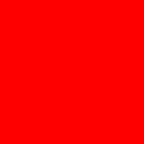




ORACLE®

JVMLS 2013 Keynote

Guy L. Steele Jr., Software Architect
Oracle Labs Programming Language Research Group



Copyright © 2013 Oracle and/or its affiliates (“Oracle”). All rights are reserved by Oracle except as expressly stated as follows. Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted, provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers, or to redistribute to lists, requires prior specific written permission of Oracle.

An Interesting Essay

- **William R. Cook. On understanding data abstraction, revisited. Onward! Essays, 2009 ACM OOPSLA. SIGPLAN Notices 44, 10 (Oct. 2009), 557-572.**
- **Makes a distinction between abstract data types and object-oriented programming**
 - **ADT encapsulates implementation of a *type***
 - **OO encapsulates implementation of *instances***
- **I conclude:**
 - **Java is actually more ADT-like**
 - **Doing true OO in Java requires discipline—and help**

An Example (after Cook)

- A simple data structure: set of long values

```
interface LongSet {
    boolean isEmpty();
    LongSet adjoin(long x);
    LongSet union(LongSet other);
    boolean contains(long x);
    static LongSet Empty = new EmptySet();
}
```

- **Example:** `LongSet.Empty.adjoin(1).adjoin(2)`
`.union(LongSet.Empty.adjoin(3)).contains(2)`
should be true

Implementation of EmptySet

```
class EmptySet implements LongSet {
    EmptySet() { }
    public boolean isEmpty() { return true; }
    public LongSet adjoin(long x) {
        return new AdjoinSet(x, this); }
    public LongSet union(LongSet other) {
        return other; }
    public boolean contains(long x) {
        return false; }
}
```

Implementation of AdjoinSet (1 of 3)

```
class AdjoinSet implements LongSet {
    long item; LongSet rest;
    AdjoinSet(long x, LongSet r) {
        item = x; rest = r; }
    public boolean isEmpty() { return false; }
    public LongSet adjoin(long x) {
        return new AdjoinSet(x, this); }
    public LongSet union(LongSet other) {
        return new UnionSet(this, other); }
    public boolean contains(long x) {
        return (x == item) || rest.contains(x); }
}
```

Implementation of AdjoinSet (2 of 3)

```
class AdjoinSet implements LongSet {
    long item; LongSet rest;
    AdjoinSet(long x, LongSet r) {
        item = x; rest = r; }
    public boolean isEmpty() { return false; }
    public LongSet adjoin(long x) {
        if (contains(x)) return this;
        else return new AdjoinSet(x, this); }
    public LongSet union(LongSet other) {
        return new UnionSet(this, other); }
    public boolean contains(long x) {
        return (x == item) || rest.contains(x); }
}
```

Implementation of AdjoinSet (3 of 3)

```
class AdjoinSet implements LongSet {
    long item; LongSet rest;
    AdjoinSet(long x, LongSet r) {
        item = x; rest = r; }
    public boolean isEmpty() { return false; }
    public LongSet adjoin(long x) {
        if (contains(x)) return this;
        else return new AdjoinSet(x, this); }
    public LongSet union(LongSet other) {
        return new UnionSet(this, other); }
    public boolean contains(long x) {
        if (x == item) return true;
        else return rest.contains(x); }
}
```


Implementation of UnionSet

```
class UnionSet implements LongSet {
    LongSet set1, set2;
    UnionSet(LongSet x1, LongSet x2) {
        set1 = x1; set2 = x2; }
    public boolean isEmpty() {
        return set1.isEmpty() && set2.isEmpty(); }
    public LongSet adjoin(long x) {
        if (contains(x)) return this;
        else return new AdjoinSet(x, this); }
    public LongSet union(LongSet other) {
        return new UnionSet(this, other); }
    public boolean contains(long x) {
        return set1.contains(x)
            || set2.contains(x); }
}
```

Implementation of MultiplesOf

```
class MultiplesOf implements LongSet {
    long factor;
    public MultiplesOf(long x) { factor = x; }
    public boolean isEmpty() { return false; }
    public LongSet adjoin(long x) {
        if (contains(x)) return this;
        else return new AdjoinSet(x, this); }
    public LongSet union(LongSet other) {
        return new UnionSet(this, other); }
    public boolean contains(long x) {
        return (factor == 0) ?
            (x == 0) :
            (x % factor == 0); }
}
```

Using “Infinite” Sets

Now we can write an expression such as

```
new MultiplesOf(2) .union(new MultiplesOf(3))  
    .adjoin(7) .adjoin(43)
```

and expect it to behave as if it contained all long values that are a multiple of either 2 or 3 (or both), as well as the values 7 and 43.

A Simple Use Case (1 of 2)

```
class SetTest {
    public static void main(String[] args) {
        LongSet s = Empty.adjoin(2);
        long candidate = 3;
        int count = 1;
        while (count < 1000000) {
            if (isPrime(candidate)) {
                s = s.adjoin(candidate);
                ++count;
            }
            candidate += 2;
        }
        System.out.println(s.contains(13));
    }
}
```

What does it print?

Implementation of AdjoinSet (again)

```
class AdjoinSet implements LongSet {
    long item; LongSet rest;
    AdjoinSet(long x, LongSet r) {
        item = x; rest = r; }
    public boolean isEmpty() { return false; }
    public LongSet adjoin(long x) {
        if (contains(x)) return this;
        else return new AdjoinSet(x, this); }
    public LongSet union(LongSet other) {
        return new UnionSet(this, other); }
    public boolean contains(long x) {
        if (x == item) return true;
        else return rest.contains(x); }
}
```

Implementation of AdjoinSet (modified)

```
class AdjoinSet implements LongSet {
    long item; LongSet rest;
    ...
    boolean contains(long x) {
        // This for loop has an interesting idea
        // but is erroneous.
        for (AdjoinSet z = this;
            !z.isEmpty();
            z = z.rest) {
            if (z.item == x)
                return true;
        }
        return false;
    }
}
```

Implementation of AdjoinSet (modified more)

```
class AdjoinSet implements LongSet {
    long item; LongSet rest;
    ...
    boolean contains(long x) {
        LongSet z;
        for (z = this;
             !(z instanceof AdjoinSet);
             z = ((AdjoinSet) z).rest) {
            if (((AdjoinSet) z).item == x)
                return true;
        }
        return z.contains(x);
    }
}
```


Implementation of AdjoinSet (modified more)

```
class AdjoinSet implements LongSet {
    long item; LongSet rest;
    ...
    boolean contains(long x) {
        for (LongSet z = this;
            !(z instanceof AdjoinSet);
            z = ((AdjoinSet) z).rest) {
            if (((AdjoinSet) z).item == x)
                return true;
        }
        return z.contains(x);
    }
}
```

But what if there are 2 sorts of set that form long chains?
(Use an iterator??)

Tail Calls Can Enhance Modularity

- **Sometimes what you want is a Finite-State Machine**
 - **The JVM model is a stack machine**
 - **If you're *forced* to carry along a stack of return addresses, the state ain't finite!**
 - **So we use loops for simple cases and simulations (simulated program counter and a loop around a big case statement) for complicated cases**
 - **No good way to break up the FSM graph**
 - **This matters when method size is limited**
 - **In fact, tail calls provide a way around this limit!**
 - **So you cannot modularize the FSM**

An Analogy: Airplane Tickets

- **What if there were no one-way tickets, only round trips?**
- **How can you start at SFO, visit BOS and DFW, then return to SFO?**
 - **Round-trip SFO-BOS, then round-trip SFO-DFW**
 - **Round-trip SFO-BOS, but while in BOS, round-trip BOS-DFW**
- **What you really want is a triangle trip**
 - **From SFO go to BOS; remember SFO as return point**
 - **From BOS go to DFW; pass along SFO return point**
 - **Return to SFO**

I Think Tail Calls Should Be Marked (1 of 2)

```
class AdjoinSet implements LongSet {
    long item; LongSet rest;
    ...
    public boolean contains(long x) {
        if (x == item) return true;
        else goto rest.contains(x); } // OO style
}
```

I Think Tail Calls Should Be Marked (2 of 2)

```
class AdjoinSet implements LongSet {
    long item; LongSet rest;
    ...
    boolean contains(long x) {    // ADT style
        LongSet z;
        for (z = this;
            !(z instanceof AdjoinSet);
            z = ((AdjoinSet) z).rest) {
            if (((AdjoinSet) z).item == x)
                return true;
        }
        goto z.contains(x);    // But OO for others
    }
}
```

Advantages of Proper Tail Calls in JVM

- **Support for coding Finite-State Machine models**
 - **Modularity of Finite State Machine models**
 - **OO abstraction, not just ADT abstraction**
- **The ability to automatically break up large methods**
 - **Each JVM method is a Finite-State Machine**
 - **Just a matter of modularizing the control-flow graph**
- **Support for other JVM-targeted languages that want to provide these advantages**
 - **A recent paper on a functional language (paraphrased):**
We have all sorts of features that differ from Java, and this paper is about how we manage to compile them all to the JVM—but tail calls remain a problem.

Please, Please: Proper Tail Calls in JDK9

- **So many advantages**
 - **This we have known for over 17 years**
 - **No, for over 34 years**
- **Now not that hard to do**
 - **We understand the JVM implementation issues**
 - **We understand the JVM security issues**

Parallelism: Streams and Spliterators

- **Java is becoming somewhat more functional in style**
- **Guess what: so are a lot of other languages**
- **There seems to be a sort of convergence happening, a consensus on how represent and process collections**
- **Not surprising: avoiding side effects**
- **Surprising: use of higher-order functions and lambdas**
 - **Java dragged a lot of C programmers halfway to Lisp**
 - **Killer feature: garbage collection (*memory*)**
 - **Maybe now it will drag them halfway to Haskell?**
 - **Killer feature: automatic parallelism (*processors*)**
- **Hurray for JDK8!**

A Related Project:

Generating Pseudorandom Numbers

- Not obvious that it fits in the framework
 - Side effect? Call the same method, get different values
 - But a side effect of a limited form
 - Actually a form of resource allocation, like “new”
- So consider a class like `java.util.Random`, but with these two methods:
 - `nextLong()`
 - `split()`
- It's sort of collection-like
- You can make a spliterator for it
- You can get a stream from it: `rng.longs()`

Guaranteeing Good Behavior

- You don't want the result of `rng.split()` to generate the same set of values as `rng`
- In fact, characterizing what you do want is subtle
 - Requiring a precise partition is overkill
 - But you want good mathematical properties
- So is making it happen!
 - Leiserson, Schardl, and Sukha (2012 ACM PpoPP)
- We're still exploring alternatives
 - Salmon, Moraes, Dror, and Shaw (2011 ACM SC)

An Implementation in Pure Java

We have implemented a class `SplittableRandom` that compares favorably to `java.util.Random`

- Much longer period (2^{64} instead of 2^{48})
- Passes statistical tests that `java.util.Random` fails
 - DieHarder, including Marsaglia-Tsang GCD test
- Sequential Monte Carlo benchmark (calculation of π by measuring the area under a quarter-circle):
4.5x speedup
- Parallel sum-of-stream-of-longs benchmark:
2900x speedup!

Confirmation of a Good Framework

It's always very satisfying when a design generalizes to applications not originally envisioned.

The spliterator framework helped us to organize the code better and to see how we could better tune it for speed.

War Story about Low-Level Details (1 of 4)

```
long  seed;           // 64-bit value
long  gamma;         // 64-bit value
```

```
long s = seed;
seed = s + gamma;
if (seed < s) seedFixup();
```

War Story about Low-Level Details (2 of 4)

```
long  seedHi,  seedLo;  // 128-bit value
long  gammaHi, gammaLo; // 128-bit value
```

```
long s = seedLo, h = seedHi, gh = gammaHi;
seedLo = s + gammaLo;
if (seedLo < s) ++gh;
seedHi = h + gh;
if (seedHi < h) seedFixup();
```

War Story about Low-Level Details (3 of 4)

```
long  seedHi,  seedLo;  // 128-bit value
long  gammaHi, gammaLo; // 128-bit value
```

```
long s = seedLo, h = seedHi, gh = gammaHi;
seedLo = s + gammaLo;
if (seedLo < s) ++gh;
seedHi = h + gh;
if (seedHi < h) seedFixup();
```

```
gamma: 000nnnnnnnnnnnnnnnnn nnnnnnnnnnnnnnnnnn
```

War Story about Low-Level Details (4 of 4)

```
long  seedHi,  seedLo;  // 128-bit value
long  gammaHi, gammaLo; // 128-bit value
```

```
long s = seedLo, h = seedHi, gh = gammaHi;
seedLo = s + gammaLo;
if (seedLo < s) ++gh;
seedHi = h + gh;
if (seedHi < h) seedFixup();
```

```
gamma: 000nnnnnnnnnnnnnnnnnn 0nnnnnnnnnnnnnnnnnn
```


Conclusions

- **Parallelism** is an important part of the big picture
 - We have a good story for JDK8
 - We hope we can support other languages well
- But **sequential programming** continues to be important
 - We hope we can support other languages well
 - At long last, let's implement proper tail calls!
- And sometimes **low-level detail** matters for performance
- We need good tools and good JVM support for all three