Issues in High-Performance Programming in Java

HPCS Tutorial

June 13th, 1999

Bowen Alpern
alpern@watson.ibm.com

Susan Flynn Hummel
hummel@watson.ibm.com

http://www.research.ibm.com/jalapeno

Welcome

This tutorial will:

- Identify Java performance issues
- Survey ongoing projects to address these issues

Intended audience:

- Programmers concerned with performance
  (but not obsessed with performance)
- Either building a large software project in Java
- Or considering building such a project in Java

Assumptions:

- Experience writing high-performance programs
- Some knowledge of Java

This tutorial is not:

- An introduction to performance programming
- An introduction to Java
- An introduction to Object Oriented programming
- An assessment of current Java performance
- Techniques for attaining C-like performance
- An introduction to HotSpot or Jalapeño

Tutorial outline

- Performance issues with Java
  - How performance issues can be addressed
  - Economic considerations
  - Problematic features of Java
    - Object layout and method dispatch
    - Exception handling
    - Dynamic type checking
    - Threads and synchronization
    - Garbage collection
    - Dynamic class loading
  - A note on JIT compilers
  - Missing features for scientific programming
- Addressing Java performance issues
  - The Java Grande forum
  - Micro scale performance: fused multiply-adds
  - Micro scale performance: array bounds checking
  - Mini scale performance: simulated forall
  - Macro scale performance: distributed computing

How issues can be addressed

- Change the language (not easy)
- Use a dialect
  - Pizza, Titanium
- Object oriented preprocessor
- Static compiler
  - Source to machine code
- Java compiler
  - Source to bytecodes
- Bytecode optimizer
  - Bytecodes to bytecodes
- JIT compiler
  - Bytecode to machine code
- JVM design
- Programming idioms
  - Avoiding the most problematic features

Economics of programming

Larry Carter's Law:

- Programmers are more expensive than computers
- 1 programmer/year ~ $100,000
- 1 processor/year (~10^{16} cycles) ~ $1,000

Expeditious programming

- Development costs dominate
  - 1 programmer year effort
  - Less that 10^{17} cycles over program lifetime

Performance programming

- Execution costs dominate
  - 1 programmer year effort
  - More than 10^{19} cycles over program lifetime

Semi-performance programming

- Parity between execution and development costs
- Automatic memory management justified
  - On a 1 programmer year effort
  - If it reduces development cycle by three months
  - And performance by 20%

Problematic Java features

Object orientation

- Objects
- Virtual methods
- Interfaces

Runtime checks

- Null pointer
- Array bounds
- Dynamic typing

Threads (shared-memory parallelism)

Automatic memory management

Exceptions

Dynamic class loading

Portability (bit-wise reproducible results)

Security mechanisms ???

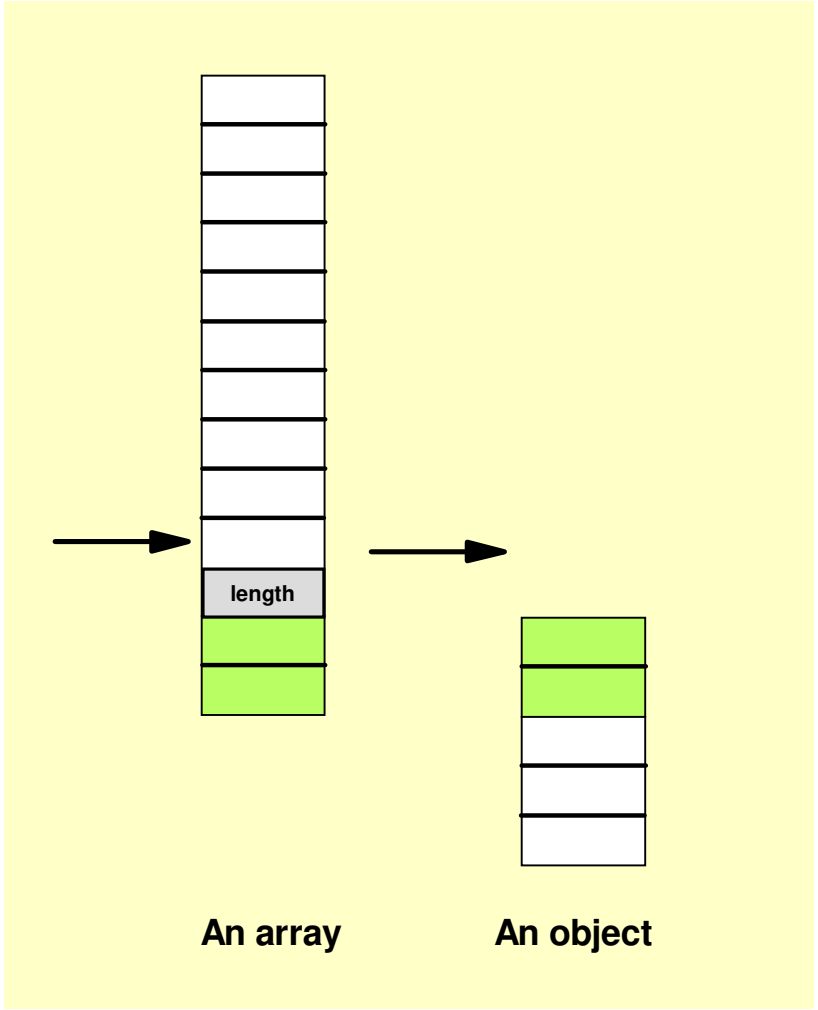
Java objects

An object is either an array or a scalar
- Array components are primitive or reference
- Scalar has primitive and reference fields
- No embedded objects

Auxiliary information for an object supports
- Virtual method dispatch
- Obtaining information about the object's class
- Thread synchronization (locking)
- Hashing
- Garbage collection

Maintaining auxiliary information
- Handle -- away from the data
  - Allows object movement without updating references
  - Requires additional word to point to the data
  - Requires additional indirection to access the data
- Header -- adjcent to data
  - Object movement requires updating all references to it
  - Less space overhead
  - Easier access to the data

Jalapeño object layout



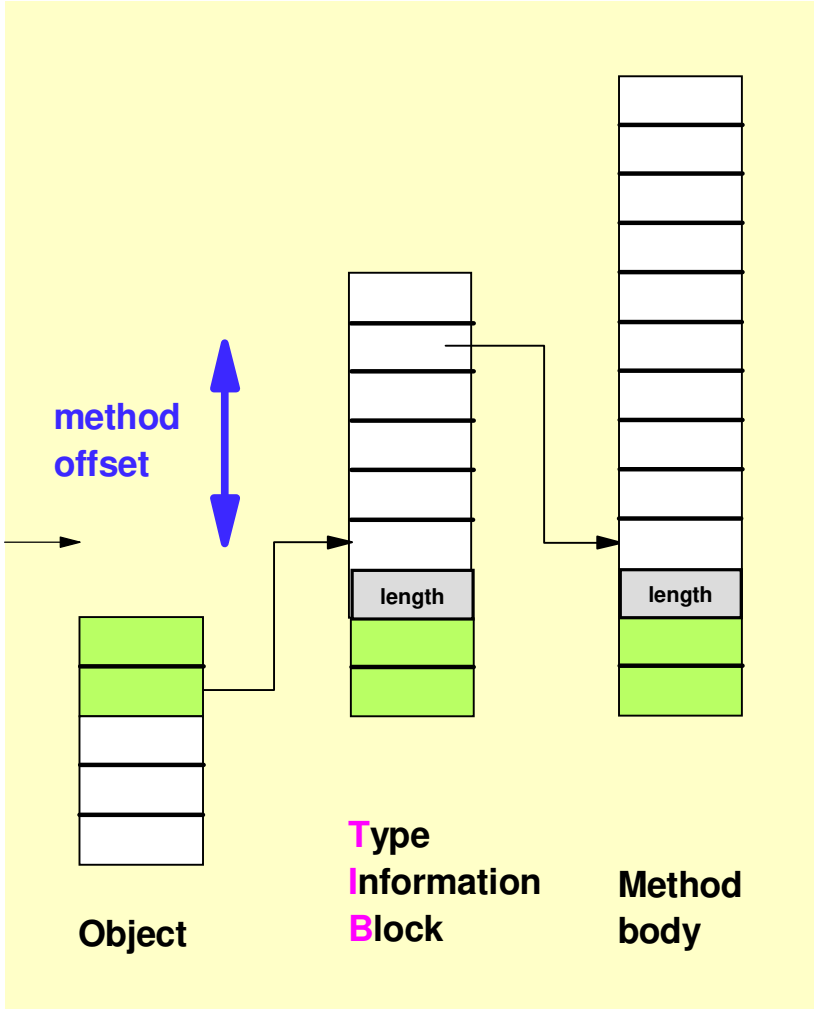
Reducing object overhead

- Traditional object allocation
  - Heap allocated
    - Requires periodic garbage collection
  - No embedded objects
    - Indirection overhead to access objects
    - Space overhead for object headers
- Escape analysis [Jong *et.al.*, OOPSLA'99]
  - Objects that do not survive the current method
  - Can be allocated in current stack frame
- Deep stack allocation [Reid *et.al,* WCSSS'99]
  - Other objects
  - Can be allocated further up the stack
    - Buffers associated with each stack frame
    - Deallocated when the frame is popped
- Embedded objects
  - Private reference fields of a class
  - Not returned from the classes methods
  - Can be subsumed into the object

Virtual method invocation

- Virtual methods depend on an object's class
  - Derived class may override superclass definition
- Virtual method invocation overhead
  - Virtual method dispatch is cheap
  - Method prologue/epilogue may not be
    - stackframe allocation/deallocation
    - register save/restore
    - dynamic profiling (possible)
- JIT compiler can help
  - Inline static and final methods and constructors
    - Inline virtual methods is problematic
      - Subsequent class load may invalidate inline decision
    - Tip: declare final methods final
  - Leaf methods may not need a stackframe
- Language fix
  - Pragma: inline method

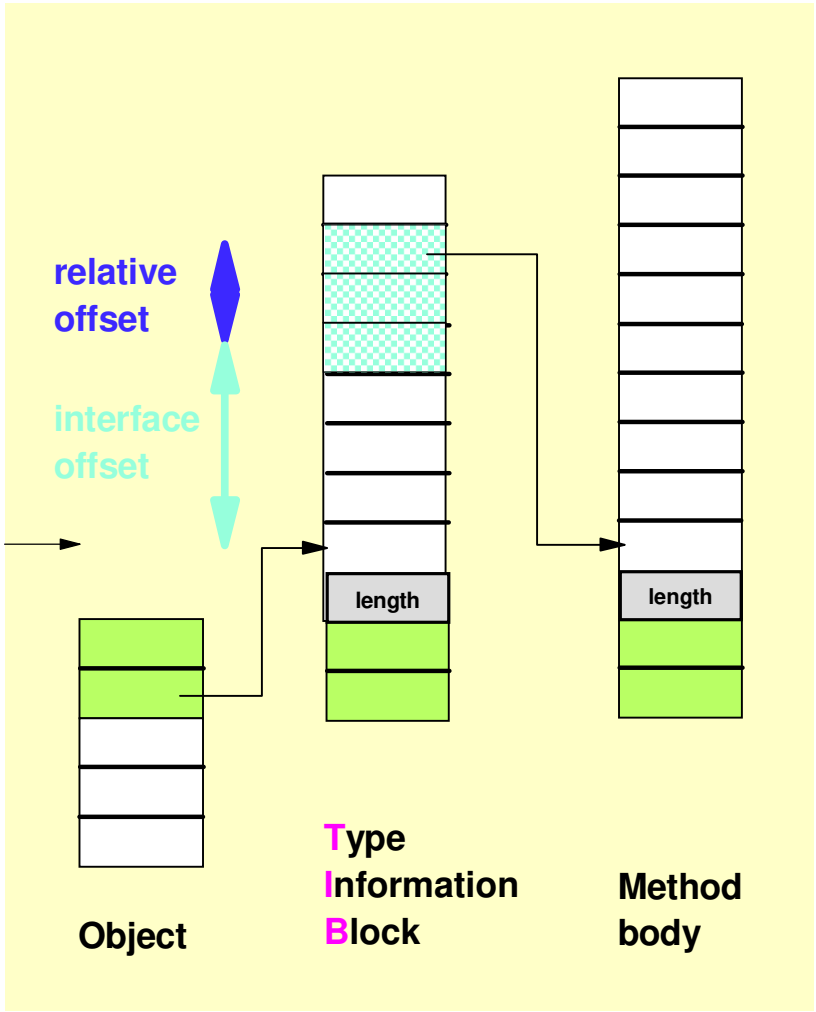
Virtual method dispatch



Interface invocation

- Interfaces allow multiple inheritance
  - Implementing classes have required methods
  - Problem: method offsets will differ
    - actual offset = interface offset + relative offset
    - JIT compiler knows relative offset
- Reserve TIB space for *important* interfaces
  - Wastes space
    - For non final classes that don't implement them
    - Can reclaim for final classes
    - Tip: declare final classes final
  - Which interfaces are important?
- Caching interface offsets
  - With the class (last interface invoked)
    - Minimal space overhead
    - Limited utility
  - At the call site (last class)
    - Greater utility
    - Greater space overhead
  - Imperfect

Interface dispatch



Null pointer checks

Every object access requires a check
- NullPonterException, if reference isn't valid

Normally, this is not useful work
- Tip: don't use exception to find the end of a list

Java compiler's can't help
- The check is inherent in the relevant bytecodes

JIT compiles can help (often)
- Redundant checks can be eliminated
- This entails some "compile-time" cost

JVM can help (on some hardware)
- Generate hardware interrupt
- On Windows NT
  - Null pointer has value 0
  - Page 0 is not readable (or writable)
- On AIX
  - Null pointer has value 0
  - Page -1 is not readable (or writable)

Array bounds checks

Access to A[i]

- requires checking: 0 <= i < A.length

Normally, this is not useful work

- Tip: don't use this exception for loop termination

JIT compiler can help (some)

- i logically less than A.length (one instruction)
- Eliminate redundant checks
- Can't prove check is unnecessary
  - too expensive
- Could verify proof check is unnecessary
  - proof supplied by Java compiler or bytecode optimizer

Program transformation can help

- Ninja

Language fix:

- Assertions
  - Proof with bytecodes, or
  - Checked at runtime

Dynamic type checking

Down cast bytecodes

- instanceof: x instanceof Y
- checkcast: (Y) x
- aastore: y[i] = x

Normally, not useful work

JIT Compiler can eliminate redundant checks

- If (x instanceof Y) yy = (Y) x;

Fast checks

- If X == Y then yes
- Otherwise if Y is final then no
  - Tip: declare final classes final

Cache previous results

- At X, Y, or site
- Previous value or last success and failure

Language fix:

- Templates (Vector<Y>)

Array stores

The problem

Truck [] a = y;

Java compiler checks y is a Truck[]

a[i] = x;

Java compiler checks x is a Truck

Dynamic check

if y is a Y[], x must be a Y

Fast checks

If Y is Truck ??

Declared type available for fields

Not for local variables

If Y is Object

Note: checking if Y is final isn't useful

Almost all array stores succeed

Speeding up the fail case doesn't help

Threads

Addresses SMP parallelism

Threads designed to run concurrently

Hardware can run several simultaneously

Modeled after operating system threads

OS threads modeled after light weight processes

Mostly independant

Occasionally, interact with global data structures

To synchronize access to data structures

Monitors [Hoare '74]

wait(), notify(), notifyAll()

Java memory model

Orders *unsynchronized* accesses to shared data

Tip: avoid "false sharing" where possible

Sharing: same data accessed by different threads

False sharing: near by data accessed

Thread scheduling

- Java threads as operating-system threads
  - System handles thread-switching
    - Threads can be interrupted anywhere
  - System services for heavy weight locking
    - System services are expensive
      - ~50,000 cycles per kernel call
- Virtual processors multiplex Java threads
  - System threads implement virtual processors
  - Virtual processors handle thread-switching
    - (Timer interrupt from system)
  - Thread-switching at defined yield points
    - Allows more compiler optimization
    - Less overhead for garbage collection support
  - Allows locking and thread-switch integration
- Design considerations
  - Minimize thread-switch overhead
  - Minimize contention for global queues
  - Load balancing
    - Keep processors busy (throughput)
    - Fairness (response time)

Shared-Memory Multiprocessors

- SMP model
  - Several processors
  - One global memory
  - One local cache memory per processor
- Problems
  - Processors need to be able to synchronize
  - Access to variables need to be ordered
- Coherent cache:
  - Accesses *to each variable* are in program order
- PowerPC architecture
  - Synchronization primitives:
    - lwarx -- load with reservation
    - stwxc -- store conditional
  - Access ordering
    - Cache line are held shared or exclusive
    - Exclusive required for writes
    - Writes cause invalidates to be broadcast
    - sync -- forces invalidates to be processed

Java Memory Model

Java memory model is a disaster
- Almost impossible to understand
- Prevents common compiler optimizations
- Allows bizarre behavior
- Many implementations are wrong (most? all??)
- http://www.cs.umd.edu/~pugh/java/index.htm

Simplified memory model
- Accesses (4 byte reads or writes) atomic
- Data flow order within an assignment
- Coherent access to each variable
  - Weaker: allow out of order reads
- Accesses within a synchronized block or method
  - After monitorenter
  - Before monitorexit
- At most one thread in a monitor at a time

Tip: synchronize to access shared data
- Safe idiom:
  - Unsynchronized reads
  - Synchronized atomic writes
    - Probably not visible immediately

Monitors

Conceptually:
- One monitor for each object (and class)
- Only one thread can own a monitor at a time
- A thread can acquire a monitor recursively

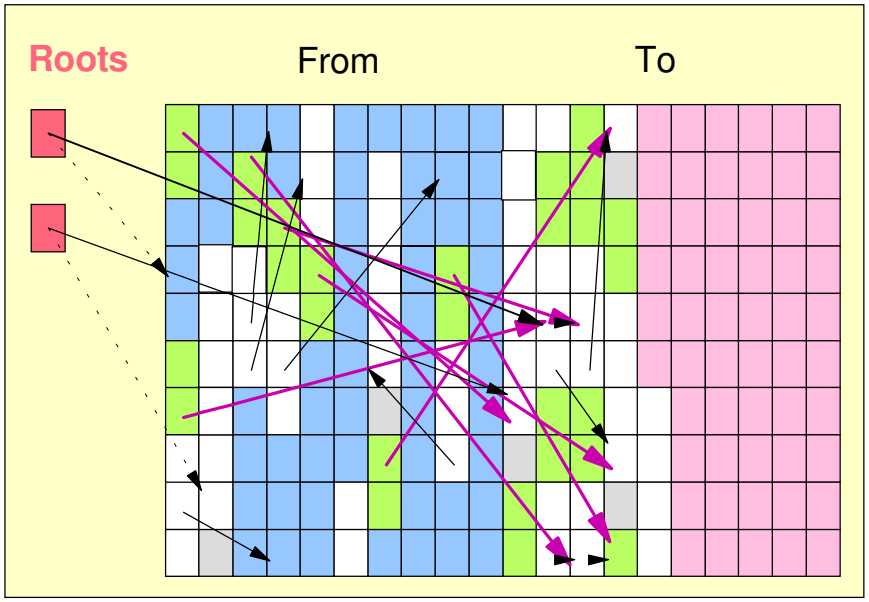
Pragmatically:
- Heavy weight synchronization expensive
- Almost all monitors are never acquired
- Contention rare even for acquired monitors
  - Library methods synchronize on thread local objects
- Shared monitors likely to be contended for again
- Recursion almost always less than 7 deep

Thin locks [Bacon, et. al., PLDI '98]
- Bits reserved in status word of object header
- Fat bit
  - 1, other bits: index a heavy weight lock
  - 0, other bits: thread id and recursion count
- Busy wait to promote a contended thin lock
- Heavy weight locks not demoted

Synchronization issues

- Concurrent programming is hard
  - Deadlock
  - Starvation
  - Lock contention
- Tip: avoid synchronization where possible
  - Needed to update global data
  - Needed to read globlal data
    - Unless update is atomic
- Course grain vs. fine grain
  - Course grain -- more contention
    - Less locking
    - Locks held longer
    - Less prone to deadlock
  - Fine grain -- more overhead
    - More locking
    - Total lock time shorter
- Sophisticated locking mechanisms
  - Example: read-locks and write-locks
  - Build on top of monitors

Two-space garbage collection



- Simple copying collector

```
put root references on a work queue
for each reference on the work queue
    if its object hasn't been forwarded
        forward it
        add its references to the work queue
update roots
swap From and To
```

Automatic memory management

Two-space garbage collectors
- Wastes half of memory
- Visits each live object
- Copies each live object

Generational collectors
- Ignores most old objects during minor collection
- Write barrier during mutation
- Short pause times for minor collections
- May take longer overall

Train collectors
- Breaks major collections into small pieces
- Short pause times
- May take longer overall

Tip: use the largest heap you can

Tip: avoid unnecessary object creation

Tip: don't hang on to objects unnecessarily

Exceptions

Tip: exceptions should be exceptional
- Should be free, if there are no exceptions
- Handling an exception is expensive
- Resist temptation to use for normal control flow

Precise exception semantics
- Everything before the exception happens
- Everything after the exception doesn't
- Prevents instruction reordering

Inhibited JIT compiler optimizations
- Loop transformation
- Common subexpression elimination
- Instruction scheduling
- (anything that reorders code)

Idiom: transform the program yourself

Language fix:
- Method modifier: imprecise exceptions
  - Same behavior, if no exception occurs
  - Otherwise, some exception must be reported

Dynamic class loading

- Bytecodes that may require class loading
  - invokestatic, invokevirtual, invokespecial, getstatic, putstatic, getfield, putfield
- A class is loaded, resolved, instantiated
  - At most once
  - First execution of a bytecode that mentions it
    - Easy for interpreters
    - JIT compiler defers class loading
- Deferred class loading
  - JIT compiler emits call to a conditional loader
  - Conditional loader loads a class (if it hasn't been)
- Backpatching calls to the conditional loader
  - Overwrite call site with compiled code
  - Tricky on an SMP
    - Idenpotent in case two thread execute at once
    - Also handles executing partially backpacthed code
    - Must synchronize to avoid loading a class twice
  - Some code space overhead

A portability issue

- Exceptions to "bit-wise reproducible results"
  - Environmental queries
    - java.lang.File.pathSeparatorChar
    - java.lang.System.currentTimeMillis()
  - Thread interleaving
- Java floating-point arithmetic:
  - IEEE 754 standard (sort of)
  - Forbids extended precision intermediate results
    - Defeats Intel and PowerPC floating-point hardware
    - Java 2 strict modifier --- hack for Intel
- PowerPC fma (multiply-add) instruction
  - More accurate than Java's required result
    - Intermediate result has 106 bit mantissa
  - Fully compliant with IEEE 754
  - Twice as fast as required fm and fa
- Language fix:
  - Make floating-point a third exception
  - Nobody should care about floating-point bits
    - (except a certified numerical analyst)

About JIT compilers

- Disadvantage
  - Traditional compiler's effort amortized
  - JIT effort repeated for each run
- Advantages
  - Target machine specific
  - Can exploit online performance information
    - Focuses on performance bottlenecks
- Implications
  - Techniques for amortizing JIT effort needed
    - Cache work in augmented classfiles
    - Or in JVM image files
  - Division of labor between Java and JIT compilers
    - Proofs are hard -- amortize (Java compiler)
      - Programmer input could be very helpful
    - Verification is easy -- repeat (JIT compiler)
  - JIT won't benefit performance programmer
    - Performance programmer codes for target
    - Hand tunes bottlenecks anyway
    - Relies on compiler for non-bottleneck code

Features missing from Java

- Templates
- Multidimensional arrays
- Complex arithmetic
- Support for parallelism
  - Parallel loop constructs (SPMD programming)
  - Distributed memory parallelism
- Memory hierarchy support

Templates

Java 1.1:

Collections always contain objects

```
Truck t;
Vector v;
...
v.addElement(t);              // no possible error
...
t = (Truck) v.elementAt(i); // error, if not a Truck
```

Java 2 collections have prettier syntax

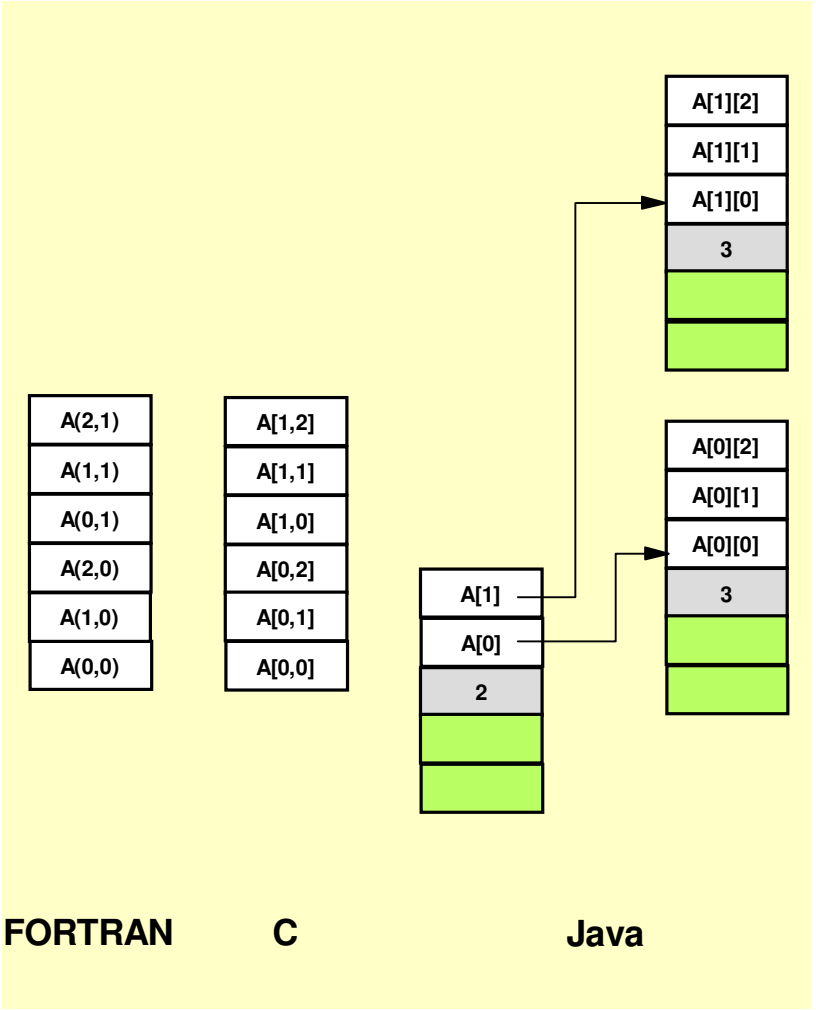
Still always contain objects

Templated version:

Typed collections

```
Truck t;
Vector<Truck> v;
...
v.addElement(t);              // error, if not a Truck
...
t = v.elementAt(i);           // no possible error
```

Multidimensional arrays



Complex arithmetic

```
class Complex {

  private double real;
  private double imaginary;

  Complex (double re, double im) {
    real = re;
    imaginary = im;
  }

  final double getReal () {return real;}
  final double getImaginary () {return imaginary;}

  static Complex add (Complex x, Complex y) {
     return new Complex (x.real + y.real,
                         x.imaginary + y.imaginary);
  }
  ...
}
```

Complex arithmetic continued

Complex.add[C[i,j],Complex.multiply(A[i,k],B[k,j])

Aesthetic overhead

- Ugly as sin (should be C[i,j] + A[i,k]*B[k,j])
- Java doesn't have operator overloading

Space overhead

- 2 word header per 4 words of data
- 1 word reference per 4 words of data
- An object creation per operation
  - JIT compiler will reclaim much of this

Computation overhead

- 1 level of indirection per access
  - JIT compiler will eliminate redundant indirections
- Method overhead per getter
  - JIT compiler will inline this
- An object creation per operation
  - JIT compiler will streamline this

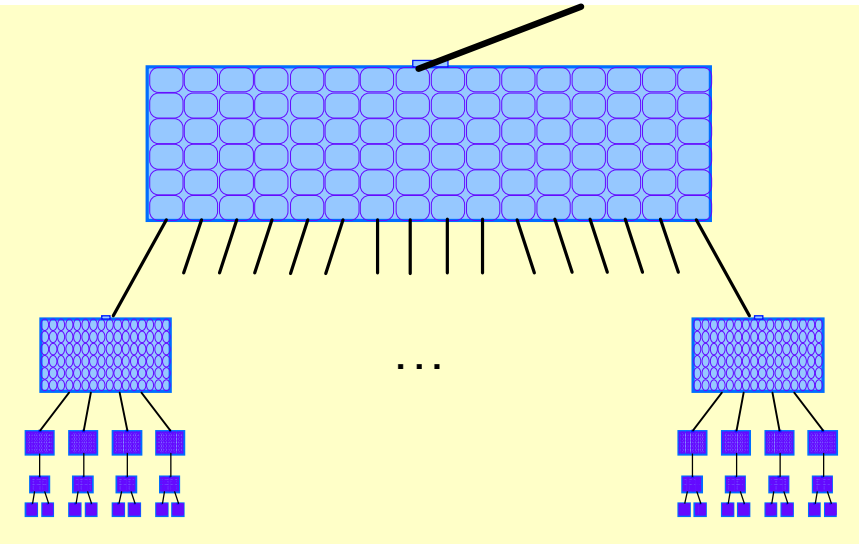
Performance programming

Methodology
- Model target architecture
- Estimate best possible performance
- Measure performance and identify bottlenecks
- Implement more appropriate mechanisms, repeat

Mantra: "keep processing elements busy"
- (doing useful work)
- Break problems into independent subproblems
- Distribute subproblems appropriately
- Locate data close to its processing element
- Overlap communication with computation

Processing elements can be
- Stages of a (vector or arithmetic) pipeline
- Functional units of a CPU
- Processors of a tightly coupled SMP
- Nodes of a distributed-memory supercomputer
- Heterogeneous computers on a network

Parallel Memory Hierarchy



PMH model of a computer
- Tree of *memory modules*
- Processors at the leaves
- Data moves in parallel along the branches
- Memory module parameters
  - Inverse transfer rate (cycles per block)
  - Block size
  - Block count
  - Number of children