Java: 1.5 and Beyond

Doug Lea
SUNY Oswego
dl@cs.oswego.edu
Outline

- J2SE 1.5
  - Language
    - Generics, Metadata, Syntax enhancements
  - Behind the scenes
    - Classfile updates, Memory model
  - Packages
    - Monitoring and management
    - Concurrency utilities
    - Other package enhancements
- Some possible next steps

credits: Thanks to Josh Bloch and Graham Hamilton for some slide material
Generics

- Parameterized classes and methods
  - Supports `List<E>`, `AtomicReference<T>`, etc
- An extension of GJ (Wadler et al)
  - Adds wildcards (“?”)
  - Supports `void add(List<? extends Number>)`
- A compile-time only language extension
  - Parameterized types are NOT macro expanded
  - Compiler can “erase” type parameters after checking them
  - Some usage limitations (arrays, `instanceof`) due to lack of guaranteed run-time type knowledge
- Improves safety and (usually) readability
  - Collections and related APIs are now generic
Using Generic Collections

Old

// Removes 4-letter words from c; elements must be strings
static void expurgate(Collection c) {
    for (Iterator i = c.iterator(); i.hasNext(); )
        if (((String) i.next()).length() == 4)
            i.remove();
}

New:

static void expurgate(Collection<String> c) {
    for (Iterator<String> i = c.iterator(); i.hasNext(); )
        if (i.next().length() == 4)
            i.remove();
}
Metadata Annotations

- Act as user-defined qualifiers
  - javadoc-like syntax
- Compiler knows nothing about semantics
  - Just places annotations in classfile
  - Accessible at runtime using extensions to java reflection API
- Manipulated by user-defined tools at any of several stages
  - IDEs, Compile-time preprocessors
  - J2EE code generation tools
  - Bytecode analysis and rewriting tools
  - Load-time tools
  - Runtime support packages
Annotation example

Using:

@Persistent
@UseCases({"payroll", "taxprep"})
@Author(@Name(first = "John", last = "Doe"))
class Employee {
    @Nonnull private String name;

    @Transactional(mode = Transactional.readOnly)
    public getName();
    // ...
}

Defining:

public @interface Persistent {}
Enhanced For Loops

❖ Old:

```java
void cancelAll(Collection c) {
    for (Iterator i = c.iterator(); i.hasNext(); ) {
        TimerTask tt = (TimerTask) i.next();
        tt.cancel();
    }
}
```

❖ New:

```java
void cancelAll(Collection<TimerTask> c) {
    for (TimerTask task : c)
        task.cancel();
}
```

❖ Also works for arrays:

```java
int sum(int[] a) {
    int result = 0;
    for (int i : a) result += i;
    return result;
}
```
Autoboxing/Unboxing

❖ Old:

```java
public class Freq {
    public static void main(String[] args) {
        Map m = new TreeMap();
        for (int i=0; i<args.length; i++) {
            Object freq = m.get(args[i]);
            m.put(args[i], (freq==null ? new Integer(1) : new Integer(((Integer)freq).intValue() + 1)));
        }
        System.out.println(m);
    }
}
```

❖ New:

```java
public class Freq {
    public static void main(String[] args) {
        Map<String, Integer> m = new TreeMap<String, Integer>();
        for (String word : args) {
            Integer freq = m.get(word);
            m.put(word, (freq == null ? 1 : freq + 1));
        }
        System.out.println(m);
    }
}
```
 Enums

 Old:

```java
class CardGame {
    public static final int SUIT_CLUBS = 0;
    public static final int SUIT_DIAMONDS = 1;
    public static final int SUIT_HEARTS = 2;
    public static final int SUIT SPADES = 3; // ...
}
```

 New:

```java
enum Suit { CLUBS, DIAMONDS, HEARTS, SPADES }
enum Rank { DEUCE, THREE, FOUR, FIVE, SIX, SEVEN,
             EIGHT, NINE, TEN, JACK, QUEEN, KING, ACE }
class Card { Suit suit; Rank rank; /*...*/ }

class CardGame {
    List<Card> deck = new ArrayList<Card>();
    void play() {
        for (Suit suit : Suit.values())
            for (Rank rank : Rank.values())
                deck.add(new Card(suit, rank));
        Collections.shuffle(deck);
    }
}
```
VarArgs

Old:

```java
public static String format(String pattern, Object[] arguments);

Object[] arguments = { new Integer(7), new Date(),
    "a disturbance in the Force"}
;

String result = MessageFormat.format(
    "At {1,time} on {1,date}, there was {2} on planet "+ 
    "{0,number,integer}.", arguments);
```

New:

```java
public static String format(String pattern, Object... arguments);

String result = MessageFormat.format(
    "At {1,time} on {1,date}, there was {2} on planet "+ "
    "{0,number,integer}".,
    7, new Date(), "a disturbance in the Force");
```
public class Physics {
    public static final double AVOGADROS_NUMBER = 6.02214199e23;
    public static final double BOLTZMANN_CONSTANT = 1.3806503e-23;
    ...
}

Old:

import org.iso.Physics;
...
double molecules = Physics.AVOGADROS_NUMBER * moles;
x = Math.cos(Math.PI * theta);

New:

import static org.iso.Physics.*;
import static java.Math.*;
...
double molecules = AVOGADROS_NUMBER * moles;
x = cos(PI * theta);
Classfile updates

- **Split Verification**
  - Compiler or tool includes verification hints in classfile
  - First used in J2ME to save time/space
  - Speeds up and simplifies runtime verifier
  - Can speed up generation and improve quality of native code

- **Support for Tiger language features**
  - New attributes and flags; remove some size constraints
  - No new bytecodes

- **Support for class file compression**
  - Pack200 format uses efficient Java-specific compression
Extended Unicode

- Unicode 3.1 adds extra characters
  - Some characters don't fit in 16 bits
- Java “char” remains 16 bits
  - Extended chars represented as pair of values encoded into a string
  - Library char processing APIs now support this
- Not pretty but necessary
  - and rarely used
Monitoring and Manageability

- Collects Reliability, Availability, Serviceability (RAS) features
- JVM Monitoring & Management API (JSR-174)
  - Low memory detection, heap sizes, gc info, threads, etc.
  - Allows access to internal VM status
  - Supports SNMP
- JMX Management (JSR-003, 160)
  - Support remote management using RMI
  - Works with existing J2EE application servers
- New JVM profiling API (JSR-163)
  - "C" level API capturing allocation, contention, etc events
  - Allows fine-grained performance analysis
- Improved diagnosability
  - Stack trace API, Error handling.
Other package updates

- JDBC extensions
  - CachedRowSet contains in-memory collection of rows
  - Resynchronizable; allows disconnected use
  - WebRowSet uses XML for data transfer
- Updated JAXP now in J2SE
  - Supports DOM, SAX, XML schema, etc
- BigDecimal enhancements
  - Additional methods, rounding and scaling modes
- Swing
  - New look and feel; performance enhancements
- Lots of other minor RFEs accepted into other packages
  - Example: `java.lang.StringBuilder` is a more efficient, unsynchronized alternative to `StringBuilder`
JSR-133 Memory Model

- A memory model specifies how threads and objects interact
  - Atomicity
    - Ensuring mutual exclusion for field updates
  - Visibility
    - Ensuring changes made in one thread are seen in other threads
  - Ordering
    - Ensuring that you aren’t surprised by the order in which statements are executed

- Original JLS spec was broken and impossible to understand
  - Included unwanted constraints on compilers and JVMs, omissions, inconsistencies

- The basic JSR-133 rules are easy. The formal spec is not.
JSR-133 Main Rule

Thread 1

- \( y = 1 \)
- lock \( M \)
- \( x = 1 \)
- unlock \( M \)

Everything before the unlock on \( M \) ...

Thread 2

- lock \( M \)
- \( i = x \)
- unlock \( M \)
- \( j = y \)

... visible to everything after the lock on \( M \)
Additional JSR-133 Rules

- Variants of lock rule apply to volatile fields and thread control
  - Writing a volatile has same basic memory effects as unlock
  - Reading a volatile has same basic memory effects as lock
  - Similarly for thread start and termination
  - Details differ from locks in minor ways

- Final fields
  - All threads will read the final value so long as it is guaranteed to be assigned before the object could be made visible to other threads. So DON'T write:

    ```java
    class Stupid implements Runnable {
        final int id;
        Stupid(int i) { new Thread(this).start(); id = i; }
        public void run() { System.out.println(id); }
    }
    ```

- Extremely weak rules for unsynchronized, non-volatile, non-final reads and writes
  - type-safe, not-out-of-thin-air, but can be reordered, invisible
New Concurrency Utilities

- New package `java.util.concurrent`
- Queue framework
  - Queues & blocking queues
- Other concurrent collections
  - List, Set, Map implementations geared for concurrent use
- Executor framework
  - ThreadPoolExecutors, Futures
- Lock framework (subpackage `java.util.concurrent.locks`)
  - Conditions & ReadWriteLocks
- Synchronizers
  - Semaphores, Barriers, Exchangers, CountDownLatches
- Atomic variables (subpackage `java.util.concurrent.atomic`)
  - JVM support for compareAndSet operations
Main JSR166 components

- **Lock**
  - `void lock()`
  - `void unlock()`
  - `boolean trylock()`
  - `newCondition()`

- **Condition**
  - `void await()`
  - `void signal()`
  - `...`

- **Collection<E>**
  - `boolean add(E x)`
  - `E poll()`
  - `...`

- **Queue<E>**
  - `void put(E x)`
  - `E take();`
  - `...`

- **Executor**
  - `void execute(Runnable r)`

- **ReadWriteLock**

- **ReentrantLock**

- **ArrayBQ**

- **LinkedQ**

- **BlockedQueue<E>**
  - `void put(E x)`
  - `E take();`
  - `...`

- **ThreadPoolExecutor**
  - `T get()`
  - `boolean cancel()`
  - `...`

- **ScheduledExecutor**

- **Future<T>**
  - `T get()`
  - `boolean cancel()`
  - `...`

- **Semaphore**

- **CyclicBarrier**

- **AtomicInteger**

- **locks**
class LoggedService { // ...
    final BlockingQueue msgQ = new LinkedBlockingQueue();
    public void serve() throws InterruptedException {
        // ... perform service ...
        String status = ... ;
        msgQ.put(status);
    }
    public LoggedService() { // start background thread
        Runnable logr = new Runnable() {
            public void run() {
                try {
                    for(;;)
                        System.out.println(msgQ.take());
                } catch(InterruptedException ie) {} };
        Executors.newSingleThreadExecutor().execute(logr);
    }
}
Executor Example

class NetworkService {

    public static void main(String[] args) {
        Executor pool = Executors.newFixedThreadPool(7);
        try {
            ServerSocket socket = new ServerSocket(9999);

            for (;;) {
                final Socket connection = socket.accept();
                pool.execute(
                    new Runnable() {
                        public void run() {
                            new Handler().process(connection);
                        }
                    }));
            }

        } catch(Exception e) { } // die
    }

    class Handler { void process(Socket s); }
}
Future Example

class ImageRenderer { Image render(byte[] raw); }

class App { // ...
   Executor exec = ...;  // any executor
   ImageRenderer renderer = new ImageRenderer();

   public void display(final byte[] rawimage) {
      try {
         Future<Image> image = Executors.invoke(exec, new Callable()
            public Object call() {
               return renderer.render(rawImage);
            }});

            drawBorders(); // do other things while executing
            drawCaption();

            drawImage(image.get()); // use future
      }
      catch (Exception ex) {
         cleanup();
         return;
      }
   }
}
class Cell {
    private long val;
    private final Lock mutex = new ReentrantLock();

    void swapVal(Cell other) {
        if (this == other) return; // alias check
        for (;;) {
            mutex.lock();
            try {
                if (other.mutex.tryLock()) {
                    try {
                        long t = val;
                        val = other.val;
                        other.val = t;
                        return;
                    } finally {
                        other.mutex.unlock();
                    }
                }
            } finally {
                mutex.unlock();
            }
            Thread.sleep(100); // heuristic retry interval
        }
    }
}
Bounded Buffer with Conditions

class BoundedBuffer {
    Lock lock = new ReentrantLock();
    Condition notFull = lock.newCondition();
    Condition notEmpty = lock.newCondition();
    Object[] items = new Object[100];
    int putptr, takeptr, count;

    public void put(Object x) throws IE {
        lock.lock(); try {
            while (count == items.length) notFull.await();
            items[putptr] = x;
            if (++putptr == items.length) putptr = 0;
            ++count;
            notEmpty.signal();
        } finally { lock.unlock(); }
    }

    public Object take() throws IE {
        lock.lock(); try {
            while (count == 0) notEmpty.await();
            Object x = items[takeptr];
            if (++takeptr == items.length) takeptr = 0;
            --count;
            notFull.signal();
            return x;
        } finally { lock.unlock(); }
    }
}
Semaphore Example

class ResourcePool {
    final Semaphore available = new Semaphore(N);
    Object[] items = ... ;

    Object getItem() throws InterruptedException {
        available.acquire();
        return getNextAvailableItem();
    }

    void putItem(Object x) {
        if (markAsUnused(x))
            available.release();
    }

    private Object getNextAvailableItem();
    private boolean markAsUnused(Object x);
}
class Solver { // Code sketch
    void solve(final Problem p, int nThreads) {

        final CyclicBarrier barrier = new CyclicBarrier(nThreads,
                new Runnable() {
                    public void run() { p.checkConvergence(); }
                });

        for (int i = 0; i < nThreads; ++i) {
            final int id = i;
            Runnable worker = new Runnable() {
                final Segment segment = p.createSegment(id);
                public void run() {
                    try {
                        while (!p.converged()) {
                            segment.update();
                            barrier.await();
                        }
                    } catch (Exception e) { return; }
                }
            };
            new Thread(worker).start();
        }
    }
}
class Random {
    // snippets
    private final AtomicLong seed;
    Random(long s) { seed = new AtomicLong(s); }

    private long next(){
        long oldseed, nextseed;
        for(;;) {
            oldseed = seed.get();
            nextseed = oldseed * ... + ...;
            if (seed.compareAndSet(oldseed,nextseed))
                return oldseed;
        }
    }
}
Building new synchronizers

class FIFOMutex {
    AtomicBoolean locked = new AtomicBoolean();
    Queue<Thread> waiters = new ConcurrentLinkedQueue<Thread>();

    void lock() {
        Thread current = Thread.currentThread();
        waiters.add(current);
        while (waiters.peek() != current || !locked.compareAndSet(false, true))
            LockSupport.park();
        waiters.remove();
    }

    void unlock() {
        locked.set(false);
        LockSupport.unpark(waiters.peek());
    }

    // ... }
}
Beyond 1.5

- Very likely -- existing in-process JSRs
  - Isolates (illustrated next)
  - Plus follow-on resource constraint API
  - Additional IO extensions
    - FileSystems, asynch IO, completions
  - High-performance numerics and Linear Algebra support
- Possible -- discussions but no JSRs yet
  - Aspect-oriented programming support
  - Tighter integration of enterprise and scripting support
- Further out
  - Native XML data types?
  - Transactional concurrency support?
Overview of Isolates

Isolate *noun*. pronunciation: *isolet*. 1. A thing that has been isolated, as by geographic, ecologic or social barriers - *American Heritage Dictionary*

- Outline
  - Motivation
  - Some design and implementation issues
  - API overview and code examples

- Status
  - At public review draft in JSR-121.
  - NOT scheduled for inclusion in JDK1.5, but in some future release.
  - J2ME versions will probably appear sooner.
Aggregates vs Isolates vs Threads

- Bytecodes
- Exec code
- Class reps
- Statics, heap

An isolate acts as a separate logical virtual machine.

Possibly shared run-time data

RMI etc

Each isolate and aggregate can have OS resources and services.
Three Implementation Styles

- **One Isolate per OS process**
  - Internal sharing via OS-level shared memory, comms via IPC
    - class representations, bytecodes, compiled code, immutable statics, other internal data structures

- **All Isolates in one OS address space / process managed by aggregate**
  - Isolates still get own versions of all statics/globals
    - including AWT thread, shutdown hooks, ...

- **LAN Cluster JVMs**
  - Isolates on different machines under a common administrative domain. *NOT* a substitute for RMI
    - Little or no internal sharing
Main Classes

- **public final class Isolate**
  - Create with name of class with a "main", arguments to main, plus optional standard IO bindings, classpath, security, system property and other context settings.
  - Methods to start, stop, and terminate created isolate
  - Event-based monitoring of life cycle events

- **public abstract class Link**
  - A pipe-like data channel to another isolate
    - byte arrays, ByteBuffers, Strings and serializable types
    - SocketChannels, FileChannels and other IO types (Descriptor Bearing Doobers, aka DBDs)
    - Isolates, Links
void runProgram(String classname,
    String[] args) {
    try {
        new Isolate(classname, args).start();
    }
    catch (SecurityException se) { ... }
    catch (Exception other) { ... }
}
class Runner {
    LinkMessageDispatcher d = new LinkMessageDispatcher();
    
    LinkMessageDispatcher.Listener l =
        new LinkMessageDispatcher.Listener() {
            public void messageReceived
                (IsolateMessageDispatcher d, Link l, LinkMessage m){
                IsolateEvent e = m.getEvent();
                System.out.println("State change"+ e.getType());
            }
        };

    void runStarlet(...) throws ... {
        TransientPreferences ctx = new TransientPreferences();
        ctx.node("java.properties").put("java.class.path", ...);
        
        IsolateMessage stdIn =
            IsolateMessage.
                newFileInputStreamMessage(new FileInputStream(...));
        
        Isolate p = new Isolate(..., ctx, stdIn, ...);
        d.add(p.newEventLink(Isolate.currentIsolate()), l);
        p.start();
    }
void appRunner() throws ... {
    Isolate child = new Isolate("Child", ...);
    Link toChild = Link.newLink(Isolate.currentIsolate(), child);
    Link fromChild = Link.newLink(child, Isolate.currentIsolate());
    app.start(new IsolateMessage[] {
        IsolateMessage.newLinkMessage(toChild),
        IsolateMessage.newLinkMessage(fromChild) });
    toChild.send(IsolateMessage.newStringMessage("hi"));
    String reply = fromChild.receive().getString();
    System.out.println(reply);
    child.exit(0);
    Thread.sleep(10 * 1000);
    if (!app.isTerminated()) app.halt(1);
}

class Child {
    public static void main(...) {
        Link fromParent = Isolate.currentIsolateStartMessages()[0];
        Link toParent = Isolate.currentIsolateStartMessages()[1];
        String hi = fromParent.receive().getString();
        toParent.send(IsolateMessage.newStringMessage("bye"));
        System.exit(0);
    }
}
Target Usage Patterns

- Minimizing startup time and footprint
  - User-level "java" program, web-start, etc can start JVM if not already present then fork Isolate
  - OS can start JVM at boot time to run daemons

- Partitioning applications
  - Contained applications (*lets)
    - Applets, Servlets, Xlets, etc can run as Isolates
    - Container utility services can run as Isolates
  - Service Handler Forks
    - ServerSocket.accept can launch handler for new client as Isolate
    - Pools of "warm" Isolates
More Usage Patterns

- Parallel execution on cluster JVMs
  - Java analogs of Beowulf clusters
    - Can use MPI over Links
  - Need partitioning and load-balancing frameworks
- Fault-tolerance
  - Fault detection and re-activation frameworks
  - Redundancy via multiple Isolates
- CSP style programming
  - Always use Isolates instead of Threads
  - Practically suitable only for coarse-grained designs