Java SE: Where We've Been, Where We're Going

Alex Buckley
Spec Lead, Java Language & VM

November 2011

The following is intended to outline our general product direction. It is intended for information purposes only, and may not be incorporated into any contract. It is not a commitment to deliver any material, code, or functionality, and should not be relied upon in making purchasing decisions.
The development, release, and timing of any features or functionality described for Oracle’s products remains at the sole discretion of Oracle.

Executive Summary

- Java SE 7 released Summer 2011
- Java SE 8 targeted for Summer 2013
- More transparent JCP and OpenJDK

Audience survey

- Downloaded JDK7 GA or 7u1?
  - JDK7 Mac OS X Port – Developer Preview Release
    http://jdk7.java.net/macportpreview/
- Subscribed to coin-dev @ OpenJDK ?
- Subscribed to lambda-dev @ OpenJDK ?
- OpenJDK contributor?
  - http://openjdk.java.net/contribute/

Java SE 7

Features in Java SE 7

- Small language changes (JSR 334)
- Dynamic language support (JSR 292)
- Fork/Join Framework (JSR 166y)
- ClassLoader, Swing, and Java2D improvements
- Unicode 6.0 and improved regex support
- HotSpot: G1 GC, Tiered Compilation, Compressed OOPs
- Specs: JLS7 and JVMS7 on jcp.org now, books coming soon

Small language changes in Java SE 7

- Consistency and clarity
  - Numeric literals
  - Strings in switch
- Ease of use for generics
  - Varargs warnings
  - Diamond operator
- Concise error handling
  - Multi-catch and precise rethrow
  - try-with-resources

NetBeans 7.0

Eclipse 3.7.1

IntelliJ IDEA 10.5

Consistency and clarity

- Underscores in numeric literals
  - `650_506_7000`
  - `0xCAFE_BABE`
- Binary literals
  - `0b0010_1100`
- Strings in switch
  - ```
    int daysInMonth(String month, int year) {
        switch (month) {
            case "April":
            case "June":
            case "September":
            case "November":
                return 30;
    ```
    ```

@SafeVarargs

- Unchecked warnings indicate possible heap pollution
- Prior to Java SE 7, calling certain varargs methods in the platform resulted in unchecked warnings
  - <T> List<T> Arrays.asList(T… a)
  - <T> boolean Collections.addAll(Collection<? super T> c, T… elements)
  - <E extends Enum<E>> EnumSet<E> EnumSet.of(E first, E… rest)
  - void javax.swing.SwingWorker.publish(V… chunks)
  - Warnings were due to a poor interaction of arrays and generics, but nothing bad actually happens
- In Java SE 7, @SafeVarargs suppresses these warnings

Diamond

- `List<String> ls = new ArrayList<>();`
- `List<List<String>> ls = new ArrayList<>();`
- `List<? extends String> ls = new ArrayList<>();`
- `Map<? extends Number, ? extends String> m = new HashMap<>();`

- Types of constructor arguments are taken into account
- Important to keep static type information on the left-hand side
- <> turns out to be very useful for using lambda-fied libraries

Concise error handling

```
void exampleMethod(Future future) throws
    InterruptedException, ExecutionException, TimeoutException
{
    Object result = future.get(5, SECONDS);

    // Future.get(long, TimeUnit) is declared to throw
    // InterruptedException, ExecutionException, TimeoutException
}
```

- How would we catch, clean up, and rethrow?

Multiple catch clauses in Java SE 6

```
void exampleMethod(Future future) throws
    InterruptedException, ExecutionException, TimeoutException
{
    try {
        Object result = future.get(5, SECONDS);
    } catch (InterruptedException ex) {
        cleanup();
        throw ex;
    } catch (ExecutionException ex) {
        cleanup();
        throw ex;
    } catch (TimeoutException ex) {
        cleanup();
        throw ex;
    }
}
```

Multi-catch in Java SE 7

```
void exampleMethod(Future future) throws
    InterruptedException, ExecutionException, TimeoutException
{
    try {
        Object result = future.get(5, SECONDS);
    } catch (InterruptedException |
             ExecutionException |
             TimeoutException ex) {
        cleanup();
        throw ex;
    }
}
```

Precise rethrow instead of multi-catch

```
void exampleMethod(Future future) throws
    InterruptedException, ExecutionException, TimeoutException
{
    try {
        Object result = future.get(5, SECONDS);
    } catch (Exception ex) {
        cleanup();
        throw ex;
    }
}
```

Type of `ex` *is inferred as:*
`InterruptedException | ExecutionException | TimeoutException`

- An entirely new exception handling idiom!
- The exception variable must be final or effectively final

try-with-resources

- Aims to avoid leaks of external resources

- You type this:

```
try (Resource r = ...) {
    ...
} catch (Exception e) {
    ...
} finally {
    ...
}
```

- Compiler generates this:

```
try {
    Resource r = null;
    try {
        r = ...;
        ...
    } finally {
        if (r != null) r.close();
    }
} catch (Exception e) { ... }
} finally { ... }
```

- Allows initialization of one or more resource variables

- Allows a try block with no catch or finally blocks

Example

Applying Java SE 7 features in the JDK

- 7018392 “update URLJarFile.java to use try-with-resources”
- Example method: URLJarFile.retrieve()
  - Takes a URL
  - Opens it
  - Downloads content into a temporary file
  - Creates and returns a JarFile instance backed by the temp file
  - Removes temp file if there was an error
  - Mustn’t leak anything
  - Must handle all errors without loss of information

Original code (Part 1 of 3)

```
JarFile retrieve(URL url) throws IOException {
    InputStream  in       = url.openStream();
    OutputStream out      = null;
    File         tmpFile = null;
    try {
        tmpFile = File.createTempFile("jar_cache", null);
        out     = new FileOutputStream(tmpFile);
        ...
```

Original code (Part 2 of 3)

```
...
int read = 0;
byte[] buf = new byte[BUF_SIZE];
while ((read = in.read(buf)) != -1) {
    out.write(buf, 0, read);
}
out.close();
out = null;
return new JarFile(tmpFile);
...
```

Original code (Part 3 of 3)

```
        ...
    } catch (IOException e) {
        if (tmpFile != null) {
            tmpFile.delete();
        }
        throw e;
    } finally {
        if (in != null) {
            in.close();
        }
        if (out != null) {
            out.close();
        }
    }
}
```

Original code (Part 3 of 3)

```
        ...
    } catch (IOException e) {
        if (tmpFile != null) {
            tmpFile.delete();
        }
        throw e;
    } finally {
        if (in != null) {
            in.close();
        }
        if (out != null) {
            out.close();
        }
    }
}
```

Bug: If non-IOException is thrown, temp file will not be deleted

Bug: If in.close() fails, out will remain open

Problem: Suppressed exceptions are mishandled

Problem: Uses null references to keep track of what needs cleanup

Pathology: trying to do too much in a single try/catch/finally block

Alternative (nested try-statements) is arguably worse

Improvement #1: NIO2

- Allows us to replace this…

```
out = new FileOutputStream(tmpFile);
int read = 0;
byte[] buf = new byte[BUF_SIZE];
while ((read = in.read(buf)) != -1) {
    out.write(buf, 0, read);
}
out.close();
out = null;
return new JarFile(tmpFile);
```

- With…

```
import java.nio.file.*;
Files.copy(in, tmpFile, REPLACE_EXISTING);
return new JarFile(tmpFile.toFile());
```

Improvement #2: try-with-resources

- Allows us to replace this…

```
InputStream in = url.openStream();
try { … }
catch (…) { … }
finally {
    if (in != null) {
        in.close();
    }
}
```

- With…

```
try (InputStream in = url.openStream()) { … }
catch (…) { … }
```

Improvement #2: try-with-resources

```
JarFile retrieve(URL url) throws IOException {
  InputStream in = url.openStream();
  Path tmpFile = null;
  try {
    tmpFile =
      Files.createTempFile("jar_cache", null);
    Files.copy(in, tmpFile, REPLACE_EXISTING);
    return new JarFile(tmpFile.toFile());
  } catch (IOException e) {
    if (tmpFile != null) {
      Files.delete(tmpFile);
    }
    throw e;
  } finally {
    if (in != null) {
      in.close();
    }
  }
}
```

```
JarFile retrieve(URL url) throws IOException {
  Path tmpFile = null;
  try (InputStream in = url.openStream()) {
    tmpFile =
      Files.createTempFile("jar_cache", null);
    Files.copy(in, tmpFile, REPLACE_EXISTING);
    return new JarFile(tmpFile.toFile());
  } catch (IOException e) {
    if (tmpFile != null) {
      Files.delete(tmpFile);
    }
    throw e;
  }
}
```

Improvement #3: Drop the null sentinels

- The `in` and `out` variables are now handled for us
  - `in` is a resource variable, `out` is “buried” inside Files.copy()
- The only thing to clean up after an IOException is the temp file, so we can create it first and drop its null sentinel

```
JarFile retrieve(URL url) throws IOException {
    Path tmpFile = Files.createTempFile("jar_cache", null);
    try (InputStream in = url.openStream()) {
        Files.copy(in, tmpFile, REPLACE_EXISTING);
        return new JarFile(tmpFile.toFile());
    } catch (IOException e) {
        Files.delete(tmpFile);
        throw e;
    }
}
```

Improvement #4: Precise rethrow

- We want to delete the temp file on any error

- Precise rethrow to the rescue!

```
JarFile retrieve(URL url) throws IOException {
    Path tmpFile = Files.createTempFile("jar_cache", null);
    try (InputStream in = url.openStream()) {
        Files.copy(in, tmpFile, REPLACE_EXISTING);
        return new JarFile(tmpFile.toFile());
    } catch (~~IOException~~Throwable e) {
        Files.delete(tmpFile);
        throw e;
    }
}
```

Improvement #5: Suppressed exceptions

- An exception from Files.delete() in the catch clause would suppress the exception thrown by the try block
- Better to catch an exception from Files.delete() and add it to the *suppressed exception list* of the try block's exception
  - Throwable.addSuppressed()
  - Throwable.getSuppressed()
  - Suppressed exceptions are orthogonal to an exception's *cause*
- More verbose, but closes a big gap in exception handling

Improvement #5: Suppressed exceptions

```
JarFile retrieve(URL url) throws IOException {
    Path tmpFile = Files.createTempFile("jar_cache", null);
    try (InputStream in = url.openStream()) {
        Files.copy(in, tmpFile, REPLACE_EXISTING);
        return new JarFile(tmpFile.toFile());
    } catch (Throwable e) {
        try {
            Files.delete(tmpFile);
        } catch (Throwable e2) {
            e.addSuppressed(e2);
        }
        throw e;
    }
}
```

Before and After

```java
JarFile retrieve(URL url) throws IOException {
    InputStream in = url.openStream();
    OutputStream out = null;
    File tmpFile = null;
    try {
        tmpFile = Files.createTempFile("jar_cache", null);
        out = new FileOutputStream(tmpFile);
        int read = 0;
        byte[] buf = new byte[BUF_SIZE];
        while ((read = in.read(buf)) != -1) {
            out.write(buf, 0, read);
        }
        out.close();
        out = null;
        return new JarFile(tmpFile);
    } catch (IOException e) {
        if (tmpFile != null) {
            tmpFile.delete();
        }
        throw e;
    } finally {
        if (in != null) {
            in.close();
        }
        if (out != null) {
            out.close();
        }
    }
}
```

```java
JarFile retrieve(URL url) throws IOException {
    Path tmpFile = Files.createTempFile("jar_cache", null);
    try (InputStream in = url.openStream()) {
        Files.copy(in, tmpFile, REPLACE_EXISTING);
        return new JarFile(tmpFile.toFile());
    } catch (Throwable e) {
        try {
            Files.delete(tmpFile);
        } catch (Throwable e2) {
            e.addSuppressed(e2);
        }
      throw e;
    }
}
```

Proving language changes on the JDK

- Wrote annotation processors to detect idiomatic *types*
  - Types to be retrofitted as `AutoCloseable`
  - E.g. JDBC's `Connection`, `ResultSet`, `Statement`
  - Methods and constructors to be annotated with @SafeVarargs
- Extended javac to detect idiomatic *statements*
  - Instance creation expressions that could use diamond
  - try/catch/finally blocks that could use try-with-resources
  - Null handling of try-with-resources was changed in part from experiences applying it in JDK code
- Seven-dimensional test case generation for @SafeVarargs

Java SE 8

Platform evolution for multicore

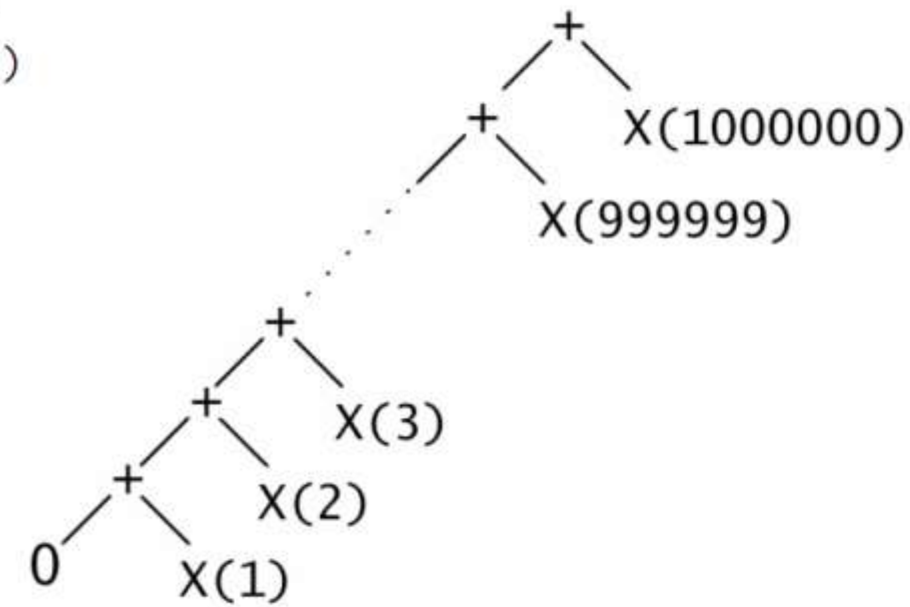
- Multicore hardware is now the default
- Our goal is to exploit it *gracefully*, with a small syntactic and semantic gap between serial and parallel code
- Better libraries are the key to graceful parallelization
  - Libraries can hide domain-specific concerns (e.g. task scheduling)
  - Libraries are easier to evolve than languages
  - Fork/Join in Java SE 7 is a good start, but we need more
- Libraries need some help from the language
  - A concise “code as data” construct
  - In turn, the language may need help from the VM

Inspiration

Inspiration

Sequential Computation Tree

```
SUM = 0
DO I = 1, 1000000
  SUM = SUM + X(I)
END DO
```

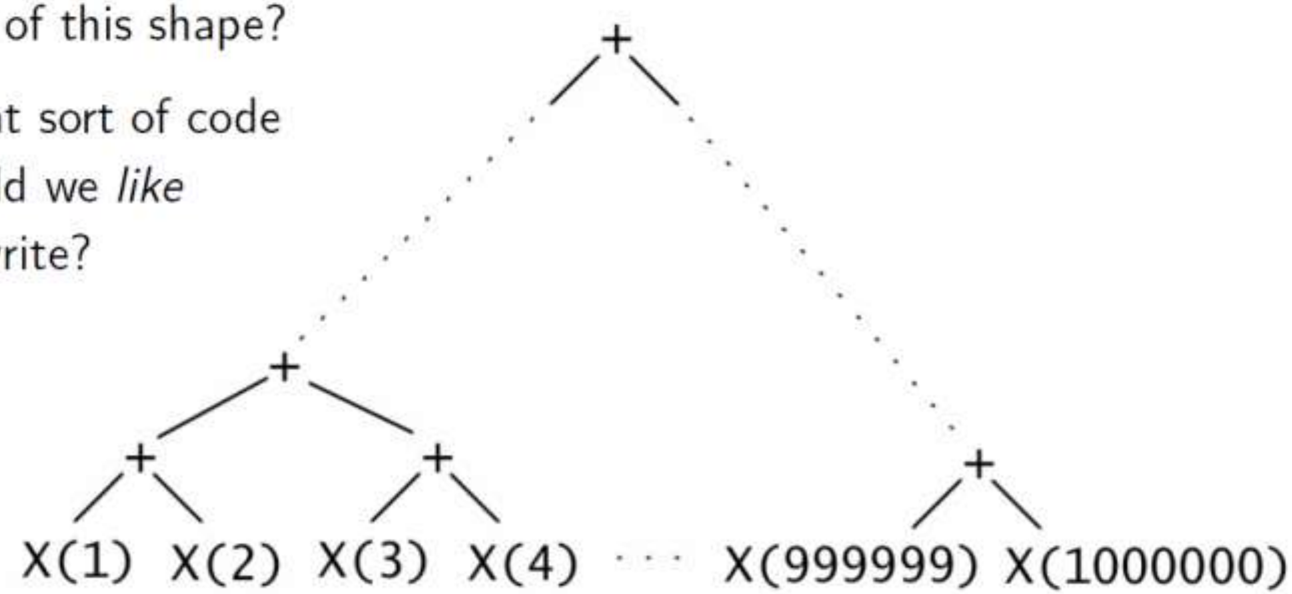


Inspiration

Parallel Computation Tree

What sort of code
should we write
to get a computation
tree of this shape?

What sort of code
would we *like*
to write?



Inspiration

We Need a New Mindset

- DO loops are so 1950s! (Literally: Fortran is now 50 years old.)
- So are linear linked lists! (Literally: Lisp is now 50 years old.)
- Java™-style iterators are **so** last millennium!
- Even arrays are suspect! (Constant-time indexing is an illusion.)
- As soon as you say "first, `SUM = 0`" you are hosed.
- Accumulators are BAD. They encourage sequential dependence and tempt you to use nonassociative updates.
- If you say, "process subproblems in order," you lose.
- The great tricks of the sequential past WON'T WORK.
- The programming idioms that have become second nature to us as everyday tools for the last 50 years WON'T WORK.

Java encourages external iteration

```
List<Student> students = ...
double highestScore = 0.0;
for (Student s : students) {
  if (s.getGradYear() == 2011) {
    if (s.getScore() > highestScore) {
      highestScore = s.getScore();
    }
  }
}
```

- Client controls iteration
- Inherently serial: iterates from beginning to end
- Not thread-safe because business logic is stateful (mutable accumulator variable)

Internal iteration with inner classes

```
SomeCoolList<Student> students = ...
double highestScore =
  students.filter(
    new Predicate<Student>() {
      public boolean op(Student s) {
        return s.getGradYear() == 2011;
      }
    }
  ).map(
    new Mapper<Student,Double>() {
      public Double extract(Student s) {
        return s.getScore();
      }
    }
  ).max();
```

- Library controls iteration and accumulation
- Not inherently serial: traversal may be done in parallel
- Traversal may be done lazily, so one pass not three
- Thread-safe because business logic is stateless
- But ... ugly, and new libraries

Internal iteration with lambda expressions

```
SomeCoolList<Student> students = ...
double highestScore =
  students.filter(Student s -> s.getGradYear() == 2011)
          .map(Student s -> s.getScore())
          .max();
```

- More readable
- More abstract
- Less error-prone
- No reliance on mutable state
- Easier to make parallel
- Something Must Be Done About The Libraries

Lambda expressions

- A lambda expression is an anonymous function

```
Comparator<String> c = new Comparator<String>() {
    public boolean compare(String x, String y) {
        return x.length() - y.length();
    }
};
boolean x = c.compare(“hello”, “goodbye”);

Comparator<String> c = (String x, String y) -> x.length() - y.length();
boolean x = c.compare(“hello”, “goodbye”);
```

- Do not assume implementation is with anonymous classes!

Lambda expression typing

- The type of a lambda expression is a *functional interface*
- “An interface with one method”

```
interface Runnable         { void run(); }
interface ActionListener { void actionPerformed(…); }
interface Comparator<T>  { boolean compare(T x, T y); }
interface FileFilter      { boolean accept(File x); }
interface Callable<T>     { T call(); }
interface DirectoryStream.Filter<T> { boolean accept(T x); }
```

- We’ve used interfaces like this to describe functions forever
  - No need for A=>B function types which would bifurcate the libraries

Target typing

- The functional interface of a lambda expression is inferred from the assignment or method invocation context

```
Collections.sort(lst, new Comparator<String>() {
    public boolean compare(String x, String y) {
        return x.length() - y.length();
    }
});

Collections.sort(lst, (String x, String y) -> x.length() - y.length());
```

Variable capture

- Can refer to any *effectively final* variables in enclosing scope
  - Effectively final means the variable meets the requirements for a final variable (e.g. assigned to once), even if not declared final
  - A kind of type inference, introduced for precise rethrow in Java SE 7

```
void expire(File root, long before) {
  … root.listFiles((File p) -> p.lastModified() <= before)
}
```

Lexical scoping

- Meaning of a name is the same inside the lambda as outside
- Meaning of `this` is the enclosing object, not the lambda

```
class SessionManager {
  long before = ...;

  boolean check(long time, long expiry) { … }

  void expire(File root) {
    … root.listFiles((File p) -> check(p.lastModified(), this.before))
  }
}
```

Improved type inference

- Parameter types in a lambda expression are inferred based on the method signature in the target functional interface
  - Below, the lambda expression will be inferred as `Comparator<String>`
  - The formal parameters must therefore each be `String`

```
Collections.sort(lst, (String x, String y) -> x.length() - y.length());

Collections.sort(lst, (x, y) -> x.length() - y.length());
```

- Fully statically typed – no dynamic typing here
  - Builds on type inference for generic methods (SE 5.0) and diamond (SE 7)

Method references

- "A way to reuse a method as a lambda expression"

```
FileFilter x = new FileFilter() {
    public boolean accept(File f) {
        return f.canRead();
    }
}

FileFilter x = (File f) -> f.canRead();

FileFilter x = File::canRead;
// canRead is a function from File (receiver) to boolean
```

Putting it all together

- With some library improvements, we can make common tasks more expressive, reliable, and compact

```
Collections.sort(people, new Comparator<Person>() {
    public int compare(Person x, Person y) {
        return x.getLastName().compareTo(y.getLastName());
    }
});

Collections.sort(people,
  (Person x, Person y) -> x.getLastName().compareTo(y.getLastName()));
```

Putting it all together

- With some library improvements, we can make common tasks more expressive, reliable, and compact

```
Collections.sort(people, new Comparator<Person>() {
    public int compare(Person x, Person y) {
        return x.getLastName().compareTo(y.getLastName());
    }
});

Collections.sort(people, comparing(Person p -> p.getLastName()));
```

Putting it all together

- With some library improvements, we can make common tasks more expressive, reliable, and compact

```
Collections.sort(people, new Comparator<Person>() {
    public int compare(Person x, Person y) {
        return x.getLastName().compareTo(y.getLastName());
    }
});

Collections.sort(people, comparing(p -> p.getLastName()));
```

Putting it all together

- With some library improvements, we can make common tasks more expressive, reliable, and compact

```
Collections.sort(people, new Comparator<Person>() {
    public int compare(Person x, Person y) {
        return x.getLastName().compareTo(y.getLastName());
    }
});

Collections.sort(people, comparing(Person::getLastName));
```

Putting it all together

- With some library improvements, we can make common tasks more expressive, reliable, and compact

```
Collections.sort(people, new Comparator<Person>() {
    public int compare(Person x, Person y) {
        return x.getLastName().compareTo(y.getLastName());
    }
});

people.sort(comparing(Person::getLastName));
```

The real challenge: Library evolution

- If Java had lambdas on day 1, all APIs would look different
- Adding lambdas now makes aging APIs show their age more
- The most important APIs (Collections) are based on interfaces
  - Can’t add methods to interfaces without breaking <u>source</u> compatibility
  - But adding lambdas and not upgrading the APIs would be silly
- We need a mechanism for API evolution
  - Burden of API evolution should fall on *implementers*, not users
  - A solution that requires users to permanently cruft up their code to use new methods is unacceptable

Library-based options for API evolution

- Add more static methods to java.util.Collections helper class
  - `Collections.map`
- Add aggregate methods to existing collection implementations
  - `ArrayList.map`
- Add new collection implementations with aggregate methods
  - `ParallelArrayList.map`
- Add new subinterfaces to existing collection interfaces
  - `interface ParallelList<T> extends List<T>`
- Introduce an entirely new set of collection interfaces

Language-based options for API evolution

- Static extension methods
  - What appears to be an instance method call is really a static method call
  - Client says: `import static Collections.sort(List);`
  - Then a call to `ls.sort()` would compile to `Collections.sort(ls)`

  - Simple to implement, and “proven” in C#
  - Programmer must reason about extension in scope for each call
  - Classes don’t know about their extension methods, so cannot provide better implementations
  - Poor interaction with instance methods of the same name
  - Poor interaction with reflection

API evolution is a cross-cutting concern

- Pure library changes don’t help, nor most language proposals
- Let’s revisit “Can’t add a method to an interface”
- This is a problem for implementers, not callers, but an interface has relatively few implementers compared to callers
- A new interface method often has an “obvious” implementation
- Let the interface provide a *default* implementation for a method
- Implementers may override it at their leisure

Virtual extension methods

- “Allow an interface method to have an implementation”

```
interface List<T> … {
    // existing methods, plus
    void sort(Comparator<? super T> cmp)
        default { Collections.sort(this, cmp); }
}
```

- From caller’s perspective, an ordinary interface method
  - Default is only invoked if the receiver class does not override the interface method
  - “If you cannot afford an implementation of `sort`, one will be provided for you at no charge”

Implications of virtual extension methods

- Java always had multiple inheritance of types
- Java now has multiple inheritance of behavior (not state)
- Multiple inheritance can be fairly benign
  - Invocation requires a unique, most-specific, default-providing interface
  - Some situations can be automatically resolved by “pruning”
  - Some situations can be detected at compile-time (under globally consistent compilation) and resolved automatically or manually
  - Only some situations (which imply inconsistent separate compilation) need to be handled at runtime

Implementation of virtual extension methods

- Many possible techniques
  - Inline default body into calling class at compile-time
  - Inject default body into calling class at class load-time
  - Invoke extension method via `invokedynamic`, and let bootstrap method resolve default
- Best technique is to view extension methods as a VM feature
  - Integrate extension methods into vtables at runtime
  - `invokeinterface` prefers declaration in class to declaration in interface
  - `invokeinterface` prefers declaration in a more specific interface than a less specific interface, and must detect ambiguity
- Non-Java languages benefit too

Application: Optional methods

- Virtual extension methods can reduce boilerplate
- Most implementations of `Iterator` don't provide a useful remove() method, so why make the implementer declare it?

```
interface Iterator<T> {
    boolean hasNext();
    T next();
    void remove()
        default { throw new UnsupportedOperationException(); }
}
```

Application: Retrofits

- JDK 1.0 had `Enumeration`, later replaced with `Iterator`
- APIs that returned `Enumeration` became second-class citizens
- Let's retrofit `Enumeration` to implement `Iterator`

```
interface Enumeration<E> extends Iterator<E> {
    boolean hasMoreElements();
    E nextElement();
    boolean hasNext()
        default { return hasMoreElements(); }
    E next()
        default { return nextElement(); }
    void remove()
        default { throw new UnsupportedOperationException(); }
}
```

Application: Simple extensions

```
interface Collection<E> {
    void removeAll(Predicate<? super E> p) default { ... };
}
interface List<E> {
    void sort(Comparator<? super E> c) default { ... };
}
interface Reader {
    void eachLine(Block<String> block) default { ... };
}
```

```
collection.removeAll(s -> s.length() > 20);
collection.removeAll(String::isEmpty);

list.sort(comparing(Person::getLastName).reverse());

reader.eachLine(s -> { System.out.println(s); });
```

Parallel Collections

Remember what we wanted to write?

```
List<Student> students = ...
double highestScore =
  students.filter(s -> s.getGradYear() == 2011)
          .map(s -> s.getScore())
          .reduce(0.0, Math::max);
```

- Need to add extension methods to `Collection` / `List`
- Many design choices
  - Eager v. lazy
  - In-place v. create-new
  - Serial v. parallel
  - Exactly where in the hierarchy to put new methods

Attractive target: Iterable

```
public interface Iterable<T> {
    Iterator<T>     iterator();
    boolean         isEmpty()                                default ...;
    void            forEach(Block<? super T> block)          default ...;
    Iterable<T>     filter(Predicate<? super T> predicate)   default ...;
    <U> Iterable<U> map(Mapper<? super T, ? extends U> mapper) default ...;
    T               reduce(T base, Operator<T> reducer)      default ...;
    Iterable<T>     sorted(Comparator<? super T> comp)       default ...;
    <C extends Collection<? super T>> C into(C collection)   default ...;
    // and more...
}
```

- All collections get these for free
- Defaults easily implemented in terms of iterator()
- Scala `Traversable`, Ruby `Enumerable`, .NET `IReadOnlyList`

Eager v. lazy

- New methods are a mix of lazy and eager
  - `filter`, `map`, `cumulate` – naturally lazy
  - `reduce`, `forEach`, `into` – naturally eager
- Many useful operations can be represented as pipelines of source – lazy … lazy – eager
  - collection – `filter` – `map` – `reduce`
  - array – `map` – `sorted` – `forEach`
- Laziness is mostly invisible
  - No new abstractions for LazyCollection, LazyList, etc

Serial v. parallel

- A collection knows how to operate on elements serially
  - By adding to `Iterable`, all `Collection` types now expose serial bulk data operations without changing implementations ☺
- Can we do the same for parallel operations?
  - i.e. what is the parallel equivalent of `Iterable`?
- Many problems yield to "divide-and-conquer" recursive decomposition
  - Break down problems into subproblems, solve in parallel, combine results
  - Break down subproblems recursively until small enough for serial solution
  - Embodied by the Fork/Join framework in Java SE 7

Parallel high score finder with Fork/Join

```
ForkJoinExecutor pool = new ForkJoinPool(nThreads);
ScoreFinder finder = new ScoreFinder(problem);
pool.invoke(finder);

class ScoreFinder extends RecursiveAction {
  private final ScoreProblem problem;
  double highestScore;

  protected void compute() {
    if (problem.size < THRESHOLD)
      highestScore = problem.solveSequentially();
    else {
      int m = problem.size / 2;
      ScoreFinder left, right;
      left = new ScoreFinder(
               problem.subproblem(0, m));
      right = new ScoreFinder(
                problem.subproblem(m, problem.size));
      forkJoin(left, right);
      highestScore = Math.max(left.highestScore,
                              right.highestScore);
    }
  }
}
```

```
class ScoreProblem {
  final List<Student> students;
  final int size;

  ScoreProblem(List<Student> ls) {
    this.students = ls;
    this.size = this.students.size();
  }

  public double solveSequentially() {
    double highestScore = 0.0;
    for (Student s : students) {
      if (s.gradYear == 2011) {
        if (s.score > highestScore) {
          highestScore = s.score;
        }
      }
    }
    return highestScore;
  }

  public ScoreProblem subproblem(int start, int end) {
    return new ScoreProblem(students.subList(start, end));
  }
}
```

Going parallel

- Fork/Join relies on the client to decompose the problem
  - Configure thread pool
  - Choose serial v. parallel threshold
  - Determine decomposition approach (2-way, 3-way, n-way…)
  - Fork/Join is really “parallelism assembly language”
- It’s more powerful (and object-oriented) for a collection to decompose itself
- `Iterable` embodies internal iteration, so let’s define a type to embody *parallel* internal iteration

Spliterable: the parallel version of Iterable

- A `Spliterable` can be decomposed into two smaller chunks
  - Most data structures (arrays, trees, maps) admit a natural means of subdividing themselves
  - Eventually, small chunks can be processed sequentially

```
public interface Spliterable<T> {
    Iterator<T>     iterator();
    Spliterable<T>  left();
    Spliterable<T>  right();
    Iterable<T>     sequential();
    // plus extension methods
}
```

```
public interface Iterable<T> {
    Iterator<T>     iterator();
    Spliterable<T>  parallel();
    // plus extension methods
}
```

Methods on Spliterable analogous to Iterable

```
public interface Spliterable<T> {
    Iterator<T>     iterator();
    Spliterable<T>  left();
    Spliterable<T>  right();
    Iterable<T>     sequential();

    boolean            isEmpty()                                    default ...;
    void               forEach(Block<? super T> block)              default ...;
    Spliterable<T>     filter(Predicate<? super T> predicate)       default ...;
    <U> Spliterable<U> map(Mapper<? super T, ? extends U> mapper) default ...;
    T                  reduce(T base, Operator<T> reducer)          default ...;
    Spliterable<T>     sorted(Comparator<? super T> comp)           default ...;
    <C extends Collection<? super T>> C into(C collection)          default ...;
    // and more...
}
```

Explicit but unobtrusive parallelism

```
List<Student> students = ...
double highestScore =
  students.parallel()
          .filter(s -> s.getGradYear() == 2011)
          .map(s -> s.getScore())
          .reduce(0.0, Math::max);
```

- Implementation fuses three operations into one parallel pass
- Big win for data locality
- Works on any data structure that knows how to subdivide itself

Example

Given a music library, get the set of albums for which at least one track is highly rated

```
class Library {
    Set<Album> albums;

    Set<Album> allAlbums() {
        return albums;
    }

    Set<Album> favoriteAlbums() {
        // TODO
    }
}
```

```
class Album {
    String      title;
    List<Track> tracks;
}

class Track {
    String title;
    String artist;
    int    rating;
}
```

Identifying a favorite album

```
// Set hasFavorite to true if some track in album a is rated >= 4

boolean hasFavorite = false;
for (Track t : a.tracks) {
    if (t.rating >= 4) {
        hasFavorite = true;
        break;
    }
}

boolean hasFavorite = a.tracks.anyMatch(t -> t.rating >= 4);
```

Making a set of favorite albums

```
// Initialize favs as a set of favorite albums drawn from albums

Set<Album> favs = new HashSet<>();
for (Album a : albums) {
    if (a.tracks.anyMatch(t -> (t.rating >= 4)))
        favs.add(a);
}

Set<Album> favs =
  albums.filter(a -> a.tracks.anyMatch(t -> t.rating >= 4))
        .into(new HashSet<>());
```

Loops v. Lambdas

```
Set<Album> favs = new HashSet<>();
for (Album a : albums) {
    boolean hasFavorite = false;
    for (Track t : a.tracks) {
        if (t.rating >= 4) {
            hasFavorite = true;
            break;
        }
    }
    if (hasFavorite) favs.add(a);
}
```

```
Set<Album> favs =
  albums.filter(a -> a.tracks.anyMatch(t -> t.rating >= 4))
        .into(new HashSet<>());
```

Loops v. Lambdas – Adding parallelism

```
Set<Album> favs = new HashSet<>();
for (Album a : albums) {
    boolean hasFavorite = false;
    for (Track t : a.tracks) {
        if (t.rating >= 4) {
            hasFavorite = true;
            break;
        }
    }
    if (hasFavorite) favs.add(a);
}
```

```
Set<Album> favs =
  albums.parallel()
        .filter(a -> a.tracks.anyMatch(t -> (t.rating >= 4)))
        .into(new ConcurrentHashSet<>());
```

In summary

- Lambdas are the on-ramp to productive parallel programming
- Adding lambdas to Java without lambda-fying Collections would be poor, and replacing Collections is a non-starter
- Compatibly evolving interface-based APIs is impossible without directly addressing the problem in the language and VM
- The solution – an enhanced inheritance model – enables new idioms in Java and helps other languages too
- JSR 335 in Early Draft Review
- Prototype binaries at OpenJDK Project Lambda

Other features in Java SE 8

- Java Module System (JSR TBD) + JDK modularization
- Annotations on type names (JSR 308)
- Repeating annotations
- Parameter name access at runtime
- Refinements to Project Coin
  - try-with-resources on an effectively final variable?
  - Remove restrictions on diamond operator?
  - @SafeVarargs on more kinds of method?

Transparency

Java Community Process 2.8

- Expert Group transparency
  - EG must do all substantive business on a public mailing list
  - EG must track issues in a public issue tracker
  - EG must respond publicly to all comments
- Executive Committee transparency
  - EC must hold public meetings and teleconferences, and publish minutes
  - EC must provide a public mailing list for JCP member feedback
- TCK and License transparency
  - TCK licensing must permit public discussion of testing process and results
  - Spec Lead cannot withdraw a spec/RI/TCK license once offered

Java Community Process 2.8

- Participation
  - EG nominations and Spec Lead responses must be public
  - EG members are identified by name and company
- Agility
  - JSRs must reach Early Draft Review within nine months
  - JSRs must reach Public Review within 12 months after EDR
  - JSRs must reach Final Release within 12 months after PR
  - Faster and simpler Maintenance Releases
- JCP 2.8 is mandatory for new JSRs, and in-flight JSRs are encouraged to adopt it

JDK Enhancement Proposal (JEP) Process

- “A process for collecting, reviewing, sorting, and recording the results of proposals for enhancements to the JDK”
- Goal: Produce a regularly-updated list of proposals to serve as the long-term Roadmap for JDK Release Projects
- Looks at least three years into the future to allow time for the most complex proposals to be defined and implemented
- Open to every OpenJDK Committer
- Does not in any way supplant the Java Community Process

JEPs as of 11/11/11 (11:11:11)

1	JDK Enhancement-Proposal & Roadmap Process
2	JEP Template
101	**Generalized Target-Type Inference**
102	Process API Updates
103	**Parallel Array Sorting**
104	**Annotations on Java Types**
105	DocTree API
106	Add Javadoc to javax.tools
107	**Bulk Data Operations for Collections**
108	**Collections Enhancements from Third-Party Libraries**
109	**Enhance Core Libraries with Lambda**
110	New HTTP Client
111	Additional Unicode Constructs for Regular Expressions
112	Charset Implementation Improvements
113	MS-SFU Kerberos 5 Extensions
114	TLS Server Name Indication (SNI) Extension
115	AEAD CipherSuites
116	Extended Validation Certificates
117	Remove the Annotation-Processing Tool (apt)
118	**Access to Parameter Names at Runtime**
119	javax.lang.model Implementation Backed by Core Reflection
120	**Repeating Annotations**
121	Stronger Algorithms for Password-Based Encryption
122	Remove the Permanent Generation
123	Configurable Secure Random-Number Generation
124	Enhance the Certificate Revocation-Checking API
125	Network Interface Aliases, Events, and Defaults
126	**Lambda Expressions and Virtual Extension Methods**

In Conclusion

Where We've Been, Where We're Going

- Java SE 7 laid the groundwork
  - Language and library changes for productivity
  - Library and VM plumbing for concurrency and functional idioms
- Java SE 8 evolves the language, libraries, and VM together
  - Lambdas + Virtual extension methods + Parallel collections
  - Modules + Modularized libraries and tools
- Java SE's best days lie ahead
  - Multi-language interoperability through a unified, reified type system
  - A two-year tick for Java SE releases
  - Collaboration in OpenJDK and the JCP

ORACLE®