Dirty Hacks With Java Reflection (includes one or two useful hints)

Dr Heinz M. Kabutz

Last updated 2016-10-21

Heinz Kabutz

- **Created The Java Specialists' Newsletter**
  - **Advanced newsletter for gifted Java programmers**
    - **No "Hello World" tutorials, except maybe**

```
System.out.println("Hello world!");
   -> "Goodbye, cruel world!"
```

- **Special offer - sign up today and get an invite to join our new JavaSpecialists.slack.com team**
  - **Chat to over 1200 Java experts from around the world in real time**
    - **And listen to their conversations**
  - **http://tinyurl.com/geecon-slack**

Reflection is like Opium

- **A bit too strong for every day use**
  - **But can relieve serious pain**
- **Please do not become a reflection addict!**

Modifying/Reading Private/Final Fields

- **We can access private fields by making it accessible**
  - **Requires security manager support**
- **Note: value field is final and private!**

```
import java.lang.reflect.*;

public class PrivateFinalFieldTest {
  public static void main(String... args)
      throws NoSuchFieldException, IllegalAccessException {
    Field value = String.class.getDeclaredField("value");
    value.setAccessible(true);
    value.set("hello!", "cheers".toCharArray());
    System.out.println("hello!");
  }
}
```

cheers

Stack Interface

- **Yes, you can define classes in interfaces!**

```
public interface Stack<E> {
  void push(E item);
  E pop();

  class Node<E> {
    private final E item;
    private final Node<E> next;

    public Node(E item, Node<E> next) {
      this.item = item;
      this.next = next;
    }
    public E getItem() { return item; }
    public Node<E> getNext() { return next; }
  }
}
```

Synchronized Stack

- **Plain old synchronized locking**

```
public class SynchronizedStack<E> implements Stack<E> {
  private Node<E> top = null;

  public synchronized void push(E item) {
    top = new Node<>(item, top);
  }

  public synchronized E pop() {
    if (top == null) return null;
    E item = top.getItem();
    top = top.getNext();
    return item;
  }
}
```

```
public class ConcurrentStackAR<E> implements Stack<E> {
  private final AtomicReference<Node<E>> top = new AtomicReference<>();

  public void push(E item) {
    Node<E> oldHead, newHead;
    do {
      oldHead = top.get();
      newHead = new Node<>(item, oldHead);
    } while (!top.compareAndSet(oldHead, newHead));
  }

  public E pop() {
    Node<E> oldHead, newHead;
    do {
      oldHead = top.get();
      if (oldHead == null)
        return null;
      newHead = oldHead.getNext();
    } while (!top.compareAndSet(oldHead, newHead));
    return oldHead.getItem();
  }
}
```

AtomicReferenceFieldUpdater

- **Used to be slow, but fast in later Java 8 versions**

```
public class ConcurrentStackARFU<E> implements Stack<E> {
 private volatile Node<E> top = null;

  public void push(E item) {
    Node<E> oldHead, newHead;
    do {
      oldHead = top;
      newHead = new Node<>(item, oldHead);
    } while (!topUpdater.compareAndSet(this, oldHead, newHead));
  }

  public E pop() { ... }

  private final static
     AtomicReferenceFieldUpdater<ConcurrentStackARFU, Node>
       topUpdater = AtomicReferenceFieldUpdater.newUpdater(
           ConcurrentStackARFU.class, Node.class, "top");

}
```

sun.misc.Unsafe

- **Similar to atomics, but uses pointer arithmetic**
  - **compareAndSwapObject()**

```
public class ConcurrentStackUnsafe<E> implements Stack<E> {
  private volatile Node<E> top = null;

  public void push(E item) {
    Node<E> oldHead, newHead;
    do {
      oldHead = top;
      newHead = new Node<>(item, oldHead);
    } while (!UNSAFE.compareAndSwapObject(
        this, TOP_OFFSET, oldHead, newHead));
}

public E pop() { ... }
```

sun.misc.Unsafe Plumbing

- **Usually we hide gory details at end of class**
  - **Dangerous, don't use: sun.misc dependency, direct memory access**

```
private final static Unsafe UNSAFE;
private static final long TOP_OFFSET;

static {
  try {
    Field theUnsafeField =
      Unsafe.class.getDeclaredField("theUnsafe");
    theUnsafeField.setAccessible(true);
    UNSAFE = (Unsafe) theUnsafeField.get(null);
    TOP_OFFSET = UNSAFE.objectFieldOffset(
        ConcurrentStackUnsafe.class.getDeclaredField("top"));
  } catch (ReflectiveOperationException e) {
    throw new ExceptionInInitializerError(e);
  }
}
}
```

VarHandles in Java 9

- **Replacement for Unsafe and AtomicXXXFieldUpdater**

```
import java.lang.invoke.*;

public class ConcurrentStackVarHandles<E> implements Stack<E> {
  private final static VarHandle topHandle;

  static {
    try {
      topHandle = MethodHandles.lookup().findVarHandle(
          ConcurrentStackVarHandles.class,
          "top", Node.class
      );
    } catch (ReflectiveOperationException e) {
      throw new ExceptionInInitializerError(e);
    }
  }
```

VarHandles in Java 9

```
private volatile Node<E> top = null;

public void push(E item) {
  Node<E> oldHead, newHead;
  do {
    oldHead = top;
    newHead = new Node<>(item, oldHead);
  } while (!topHandle.compareAndSet(this, oldHead, newHead));
}
  public E pop() { ... }
}
```

VarHandle compareAndExchange()

- **Does a true compareAndSwap**
  - **Always returns the value that was found**
  - **Faster under moderate contention**

```
public void push(E item) {
  Node<E> oldHead, newHead, swapResult = top;
  do {
    oldHead = swapResult;
    newHead = new Node<>(item, oldHead);
  } while ((swapResult = (Node<E>) topHandle.compareAndExchange(
      this, oldHead, newHead)) != oldHead);
}
```

Quick VarHandle Search

- **Where would we likely see this being used?**

Issues with VarHandles

- **Great, but**
  - **Cannot use VarHandles on fields in foreign classes**
  - **So how would you access the String.value field with VarHandles?**

```
import java.lang.invoke.*;

public class StringReader {
  public static void main(String... args)
      throws ReflectiveOperationException {
    VarHandle valueHandle = MethodHandles.lookup().findVarHandle(
        String.class, "value", byte[].class);
    valueHandle.set("Hello there", valueHandle.get("Cheerio"));
    System.out.println("Hello there");
  }
}
```

IllegalAccessException: member is private

Optimization methodology

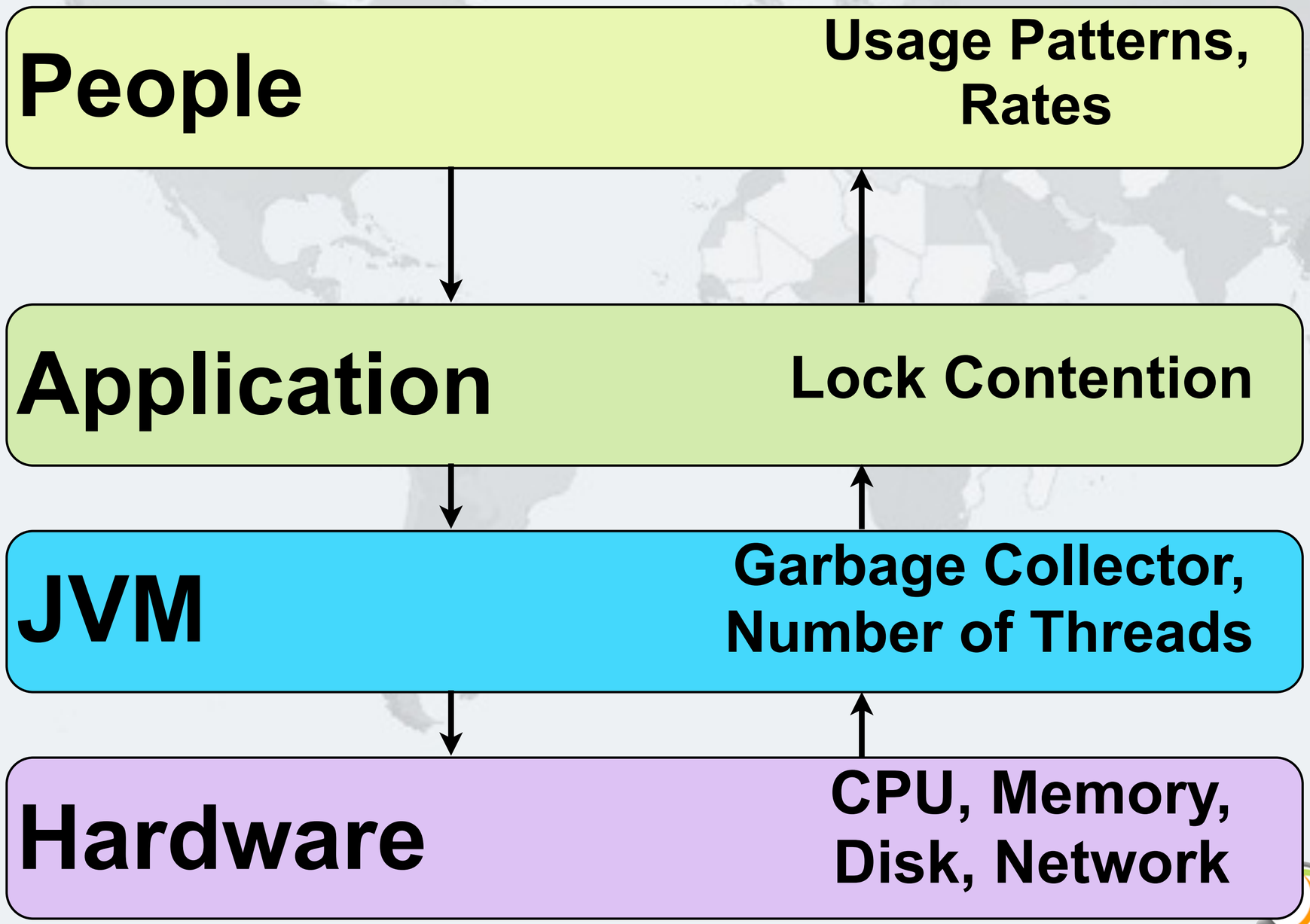
1. **Load test to identify bottlenecks**
   - **Identify the easiest to fix**
2. **Derive a hypothesis for the cause of the bottleneck**
   - **Create a test to isolate the factor identified by the hypothesis**
     - **This is important, we have often been fooled by profilers!**
3. **Alter the application or configuration**
4. **Test that the change improves the situation**
   - **Also make sure the system still works correctly**

- **Repeat process until targets are met**

Big Gains Quickly

- **Amdahl's law applies**
  - **Consider an 4 layered application**
    - **Servlet takes 10%**
    - **Business component takes 11%**
    - **EJB takes 23%**
    - **SQL takes 56%**
  - **Scenario 1, tuning Servlet gives 20x improvement**
    - **"Google" says that servlets are slow**
    - **0.10/20 + 0.11/1 + 0.23/1 + 0.56 /1 = 0.905**
  - **Scenario 2, tuning SQL give 2x improvement**
    - **We *measure* and discover SQL is the bottleneck**
    - **0.10/1 + 0.11/1 + 0.23/1 + 0.56/2 = 0.72**

System Overview - The Box



Remember:
tinyurl.com/geecon-slack