Universitatea Tehnica din Cluj-Napoca
Departament Calculatoare

# Programming Techniques in Java

## Performance Tuning and Techniques

I. Salomie, C.Pop
2017



# Introduction

- Common perception: Java is slow ??!!
  - Many assume that if a program is not complied it must be slow
  - Some of early Java versions (non-optimal coding, un-optimized JVMs, etc. ) were slow
- Java overhead
  - JVM layer - abstracts Java from the physical machine
  - Java main advantages add certain performance overhead
    - Platform independence, memory management,
    - Exception checking,
    - Dynamic resource loading,
    - Security checks, etc.
  - OO and inheritance
    - Run time linking
    - Hierarchical method invocation
    - Small methods such as getters and setters - inline optimizations
- All first run programs (no matter implementation language) could seem slow
  - Not yet performance tuned
  - Bottlenecks not yet identified

# System limitations

- Resources that may limit performance for all applications
  - CPU speed and availability
  - System memory
  - Disk (and network) input/output (I/O)
- Question - (when tuning)
  - which of these resources is causing the application to run too slowly



# System limitations

**CPU speed and availability**

- Look for possible improvements in the code
  - bottlenecks,
  - inefficient algorithms,
  - too many short-lived objects
    - Object creation and garbage collection are CPU-intensive operations
  - etc.

**System memory limits**

- Memory blocked with
  - Too many objects, or
  - A few large objects,
  - Too many allocated large arrays
    - Frequently used in buffered applications
- Check application design
  - May need to be reexamined to reduce its running memory footprint

# Tuning strategy

**Overview**

- Identify the main bottlenecks
  - look for a few bottlenecks possibilities
- Choose the quickest and easiest one to fix and address it
- Repeat for other identified bottlenecks

**Details**

**Loop the following sequence of actions:**

- Measure the performance
  - use profilers and benchmark suites
  - instrumenting code
- Identify bottleneck locations
- Think of a hypothesis for the cause of the bottleneck
  - Consider any factors that may refute your hypothesis.
- Create a test to isolate the factor identified by the hypothesis.
  - Test the hypothesis.
- Alter the application to reduce the bottleneck.
- Test
  - the alteration improves performance
  - measure the improvement
  - do regression-testing the affected code



# Tuning strategy

**Profiler** [Wikipedia]

- Profiling ("program profiling", "software profiling") is a form of dynamic program analysis that measures, for example, the space (memory) or time complexity of a program, the usage of particular instructions, or frequency and duration of function calls
- Profilers are used in the performance engineering process
- The most common use of profiling information is to aid program optimization
- Profiling is achieved by instrumenting either the program source code or its binary executable form using a tool called a *profiler* (or *code profiler*).
- A number of different techniques may be used by profilers, such as event-based, statistical, instrumented, and simulation methods

# Tuning strategy

**Profiling techniques**

- Flat profilers (determine only the average time calls)
- Call-graph profilers (call times and frequencies of function calls, call chains, etc.)
- Event-based profilers – provide hooks to trap events, class load, unload, thread enter, leave, etc.
  - For Java: JVMTI (JVM Tools Interface) API, formerly JVMPI (JVM Profiling Interface)
- Statistical based profilers
- Instrumented profilers
- Simulation based profilers



# Tuning strategy

**Recommended Java Profilers**

- Jprofiler

https://www.ej-technologies.com/products/jprofiler/overview.html

- VisualVM

https://visualvm.github.io/

- Java Mission Control

http://www.oracle.com/technetwork/java/javaseproducts/mission-control/java-mission-control-1998576.html

- YourKit

https://www.yourkit.com/

# Tuning strategy

**Program Instrumentation**

- Program instrumentation - modifies a program to analyze itself
  - The inserted code outputs the analysis data
  - Usually slows down the instrumented program
- Types
  - Manual inserted statements
  - Tools for automatic adding of the instrumented code
  - Compiler assisted adding of instrumented code, for example
    - gcc –pg
  - Runtime instrumentation- before the execution, the code is instrumented. Program execution is fully supervised and controlled by such a tool (ex. Pin, Valgrind, DynamoRIO);
  - Runtime code injection (more lightweight than Runtime instrumentation) – code modified at runtime. Inserts jumps to helper functions (ex. DynInst)



# Go faster

**Threading to appear quicker**

- Use multithreading to make an application responsive
- Use threads to ensure that any particular service is available and unblocked when needed
  - Difficult to program correctly and manage
  - Complex task - handling inter-thread communication

# Go faster

## Streaming to appear quicker

- The general case
  - A long activity that can provide results in a stream is identified
  - The results can be accessed a few at a time
- The case for distributed applications
  - Sending all data vs. streaming data while it is processed
  - Example:
    - Web browsers - display the initial page screen as soon as it is available, without waiting for the whole page to be downloaded



# Go faster

## Caching to Appear Quicker

- Moving data is expensive
  - If the same data is used multiple times = > keep a copy
  - Example: browser downloads
  - Disk access example:
    - from lowest-level hardware cache
    - to OS read / write cache
- Cached file systems
  - File reading / writing classes that provide buffered I/O
  - Cost of reading a byte is the same with reading a page

## Performance and Metrics

- Applications must meet user expectations
  - Functional requirements
  - Non-functional requirements
    - … performance among them
- Agree on performance targets
  - Agree on performance metrics
  - Identify target response times for as many subsystems as possible
- Without clear performance objectives tuning will never be completed
- System-wide throughput
  - number of transactions per minute for the system as a whole
  - response times on a saturated network
- The maximum number of resources the application should support
  - users, data, files, file sizes, objects



## Setting benchmarks

- Benchmark (Wikipedia)
  - The result of running a computer program, or a set of programs, in order to assess the relative performance of an object by running a number of standard tests and trials against it
- Setting benchmarks - precise specification stating
  - High level - what functionality needs to run in what amount of time
  - Low level - what part of the code needs to run in what amount of time
  - Specify in terms of user actions
    - the time elapsed from user pressing the button XXX until a result is displayed on the screen
- Important questions
  - How fast ?
  - In which parts ?
  - For how much effort ?

# Setting benchmarks (cont.)

- Target times specification for each benchmark
  - best times, accepted times, etc.
  - Example
    - Function X takes to execute - from pressing the button until a response is displayed
      - 2 sec in 70% of the cases
      - 2-3 sec in 15% of the cases
      - 3-5 sec in 15% of the cases
  - Anything that can vary should be controlled and reproducible



# Benchmark Harness

- Tools for testing applications
- Benchmark harness - different complexities
  - from a simple class that sets some values and starts main()
  - logging and timestamp
  - to GUI-run benchmark harness
- Examples
  - SPECjbb2013 from
  - http://www.spec.org/jbb2013/
  - http://www.spec.org/osg/jbb2005/

# Benchmark Harness
## Requirements

- Correctly reproduce user activity and data input and output
  - Reproduce and simulate all user input, including GUI input
- Support different application configurations
- Control any randomized inputs
- Random sequences used in tests should be reproducible
- Support for logging statistics
- Allow testing the system across all scales of intended use
  - up to the maximum # of users, simulate users
  - objects
  - throughputs, etc
- Each run needs to be under identical conditions (as much as possible)



# Benchmark Harness
## How to measure

- The benchmarks should be run multiple times
  - the full list of results should be stored and processed
    - i.e. not just the average and deviation or the ranged percentages
  - Assure similar conditions for benchmark runs
- Each code change should be driven by
  - Identifying exactly which bottleneck is to be improved
  - How much speedup is expected
- After each code change (or set of changes)
  - Run benchmarks to precisely identify improvements (or degradations) in the performance across all functions
- Code Optimizing
  - Can introduce new bugs, so the application should be tested during the optimization phase.
  - Validated only after the application using that optimization's code path has passed quality assessment.
  - Should be completely documented

# Benchmark Harness
## What to measure

- Main measurement reports
  - time
  - throughput
- Main way to measure time
  - **System.currentTimeMillis()**
  - two calls to determine the elapsed time
  - works well for not short periods of time
  - Call costs
    - takes up to 1/2 millisecond to execute
    - for a period of 100 milliseconds the call cost is 1%
  - Other method
  - **System.nanoTime()**
- Always small variations between test runs
    - use averages to measure differences and
    - consider whether those differences are relevant by calculating the variance in the results.
- CPU time
  - time allocated on the CPU for a particular procedure
- The # of runable processes waiting for CPU
  - this gives you an idea of CPU contention
- Memory sizes
- Disk throughput
- Disk scanning times
- Network traffic, throughput, and latency
- Transaction rates
- Other system values
- Note. Measuring these values needs:
  - system knowledge - no Java mechanisms for direct measuring of these values
  - application-specific knowledge (what is a transaction for your application?)



# Techniques for Performance
## Loops

- Move out code from loops
- Code that does not need to be executed on every pass
- Assignments, accesses, tests, and method calls that need to run only once
- Method call has costs
  - Analyze task - moving method calls out of the loop (even if this requires rewriting)

# Techniques for Performance
## Loops

- Termination test and method calls
  - Avoid method call in a loop termination test
  - Significant overhead
- Example

```
for(int i = 0; i < collection.size( ); i++)
```

- Factor out the method call

```
int max = collection.size(  );
for(int i = 0; i < max; i++)
```



# Techniques for Performance
## Loops

- Loops, temporary variables and arrays
  - VM performs bounds-checking for array-element access
  - Array access (and assignment)
    - More overhead than temporary variable access
    - Inefficiency - repeated access in each iteration
    - Do array access once - and assigned to a temporary outside the loop

- Example

```
for(int i = 0; i < repNo; i++)
  countArr[0] += 10;
```

- Optimized loop

```
count = countArr[0];
for(int i = 0; i < repNo; i++)  count += 10;
countArr[0] = count;
```

**Note.** Twice as fast

## Techniques for Performance
Loops

- Use int for Index Variables
  - int is the fastest data type
  - The VM is optimized to use ints
    - Operations on bytes, shorts, and chars - implicit casts to/from ints
- Example

Instead of

```
for(long i = 0; i < repNo; i++)
for(char i = 0; i < repNo; i++)
```

use int data types

```
for(int i = 0; i < repNo; i++)
```



## Techniques for Performance
Loops

- Loops vs. System.arraycopy( )
  - Copy arrays
    - Loop
    - System.arraycopy()
  - System.arraycopy() is faster
    - when VM has a JIT - loop could be little faster
- Conclusion
  - always use System.arraycopy()
    - even if a VM JIT is used

## Techniques for Performance
Loops

- Comparison to 0 is faster than comparisons to most other numbers.
  - The VM has optimizations for comparisons to the integers
    - -1, 0, 1, 2, 3, 4, and 5.
  - Rewrite loops to make the test a comparison against 0
  - Note. The latest VMs try to optimize the standard loop expression, so rewriting the loop may not produce faster code
- Reverse the iteration order of the loop
  - from counting up (0 to max) to counting down (max to 0).
  - Instead of

```
for(int i = 0; i < repNo; i++)
```

  - Could be coded (for better performance):

```
for(int i = repNo-1; i >= 0; i--)
for(int i = repNo; --i >= 0 ; )
```



## Techniques for Performance
Exceptions

- Exception generation know-how
  - important for good design and
  - performance achievements
  - an exception should be thrown only in exceptional situations
  - Example
- try-catch blocks costs
  - no extra time if no exception is thrown
    - some VMs may impose a slight penalty
  - significant overhead if an exception is thrown
    - due to stack snapshot creation
    - equivalent to the execution of several hundred of simple code lines

## Techniques for Performance
### Exceptions

- Implicitly generated exceptions
  - JVM generated exceptions
    - ClassCastException
    - ArrayIndexOutOfBoundsException
- Explicitly thrown exceptions
  - using throw statement
  - StackTrace generation overhead
- Reusing an existing exception object
  - reduce cost by not creating a new exception object
  - two orders of magnitude faster
  - avoids stack trace generation = > reduce the overhead



## Techniques for Performance
### Exceptions

**Reusable Exceptions - Example**

```
public static Exception
    REUSE_EXCEPTION =
                new Exception( );
...
//Faster by exception reusing
try {
  … throw REUSE_EXCEPTION;
}
catch (Exception e) {...}

//The following try-catch is 50..100
// times slower
try {
   … throw new Exception( );
}
catch (Exception e) {...}
```

- Disadvantage
  - the stack trace is incorrect for the thrown exception object
  - it is the one that is generated when the exception is created
  - this could be important and may generate maintenance future problems
- Note
  - fillInStackTrace() should be invoked to get the stack filled with the current situation
  - this call generates the mentioned overhead!!

# Techniques for Performance
## try/catch vs. instanceof

**// pesimistic approach**
```
public static boolean test1(Object obj) {
   if (obj instanceof Integer) {
     Integer i = (Integer) obj;
     return false;
   }
   else
     return true;
 }
}
```

**// optimistic approach (speculative cast)**
```
public static boolean test2(Object obj) {
   try {
     Integer i = (Integer) obj;
     return false;
   }
   catch (Exception e) {return true;}
}
```

**Conclusion**
- test1() takes 1 TU (assumption)
- test2() with ClassCastException thrown
  - 100 TU
- Recommendation
  - use instanceOf instead of speculative cast



# Techniques for Performance
## Method parameter checking

- Check method arguments for validity
  - Ok during development / testing time
  - Significant overhead for the released product
  - Error checking could be removed for the released product
- Next is a technique for error parameter checks that can be optionally removed (through an extra compilation)

**// Conditional Error Checking)**
```
public class GLOBAL_CONSTANTS {
  public static final boolean ERR_CHECK_ON =  true;
  ...
}

//code in methods of other classes
if (GLOBAL_CONSTANTS.ERR_CHECK_ON) {
   //error check code of some sort
   ...
}
```

- Allows turning ON/OFF error checking by recompiling
- A compiler feature – eliminates the overhead of the if statement for each check

# Techniques for Performance
## Assertions

- Forms
  **assert boolean_expression;**
  - if boolean expression evaluates to false => AssertionError is thrown.

  **assert boolean_expression : String_expression;**
  - allows message customization using a String parameter
  - Example

  assert val >= 0 : "Parameter val must be non-negative, but was " + val
- assert statement - normal statement comparison
  - assert can be enabled / disabled at runtime
- Assertions enabled
  - Better quality code
  - Assists in diagnosing problems
- Assertion enable / disable
  - Assertion enable / disable granularity - class level
  - **-ea** and **-da** parameters of **java** command can be used to specify for each class if assertions are enabled or disabled




# Techniques for Performance
## Assertions


### How assertion works

- At compile time
  - When an assertion is found in a class
    - A new field is added to the class - and is left unassigned (internal legal operation)
    - $assertionsDisabled
  - How assertion is compiled

```
if (! $assertionsDisabled)
  if (!boolean_expression)
    throw new AssertionError(String_expression);
```

- At class loading time
  - class loader determines assertion status of the class using the specified assertion rules and sets the value of  $assertionsDisabled

# Techniques for Performance
## Assertions

### Conclusion

- Assertions are disabled
  - Without any further optimization every assert costs a minimum one runtime test
  - Significant overhead for short methods
  - Setters, getters and other short frequently called methods - think of avoiding
- Assertions are enabled
  - Any assertion takes at least as long to run as its boolean_expression evaluation takes
  - Code running with assertions enabled will be slower than code running with assertions disabled - even if only a few percent slower



# Techniques for Performance
## Assertions

**Assertion vs. explicitly checks**

- Facts
  - Assertions can be turned off
  - Explicit checks cannot be turned off
- Conclusion
  - Consider changing all explicit checks for incorrect parameters and state in your code to use assertions instead of explicitly using **if...throw** statements
  - Example
    - IllegalArgumentExceptions often test for documented incorrect conditions
    - These tests could be changed to assertions
- Ultimate decision
  - The test should always be present
    - Don't make it an assertion
  - The test is optional and provides extra robustness, especially during development and testing
    - Use assertion

# Techniques for Performance
## Assertions

**Examples**

```
public int noCheckMethod(int iVal) {
  // … method body
  return …
}
public int explicitlyCheckMethod(int iVal) {
  if(iVal < 0)
  throw new IllegalArgumentException("parameter val should be positive, but is " +
    val);
  // … method body
  return …
}
public int usingAssertMethod(int iVal) {
  assert (val >= 0) : "parameter val should be positive, but is " + val;
  // … method body
  return …
}
```



# Techniques for Performance
## Casts

- Resolve time
  - Some casts can be resolved at compile time
    - They are eliminated by the compiler
  - Casts not resolvable at compile time must be executed at runtime
- Casts cost dependences
  - Depends on the depth of the hierarchy
    - The further back in the hierarchy the longer the cast takes to execute
  - Depends whether the casting type is an interface or a class
    - Interfaces are generally more expensive to use in casting
- Important note
  - Never change the design of the application for minor performance gains

# Techniques for Performance
## Casts

**Good practices**

- Avoid casts whenever possible
  - Use type-specific collection classes instead of generic collection classes
    - Instead using a standard List to store a list of Strings
    - Better performance with a StringList class
  - Type the variable as precisely as possible
- Variable needing casting several times
  - Cast once and save the object into a temporary variable of the cast type
  - Use the temporary variable instead of repeatedly casting



# Techniques for Performance
## Casts

**Bad practices**

- Avoid the following type of code

```
if (obj instanceof Something)
  return ((Something)obj).x + ((Something)obj).y + ((Something)obj).z;
...
```

- Use a temporary variable
  - more readable code

```
if (obj instanceof Something) {
  Something something = (Something) obj;
  return something.x + something.y + something.z;
}
...
```

# Techniques for Performance
## Variables

- Local (temporary) and method-argument variables
  - The fastest variables to access and update
  - Local variables
    - Remain on the stack, so they can be manipulated directly
  - There are special bytecodes for accessing the first four local variables and parameters on a method stack
- Static and instance variables
  - Stored and manipulated in the heap memory
  - Instance and static variables vs. local variables and method arguments
    - Up to an order of magnitude slower to operate on



# Techniques for Performance
## Variables

**Temporary objects**

- Created from the heap
- Object reference itself is allocated on the stack
- Operations on any object are slower than on any of the primitive data types for temporary variables
- Temporary primitive variables vs. temporary objects
  - As soon as variables are discarded at the end of a method call, the memory from the stack can immediately be reused for other temporaries.
  - Any temporary objects remain in the heap until garbage collection reallocates the space
- **Conclusion**: Temporary variables using primitive (non object) data types are better for performance

# Techniques for Performance
Variables

## Good practice

- When many manipulations on instance or static variable
  - Execute them on a temporary variable
  - Reassign to the instance variable at the end
    - This is true for instance variables that hold arrays as well
- Where possible
  - Access public instance variables rather than use methods (getters and setters)
  - This breaks encapsulation (bad design)
  - Java SDK uses this techniques in some of the cases



# Techniques for Performance
Variables

## Good practice

- Array-element access - typically two to three times as expensive as accessing non-array elements
  - Due to range checking and null pointer checking (for the array itself) done by the VM
  - The VM JIT compiler manages to eliminate almost all the overhead in the case of large arrays
- Arrays overhead, due to the range checking in Java
  - If an array element is manipulating many times = > assign it to a temporary variable for the duration

```
for(int i = 0; i < repNo; i++)
  countArr[0]+=i;
```

Replace with:

```
int count = countArr[0];
for(int i = 0; i < repNo; i++) count+=i;
countArr[0]=count;
```

# Techniques for Performance
## Variables

**Good practice**

- ints
  - The fastest variable type to operate on
- shorts, bytes, and chars
  - All widened to ints for almost any type of arithmetic operation
  - Cast back is required if you want to end up with the data type you started with
  - Example
    - Adding two bytes produces an int and requires a cast to get back a byte
- longs and doubles
  - Can take longer to access and update than other variables
    - Due to … they are twice the basic storage length for Java (which is four bytes)
- Floating-point arithmetic
  - The worst in terms of performance