Writing Optimized Applications for High-Performance Java™ ME Runtime Environments

Kyle Buza
Sun Microsystems

Oleg Pliss, Ph.D.
Sun Microsystems

TS-1451

Speaker Qualification

- Kyle Buza has been a software engineer at Sun Microsystems for five years, implementing numerous performance enhancements for both the CLDC and CDC HotSpot implementation VMs
- Oleg Pliss is a Senior Staff Engineer with the Client Systems Group at Sun; he is working on high performance Java ME virtual machines and specializes in compilers and garbage collection

Goal of This Talk

Learn about features and capabilities of high performance Java™ Platform, Micro Edition runtime environments and how your applications can take advantage of them

Agenda

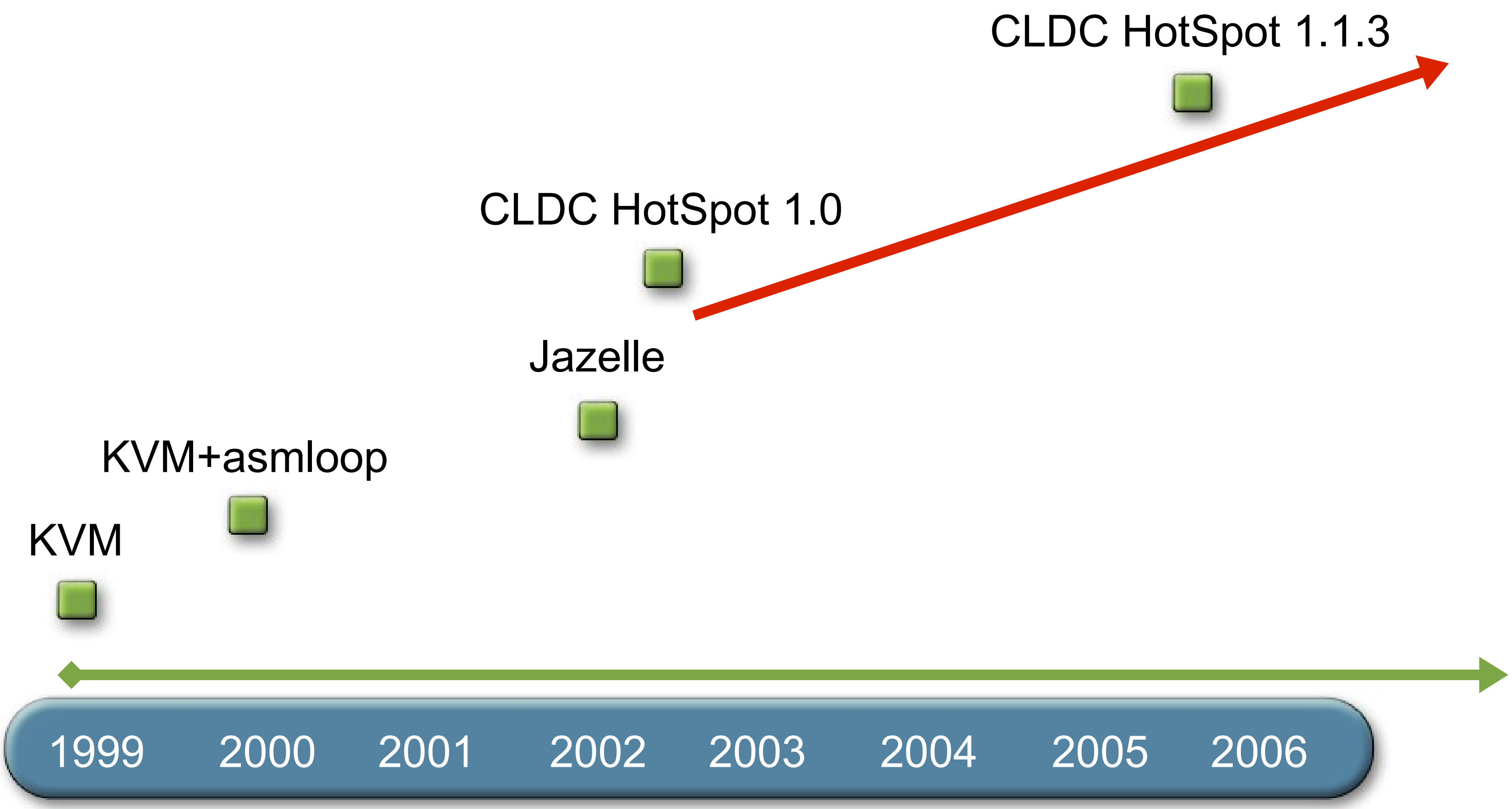
The Evolution of Java ME Performance

Optimization Techniques

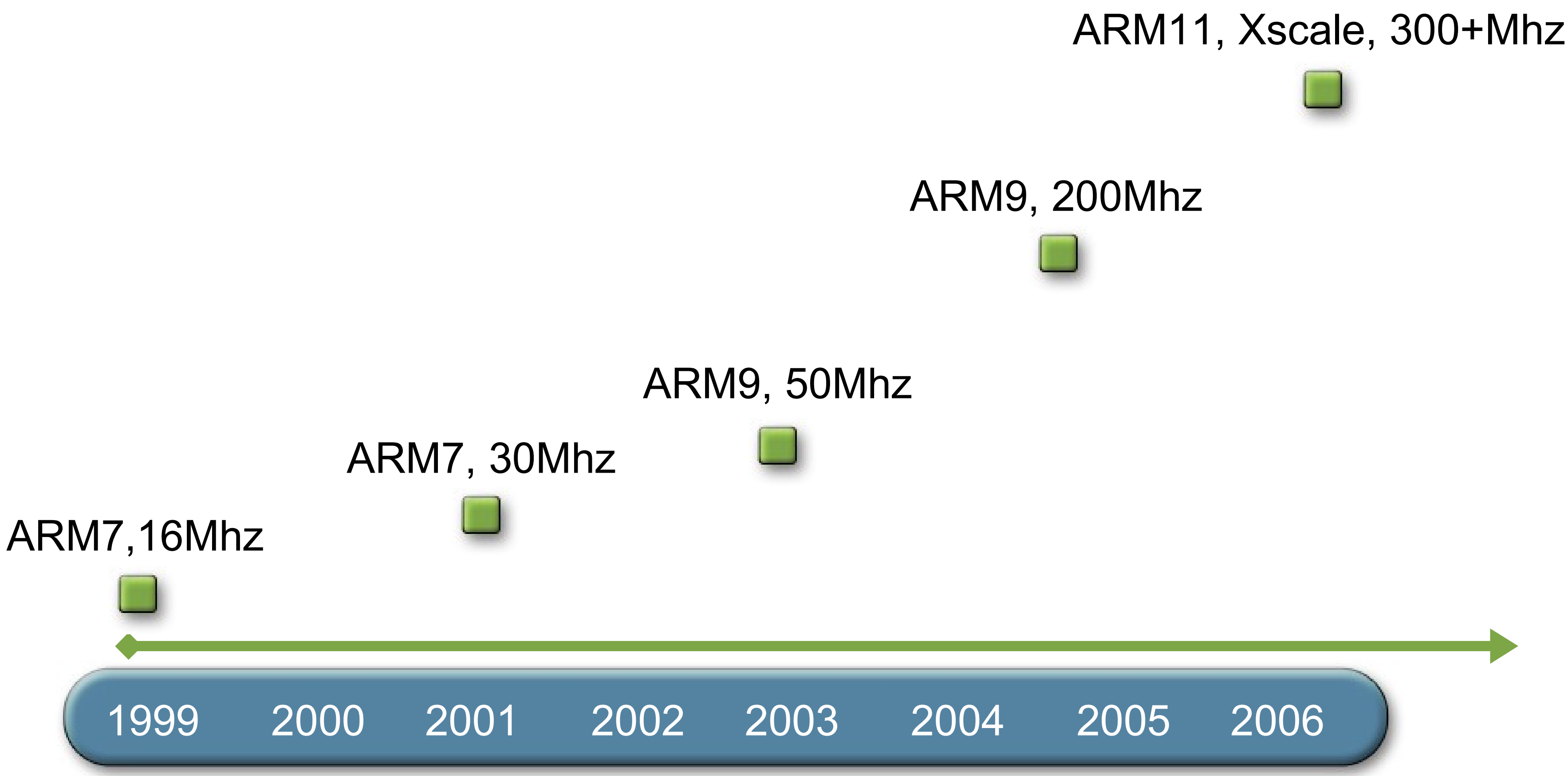
Summary

Q&A

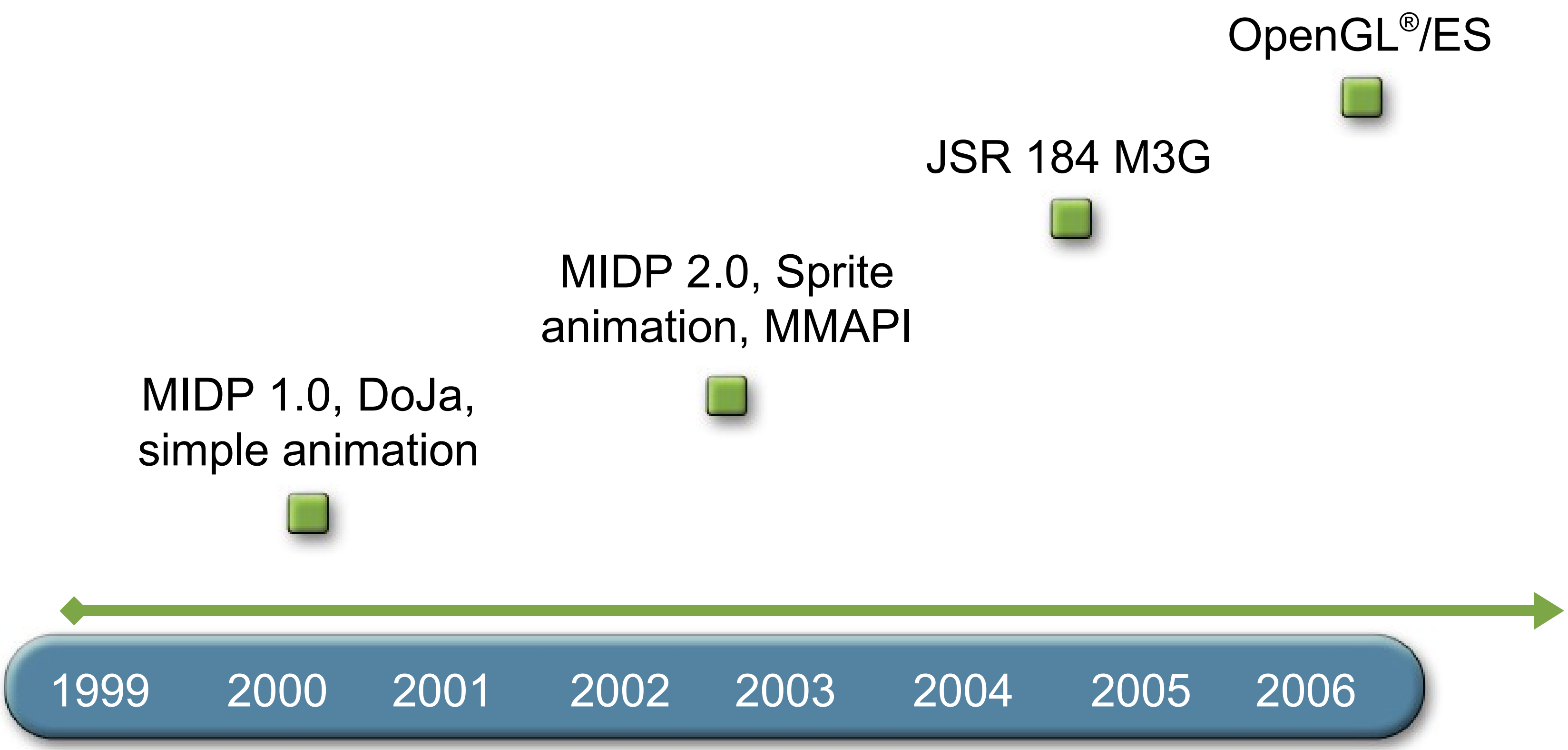
The Evolution of Java ME Performance



The Evolution of Java ME Performance (2)



The Evolution of Java ME Performance (3)



Techniques for Java ME Runtime Environments

- Know what works well with Java ME software dynamic compilers
- Manage Garbage-collection pauses
- Tune for a range of platforms (compilers, hardware engines, interpreters)
- Know your platforms

A Developer's Cookbook

- 15 guidelines for application design and coding
  - Focus on straightforward, most effective improvements
  - Guidelines are high-level and generic to make them as universal as possible
- Organized according to user experience impact
  - Initialization
  - Runtime optimizations
  - Execution
  - Memory Use

Initialization

1) Static Array Initialization

- Problem
  - Array initialization performed in static initializer
  - Significant code bloat and slow down class loading
- Impact
  - Application startup, memory footprint
- Guideline
  - Move initialization out of static initializer and do programmatic initialization or read data from I/O
    - Note: Time vs. space trade-off (I/O can be slow)

Runtime Optimizations

2) Locality of Hot Code

- Problem
  - Hot code spread out over multiple individual methods
  - Optimization techniques typically operate on method boundaries (for semantical reasons)
  - This multiplies overhead for hot spot detection, optimization, and subsequent management
- Impact
  - Liveliness, execution consistency
- Guideline
  - Factor/concentrate hot code in a few methods (as practical from design perspective)

Example: Locality of Hot Code

```
gameLoop() {
  while (!done)
    advance();
}

advance() {
  updateModel();  // update game state
  updateScreen(); // refresh screen
  // Check collision with different objects: Non-optimal!
  checkCollisionObjectA();
  checkCollisionObjectB();
  checkCollisionObjectC();
  checkCollisionObjectD();
  checkCollisionObjectE();
  ...
}
```

Example: Locality of Hot Code (Cont.)

```
advance() {
  updateModel();
  updateScreen();
  // Better - Collapse multiple related hot methods:
  // - Management overhead is reduced to 1/5th
  // - Optimization occurs 5x sooner
  // - Concentrates hot code
  // - All related code is optimized as an entity
  checkCollisions();
  ...
}
```

Runtime Optimizations

3) Large Methods Containing Hot Code

- Problem
  - Hot code embedded in large methods (good for K Virtual Machine)
  - High cost for dynamic compilation.
  - Failed compilation == fall back to interpretation
  - Some dynamic compilers may show visible pauses (though not a problem in CLDC-HotSpot implementation compiler)
- Impact
  - Liveliness, execution consistency
- Guideline
  - Keep methods with hot code compact

Example: Large Methods

```
// Initialization and cleanup plus main loop
// all in one method: Non-optimal
gameMain() {
  ... // lots of code here, executed only once
  while (!done)  // only section of hot code
    advance();
  ... // more code here, executed only once
}
```

Example: Large Methods (Cont.)

```
// Better - Factor out code to reduce method size:
// - Less overhead to optimize
// - Reduced resource requirements
gameMain() {
  initialize();
  while (!done)
    advance();
  cleanup();
}

initialize() {
  ...
}
cleanup() {
  ...
}
```

Runtime Optimizations

4) Mixing Hot and Cold Code

- Problem
  - A section of code is hot but contains a number of code paths that don't execute often
  - This results in large methods (see previous slide) and disrupts optimization and execution due to branches
- Impact
  - Execution speed
- Guideline
  - Avoid mixing hot and cold code—All code in a hot method should really be hot

Example: Mixing Hot and Cold Code

```
// Testing of rare conditions: Non-optimal
advance() {
  updateModel();
  updateScreen();

  if (condition == 1) { // occurs in 1% of loops
    ... // code to add object A (cold code)
  }
  if (condition == 2) { // occurs in 5% of loops
    ... // code to add object B (cold code)
  }
  if (condition == 3) { // occurs in 2% of loops
    ... // code to add object C (cold code)
  }
  ...
}
```

Example: Mixing Hot and Cold Code

```
advance() {
  updateModel();
  updateScreen();
  // Better - Factor out code to streamline execution
  // - Reduced method size
  // - Reduced number of branches
  // - Reduced size of branches
  // - Concentrated hot spot
  if (condition > 0) {
    addObject(condition);
  }
  ...
}
```

Runtime Optimizations

5) Conditional Exceptions in Hot Code

- Problem
  - Code conditionally throws exceptions in the hot path
  - This precludes certain optimizations (code reordering/rescheduling); exceptions are expensive
- Impact
  - Execution speed
- Guideline
  - Never throw unconditional exceptions in hot code
  - Avoid conditional exceptions if possible
    - Caveat: Might not always be practical due to implicit exception points such as null checks, array bounds, etc.

Example: Conditional Exceptions

```
// Conditional exception - Not optimal!
void process(data) throws MyException {
  ... // some processing of data
  if (problem) throw new MyException();
}

for (int i = data.length-1; i >= 0; i--) {
  try {
    process(data[i]);
    ... // additional operations
  } catch MyException {  // can happen at any time
    ... // handle error
  }
}
```

Example: Conditional Exceptions

```
// Better - Eliminate conditional exception:
// - Array bounds check can be eliminated by runtime
// - Code reordering/rescheduling in hot loop possible
int process(data) {
   ...
   if (problem) return 1 else return 0;
}

int errors = 0;
for (int i = data.length-1; i >= 0; i--) {
   errors += process(data[i])    // no exceptions
   ... // additional operations, always executed
}
if (errors > 0) {
   ... // handle error
}
```

Runtime Optimizations

6) Optimization Hints for the Target Device

- Problem
  - Runtime optimizations must be dynamically detected
  - This adds overhead and delays optimized execution
- Impact
  - Execution speed and consistency
- Guideline
  - Give hints to build system/runtime environment, e.g. define known hot code for eager optimization
    - Note: Expected to become more widely supported in future platforms

Code Execution

7) Code Reuse

- Problem
  - Application has multiple variants of similar hot code
  - This adds overhead for extra optimizations and dilutes hot spots
    - Code reuse is generally a good idea, but even more important in optimizing platforms
- Impact
  - Liveliness, execution speed and consistency
- Guideline
  - Design for code reuse as much as practical, in particular with hot code

Code Execution

8) Native Code

- Problem
  - Frequent calls to native code for “optimization” purposes (as opposed to functional reasons)
  - In optimized platforms the transitions between native and Java code may incur significant overhead
- Impact
  - Execution speed
- Guideline
  - Avoid frequent calls to native code unless the work performed is worth the transition overhead
    - Note: Determining the trade-off might be difficult

Example: Native Code

```
// System.arraycopy() is often implemented in native.
// In this example, srcArr and dstArr hold primitive
// values. Threshold is implementation dependent.
if (length <= threshold) {
  for (i = length-1; i >= 0; i--) {
    dstArr[i] = srcArr[i]; // better off copying directly
  }
}
else {
  System.arraycopy(srcArr, 0, dstArr, 0, length);
}

//(smart VMs can optimized this ...)
```

Code Execution

9) Qualifiers

- Problem
  - The Java language allows liberal use of features such as polymorphism and a wide scope of visibility
  - This lack of restrictions prevents certain optimizations (fast access to members, fast calling, simplified code transformation)
- Impact
  - Execution speed
- Guideline
  - Use **`private/static/final`** where possible

Code Execution

10) System.gc() (and Many VMs This Is No-op)

- Problem
  - Application periodically calls System.gc()
  - Sophisticated memory management systems already dynamically adapt to a variety of conditions
  - This means calls to System.gc() likely add overhead without any benefit (or even cause disruption)
- Impact
  - Execution speed
- Guideline
  - Don’t call System.gc()

Code Execution

11) Complex Byte Codes

- Problem
  - Use of complex Java byte codes (e.g. **`aastore`**, ***`new`***`,` **`instanceof`**`,` ...) in tight loops
  - Operation might be heavyweight or cause transition to a different execution state (e.g. software emulation) with lots of associated overhead
- Impact
  - Execution speed
- Guideline
  - If possible, avoid complex byte codes in hot code
    - Hint: Look at class file (byte stream of method) to verify

Example: Complex Byte Codes

```
// Use instanceof to determine object type: Non-optimal!
class BaseObject { ... }
class Object1 extends BaseObject { ... }
class Object2 extends BaseObject { ... }

BaseObject[] objList;
for (int i = objList.length-1; i >= 0; i--) {
  if (objList[i] instanceof Object2) // complex byte code
    ... // do something
}
```

Example: Complex Byte Codes (Cont.)

```
// Better - Add tag, allow easy runtime type determination
// Not the best OOP, but an acceptable tweak for specific
// situations
class BaseObject {
  bool isObject2;      // tag
}
class Object2 extends BaseObject {
  Object2() {
    isObject2 = true; // constructor sets tag
  }
}

for (int i = objList.length-1; i >= 0; i--) {
  if (objList[i].isObject2) // simple member access
    ...
}
```

When Compilers and Interpreters Disagree

12) Write Platform-Specific Hot Code

- Some new JSRs are designed to work with an optimizing compiler; for example, to manipulate vertex data with JSR 239, Java Bindings for OpenGL ES

```
intBuffer.put(value1);    →    str r0, [r1], #1
intBuffer.put(value2);    →    str r0, [r2], #1
```

- Techniques good for interpretation actually hurts compiled performance

```
javaArray[n++] = value1; javaArray[n++] = value2;
intBuffer.put(javaArray);
```

JIT Warmup

13) Avoid FPS Ramp-up

- Problem
  - The game reaches optimal frame rate only after dynamic compilation is complete
  - User may see a “ramp up” of FPS
- Impact
  - Execution speed and consistency
- Guideline
  - Run the game loop in an invisible “warm up” loop before starting the game

Memory Use

14) Allocation Rate and Overhead

- Problem
  - Application tries to avoid memory allocation and gc, and recycle memory itself
  - This likely imposes much more overhead (and bugs) than the VM-level memory management
- Impact
  - Execution speed and consistency
- Guideline
  - Leave most (or all) memory management to the Java VM
    - But avoid high allocation rates and spikes (often a sign of poor application design)

Memory Use

15) Spikes in Memory Usage

- Problem
  - Allocation of large objects or periodic increases in allocation activity cause large spikes in memory usage
  - This may cause low memory conditions and/or undo existing optimizations (e.g. trash code and information)
- Impact
  - Liveliness, execution speed and consistency
- Guideline
  - Maintain consistent and low to medium allocation rate, cleanup and free objects in timely manner
  - Pool/reuse objects with large or heavyweight allocations

Example: Memory Use

```
// Creates garbage on every collision: Non-optimal
class Fragment {
  ... // graphics and/or image data
  init() {
    ... // simple initialization to initial values
  }
}
initFragments() {
  for (int i = fragments.length-1; i >= 0; i--)
    fragments[i].init();
}
handleCollision() {
  // Throw away and allocate 30 objects on every collision
  Fragment[] fragments = new Fragment[30];
  initFragments();     // must initialize before using
  animateFragments(); // animate collision
}
```

Example: Memory Use (Cont.)

```
// Better - Reuse objects and do eager initialization
// - Low init. overhead compared to allocation and gc
//   makes reuse worthwhile
// - Removes allocation and gc spikes
// - Improves visuals with eager initialization
initApp() {
  Fragment[] fragments = new Fragment[30];
  initFragments();  // init. for first use
}

handleCollision() {
  // Fragments already initalized, no delay
  animateFragments();
  initFragments();    // reset for next collision
}
```

Know Your Platforms

www.eembc.com

- Good source of Java VM performance; fairly reliable

www.jbenchmark.com

- Good source of graphics and game performance, including 3D
- Simple benchmarks, beware of cheating!

Summary

- Be aware of the properties and techniques of advanced Java ME platforms
- Make your application “Java VM-friendly” and maximize your chances for a substantial performance boost
- This is only a snapshot in time
  - Java ME platforms will become more capable and push the envelope on optimization techniques
  - Application design must continue to adapt in order to deliver the best user experience

For More Information

URLs

- www.eembc.org (click on Java subsection)
- www.jbenchmark.com

Books

- Effective Java, http://java.sun.com/docs/books/effective/

Q&A

Writing Optimized Applications for High-Performance Java™ ME Runtime Environments

Kyle Buza
Sun Microsystems

Oleg Pliss, Ph.D.
Sun Microsystems

TS-1451