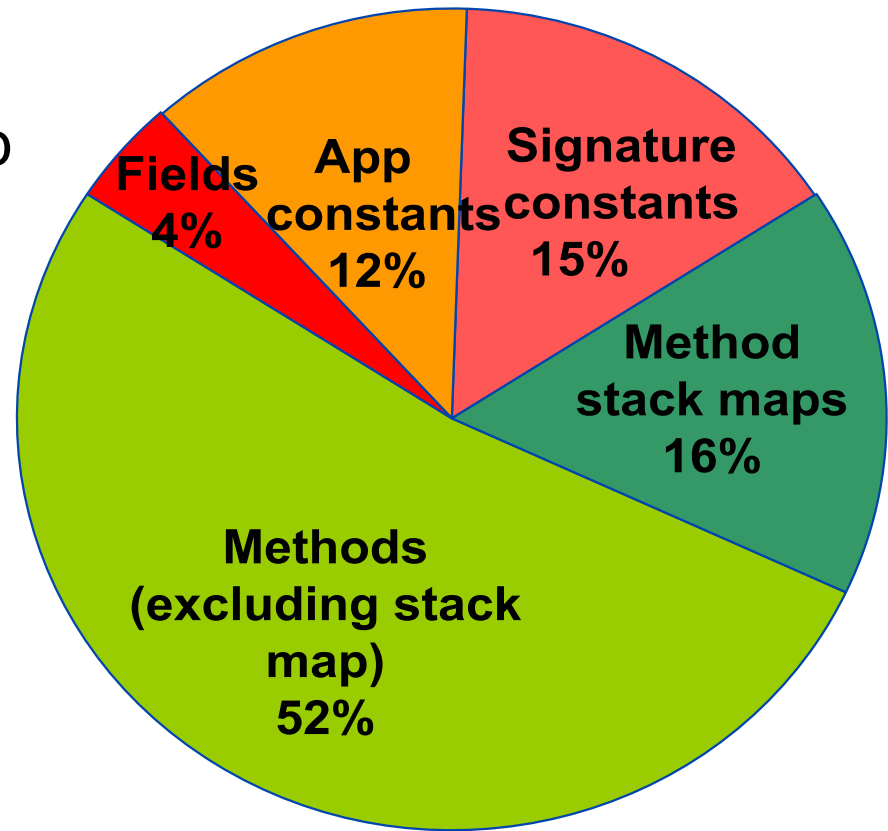
Optimizing for Jazelle DBX and
Java HotSpot technology

Case study

Where Should We Focus?



Typical
Business or
Consumer App



Source: Innaworks' customer study

Merging Classes

- Takes two classes and combine them
- Reduces the ZIP overhead
- Removes Java class file 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 doAction();
    void runAway() {...};
    void drinkMore() {...};
}

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

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

Merging Sibling Classes

Optimized:

```
// Combined the TimidMonster and
// DrunkMonster into one class
class CombinedMonster extends AbstractMonster {
    int monsterType;          // 0=TimidMonster,
                              // 1=DrunkMonster

    void doAction() {
        switch (monsterType) {
            case 0: runAway(); break;
            case 1: drinkMore(); break;
        }
    }
}
```

Merging Classes

Very powerful and dangerous

Look out for traps

- instanceof 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 stack map 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

- From 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 JVM machines (e.g., Java HotSpot technology-based JVM machines) impose limits on method size to compile to native code

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, ..., 99};
```

Array Initialization

What code is generated by the Java compiler?

```
int[] map = {0, 1, 2, 3, ..., 99};
```

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 < 100; i++)  
    map[i] = i;
```

Array Initialization

Optimized: Store the array data in a resource

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

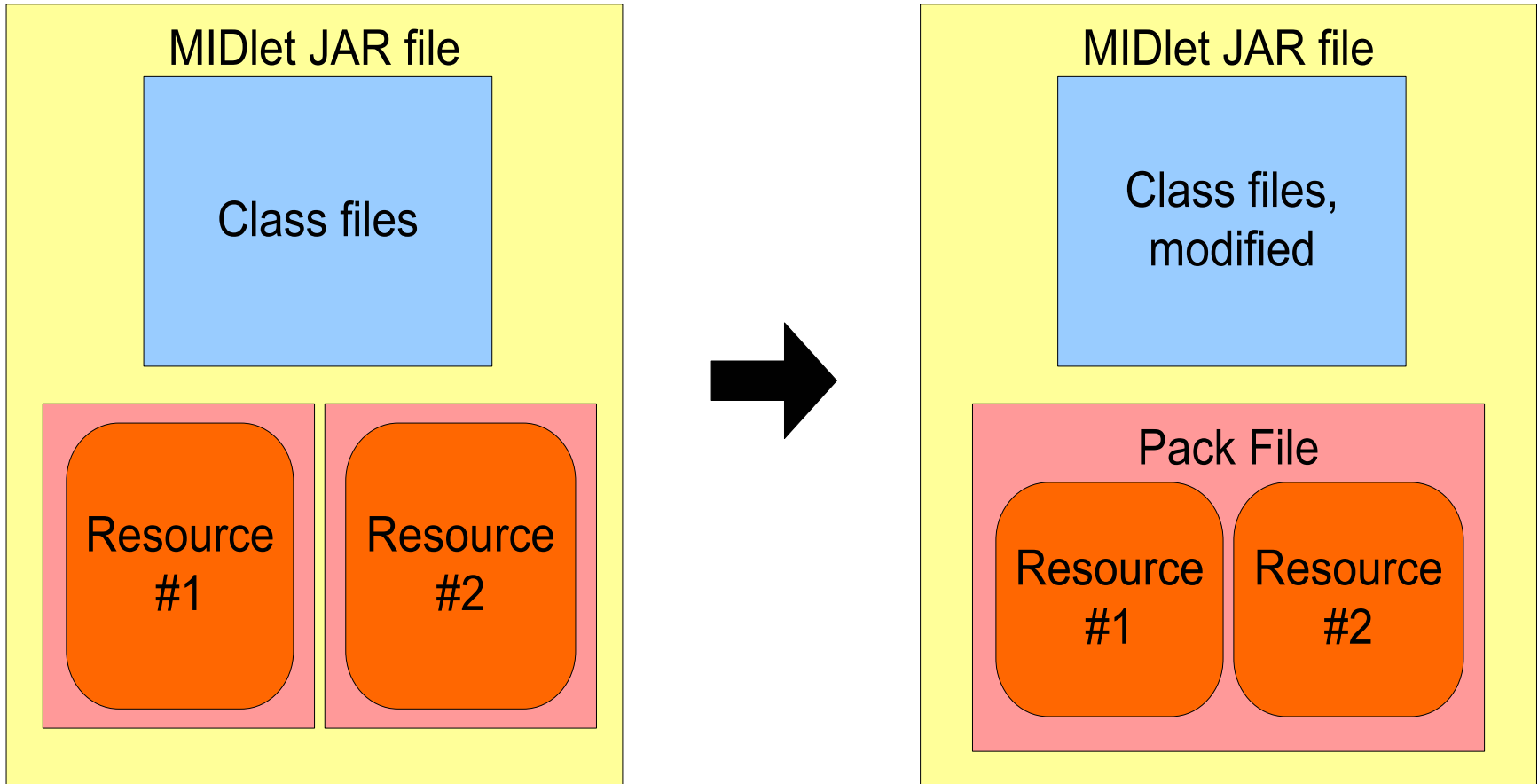
int len = dis.readInt();

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 size for file
    is.skip(imageOffset);
    byte[] buffer = new byte[imageLength];
    for (int i = 0; i < imageLength; i++) {
        buffer[i] = is.read();
    }

    is.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

- Improves compressibility when used in conjunction with resource packing, by:
- Reducing the palette of each subsequent PNG to 2 bytes (compressed)
- Increasing compressibility of image data

Optimization Summary

Tuned for **minimum JAR file size**

	JAR file size	Heap usage	Speed
Class merging	▼ ▼ ▼ ▼	▲ ▲	
Eliminating variables	▼		
Method inlining	▼ ▼	▼	▲
Flattening 2D arrays			▲ ▲
Array initialization	▼ ▼	▲	▼
Resource packing	▼ ▼ ▼	▲ ▲	▼
Sharing palette	▼		

Agenda

Why size and performance matters

Under the bonnet of a Java ME
Platform MIDlet

Optimization strategy

Optimization techniques

**Optimizing for Jazelle DBX and
Java HotSpot technology**

Case study

What Is ARM Jazelle DBX?

- Some handsets now make use of Jazelle **DBX** (**D**irect **B**ytecode **E**xecution)
- Provides performance improvements by directly supporting **some** byte codes for execution in hardware
- Handsets include **K700, K800, S700, O₂ X4**

Optimizing for Jazelle DBX

The following are accelerated through Jazelle technology:

- 32-bit mathematical operations
- Bitwise manipulation
- Conditional branching
- Local data access

Strategy:

- Switch to hardware enabled byte codes
- Reduce the number of byte codes
- Focus on performance critical sections

Optimizing for Jazelle DBX

Techniques:

- Elimination of unnecessary field access
- Redundancy elimination
- Use if/else in place of switch instructions
- Method elimination through inlining

Working With HotSpot

- Sun's Connected Limited Device Configuration HotSpot™ Implementation JVM machine is present on a number of high-end handsets in the market
- Picks 'hot' methods in your MIDlet to compile to native code
- **But large methods might not be considered**
- **To compile code can mean a short pause!**

Strategy for Java HotSpot Technology

- To optimize a build targeted to a Java HotSpot technology handset:
 - Use Method Inlining to eliminate trivial methods
 - Don't form large methods through method inlining, if you think they might be 'hot'
 - If you notice pauses at an inappropriate time, you may be able to trick HotSpot into moving them earlier



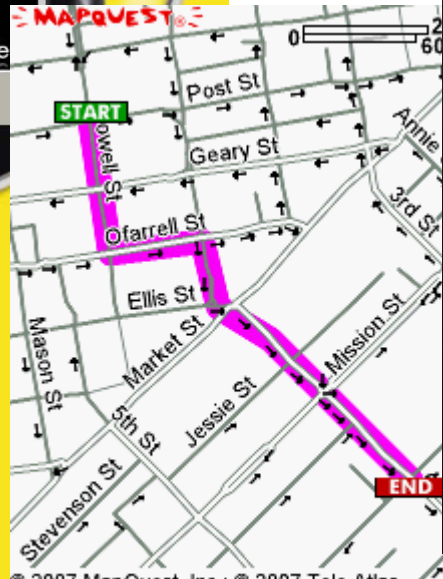
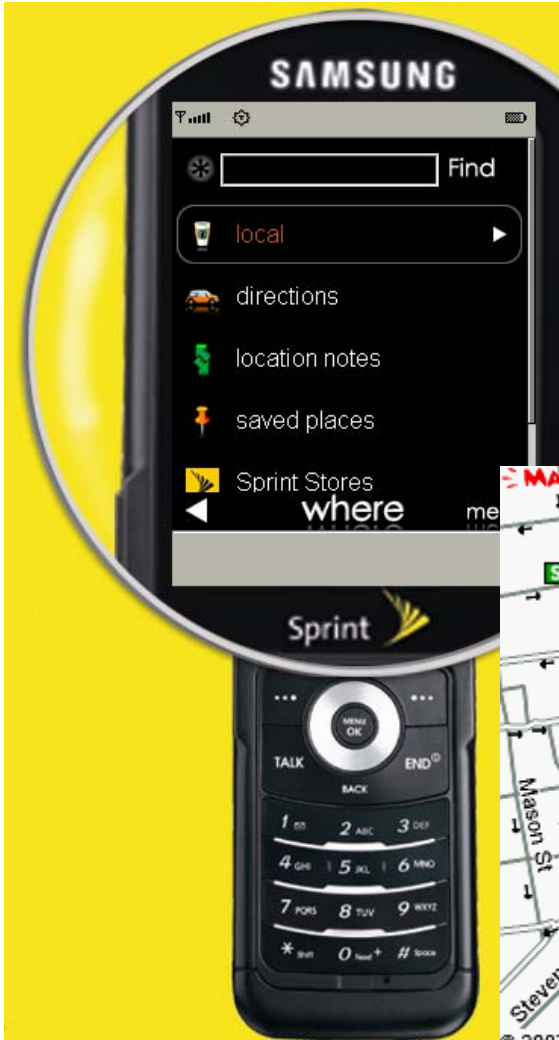
Case Study



Where™

Ulocate

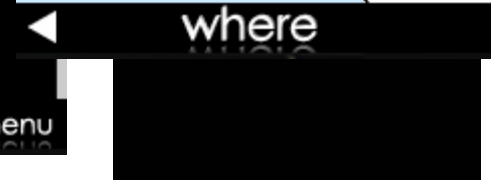
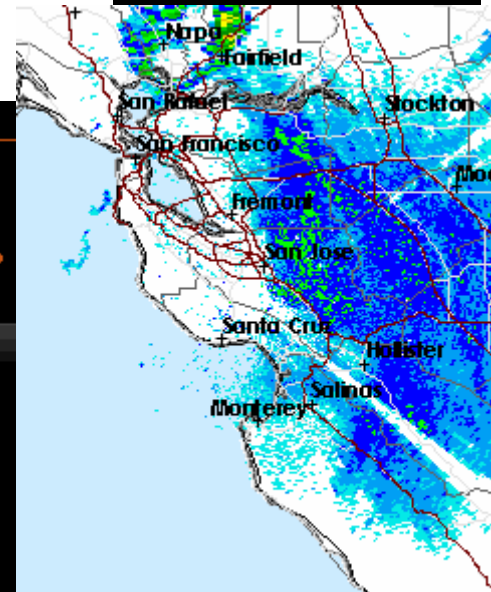
the power of where™



Manage Widgets

Did you know that you can also manage your widgets online at www.where.com?

- Airports/Flights
- ThinAir
- GPS Twitter
- Pubwalk-Nightlife
- US Reps
- BURGER KING
- NearBio
- Irish Pubs

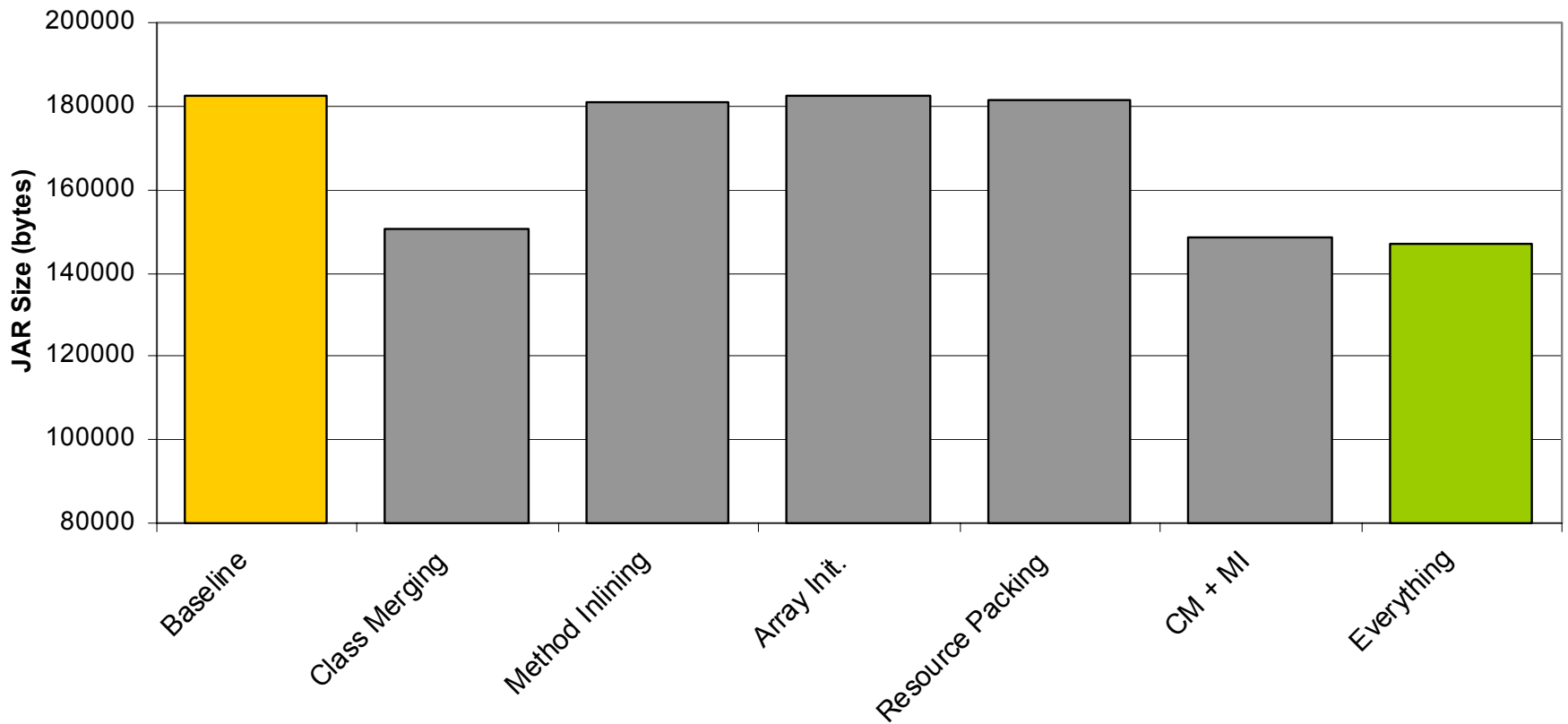




Where™

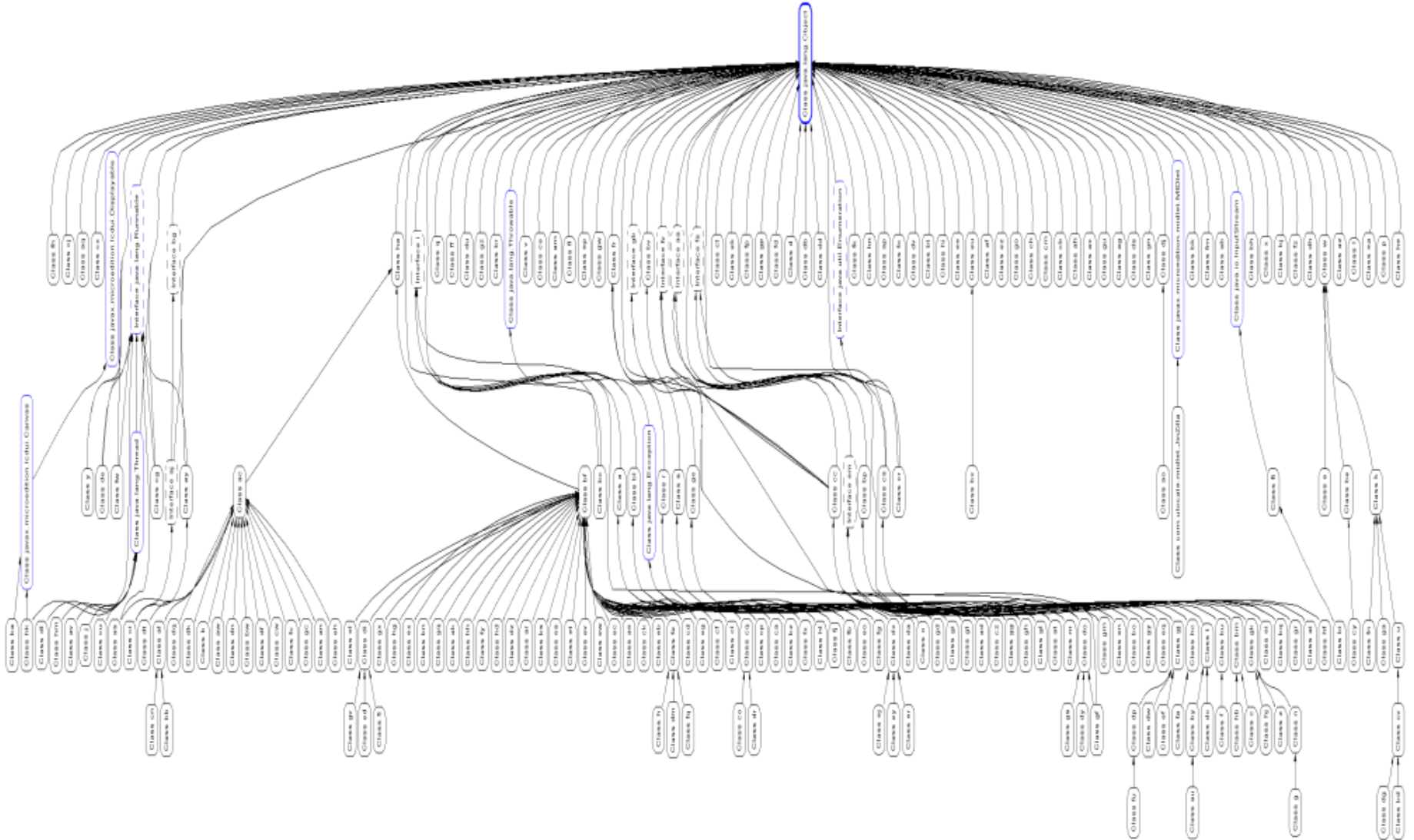
	Original	Baseline	Class Merging	Method Inlining	Array Init.	Resource Packing	CM+MI	CM+MI+ AI+RP
JAR file size	192748	182603	150706	181081	182470	181497	148723	147495
JAR file size difference (from baseline)		0	-31897	-1522	-133	-1106	-33880	-35108
% difference (from baseline)		0.0%	-17.47%	-0.83%	-0.07%	-0.61%	-18.55%	-19.39%
Number of classes total	223	223	146	223	223	224	146	147
Number of methods	1446	1433	1409	1334	1434	1438	1219	1225
Number of fields	520	511	415	511	511	514	415	416
Constant Pool Entries count	18953	18773	14998	18434	18784	18878	14453	14577

Size Comparison

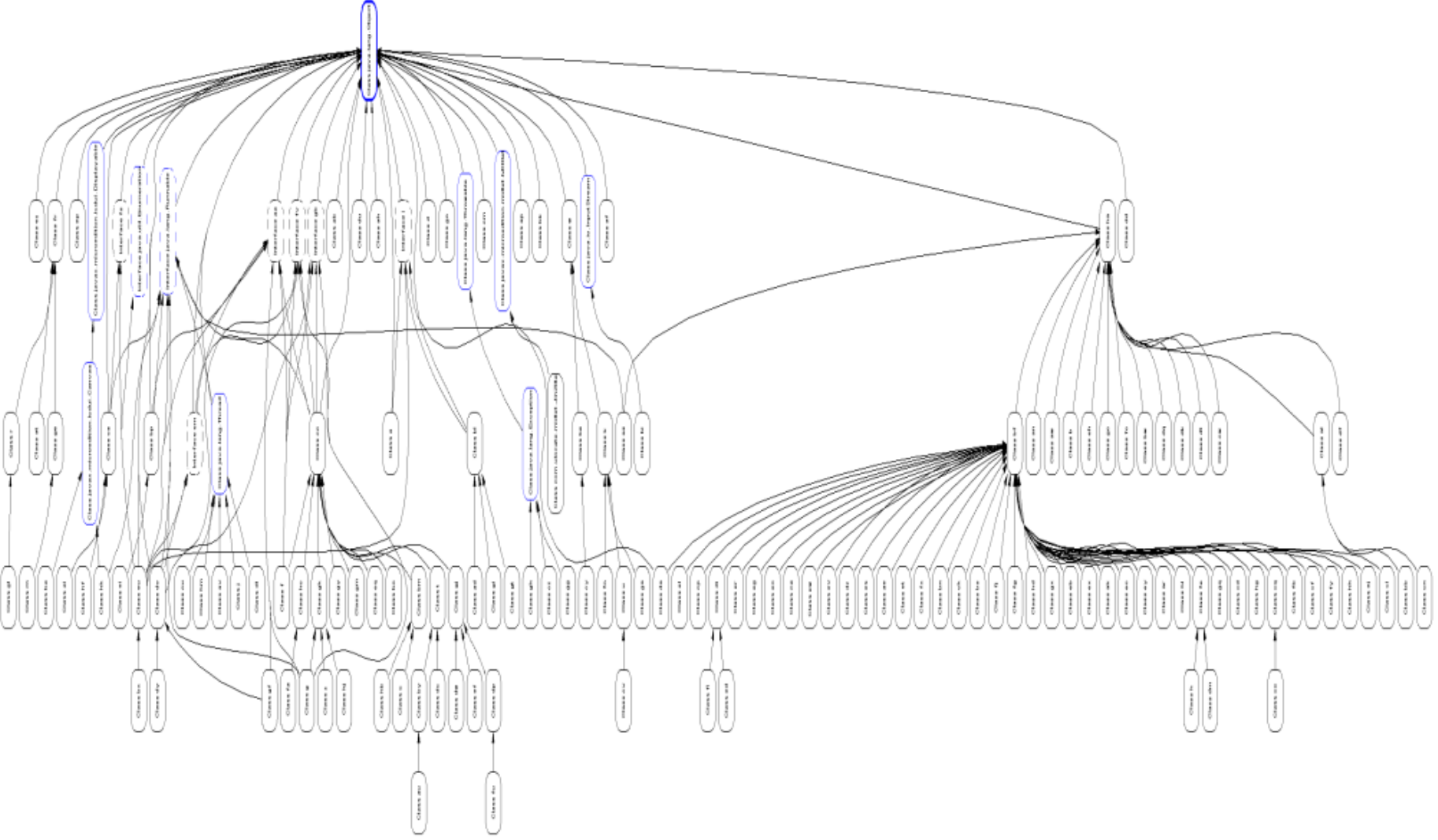




Class Hierarchy (Before)



Class Hierarchy (After)



Summary

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



Q&A





JavaOne

Optimizing Midlets for Size and Performance

Simon Robinson

Innaworks

www.innaworks.com

TS-5109

Appendix: Where™

	Original	Baseline	CM	MI	AI	RP	CM+MI	ALL
Size	192748	182603	150706	181081	182470	181497	148723	147495
Diff from baseline		0	-31897	-1522	-133	-1106	-33880	-35108
%Diff from baseline		0.00%	-17.47%	-0.83%	-0.07%	-0.61%	-18.55%	-19.39%
Num interfaces	8	8	6	8	8	8	6	6
Num abstract	21	21	4	21	21	21	4	4
Num concrete	194	194	136	194	194	195	136	137
Num classes	223	223	146	223	223	224	146	147
Num methods	1446	1433	1409	1334	1434	1438	1219	1225
Num fields	520	511	415	511	511	514	415	416
CP Entries	18953	18773	14998	18434	18784	18878	14453	14577