

Welcome to The Java Specialists' Newsletter

This selection of articles from our newsletter covers some of the topics we will be doing in the Extreme Java - Concurrency and Performance for Java 8 Course. We recommend that you look through some of them before coming to class as it will make the discussions far more interesting.

All the articles in this PDF have been carefully chosen as being most relevant to what you will be learning. We list them in reverse order, with newest first.

Enjoy!

Heinz

Issue 238 - java.util.Optional - Short Tutorial By Example

Author: Dr. Heinz M. Kabutz

Date: 2016-05-09

Category: Language

Java Versions: 8, 9

Abstract:

Java 8 introduced the `java.util.Optional` class, based on the famous Guava class by the same name. It was said that we should hardly ever call `get()`. In this newsletter we offer a short tutorial that demonstrates coding examples of alternatives to `get()`.

Welcome to the 238th edition of **The Java(tm) Specialists' Newsletter**, written en route back from JAX Finance in London and completed on another flight from Greece to Spain. I'm in Malaga for a few days and will speak at the **Malaga Java User Group tomorrow evening the 10th May 2016**. I like speaking at JUGs whilst traveling, since I meet people who are truly passionate about Java and learning.

A few weeks ago, Sanjeev Kumar made the suggestion that since he really liked **my instructional videos on Vimeo**, I should try to record a short message per newsletter. I liked the idea and have started doing it. I hope you find it interesting. **Please let me know** if you do and would like to see more videos.

[JavaSpecialists] Issue 238 - `java.util.Optional` - Short Tutorial By Example

`java.util.Optional` - Short Tutorial By Example

I focus my research mostly on core Java SE. Even though I look at many other frameworks like Spring, Guava, Eclipse Collections, JEE, they are not stored in my brain's long-term memory. Unfortunately I had not spent a lot of time with `Optional` from Google Guava, so when `java.util.Optional` arrived in Java 8, it was fairly new to me. The point of `Optional` and its cousins `OptionalInt`, `OptionalDouble` and `OptionalLong`, is to avoid returning `null` from methods. A noble goal.

We should avoid calling `get()` and instead use some other methods that I will show in a moment. If we have to use `get()`, we should first check if the value is present. A `get()` not preceded by a call to `isPresent()` is a bug. Since we have alternatives to `get()` for most use cases, Stuart Marks recently suggested `get()` should be deprecated and replaced with the method `getWhenPresent()`. I think only Brian Goetz agreed with him! Hopefully it won't happen, since I don't think `getWhenPresent()` is that well named either. What if it's not present? What then? Does it block? Oh, it throws a `NoSuchElementException`! So why not call it `getWhenPresentOrElseThrowNoSuchElementExceptionIfYouCallIt()`? A change like that in Java is cumbersome. First off, they need to add the new method and

deprecate the old `get()` method. They then have to wait at least a release to delete `get()` altogether. In the meantime, we have to change all our code that innocently calls `get()` after checking that it exists with `isPresent()` to either use the new name or to instead annotate our code with `@SuppressWarnings("deprecated")`. Or we could just ignore deprecation warnings, like we have done for the past 20 years ...

Stuart Marks sent a very interesting post, where he shows examples in the JDK where `Optional.get()` was used and where it could have been replaced with another mechanism. He very kindly agreed to let me publish his findings in this newsletter. I am hoping that it will be a "tutorial by example" for those who have not used `Optional` before and would like to learn how it works.

The first method we should learn is `Optional.ifPresent(Consumer<? super T> action)`. Not `isPresent()`, but `ifPresent()`. I must admit that when I saw his reference to this method I scratched my head and wanted to see if it really existed. I had seen `isPresent()` before, but somehow didn't see that we also had another method that looked almost the same. After all, "s" and "f" are just separated by a "d" :-) Thus if you are tempted to write code like this:

```
if (source.isPresent()) {
    doSomethingWith(source.get());
}
```

As long as `doSomethingWith()` does not throw any checked exceptions, you could easily transform the code to:

```
source.ifPresent(s -> doSomethingWith(s));
```

Or if like me you aren't scared of method references, you could do this:

```
source.ifPresent(this::doSomethingWith);
```

Stuart dug around in the JDK and came up with a bunch of examples where this could have been used. Remember, these are JDK developers, so hopefully not complete beginner programmers. The first one is from the [DependencyFinder.java](#):

```
206 if (source.isPresent()) {
207     executor.runTask(source.get(), deque);
208 }
```

This could be rewritten as:

```
source.ifPresent(archive -> executor.runTask(archive, deque));
```

Our next example is from [JdepsTask.java](#)

```
476 Optional<String> req = options.requires.stream()
477     .filter(mn -> !modules.containsKey(mn))
478     .findFirst();
479 if (req.isPresent()) {
480     throw new BadArgs("err.module.not.found", req.get());
481 }
```

could be rewritten as

```
options.requires.stream()
    .filter(mn -> !modules.containsKey(mn))
    .findFirst()
    .ifPresent(s -> throw new BadArgs("err.module.not.found", s));
```

Next we have a code snippet where the programmer did not check that the Optional contained a value using `isPresent()`. If for some reason the `subList()` was empty, the `reduce()` would return an empty Optional and thus `get()` would throw a `NoSuchElementException`. IDEs should pick this up and warn us. Furthermore, we have a shorter and probably more efficient way of joining the Strings together using `String.join()`. The code this time is from [JShellTool.java](#)

```
1203 String hist = replayableHistory
1204     .subList(first + 1, replayableHistory.size())
1205     .stream()
1206     .reduce( (a, b) -> a + RECORD_SEPARATOR + b)
1207     .get();
```

could be rewritten as

```
String hist = String.join(RECORD_SEPARATOR,
    replayableHistory.subList(first+1, replayableHistory.size()));
```

Another little code snipped from [Resolver.java](#)

```
100 if (mref.location().isPresent())
101     trace(" (%s)", mref.location().get());
```

could be rewritten as

```
mref.location().ifPresent(loc -> trace(" (%s)", loc));
```

The next one is a bit different. Here they check for whether a particular value is *not* present. Instead of doing it this way, we can first filter and confirm that if there is a value, that it conforms to our requirements. If it doesn't, or there never was a value, we throw the exception using `Optional.orElseThrow()`. This example is from Java 9's new `Layer` class in the Java Reflection package: [Layer.java](#):

```
364 Optional<Configuration> oparent = cf.parent();
365 if (!oparent.isPresent() || oparent.get() != this.configuration()) {
366     throw new IllegalArgumentException(
367         "Parent of configuration != configuration of this Layer");
```

could be rewritten as

```
cf.parent()
    .filter(cfg -> cfg == this.configuration())
    .orElseThrow(() -> new IllegalArgumentException(
        "Parent of configuration != configuration of this Layer"));
```

I found these examples very interesting and helpful to understand how `Optional` should be used. However, what if our idiom is slightly different, for example something like:

```
Optional<BigInteger> prime = findPrime();
if (prime.isPresent()) {
    System.out.println("Prime is " + prime.get());
} else {
    System.out.println("Prime not found");
}
```

I posed this question to Stuart Marks and he sent me back a neat solution involving `map()` and `orElse()`. In our example, `map` transforms the `BigInteger` to `String`. However, if the `Optional` is empty, then it will simply return an empty `Optional<String>`. We can then further return a default value if it is empty ("Prime not found") or we could throw an exception. Here is the code:

```
System.out.println(  
    findPrime()  
        .map(p -> "Prime is " + p)  
        .orElse("Prime not found"));
```

I made a suggestion to Stuart that perhaps we should also add `Optional.ifPresentOrElse(Consumer, Runnable)`. Turns out this is coming in Java 9, together with some other new methods for turning the `Optional` into a composite:

```
public void ifPresentOrElse(Consumer<? super T> action,  
                           Runnable emptyAction);  
public Optional<T> or(Supplier<Optional<T>> supplier);  
public Stream<T> stream();
```

I would strongly encourage using `Optional` in your code base, as it reduces the possibility of a very common issue: `NullPointerException`. And if you are tempted to use `get()`, use the tips in this newsletter to follow the best practices instead.

Kind regards

Heinz

Issue 237 - String Compaction

Author: Dr. Heinz M. Kabutz

Date: 2016-04-16

Category: Performance

Java Versions: 6, 9

Abstract:

Java 6 introduced a mechanism to store ASCII characters inside byte[] instead of a char[]. This feature was removed again in Java 7. However, it is coming back in Java 9, except this time round, compaction is enabled by default and byte[] is always used.

Welcome to the 237th edition of **The Java(tm) Specialists' Newsletter**, written at 30'000 feet en route back to Greece from Düsseldorf. I think this was the first time that I visited that great city and *did not* go to eat dinner at the Fälschen. They make an incredible pork knuckle, which is best washed down with never-ending glasses of Fälschen Alt. Great food, incredible atmosphere, drinkable Alt. Like many German breweries, they use a variation of Mark-Sweep-Compact, so you might end up being inserted in between two burly Germans at their table. I've not seen that in other countries outside of Greater Germania. Very efficient use of space and they will always find a space for a lonely traveler. But not this time. I'm working hard at improving my health and sacrifices need to be made. I didn't regret that sacrifice this morning! In fact in my 44 years of life, I have never ever woken up and regretted not drinking *more* beer the night before :-) Have you?

[JavaSpecialists] Issue 237 - String Compaction

String Compaction

I spent this week teaching a class of clever Java programmers from Provinzial Rheinland Versicherung AG the subtleties of Java Design Patterns. You might be surprised to learn that my **Patterns Course** is overall my most successful course. Not **Concurrency** (although that sells well too). Not introduction to Java (haven't sold one in years). Not even **Advanced Topics** in Java. Nope, my humble Design Patterns Course that I wrote in 2001 is still selling strong today.

Usually when I teach my patterns course, we look at all sorts of related topics in Java. So the students learn far more than they would find in any single book. They see where the patterns are used in the JDK. They learn about good design principles. They see the latest Java 8 features, even if they might still be stuck on JDK 6 or 7. We even touch a bit on threading. It is the one course that once a company has sent some programmers on it, they usually just keep on sending more and more. This is why, 15 years after writing the first edition, it is still so popular with companies.

Yesterday we were looking at the Flyweight Pattern, which is a rather strange arrangement of

classes. It isn't really a design pattern. Rather, like the Facade, it is a necessary evil because of design patterns. Let me explain. A good object oriented design will result in systems that are highly configurable and that minimize duplicated code. This is good, but it also means that you sometimes have to do a lot of work to just use the system. Facade helps to make a complex subsystem easier to use. Why is it complex? Usually because we have too many options of how to use it, thanks to a liberal application of design patterns. Flyweight has a similar reason for being. Usually good OO designs have more objects than bad monolithic designs, where everything is a Singleton. Flyweight tries to reduce the number of objects through sharing, which works well when we have made intrinsic state to be extrinsic instead.

I was demonstrating String deduplication to my class yesterday, where the `char[]` inside the `String` is replaced with a shared `char[]` when we have multiple `Strings` containing the same values. To use it you need to use the G1 collector (`-XX:+UseG1GC`) and also turn String deduplication on (`-XX:+UseStringDeduplication`). It worked beautifully in Java 8. I then wanted to see whether this was enabled by default in Java 9, knowing that the G1 collector was now the default collector. I was a bit surprised when my code threw a `ClassCastException` when I tried to cast the `value` field in `String` to a `char[]`.

At some point in Java 6, we got compressed Strings. These were off by default and you could turn them on with `-XX:+UseCompressedStrings`. When they were on, `Strings` containing only ASCII (7-bit) characters would automatically be changed to contain a `byte[]`. If you had one character that was more than 7 bits in size, it used `char[]` again. Things got interesting when you had UTF-16 characters such with Devanagari Hindi, because then additional objects were created and we actually had higher object creation than without compressed strings. But for US ASCII, life was good. For some reason, this feature from Java 6 was deprecated in Java 7 and the flag was completely removed in Java 8.

However, in Java 9, a new flag was introduced `-XX:+CompactStrings` and this is now enabled by default. If you look inside the `String` class, you will notice that it *always* stores the characters of a `String` inside a `byte[]`. It also has a new byte field that stores the encoding. This is currently either Latin1 (0) or UTF16 (1). Potentially it could be other values too in future. So if your character are all in the Latin1 encoding, then your `String` will use less memory.

To try this out, I have written a small Java program that we can run in Java 6, 7 and 9 to spot the differences:

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

public class StringCompactionTest {
    private static Field valueField;

    static {
        try {
            valueField = String.class.getDeclaredField("value");
            valueField.setAccessible(true);
        } catch (Exception e) {
            e.printStackTrace();
        }
    }

    public static void main(String[] args) {
        String str = "Hello, World!";
        System.out.println(str.getClass().getDeclaredField("value").get(str));
    }
}
```

```

        valueField = String.class.getDeclaredField("value");
        valueField.setAccessible(true);
    } catch (NoSuchFieldException e) {
        throw new ExceptionInInitializerError(e);
    }
}

public static void main(String... args)
    throws IllegalAccessException {
    showGoryDetails("hello world");
    showGoryDetails("hello w\u00f8rld"); // Scandinavian o
    showGoryDetails("he\u03bb\u03bbo wor\u03bbd"); // Greek l
}

private static void showGoryDetails(String s)
    throws IllegalAccessException {
    s = "" + s;
    System.out.printf("Details of String \"%s\"\n", s);
    System.out.printf("Identity Hash of String: 0x%x%n",
        System.identityHashCode(s));
    Object value = valueField.get(s);
    System.out.println("Type of value field: " +
        value.getClass().getSimpleName());
    System.out.println("Length of value field: " +
        Array.getLength(value));
    System.out.printf("Identity Hash of value: 0x%x%n",
        System.identityHashCode(value));
    System.out.println();
}
}

```

The first run is with Java 6 and -XX:-UseCompressedStrings (the default). Notice how each of the Strings contains a char[] internally.

```

Java6 no compaction
java version "1.6.0_65"

Details of String "hello world"
Identity Hash of String: 0x7b1ddcde
Type of value field: char[]
Length of value field: 11
Identity Hash of value: 0x6c6e70c7

```

```
Details of String "hello world"
Identity Hash of String: 0x46ae506e
Type of value field: char[]
Length of value field: 11
Identity Hash of value: 0x5e228a02
```

```
Details of String "heî»î»o worî»d"
Identity Hash of String: 0x2d92b996
Type of value field: char[]
Length of value field: 11
Identity Hash of value: 0x7bd63e39
```

The second run is with Java 6 and -XX:+UseCompressedStrings. The "hello world" String contains a byte[] and the other two a char[]. Only US ASCII (7-bit) are compressed.

```
Java6 compaction
java version "1.6.0_65"

Details of String "hello world"
Identity Hash of String: 0x46ae506e
Type of value field: byte[]
Length of value field: 11
Identity Hash of value: 0x7bd63e39

Details of String "hello world"
Identity Hash of String: 0x42b988a6
Type of value field: char[]
Length of value field: 11
Identity Hash of value: 0x22ba6c83

Details of String "heî»î»o worî»d"
Identity Hash of String: 0x7d2ale44
Type of value field: char[]
Length of value field: 11
Identity Hash of value: 0x5829428e
```

Java 7 the flag was ignored. In Java 8 it was removed, so a JVM started with -XX:+UseCompressedStrings would fail. Of course all Strings just contained char[].

```

Java7 compaction
Java HotSpot(TM) 64-Bit Server VM warning: ignoring option
    UseCompressedStrings; support was removed in 7.0
java version "1.7.0_80"

Details of String "hello world"
Identity Hash of String: 0xa89848d
Type of value field: char[]
Length of value field: 11
Identity Hash of value: 0x57fd54c4

Details of String "hello wÃ,rld"
Identity Hash of String: 0x38c83cf8
Type of value field: char[]
Length of value field: 11
Identity Hash of value: 0x621c232a

Details of String "hei»î»o worî»d"
Identity Hash of String: 0x2548ccb8
Type of value field: char[]
Length of value field: 11
Identity Hash of value: 0x4e785727

```

Java 9 we have a new flag `-XX:+CompactStrings`. It is on by default. Strings now *always* store their payload as a `byte[]`, regardless of the encoding. You can see that for Latin1, the bytes are packed.

```

Java9 compaction
java version "9-ea"

Details of String "hello world"
Identity Hash of String: 0x77f03bb1
Type of value field: byte[]
Length of value field: 11
Identity Hash of value: 0x7a92922

Details of String "hello wÃ,rld"
Identity Hash of String: 0x71f2a7d5
Type of value field: byte[]
Length of value field: 11
Identity Hash of value: 0x2cfb4a64

```

```
Details of String "heî»î»o worî»d"
Identity Hash of String: 0x5474c6c
Type of value field: byte[]
Length of value field: 22
Identity Hash of value: 0x4b6995df
```

Of course you can turn off this new feature in Java 9 with -XX:-CompactStrings. However, the code within String has changed, so regardless of what you do, value is still a byte[].

```
Java9 no compaction
java version "9-ea"

Details of String "hello world"
Identity Hash of String: 0x21a06946
Type of value field: byte[]
Length of value field: 22
Identity Hash of value: 0x25618e91

Details of String "hello wÃ,rld"
Identity Hash of String: 0x7a92922
Type of value field: byte[]
Length of value field: 22
Identity Hash of value: 0x71f2a7d5

Details of String "heî»î»o worî»d"
Identity Hash of String: 0x2cfb4a64
Type of value field: byte[]
Length of value field: 22
Identity Hash of value: 0x5474c6c
```

Anybody using reflection to access the gory details inside String will now potentially get ClassCastException. Hopefully the set of such programmers is infinitesimally small.

A bigger worry is performance. Methods like `String.charAt(int)` used to be fast like lightning. I could detect a slow down in Java 9. If you're doing a lot of String walking with `charAt()`, you might want to explore alternatives. Not sure what they are though! Or perhaps they will fix this in the final release of Java 9, after all the version I'm looking at is Early Release (EA).

I heard about a trick by Peter Lawrey at one of our **JCrete Unconferences**. String has a constructor that takes a `char[]` and a boolean as a parameter. The boolean is never used and you are supposed to pass in `true`, meaning that `char[]` will be used directly within the String and not copied. Here is the code:

```
String(char[] value, boolean share) {
    // assert share : "unshared not supported";
    this.value = value;
}
```

Lawrey's trick was to create Strings very quickly from a `char[]` by calling that constructor directly. Not sure of the details, but most probably was done with `JavaLangAccess` that we could get from the `SharedSecrets` class. Prior to Java 9, this was located in the `sun.misc` package. Since Java 9, it is in `jdk.internal.misc`. I hope you are not using this method directly, because you will have to change your code for Java 9. But that's not all. Since `String` in Java 9 does not have `char[]` as value anymore, the trick does not work. `String` will still make a new `byte[]` every time you call it, making it about 2.5 times slower on my machine in Java 9.

Here's the code. You will have to correct the imports depending on what version of Java you are using.

```
//import sun.misc.*; // prior to Java 9, use this
import jdk.internal.misc.*; // since Java 9, use this instead

public class StringUnsafeTest {
    private static String s;

    public static void main(String... args) {
        char[] chars = "hello world".toCharArray();
        JavaLangAccess javaLang = SharedSecrets.getJavaLangAccess();
        long time = System.currentTimeMillis();
        for (int i = 0; i < 100 * 1000 * 1000; i++) {
            s = javaLang.newStringUnsafe(chars);
        }
        time = System.currentTimeMillis() - time;
        System.out.println("time = " + time);
    }
}
```

To summarize, if you speak English or German or French or Spanish, your Strings have just become a whole lot lighter. For Greeks and Chinese, they are using about the same. For all, Strings are probably going to be a little bit slower.

Kind regards from Thessaloniki Airport

Heinz

Issue 236 - Computational Complexity of BigInteger.multiply() in Java 8

Author: Dr. Heinz M. Kabutz

Date: 2016-03-31

Category: Performance

Java Versions: 8

Abstract:

BigInteger has new algorithms for multiplying and dividing large numbers that have a better computational complexity than previous versions of Java. A further improvement would be to parallelize multiply() with Fork/Join.

Welcome to the 236th edition of **The Java(tm) Specialists' Newsletter**, written in the Aegean Airlines Lounge in Athens on my way home after renewing my South African passport. Whilst speaking to the staff at the embassy, I discovered that Marthinus van Schalkwyk has recently been appointed as the South African Ambassador to Greece. Not sure what to say about that. Van Schalkwyk became leader of the National Party in South Africa after Frederik Willem de Klerk retired. The National Party is the one that banned the ANC long before I was born. He then changed the name to "New National Party", a bit like java.nio (New IO). They then merged with their previous opposition the Democratic Party and formed yet another party, the Democratic Alliance. Shortly after some disastrous local elections, the New National Party guys left the party (so to speak) and joined the ANC, which they had previously banned. It was at this point that I gave up trying to understand politics. Good thing too, seeing how we have a coalition government in Greece consisting of an extreme left and an extreme right party. How that marriage has lasted over a year is anyone's guess. At least I get to exercise my right to vote in Greece with regularity.

A special welcome to a new subscriber country - Georgia! We have several Georgians living in Chorafakia and it's great to now also be sending **The Java(tm) Specialists' Newsletter** to their home country :)

[JavaSpecialists] Issue 238 - Computational Complexity of BigInteger.multiply() in Java 8

Computational Complexity of BigInteger.multiply() in Java 8

Whilst randomly looking through Java code a few months ago, I noticed that BigInteger had been improved. In previous versions, multiply worked the way your elementary school teacher used to torture you: $O(n^2)$. In [an earlier newsletter](#), I mentioned how Karatsuba was a better algorithm to use for larger numbers. It weighs in at $O(n^{\lg 3})$ or approximately $O(n^{1.585})$. Another algorithm is 3-way Toom Cook, which is $O(n^{1.465})$. Whilst these look very similar, the difference is huge as n grows. Here is a small program that demonstrates it:

```

public class BigOComparison {
    public static void main(String... args) {
        for (int n = 1; n <= 1_000_000_000; n *= 10) {
            double n_2 = Math.pow(n, 2);
            double n_1_585 = Math.pow(n, 1.585);
            double n_1_465 = Math.pow(n, 1.465);

            double karatsuba_faster_than_quadratic = n_2 / n_1_585;
            double toom_cook_faster_than_karatsuba = n_1_585 / n_1_465;

            System.out.printf("%d\t%.2fx\t%.2fx\n",
                n, karatsuba_faster_than_quadratic,
                toom_cook_faster_than_karatsuba);
        }
    }
}

```

We can see that as n increases in size, the difference between the computational complexities becomes more apparent. At $n=1\text{,}000\text{,}000\text{,}000$, $n^{1.585}$ would be over 5000 faster than n^2 . $n^{1.465}$ would be another 12 times faster than that:

| | | |
|------------|----------|--------|
| 1 | 1.00x | 1.00x |
| 10 | 2.60x | 1.32x |
| 100 | 6.76x | 1.74x |
| 1000 | 17.58x | 2.29x |
| 10000 | 45.71x | 3.02x |
| 100000 | 118.85x | 3.98x |
| 1000000 | 309.03x | 5.25x |
| 10000000 | 803.53x | 6.92x |
| 100000000 | 2089.30x | 9.12x |
| 1000000000 | 5432.50x | 12.02x |

Of course there are setup costs that are ignored with Big O notation. Because of these, we would only want to use Karatsuba for large numbers and Toom Cook when they are even bigger.

Java 8 now has these two algorithms built in to BigInteger. To see the improvement in performance, here is Fibonacci using Dijkstra's sum of squares algorithm:

```

import java.math.*;

public class Fibonacci {
    public BigInteger f(int n) {
        if (n == 0) return BigInteger.ZERO;
        if (n == 1) return BigInteger.ONE;
        if (n % 2 == 1) { //  $F(2n-1) = F(n-1)^2 + F(n)^2$ 
            n = (n + 1) / 2;
            BigInteger fn_1 = f(n - 1);
            BigInteger fn = f(n);
            return fn_1.multiply(fn_1).add(fn.multiply(fn));
        } else { //  $F(2n) = (2 F(n-1) + F(n)) F(n)$ 
            n = n / 2;
            BigInteger fn_1 = f(n - 1);
            BigInteger fn = f(n);
            return fn_1.shiftLeft(1).add(fn).multiply(fn);
        }
    }

    public BigInteger f_slow(int n) {
        if (n == 0) return BigInteger.ZERO;
        if (n == 1) return BigInteger.ONE;
        return f_slow(n - 1).add(f_slow(n - 2));
    }
}

```

The `f_slow()` method is only there to help us test our fast algorithm, but would not be useful beyond about $n=30$.

Here is a test class that we can run in Java 7 and 8 to see how the reduced computational complexity in the `multiply()` algorithm speeds things up in Java 8:

```

public class FibonacciTest {
    public static void main(String... args) {
        Fibonacci fib = new Fibonacci();

        for (int i = 0; i < 10; i++) {
            if (!fib.f(i).equals(fib.f_slow(i))) {

```

```
        throw new AssertionError("Mismatch at i=" + i);
    }

}

for (int n = 10000; n < 50_000_000; n *= 2) {
    timeTest(fib, n);
}
}

private static void timeTest(Fibonacci fib, int n) {
    System.out.printf("fib(%d)%n", n);
    long time = System.currentTimeMillis();
    System.out.println(fib.f(n).bitLength());
    time = System.currentTimeMillis() - time;
    System.out.println("time = " + time);
}
}
```

Here is the output for Java 7:

```
heinz$ java -showversion FibonacciTest
java version "1.7.0_80"
Java(TM) SE Runtime Environment (build 1.7.0_80-b15)
Java HotSpot(TM) 64-Bit Server VM (build 24.80-b11, mixed mode)
fib(10,000)
6942
time = 14
fib(20,000)
13884
time = 10
fib(40,000)
27769
time = 11
fib(80,000)
55539
time = 23
fib(160,000)
111078
time = 51
fib(320,000)
222157
time = 108
fib(640,000)
```

```

444314
time = 181
fib(1,280,000)
888629
time = 590
fib(2,560,000)
1777259
time = 2530
fib(5,120,000)
3554518
time = 8785
fib(10,240,000)
7109037
time = 34603
fib(20,480,000)
14218074
time = 142635
fib(40,960,000)
28436148
time = 586950

```

You can see that as the value of n doubles, the time it takes roughly quadruples. You can also see by the number of bits that the results get rather large.

And now Java 8. For smaller numbers, we don't see much difference to Java 7, but we start to diverge at roughly fib(1,280,000). Java 8 calculates fib(40,960,000) about 50x faster. I wasn't patient enough to calculate larger numbers, since I have a flight to catch this afternoon :-) However, I would expect the next data point to be roughly 75x faster.

```

heinz$ java -showversion FibonacciTest
java version "1.8.0_74"
Java(TM) SE Runtime Environment (build 1.8.0_74-b02)
Java HotSpot(TM) 64-Bit Server VM (build 25.74-b02, mixed mode)
fib(10,000)
6942
time = 6
fib(20,000)
13884
time = 3
fib(40,000)
27769
time = 7

```

```

fib(80,000)
55539
time = 16
fib(160,000)
111078
time = 27
fib(320,000)
222157
time = 40
fib(640,000)
444314
time = 58
fib(1,280,000)
888629
time = 155
fib(2,560,000)
1777259
time = 324
fib(5,120,000)
3554518
time = 734
fib(10,240,000)
7109037
time = 1661
fib(20,480,000)
14218074
time = 4412
fib(40,960,000)
28436148
time = 11870

```

So, now you have seen that Java 8 has improved in at least one area. However, they did not go far enough in my opinion. Both Karatsuba and Toom-Cook can easily be parallelized with recursive decomposition. If you really want to work with such large numbers then you probably also want to throw hardware at your problem. I tried it out by modifying BigInteger and adding a little MultiplyTask:

```

private static final class MultiplyTask
    extends RecursiveTask<BigInteger> {
    private final BigInteger b1, b2;

    public MultiplyTask(BigInteger b1, BigInteger b2) {
        this.b1 = b1;
    }
}

```

```

    this.b2 = b2;
}

protected BigInteger compute() {
    return b1.multiply(b2);
}
}

```

And then inside the multiplyKaratsuba() method I changed

```

BigInteger p1 = xh.multiply(yh); // p1 = xh*yh
BigInteger p2 = xl.multiply(yl); // p2 = xl*yl

```

To instead do this:

```

MultiplyTask mt1 = new MultiplyTask(xh, yh);
mt1.fork();
BigInteger p2 = xl.multiply(yl); // p2 = xl*yl
BigInteger p1 = mt1.join(); // xh.multiply(yh); // p1 = xh*yh

```

By default my code uses the common Fork/Join Pool, but it would be marvelous to add a new method to BigInteger that allows us to multiply in parallel, for example:
`BigInteger.multiply(BigInteger, ForkJoinPool)` or more explicitly
`BigInteger.multiplyParallel(BigInteger, ForkJoinPool)`.

My modifications to BigInteger worked fairly well. I also used ManagedBlocker to implement a "reserved caching scheme" in Fibonacci to avoid calculating the same number twice. ManagedBlocker worked very nicely and kept my cores more busy.

Here is a tweet where I show my parallel solution without using the ManagedBlocker. Notice how idle my cores are, especially at the beginning of the run when the numbers I am multiplying are small. And **another tweet** with the same code, but with my "reserved caching scheme" using the ManagedBlocker to keep the ForkJoinPool more lively.
`fib(1_000_000_000)` finished 8% faster and as you can see from my CPU graph, utilized the

available hardware much better. I talk about these types of concepts in my [Extreme Java - Concurrency & Performance for Java 8 Course](#) in case you'd like to learn how to do it yourself.

I hope you enjoyed this newsletter and that it was useful to you.

Kind regards from sunny and warm Greece

Heinz

Issue 234 - Random Code Failures - Race Conditions Between Our Code and the JVM

Author: Dr. Heinz M. Kabutz

Date: 2015-11-20

Category: Concurrency

Java Versions: Java 8

Abstract:

Most of the time our code fails because of bugs in our code. Rarely, however, bugs in the JVM ecosystem cause our systems to fail. These are insanely difficult to diagnose in production systems. In this newsletter we look at two different such race conditions between our code and the JVM.

Welcome to the 234th edition of **The Java(tm) Specialists' Newsletter**, written during various flights at 30000 feet. Last weekend I was in Graz, Austria, in between two courses. On the Saturday I decided to take advantage of the stunning weather to head up the local Schoeckl mountain. Prior to starting, I asked my friend Holger whether a lot of people were walking up there. "Oh yes, half of Graz, don't worry." I did not want to go hiking in remote areas, lest something happen and I get into trouble. Holger kindly sent me a nice route I could follow on Endomondo. All keen I set out early on Saturday morning, hoping to still get some parking at the Schoecklkreuz.

Unfortunately I had forgotten to load the route into Endomondo prior to leaving the hotel, which I discovered when I parked my car. I thus had to follow the sign posts along the way. The first bit was OK, but then I got to a place where I wasn't quite sure. There were these red circles on the one sign, but I didn't notice them on the way up. The path looked OK, so I headed up. (I found out at the summit that red circle means - "Mittelschwierige Bergwege. Ausdauer und Trittsicherheit erforderlich"). A bit later, I saw some Austrians walk up the same route behind me, so I was fairly confident that I had made the right call. The path crossed a small road and then kept on going straight up through the forest. It did seem odd to me how many pine needles were lying on the path. Straight up did look like the shortest distance to get to the summit. So I trudged on. After what felt like a very long time, I checked my map and discovered that I had only gone up 1/3 of the way! And of course the Austrians who were behind me had decided to go along the small road instead of a goat path up the mountain. Down didn't sit well with me. First off there was my pride. Secondly walking down a treacherous path is more dangerous than up. So I kept on going. Fortunately when I checked the map a second time I was very close to the top. It was a very interesting little walk up the hill actually and I don't regret it, but it was a bit foolish. The pine needles did tell me that.

It all reminded me of the poem "The Road Not Taken" by Robert Frost, where he ends with: "Two roads diverged in a wood, and I - I took the one less traveled by, and that has made all the difference."

Why it makes financial sense to attend Heinz's course

Race Conditions Between Our Code and the JVM

In May this year, my esteemed friend and colleague Maurice Naftalin and I headed off to Israel to present my **Extreme Java - Concurrency and Performance for Java 8** to a bunch of smart Intel engineers. As the author of the best-selling book on Generics and Collections, Maurice has a great command of the English language. This, coupled with his vast experience in software engineering, is extremely useful in helping answer some of the tougher questions. We usually tag-team each other, with me preferring to answer questions in my native language, Java. I was flattered that Intel would invite me to teach them. They also enjoyed the course a lot, with my favourite unsolicited comment being: *"The three days that we've spent together were amazing. Java is our bread and butter and yet we've learnt many new things and techniques which I'm sure that will help us in the future."* - **Noam A.**

During the course, we got to know all the students quite well, so when I received an email from Dudu Amar in August, telling me that `String.compareTo()` sometimes gave incorrect results, I knew that I should take his claims seriously. He had found an issue in production where someone was comparing Strings with `.compareTo()` to determine equality, rather than `.equals()`, and it would occasionally spit out 0 when they were actually different.

Strangely, the failures only happened on one of his machines. They had lots of other similar machines in production. We did a binary search through Java versions to try and figure out if this failure had been introduced at some specific time. After some testing, Dudu found that it did not happen in Java 1.7.0_25-64, but it did from 1.7.0_40-64 onwards. It also failed in all Java 8 versions he tried. Usually knowing the exact version that an error starts appearing helps narrow down which change might have caused the error. Java has special intrinsic functions for Strings, including `compareTo()`. Thus the code you see in the `String.java` file is not what will really be executed.

Sadly, it being Intel with a huge amount of hardware to play with, someone went and upgraded all the CPUs, with the result that Dudu's program no longer fails. We are thus looking for someone, anyone, who can make this program fail on their machine, so that we can keep on trying to figure out what caused it. Please [contact me](#) if this fails on your machine. We would like to try some additional tests to narrow down the exact cause.

```
import java.util.concurrent.*;

public class PXMLTag {
    public final String m_type;
    private static String[] m_primitiveTypes = {
        "java.lang.String",
        "boolean",
        "int",
        "long",
        "float",
        "double",
        "char"
    };
}
```

```

"int",
"long",
"float",
"short"};
```

```

private static String[] m_allTypes = {
    "java.lang.String",
    "boolean",
    "int",
    "long",
    "float",
    "java.util.Vector",
    "java.util.Map",
    "short"};
```

```

public PXMLTag(String type) {
    this.m_type = type;
}
```

```

public boolean isPrimitiveType() {
    for (int i = 0; i < m_primitiveTypes.length; i++) {
        String type = m_type;
        String primitiveType = m_primitiveTypes[i];
        if (type.compareTo(primitiveType) == 0
            && notReallyPrimitiveType(type)) {
            System.out.println("This odd case");

            System.out.println("type '" + type + "'");
            System.out.println("currentPrimitiveType '" +
                primitiveType + "'");
            System.out.println("They are equal? " +
                type.equals(primitiveType));
            System.out.println("They are compared " +
                type.compareTo(primitiveType));

            System.out.println("m_type '" + m_type + "'");
            System.out.println("m_primitiveType[i] '" +
                m_primitiveTypes[i] + "'");
            System.out.println("They are equal? " +
                m_type.equals(m_primitiveTypes[i]));
            System.out.println("They are compared " +
                m_type.compareTo(m_primitiveTypes[i]));
            return true;
        }
    }
    return false;
}
```

```

public static boolean notReallyPrimitiveType(String m_type) {
    return m_type.contains("Vector") || m_type.contains("Map");
}

public static String getRandomType() {
    return m_allTypes[
        ThreadLocalRandom.current().nextInt(m_allTypes.length)];
}

public static void main(String[] args) {
    int threads = 1;
    if (args.length == 1) {
        threads = Integer.parseInt(args[0]);
    }
    for (int i = 0; i < threads; i++) {
        Thread t = createThread();
        t.start();
    }
}

private static Thread createThread() {
    return new Thread(new Runnable() {
        @Override
        public void run() {
            while (true) {
                PXMLTag tag = new PXMLTag(getRandomType());
                if (tag.isPrimitiveType()
                    && notReallyPrimitiveType(tag.m_type)) {
                    System.out.println(tag.m_type +
                        " not really primitive!");
                    System.exit(1);
                }
                try {
                    Thread.sleep(
                        ThreadLocalRandom.current().nextInt(100));
                } catch (InterruptedException e) {
                    return;
                }
            }
        }
    });
}
}

```

The code failed even with just one thread, but it took longer than with 5 threads.

Interestingly, we can turn off intrinsics of `compareTo()` in the String with the VM parameter `-XX:DisableIntrinsic=_compareTo`. (Thanks to Aleksey Shipilev for that tip.) We tried that and the program did not fail.

This was the first example of some very strange goings-on in Java code that could only be explained as a race-condition between our code and the JVM. I know that shouldn't happen. But it certainly seems to.

As I mentioned, I took Dudu's report seriously as he had attended my course and I knew that he was smart. I sometimes get slightly "less smart" requests from programmers who think it would be cool if I did their homework for them. My latest was a guy from South Africa asking if I wouldn't mind solving a programming task he had for a job interview! Sometimes when I feel playful I do a Socrates - ask questions that force them to think about the exercise themselves. Usually my questions take more effort to answer than the original homework assignment.

About a year ago someone sent me his homework assignment. He was studying computer science at some university in darkest Africa. His answer was so far off the mark that I gave him the following advice: *"Programming is probably not the right career for you. If you struggled with that exercise, then it's not the right thing for you to be doing as a career. I think I would have been able to figure out that exercise when I was about 15 or 16 years old. Sorry for being harsh, but I don't want to get your hopes up."*

I expected to never hear from him again. I knew it was a cruel thing to say to someone who had told me: "help me actualize my dreams in programming" I was rather surprised when I received an email from him about six months later. He told me that even though he could've easily been offended, he chose not to and took my advice. Instead of programming, he changed his focus and is now a database administrator and absolutely loves his work. I would be useless as a db admin and I admire him for being so mature to listen without getting angry. Well done mate!

On Stack Replacement (OSR) Causing Failures

Back to race conditions between our code and the JVM :-)

Another very smart ex-student of mine is Dr Wolfgang Laun. He sent me a code sample that he found on StackOverflow. This code started failing sporadically on the JVM from version 1.8.0_40 onwards. (Actually, isn't that a funny coincidence? Dudu's race condition happened after 1.7.0_40. Maybe 40 is to Java what is to us the number 13?) Before I show you the code (slightly modified from the original StackOverflow example), I would like to show just an excerpt that will make it clear that this simply should not fail:

```

double array[] = new double[1];
// some code unrelated to array[]
array[0] = 1.0;
// some more code unrelated to array[]
if (array[0] != 1.0) { // could not possibly be true
    if (array[0] != 1.0) {
        // and we should definitely not get here either!
    } else {
        // I never saw the code get into this else statement
    }
}

```

Seeing the code above, it should be clear that if we have just set the array[0] element to 1.0, that it would make no sense for it to be anything else but 1.0. In fact it might come back as 0.0!

```

import java.util.*;

// based on http://www.stackoverflow.com/questions/32994608/
// java-8-odd-timing-memory-issue
public class OnStackReplacementRaceCondition {
    private static volatile boolean running = true;
    public static void main(String... args) {
        new Timer(true).schedule(new TimerTask() {
            public void run() {
                running = false;
            }
        }, 1000);
        double array[] = new double[1];
        Random r = new Random();
        while (running) {
            double someArray[] = new double[1];
            double someArray2[] = new double[2];

            for (int i = 0; i < someArray2.length; i++) {
                someArray2[i] = r.nextDouble();
            }

            // for whatever reason, using r.nextDouble() here doesn't
            // seem to show the problem, but the # you use doesn't seem
            // to matter either...
        }

        someArray[0] = .45;
    }
}

```

```

array[0] = 1.0;

// can use any double here instead of r.nextDouble()
// or some double arithmetic instead of the new Double
new Double(r.nextDouble());

double actual;
if ((actual = array[0]) != 1.0) {
    System.err.println(
        "claims array[0] != 1.0....array[0] = " + array[0] +
        ", was " + actual);

    if (array[0] != 1.0) {
        System.err.println(
            "claims array[0] still != 1.0...array[0] = " +
            array[0]);
    } else {
        System.err.println(
            "claims array[0] now == 1.0...array[0] = " +
            array[0]);
    }

    System.exit(1);
} else if (r.nextBoolean()) {
    array = new double[1];
}
System.out.println("All good");
}
}

```

Output on all Java 8 versions from 1.8.0_40 to 1.8.0_72-ea, and also 1.9.0-ea was sometimes:

```

claims array[0] != 1.0....array[0] = 1.0, was 0.0
claims array[0] now == 1.0...array[0] = 1.0

```

The most likely reason for this bug is a race condition between on-stack-replacement (OSR) and our code. OSR does a hot-swap of our method that is currently at the top of the stack with faster, more optimized code. OSR is usually not the fastest machine code. To get that,

we need to exit completely from the method and come back in again. For example, consider this class OptimizingWithOSR:

```
public class OptimizingWithOSR {
    public static void main(String... args) {
        long time = System.currentTimeMillis();
        double d = testOnce();
        System.out.println("d = " + d);
        time = System.currentTimeMillis() - time;
        System.out.println("time = " + time);
    }

    private static double testOnce() {
        double d = 0;
        for (int i = 0; i < 1_000_000_000; i++) {
            d += 0;
        }
        return d;
    }
}
```

When we run this code with `-XX:+PrintCompilation`, we see that our `testOnce()` method is compiled, but not `main()`. This makes sense, since we hardly execute any code in `main()`. It all happens in `testOnce()`, but only once. Here is the output on my machine:

```
84 73 % 3 OptimizingWithOSR::testOnce @ 4 (22 bytes)
84 74     3 OptimizingWithOSR::testOnce (22 bytes)
84 75 % 4 OptimizingWithOSR::testOnce @ 4 (22 bytes)
86 73 % 3 OptimizingWithOSR::testOnce @ -2 (22 bytes)
```

The "%" means that it is doing on-stack-replacement. Now in terms of performance, with the default `-XX:+UseOnStackReplacement` my code runs in about 850ms. If I turn it off with `-XX:-UseOnStackReplacement`, it runs in 11 seconds and we do not see `testOnce()` in the compilation output.

It would be easy to jump to the conclusion that OSR is essential and that it would make all our code run gazillions of times faster. Nope. OSR will mostly make large monolithic code run a bit faster. Microbenchmarks, which are anyway unreliable at best, could also complete

more quickly. But well-structured, well-factored code will not see much benefit. Thus if you are a good Java developer, you can safely turn it off in production without suffering any slowdown. I refactored the code a bit by breaking the long loop into three and then extracting each of the loops into separate methods:

```
public class OptimizingWithNormalHotspot {
    public static void main(String... args) {
        for (int i = 0; i < 10; i++) {
            test();
        }
    }

    private static void test() {
        long time = System.currentTimeMillis();
        double d = testManyTimes();
        System.out.println("d = " + d);
        time = System.currentTimeMillis() - time;
        System.out.println("time = " + time);
    }

    private static double testManyTimes() {
        double d = 0;
        for (int i = 0; i < 1_000; i++) {
            d = unrolledTwo(d);
        }
        return d;
    }

    private static double unrolledTwo(double d) {
        for (int j = 0; j < 1_000; j++) {
            d = unrolledOne(d);
        }
        return d;
    }

    private static double unrolledOne(double d) {
        for (int k = 0; k < 1_000; k++) {
            d = updateD(d);
        }
        return d;
    }

    private static double updateD(double d) {
        d += 0;
        return d;
    }
}
```

}

The code now runs exactly the same, whether I have OSR turned on or not.

Thus my recommendation would be to run production systems that are using Java 1.8.0_40 or later with OSR turned *off* with the switch -XX:-UseOnStackReplacement. Of course you need to test that you don't have any performance degradation, but I would doubt it. Please let me know if you do.

The other race condition I mentioned at the start of this newsletter is more nebulous. Please shout if you manage to reproduce it on your hardware.

Interestingly, the Azul's Zulu virtual machine has exactly the same bug. It is to be expected, as their intention is to have an identical build to the OpenJDK, but with additional support.

I went to see the Danube yesterday with my friend Frans Mahrl who lives in "Orth an der Donau". The level is the lowest it has been since they started recording about 400 years ago. A bit more and we'll be able to wade across.

Kind regards from a sunny Crete

Heinz

Issue 232 - ByteWatcher from JCrete

Author: Daniel Shaya, Chris Newland and Dr. Heinz M. Kabutz

Date: 2015-09-29

Category: Performance

Java Versions: Java 8

Abstract:

Even though standard Java is not real-time, we often use it in time-sensitive applications.

Since GC events are often the reason for latency outliers, such code seeks to avoid allocation. In this newsletter we look at some tools to add to your unit tests to check your allocation limits.

Welcome to the 232nd edition of **The Java(tm) Specialists' Newsletter**, sent to you from the beautiful Island of Crete. Thank you to all those who were willing to take my place on Martin Thompson's course. In the end, we had such a lot of interest that we had to raffle the place. Thompson certainly is a popular guy! In a funny twist of fate, the winner works for LMAX, the company that Martin used to work for :-)

I spent the last few days as a "synodos" to my dear wife at the Heraklion Venizelou Hospital. You are probably wondering what a "synodos" entails? Let me explain and you will learn a little bit about Greek culture. In South Africa, hospitals have two categories of assistants to the doctors. Right below the doctor is the "sister". These have medical degrees and can do a surprising number of tasks, such as take the blood pressure, dress wounds, check your heart, save your life, etc. As a child, I preferred injections from a sister rather than from a doctor as they had more practice and were generally more skilled at those tasks. Below the sisters are the "nurses". These would also have some medical training I think, but not nearly as much as the sisters. They do the jucky stuff, like washing the patients, changing bedpans, etc. Don't ever ever ever call a sister a "nurse" in South Africa or your next injection will be painful ;-)

In Greece, the hospitals do not employ nurses, but only sisters. You will get the essential medical help, but that's it. So without nurses, how do the old frail people get to the bathrooms? This is where the "synodos" comes in. Every patient has at least one "synodos" sitting with them 24/7. This would be a relative or a friend. The word "synodos" is a companion or an escort. "Odos" means street, so the literal meaning is probably someone who walks the road with you. Beautiful. In most hospitals around the world, there are specific visiting hours. We had un-visiting hours! Whenever the doctors or sisters needed to do something to one of the patients, they shooed all of us out onto the balcony.

At first, this was a huge culture shock to us. Imagine a room with 8 patients, plus another 12 "synodos", all crammed together like sardines! But having experienced it all, I must say that there are some upsides to this system. Time just flies by, because you are constantly chatting to people. You don't even need a television. Plus it was heartening seeing these old

ladies surrounded by people that loved them, 24/7. The lady next to us had her son-in-law sitting next to her for several days. One lady spent 3 days and nights without sleep sitting next to a friend. Here you don't get a call in the morning "Oh, your mother passed away last night." For someone getting old, what better way, than to have excellent medical care (Venizelou is fantastic) and the companionship of those that you love? Please tell me if you have something like the "synodos" system in your country.

ByteWatcher from JCrete

After my [newsletter on String's new substring mechanism](#), Daniel Shaya wrote a series of articles on the topic of allocation. This eventually resulted in a tool we named "ByteWatcher". Daniel and I got together in a [JCrete](#) session to brainstorm and start this article together. The idea was to write and publish it in one hour. We did not manage :-) However, I hope that the result is better than what we would've given you in such a hurry.

Regression testing is important. Most projects these days are filled to the brim with unit tests. Often the amount of code in the test branch will be far more than the code in the main branch. We usually test:

1. Correctness: We make sure that no fence-post, null pointer and similar bugs have crept into our code base. We also test that the code fulfills the contracts of what it is supposed to do.
2. Performance: Most software is expected to finish within a certain amount of time. This might be explicitly stated or assumed. For example, if you open IntelliJ IDEA, you expect that to happen within a few seconds. You would also expect a large drive to take an hour or more to fully back up. A trade might need to happen in under 10 microseconds. It is generally a good idea to state what these expectations are and to have tests that check that we do not exceed them.
3. Size Matters (sometimes): With ubiquitous storage becoming the norm, size of a distribution is not nearly as important as it used to be. Some might include some tests to make sure this does not get out of hand though.

But there is another attribute that has been difficult to test in the past: How many bytes are allocated on the heap by this function?

Java is not real-time. And despite the common misconception, real-time also doesn't mean fast. It means that we are predictable in our slowness :-) However, we do use Java in applications where we want to be fast and have as predictable latencies as possible, right up to the high percentiles. These latencies get blown out of the water when a GC event occurs. For this and many other reasons, such applications seek to avoid allocating objects on the heap. With ByteWatcher we can add a couple of lines into our regression tests that will check whether the expected allocation has been exceeded or not. Let's take a method that only works with stack variables:

```
public int methodThatDoesNotAllocateAnything() {
    int x = 42;
    for (int i = 0; i < 10000; i++) {
        x += i % 10;
        x /= i % 2 + 1;
    }
    return x;
}
```

To test this really does not allocate anything, we can use the ByteWatcherRegressionTestHelper:

```
private final ByteWatcherRegressionTestHelper helper =
    new ByteWatcherRegressionTestHelper();

@Test
public void testNoAllocationMethod() {
    helper.testAllocationNotExceeded(
        this::methodThatDoesNotAllocateAnything,
        0 // no allocation!
    );
}
```

On the other hand, if we want to test that a byte[] allocation does not exceed our expected values, we could test it like this:

```
@Test
public void testByteArrayAllocation() {
    System.out.println(methodThatDoesNotAllocateAnything());
    helper.testAllocationNotExceeded(
        () -> {
            byte[] data = new byte[100];
        },
        120 // 100 bytes, plus the object header, plus int length
    );
}
```

```
}
```

The ByteWatcherRegressionTestHelper is fairly trivial and relies on the ByteWatcherSingleThread. It can be created for either the current thread or any other arbitrary thread. In most cases, the unit test would probably be monitoring the current thread in order to make the test more predictable.

```
import static org.junit.Assert.*;

public class ByteWatcherRegressionTestHelper {
    private final ByteWatcherSingleThread bw;

    public ByteWatcherRegressionTestHelper(Thread thread) {
        bw = new ByteWatcherSingleThread(thread);
    }

    public ByteWatcherRegressionTestHelper() {
        this(Thread.currentThread());
    }

    public void testAllocationNotExceeded(
        Runnable job, long limit) {
        bw.reset();
        job.run();
        long size = bw.calculateAllocations();
        assertTrue(String.format("exceeded limit: %d using: %d%n",
            limit, size), size <= limit);
    }
}
```

The code for ByteWatcherSingleThread is a combined effort by [Daniel Shaya](#), [Chris Newland \(Mr JITWatch\)](#) and myself. One of the features is that we try to figure out how many bytes are wasted when we *measure* how many bytes are wasted :-) Usually this number is 336, but if the JIT compiler runs whilst we are measuring, then it could be far more. We thus calibrate on start-up, by calling `threadAllocatedBytes()` 10000 times, sleeping for a bit, then doing that all 10 times. Hopefully by the time that the constructor returns, the code would've been all compiled and would thus not turn on the JIT Compiler on that thread during the test. The rest of the code is fairly self-explanatory. We simply grab the property `getThreadAllocatedBytes` from the `ThreadMXBean`, like I showed in my [newsletter on String.substring\(\)](#).

```

import javax.management.*;
import java.lang.management.*;
import java.util.concurrent.atomic.*;

/**
 * A class to measure how much allocation there has been on an
 * individual thread. The class would be useful to embed into
 * regression tests to make sure that there has been no
 * unintended allocation.
 */
public class ByteWatcherSingleThread {
    private static final String ALLOCATED = " allocated ";
    private static final String GET_THREAD_ALLOCATED_BYTES =
        "getThreadAllocatedBytes";
    private static final String[] SIGNATURE =
        new String[]{long.class.getName()};
    private static final MBeanServer mBeanServer;
    private static final ObjectName name;

    private final String threadName;
    private final Thread thread;

    private final Object[] PARAMS;
    private final AtomicLong allocated = new AtomicLong();
    private final long MEASURING_COST_IN_BYTES; // usually 336
    private final long tid;
    private final boolean checkThreadSafety;

    static {
        try {
            name = new ObjectName(
                ManagementFactory.THREAD_MXBEAN_NAME);
            mBeanServer = ManagementFactory.getPlatformMBeanServer();
        } catch (MalformedObjectNameException e) {
            throw new ExceptionInInitializerError(e);
        }
    }

    public ByteWatcherSingleThread() {
        this(Thread.currentThread(), true);
    }

    public ByteWatcherSingleThread(Thread thread) {
        this(thread, false);
    }
}

```

```

private ByteWatcherSingleThread(
    Thread thread, boolean checkThreadSafety) {
    this.checkThreadSafety = checkThreadSafety;
    this.tid = thread.getId();
    this.thread = thread;
    threadName = thread.getName();
    PARAMS = new Object[]{tid};

    long calibrate = threadAllocatedBytes();
    // calibrate
    for (int repeats = 0; repeats < 10; repeats++) {
        for (int i = 0; i < 10_000; i++) {
            // run a few loops to allow for startup anomalies
            calibrate = threadAllocatedBytes();
        }
        try {
            Thread.sleep(50);
        } catch (InterruptedException e) {
            Thread.currentThread().interrupt();
            break;
        }
    }
    MEASURING_COST_IN_BYTES = threadAllocatedBytes() - calibrate;
    reset();
}

public long getMeasuringCostInBytes() {
    return MEASURING_COST_IN_BYTES;
}

public void reset() {
    checkThreadSafety();

    allocated.set(threadAllocatedBytes());
}

long threadAllocatedBytes() {
    try {
        return (long) mBeanServer.invoke(
            name,
            GET_THREAD_ALLOCATED_BYTES,
            PARAMS,
            SIGNATURE
        );
    } catch (Exception e) {
        throw new IllegalArgumentException(e);
    }
}

```

```

/**
 * Calculates the number of bytes allocated since the last
 * reset().
 */
public long calculateAllocations() {
    checkThreadSafety();
    long mark1 = ((threadAllocatedBytes() -
        MEASURING_COST_IN_BYTES) - allocated.get());
    return mark1;
}

private void checkThreadSafety() {
    if (checkThreadSafety &&
        tid != Thread.currentThread().getId())
        throw new IllegalStateException(
            "AllocationMeasure must not be " +
            "used over more than 1 thread.");
}

public Thread getThread() {
    return thread;
}

public String toString() {
    return thread.getName() + ALLOCATED + calculateAllocations();
}
}

```

We also wrote a ByteWatcher, which you can use to monitor specific threads in your applications. The ByteWatcher would not work well for unit testing, since the call-back would come from a thread managed within the ByteWatcher. An assertion failure would thus only serve to stop the watching task. We need to be careful to never throw exceptions inside our callback code!

```

import java.util.*;
import java.util.concurrent.*;
import java.util.function.*;
import java.util.stream.*;

/**
 * Created by daniel on 22/07/2015.
 * This class allows a user to receive callbacks if

```

```

* threads are destroyed or created.
* Its primary function is to alert the user if any
* thread has exceeded a specified amount of allocation.
*/

public class ByteWatcher {
    public static final int SAMPLING_INTERVAL =
        Integer.getInteger("samplingIntervalMillis", 500);
    public static final Consumer<Thread> EMPTY = a -> { };
    public static final BiConsumer<Thread, Long> BI_EMPTY =
        (a, b) -> { };
    private final Map<Thread, ByteWatcherSingleThread> ams;
    private volatile Consumer<Thread> threadCreated = EMPTY;
    private volatile Consumer<Thread> threadDied =
        EMPTY;
    private volatile ByteWatch byteWatch = new ByteWatch(
        BI_EMPTY, Long.MAX_VALUE
    );

    private static class ByteWatch
        implements BiConsumer<Thread, Long>, Predicate<Long>{
        private final long threshold;
        private final BiConsumer<Thread, Long> byteWatch;

        public ByteWatch(BiConsumer<Thread, Long> byteWatch,
                          long threshold) {
            this.byteWatch = byteWatch;
            this.threshold = threshold;
        }

        public void accept(Thread thread, Long currentBytes) {
            byteWatch.accept(thread, currentBytes);
        }

        public boolean test(Long currentBytes) {
            return threshold < currentBytes;
        }
    }

    private final ScheduledExecutorService monitorService =
        Executors.newSingleThreadScheduledExecutor();

    public ByteWatcher() {
        // do this first so that the worker thread is not considered
        // a "newly created" thread
        monitorService.scheduleAtFixedRate(
            this::checkThreads,
            SAMPLING_INTERVAL, SAMPLING_INTERVAL,
            TimeUnit.MILLISECONDS);
    }
}

```

```

ams = Thread.getAllStackTraces()
    .keySet()
    .stream()
    .map(ByteWatcherSingleThread::new)
    .collect(Collectors.toConcurrentMap(
        ByteWatcherSingleThread::getThread,
        (ByteWatcherSingleThread am) -> am));
// Heinz: Streams make sense, right? ;-
}

public void onThreadCreated(Consumer<Thread> action) {
    threadCreated = action;
}

public void onThreadDied(Consumer<Thread> action) {
    threadDied = action;
}

public void onByteWatch(
    BiConsumer<Thread, Long> action, long threshold) {
    this.byteWatch = new ByteWatch(action, threshold);
}

public void shutdown() {
    monitorService.shutdown();
}

public void forEach(Consumer<ByteWatcherSingleThread> c) {
    ams.values().forEach(c);
}

public void printAllAllocations() {
    forEach(System.out::println);
}

public void reset() {
    forEach(ByteWatcherSingleThread::reset);
}

private void checkThreads() {
    Set<Thread> oldThreads = ams.keySet();
    Set<Thread> newThreads = Thread.getAllStackTraces().keySet();

    Set<Thread> diedThreads = new HashSet<>(oldThreads);
    diedThreads.removeAll(newThreads);

    Set<Thread> createdThreads = new HashSet<>(newThreads);
}

```

```

createdThreads.removeAll(oldThreads);

diedThreads.forEach(this::threadDied);
createdThreads.forEach(this::threadCreated);
ams.values().forEach(this::bytesWatch);
}

private void threadCreated(Thread t) {
    ams.put(t, new ByteWatcherSingleThread(t));
    threadCreated.accept(t);
}

private void threadDied(Thread t) {
    threadDied.accept(t);
}

private void bytesWatch(ByteWatcherSingleThread am) {
    ByteWatch bw = byteWatch;
    long bytesAllocated = am.calculateAllocations();
    if (bw.test(bytesAllocated)) {
        bw.accept(am.getThread(), bytesAllocated);
    }
}
}

```

The GIT project that Daniel and I used to build this code can be found [here](#).

Related Projects

After I sent out this newsletter, I was pointed to two similar projects: [alloccheck](#), which uses try-with-resource as a wrapper for checking allocation. Brian's mechanism also counts number of objects created, not just bytes:

```

public void doSomeWork() {
    try (Alloccheck _ = new Alloccheck(1, 1)) {
        for (int i = 0; i < 5; i++) {
            new Object().hashCode();
            Integer.parseInt(String.valueOf(i));
        }
    }
}

```

Matthew Painter wrote a Java Agent that tracks allocation directly from the byte code. See his project [sio2box](#).

Thank you so much for reading this and I hope this will be a useful addition to your testing toolbox.

Kind regards from a sunny Crete

Heinz

Issue 230 - String Substring

Author: Dr. Heinz M. Kabutz

Date: 2015-06-30

Category: Tips and Tricks

Java Versions: Java 1-9

Abstract:

Java 7 quietly changed the structure of String. Instead of an offset and a count, the String now only contained a char[]. This had some harmful effects for those expecting substring() would always share the underlying char[].

Welcome to the 230th edition of **The Java(tm) Specialists' Newsletter**, written on the Island of Crete in GREECE. By now, you would have heard about the ATMs drying up. It is surprisingly calm here. Some petrol stations ran out of gas as my fellow Cretans decided for the first time in years to completely fill their tanks. But that was resolved within a day. ATMs are issuing only 60 EURO to us locals per day (no limit for visitors), but there seems to be no limit inside the supermarkets and restaurants. It's also easy to find those ATMs with money - just check for a bunch of people standing in a queue. As my one friend said - thanks to these lines of people, he's now discovering ATMs that he didn't even know existed ;-) Chania seems as deserted as it is during winter, with a noticeable reduction in tourists. *This is the best possible time for you to come visit Crete!* The Cretan hospitality is shining through even more than usual. If you can speak Greek and you take the time to sit with a grandfather in his 70s, you will hear the reality of what he's been going through. As a tourist, you won't see any of that. You'll just be invited to drink a glass of tsikoudia with a hearty smack on your back. You can probably find excellent deals at this time with flights and hotels. The beaches are exactly the same as last year, albeit with less drunken louts. The food is still delicious. The weather fine and warm. Come. You won't regret it. And by the way, Chania is the best part of Crete, especially the Akrotiri :-)

String Substring

String is ubiquitous in Java programs. It has changed in quite a few ways over the last generations of Java. For example, in very early versions, the generated code of appending several non-constant Strings together would either be a call to concat() or a StringBuffer. In Java 1.0 and 1.1, the hashCode() function would check the size of the String and if it was too long, would add up every 8th character instead of every one. Of course, considering memory layout, that optimization would not have been all that effective anyway. In Java 2, they changed that to every character always and in Java 3, they cached the hash code. Whilst this sounds sensible, it wasn't. There are almost no cases where it helps in real code and it introduces an assumption that the hash code is unlikely to be zero. It isn't. Once you find one combination of characters that has a zero hash code, you can produce an arbitrary long series. A constant time operation like hashCode() using the cached value now potentially becomes O(n). They tried to fix this in Java 7 with the hash32() calculation, which never would allow a zero value. However, I see that is also gone again in Java 8.

Recently, my co-trainer Maurice Naftalin (author of Mastering Lambdas) and I taught our **Extreme Java 8** course together, which focuses on concurrency and performance. I always spend a bit of time on String, as it is used so much and does tend to appear near the top of many a profile. From Java 1.0 up to 1.6, String tried to avoid creating new char[]'s. The substring() method would share the same underlying char[], with a different offset and length. For example, in StringChars we have two Strings, with "hello" a substring of "hello_world". However, they share the same char[]:

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

public class StringChars {
    public static void main(String... args)
        throws NoSuchFieldException, IllegalAccessException {
        Field value = String.class.getDeclaredField("value");
        value.setAccessible(true);

        String hello_world = "Hello world";
        String hello = hello_world.substring(0, 5);
        System.out.println(hello);

        System.out.println(value.get(hello_world));
        System.out.println(value.get(hello));
    }
}
```

In Java 1 through 6, we would see output like this:

```
Hello
[C@721cdeff
[C@721cdeff
```

However, in Java 7 and 8, it would instead produce output with a different char[]:

```
Hello
```

[C@49476842
[C@78308db1

"Why this change?", you may ask. It turns out that too many programmers used `substring()` as a memory saving method. Let's say that you have a 1 MB String, but you actually only need the first 5 KB. You could then create a substring, expecting the rest of that 1 MB String to be thrown away. Except it didn't. Since the new String would share the same underlying `char[]`, you would not save any memory at all. The correct code idiom was therefore to append the substring to an empty String, which would have the side effect of *always* producing a new unshared `char[]` in the case that the String length did not correspond to the `char[]` length:

```
String hello = "" + hello_world.substring(0, 5);
```

During our course, the customer remarked that they had a real issue with this new Java 7 and 8 approach to substrings. In the past they assumed that a substring would generate a minimum of garbage, whereas nowadays the cost can be quite high. In order to measure how many bytes exactly are being allocated, I wrote a little Memory class that uses a little-known ThreadMXBean feature. The details will be the subject of another newsletter:

```
import javax.management.*;
import java.lang.management.*;

public class Memory {
    public static long threadAllocatedBytes() {
        try {
            return (Long) ManagementFactory.getPlatformMBeanServer()
                .invoke(
                    new ObjectName(
                        ManagementFactory.THREAD_MXBEAN_NAME),
                    "getThreadAllocatedBytes",
                    new Object[]{Thread.currentThread().getId()},
                    new String[]{long.class.getName()});
        }
        } catch (Exception e) {
            throw new IllegalArgumentException(e);
        }
    }
```

```
}
```

Let's say that I have a large string that I would like to break up into smaller chunks:

```
import java.util.*;

public class LargeString {
    public static void main(String... args) {
        char[] largeText = new char[10 * 1000 * 1000];
        Arrays.fill(largeText, 'A');
        String superString = new String(largeText);

        long bytes = Memory.threadAllocatedBytes();
        String[] subStrings = new String[largeText.length / 1000];
        for (int i = 0; i < subStrings.length; i++) {
            subStrings[i] = superString.substring(
                i * 1000, i * 1000 + 1000);
        }
        bytes = Memory.threadAllocatedBytes() - bytes;
        System.out.printf("%,d%n", bytes);
    }
}
```

In Java 6, the LargeString class generates 360,984 bytes, but in Java 7, it goes up to a whopping 20,441,536 bytes. That's quite a jump! You can run this code yourself to try out on your machine.

Unfortunately if we want to have the memory allocation saving of Java 6, we need to write our own String class. Fortunately that is not too hard with the CharSequence interface. Please note that my SubbableString is *not* thread safe, nor is it meant to be. I used Brian Goetz's annotation, albeit in a comment:

```
//@NotThreadSafe
public class SubbableString implements CharSequence {
    private final char[] value;
    private final int offset;
    private final int count;
```

```

public SubbableString(char[] value) {
    this(value, 0, value.length);
}

private SubbableString(char[] value, int offset, int count) {
    this.value = value;
    this.offset = offset;
    this.count = count;
}

public int length() {
    return count;
}

public String toString() {
    return new String(value, offset, count);
}

public char charAt(int index) {
    if (index < 0 || index >= count)
        throw new StringIndexOutOfBoundsException(index);
    return value[index + count];
}

public CharSequence subSequence(int start, int end) {
    if (start < 0) {
        throw new StringIndexOutOfBoundsException(start);
    }
    if (end > count) {
        throw new StringIndexOutOfBoundsException(end);
    }
    if (start > end) {
        throw new StringIndexOutOfBoundsException(end - start);
    }
    return (start == 0 && end == count) ? this :
        new SubbableString(value, offset + start, end - start);
}
}

```

If we now use CharSequence instead of String in the test, we can avoid creating all those unnecessary char[]s. Here is the revised test:

```
import java.util.*;  
  
public class LargeSubbableString {  
    public static void main(String... args) {  
        char[] largeText = new char[10000000];  
        Arrays.fill(largeText, 'A');  
        CharSequence superString = new SubbableString(largeText);  
  
        long bytes = Memory.threadAllocatedBytes();  
        CharSequence[] subStrings = new CharSequence[  
            largeText.length / 1000];  
        for (int i = 0; i < subStrings.length; i++) {  
            subStrings[i] = superString.subSequence(  
                i * 1000, i * 1000 + 1000);  
        }  
        bytes = Memory.threadAllocatedBytes() - bytes;  
        System.out.printf("%,d%n", bytes);  
    }  
}
```

With that improvement, we now use roughly 281000 bytes on Java 6, 7 and 8. For Java 7 and 8, that would be a 72x improvement!

Please keep this new "feature" in mind when you do your migration from Java 6 to Java 8. I know, too many of my customers are stuck on 6 and are finding it hard to find a business case for funding the move. Besides the syntactic advantages in Java 7 and 8, you will also want to move away from the bugs still stuck in Java 6. The sooner the better!

Kind regards

Heinz

Issue 229 - Cleaning ThreadLocals

Author: Dr. Heinz M. Kabutz

Date: 2015-05-23

Category: Tips and Tricks

Java Versions: Java 8

Abstract:

ThreadLocals should in most cases be avoided. They can solve some tough problems, but can introduce some nasty memory leaks, especially if the ThreadLocal class or value refer to our own classes, not a system class. In this newsletter we show some mechanisms that help under OpenJDK.

Welcome to the 229th edition of **The Java(tm) Specialists' Newsletter**, written mostly in Düsseldorf in Germany. Düsseldorf is mentioned in the 2005 rendition of Charlie in the Chocolate Factory. If you saw the movie, you would probably imagine Düsseldorf as this quaint German town in the mountains. Anything but! It is completely flat. Most of the buildings are modern, as 64% of the city was destroyed by 700 tons of bombs in 1942. But the modern Düsseldorf pulses with energy. Lots of young people live here and the bars are very active indeed. We've done many courses here (in German) and I keep coming back. "Fächerchen" is a firm favourite.

Cleaning ThreadLocals

In most of my courses ([master](#), [concurrency](#), [xj-conc-j8](#)), I mention ThreadLocal. It is always accompanied by stern warnings that we should avoid it if possible. ThreadLocal is designed for threads that are used and then discarded. Before the thread exits, all its thread local entries are erased. In fact, if you read [Why 0x61c88647?](#), it will explain how the actual values are stored inside a map that lives within the thread. Threads that live in thread pools generally survive a single user request, thus making them prone for memory leaks. Usually when I talk about these, at least one of my students has a story of how their production system collapsed due to a thread local hanging onto one of their classes, thus preventing the entire class loader from being unloaded. This problem is extremely common.

In this newsletter, I would like to demonstrate a ThreadLocalCleaner class that can restore the thread locals to their former glory when a thread is ready to go back into the pool. At the most basic level, we can save the state of the ThreadLocals for our current thread and then restore them again later. We would typically do this with the try-with-resource construct that was added in Java 7. For example:

```
try (ThreadLocalCleaner tlc = new ThreadLocalCleaner()) {
```

```
// some code that potentially creates and adds thread locals
}
// at this point, the new thread locals have been cleared
```

To make it easier to debug our systems, we also add an observer mechanism, so that we can be notified of any changes made to the thread local map before restoring it. This will help us discover possible thread local leaks. Here is our listener:

```
package threadcleaner;

@FunctionalInterface
public interface ThreadLocalChangeListener {
    void changed(Mode mode, Thread thread,
                 ThreadLocal<?> threadLocal, Object value);

    ThreadLocalChangeListener EMPTY = (m, t, tl, v) -> {};
}

ThreadLocalChangeListener PRINTER =
    (m, t, tl, v) -> System.out.printf(
        "Thread %s %s ThreadLocal %s with value %s%n",
        t, m, tl.getClass(), v);

enum Mode {
    ADDED, REMOVED
}
```

Some explanations might be necessary. First off, I have marked it as a `@FunctionalInterface`, which in Java 8 means it has exactly one abstract method and can be used for a lambda. Secondly, I have defined an `EMPTY` lambda internally. As you can see, the code for that is minimal. Thirdly, we also have a `PRINTER` default, which simply prints the change to `System.out`. Lastly, we also have two different events, but since I wanted to design a `@FunctionalInterface`, I had to define the marker as a separate attribute, in this case an enum.

When we construct our `ThreadLocalCleaner`, we can also pass in a `ThreadLocalChangeListener` and this will be notified of any changes that might have occurred since we created our `ThreadLocalCleaner`. Please note that these mechanisms only are applied to the current thread. Here is an example of how we could use the `ThreadLocalCleaner` in a try-with-resource block of code: Any local variable defined in the try

(...) will be automatically closed at the end of the block of code. We thus have a close() method inside our ThreadLocalCleaner that restores the thread local entries to their previous values.

```

import java.text.*;

public class ThreadLocalCleanerExample {
    private static final ThreadLocal<DateFormat> df =
        new ThreadLocal<DateFormat>() {
            protected DateFormat initialValue() {
                return new SimpleDateFormat("yyyy-MM-dd");
            }
        };

    public static void main(String... args) {
        System.out.println("First ThreadLocalCleaner context");
        try (ThreadLocalCleaner tlc = new ThreadLocalCleaner(
            ThreadLocalChangeListener.PRINTER)) {
            System.out.println(System.identityHashCode(df.get()));
            System.out.println(System.identityHashCode(df.get()));
            System.out.println(System.identityHashCode(df.get()));
        }

        System.out.println("Another ThreadLocalCleaner context");
        try (ThreadLocalCleaner tlc = new ThreadLocalCleaner(
            ThreadLocalChangeListener.PRINTER)) {
            System.out.println(System.identityHashCode(df.get()));
            System.out.println(System.identityHashCode(df.get()));
            System.out.println(System.identityHashCode(df.get()));
        }
    }
}

```

We also have two public static methods in our ThreadLocalCleaner class: forEach() and cleanup(Thread). The forEach() method takes as parameters a thread and a BiConsumer, which is then called with the ThreadLocal and its value for each entry. We skip entries with null keys, but not those with null values. The reason is that a ThreadLocal can still cause a memory leak, even if the value is null. Once we are done with using the ThreadLocal, we should always call the remove() method before letting the thread return to the pool. The cleanup(Thread) method sets the ThreadLocal map in that thread to null, thus allowing all of the entries to be garbage collected. If a ThreadLocal is used again after we have cleared it, the initialValue() method will simply be called to recreate the entry. Here are the method definitions:

```
public static void forEach(Thread thread,
    BiConsumer<ThreadLocal<?>, Object> consumer) { ... }

public static void cleanup(Thread thread) { ... }
```

The complete code of the ThreadLocalCleaner class is here. It uses a lot of reflection on private fields. It will probably only work with the OpenJDK or direct derivatives. You will also notice that I am using Java 8 for the syntax. I debated myself for a while whether I should use Java 8 or 7. Some of my clients are still on 1.4. In the end, most of my large banking customers are already using Java 8 in production. Banks are usually not the first adopters, except in the case where it makes a lot of financial sense. Thus if you're not using Java 8 in production yet, you should probably look at migrating soon, or even skipping it and going straight to Java 9. You could easily backport this to Java 7 by writing your own BiConsumer interface. Java 6 does not have the nice try-with-resource construct, so I would not go that far back.

```
package threadcleaner;

import java.lang.ref.*;
import java.lang.reflect.*;
import java.util.*;
import java.util.function.*;

import static threadcleaner.ThreadLocalChangeListener.Mode.*;

public class ThreadLocalCleaner implements AutoCloseable {
    private final ThreadLocalChangeListener listener;

    public ThreadLocalCleaner() {
        this(ThreadLocalChangeListener.EMPTY);
    }

    public ThreadLocalCleaner(ThreadLocalChangeListener listener) {
        this.listener = listener;
        saveOldThreadLocals();
    }

    public void close() {
        cleanup();
    }
}
```

```

public void cleanup() {
    diff(threadLocalsField, copyOfThreadLocals.get());
    diff(inheritableThreadLocalsField,
        copyOfInheritableThreadLocals.get());
    restoreOldThreadLocals();
}

public static void forEach(
    Thread thread,
    BiConsumer<ThreadLocal<?>, Object> consumer) {
    forEach(thread, threadLocalsField, consumer);
    forEach(thread, inheritableThreadLocalsField, consumer);
}

public static void cleanup(Thread thread) {
    try {
        threadLocalsField.set(thread, null);
        inheritableThreadLocalsField.set(thread, null);
    } catch (IllegalAccessException e) {
        throw new IllegalStateException(
            "Could not clear thread locals: " + e);
    }
}

private void diff(Field field, Reference<?>[] backup) {
    try {
        Thread thread = Thread.currentThread();
        Object threadLocals = field.get(thread);
        if (threadLocals == null) {
            if (backup != null) {
                for (Reference<?> reference : backup) {
                    changed(thread, reference,
                        REMOVED);
                }
            }
            return;
        }

        Reference<?>[] current =
            (Reference<?>[]) tableField.get(threadLocals);
        if (backup == null) {
            for (Reference<?> reference : current) {
                changed(thread, reference, ADDED);
            }
        } else {
            // nested loop - both arrays *should* be relatively small
            next:
            for (Reference<?> curRef : current) {

```

```

    if (curRef != null) {
        if (curRef.get() == copyOfThreadLocals ||
            curRef.get() == copyOfInheritableThreadLocals) {
            continue next;
        }
        for (Reference<?> backupRef : backup) {
            if (curRef == backupRef) continue next;
        }
        // could not find it in backup - added
        changed(thread, curRef, ADDED);
    }
}

next:
for (Reference<?> backupRef : backup) {
    for (Reference<?> curRef : current) {
        if (curRef == backupRef) continue next;
    }
    // could not find it in current - removed
    changed(thread, backupRef,
        REMOVED);
}
}

} catch (IllegalAccessException e) {
    throw new IllegalStateException("Access denied", e);
}
}

private void changed(Thread thread, Reference<?> reference,
                    ThreadLocalChangeListener.Mode mode)
throws IllegalAccessException {
    listener.changed(mode,
        thread, (ThreadLocal<?>) reference.get(),
        threadLocalEntryValueField.get(reference));
}

private static Field field(Class<?> c, String name)
throws NoSuchFieldException {
    Field field = c.getDeclaredField(name);
    field.setAccessible(true);
    return field;
}

private static Class<?> inner(Class<?> clazz, String name) {
    for (Class<?> c : clazz.getDeclaredClasses()) {
        if (c.getSimpleName().equals(name)) {
            return c;
        }
    }
}

```

```

throw new IllegalStateException(
    "Could not find inner class " + name + " in " + clazz);
}

private static void forEach(
    Thread thread, Field field,
    BiConsumer<ThreadLocal<?>, Object> consumer) {
try {
    Object threadLocals = field.get(thread);
    if (threadLocals != null) {
        Reference<?>[] table = (Reference<?>[])
            tableField.get(threadLocals);
        for (Reference<?> ref : table) {
            if (ref != null) {
                ThreadLocal<?> key = (ThreadLocal<?>) ref.get();
                if (key != null) {
                    Object value = threadLocalEntryValueField.get(ref);
                    consumer.accept(key, value);
                }
            }
        }
    }
} catch (IllegalAccessException e) {
    throw new IllegalStateException(e);
}
}

private static final ThreadLocal<Reference<?>[]>
copyOfThreadLocals = new ThreadLocal<>();

private static final ThreadLocal<Reference<?>[]>
copyOfInheritableThreadLocals = new ThreadLocal<>();

private static void saveOldThreadLocals() {
    copyOfThreadLocals.set(copy(threadLocalsField));
    copyOfInheritableThreadLocals.set(
        copy(inheritableThreadLocalsField));
}

private static Reference<?>[] copy(Field field) {
try {
    Thread thread = Thread.currentThread();
    Object threadLocals = field.get(thread);
    if (threadLocals == null) return null;
    Reference<?>[] table =
        () tableField.get(threadLocals);
    return Arrays.copyOf(table, table.length);
} catch (IllegalAccessException e) {
}
}

```

```

        throw new IllegalStateException("Access denied", e);
    }
}

private static void restoreOldThreadLocals() {
    try {
        restore(inheritableThreadLocalsField,
            copyOfInheritableThreadLocals.get());
        restore(threadLocalsField, copyOfThreadLocals.get());
    } finally {
        copyOfThreadLocals.remove();
        copyOfInheritableThreadLocals.remove();
    }
}

private static void restore(Field field, Object value) {
    try {
        Thread thread = Thread.currentThread();
        if (value == null) {
            field.set(thread, null);
        } else {
            tableField.set(field.get(thread), value);
        }
    } catch (IllegalAccessException e) {
        throw new IllegalStateException("Access denied", e);
    }
}

/* Reflection fields */

private static final Field threadLocalsField;

private static final Field inheritableThreadLocalsField;
private static final Class<?> threadLocalMapClass;
private static final Field tableField;
private static final Class<?> threadLocalMapEntryClass;

private static final Field threadLocalEntryValueField;

static {
    try {
        threadLocalsField = field(Thread.class, "threadLocals");
        inheritableThreadLocalsField =
            field(Thread.class, "inheritableThreadLocals");

        threadLocalMapClass =
            inner(ThreadLocal.class, "ThreadLocalMap");
    }
}

```

```

tableField = field(threadLocalMapClass, "table");
threadLocalMapEntryClass =
    inner(threadLocalMapClass, "Entry");

threadLocalEntryValueField =
    field(threadLocalMapEntryClass, "value");
} catch (NoSuchFieldException e) {
    throw new IllegalStateException(
        "Could not locate threadLocals field in Thread. " +
        "Will not be able to clear thread locals: " + e);
}
}
}
}

```

Here is an example of the ThreadLocalCleaner in action:

```

import java.text.*;

public class ThreadLocalCleanerExample {
    private static final ThreadLocal<DateFormat> df =
        new ThreadLocal<DateFormat>() {
            protected DateFormat initialValue() {
                return new SimpleDateFormat("yyyy-MM-dd");
            }
        };

    public static void main(String... args) {
        System.out.println("First ThreadLocalCleaner context");
        try (ThreadLocalCleaner tlc = new ThreadLocalCleaner(
            ThreadLocalChangeListener.PRINTER)) {
            System.out.println(System.identityHashCode(df.get()));
            System.out.println(System.identityHashCode(df.get()));
            System.out.println(System.identityHashCode(df.get()));
        }

        System.out.println("Another ThreadLocalCleaner context");
        try (ThreadLocalCleaner tlc = new ThreadLocalCleaner(
            ThreadLocalChangeListener.PRINTER)) {
            System.out.println(System.identityHashCode(df.get()));
            System.out.println(System.identityHashCode(df.get()));
            System.out.println(System.identityHashCode(df.get()));
        }
    }
}

```

```
}
```

Your output could potentially contain different system hash codes, though remember please from my [Identity Crisis Newsletter](#) that the algorithm for these numbers is just a random number generator. Here is my output. Notice how whilst we are inside the try-with-resource body, the thread local value was the same:

```
First ThreadLocalCleaner context
186370029
186370029
186370029
Thread Thread[main,5,main] ADDED ThreadLocal class \
    ThreadLocalCleanerExample$1 with value \
        java.text.SimpleDateFormat@f67a0200
Another ThreadLocalCleaner context
2094548358
2094548358
2094548358
Thread Thread[main,5,main] ADDED ThreadLocal class \
    ThreadLocalCleanerExample$1 with value \
        java.text.SimpleDateFormat@f67a0200
```

In order to make the code a bit easier to use, I have written a Facade. The Facade Design Pattern is not meant to stop users from using the subsystem directly, but are supposed to provider a simpler interface to a complicated system. Typically you provide the most common use cases of the subsystem as methods. Our Facade contains two methods: `findAll(Thread)` and `printThreadLocals()`. The `findAll()` method returns a Collection of entries found inside that thread.

```
package threadcleaner;

import java.io.*;
import java.lang.ref.*;
import java.util.AbstractMap.*;
import java.util.*;
import java.util.Map.*;
import java.util.function.*;
```

```

import static threadcleaner.ThreadLocalCleaner.*;

public class ThreadLocalCleaners {
    public static Collection<Entry<ThreadLocal<?>, Object>> findAll(
        Thread thread) {
        Collection<Entry<ThreadLocal<?>, Object>> result =
            new ArrayList<>();
        BiConsumer<ThreadLocal<?>, Object> adder =
            (key, value) ->
                result.add(new SimpleImmutableEntry<>(key, value));
        forEach(thread, adder);
        return result;
    }

    public static void printThreadLocals() {
        printThreadLocals(System.out);
    }

    public static void printThreadLocals(Thread thread) {
        printThreadLocals(thread, System.out);
    }

    public static void printThreadLocals(PrintStream out) {
        printThreadLocals(Thread.currentThread(), out);
    }

    public static void printThreadLocals(Thread thread,
                                         PrintStream out) {
        out.println("Thread " + thread.getName());
        out.println(" ThreadLocals");
        printTable(thread, out);
    }

    private static void printTable(
        Thread thread, PrintStream out) {
        forEach(thread, (key, value) -> {
            out.printf(" %s,%s", key, value);
            if (value instanceof Reference) {
                out.print("->" + ((Reference<?>) value).get());
            }
            out.println("}");
        });
    }
}

```

Thread can contain two different types of ThreadLocal: the normal one and inheritable. In

almost all cases, we use the normal one. Inheritable means that if you construct a new thread from your current thread, then all the inheritable ThreadLocals in the creating thread are inherited by the new thread. Yeah. I agree with you. Use cases are rather scarce. So just forget about this for now. Or forever :-)

An obvious use case of this ThreadLocalCleaner is with the ThreadPoolExecutor. We subclass it and override the methods beforeExecute() and afterExecute(). The class is long because we have to code all the constructors. The interesting bit is right at the end.

```
package threadcleaner;

import java.util.concurrent.*;

public class ThreadPoolExecutorExt extends ThreadPoolExecutor {
    private final ThreadLocalChangeListener listener;

    /* Bunch of constructors following - you can ignore those */

    public ThreadPoolExecutorExt(
        int corePoolSize, int maximumPoolSize, long keepAliveTime,
        TimeUnit unit, BlockingQueue<Runnable> workQueue) {
        this(corePoolSize, maximumPoolSize, keepAliveTime, unit,
            workQueue, ThreadLocalChangeListener.EMPTY);
    }

    public ThreadPoolExecutorExt(
        int corePoolSize, int maximumPoolSize, long keepAliveTime,
        TimeUnit unit, BlockingQueue<Runnable> workQueue,
        ThreadFactory threadFactory) {
        this(corePoolSize, maximumPoolSize, keepAliveTime, unit,
            workQueue, threadFactory,
            ThreadLocalChangeListener.EMPTY);
    }

    public ThreadPoolExecutorExt(
        int corePoolSize, int maximumPoolSize, long keepAliveTime,
        TimeUnit unit, BlockingQueue<Runnable> workQueue,
        RejectedExecutionHandler handler) {
        this(corePoolSize, maximumPoolSize, keepAliveTime, unit,
            workQueue, handler,
            ThreadLocalChangeListener.EMPTY);
    }

    public ThreadPoolExecutorExt(
        int corePoolSize, int maximumPoolSize, long keepAliveTime,
```

```
TimeUnit unit, BlockingQueue<Runnable> workQueue,
ThreadFactory threadFactory,
RejectedExecutionHandler handler) {
    this(corePoolSize, maximumPoolSize, keepAliveTime, unit,
        workQueue, threadFactory, handler,
        ThreadLocalChangeListener.EMPTY);
}

public ThreadPoolExecutorExt(
    int corePoolSize, int maximumPoolSize, long keepAliveTime,
    TimeUnit unit, BlockingQueue<Runnable> workQueue,
    ThreadLocalChangeListener listener) {
    super(corePoolSize, maximumPoolSize, keepAliveTime, unit,
        workQueue);
    this.listener = listener;
}

public ThreadPoolExecutorExt(
    int corePoolSize, int maximumPoolSize, long keepAliveTime,
    TimeUnit unit, BlockingQueue<Runnable> workQueue,
    ThreadFactory threadFactory,
    ThreadLocalChangeListener listener) {
    super(corePoolSize, maximumPoolSize, keepAliveTime, unit,
        workQueue, threadFactory);
    this.listener = listener;
}

public ThreadPoolExecutorExt(
    int corePoolSize, int maximumPoolSize, long keepAliveTime,
    TimeUnit unit, BlockingQueue<Runnable> workQueue,
    RejectedExecutionHandler handler,
    ThreadLocalChangeListener listener) {
    super(corePoolSize, maximumPoolSize, keepAliveTime, unit,
        workQueue, handler);
    this.listener = listener;
}

public ThreadPoolExecutorExt(
    int corePoolSize, int maximumPoolSize, long keepAliveTime,
    TimeUnit unit, BlockingQueue<Runnable> workQueue,
    ThreadFactory threadFactory,
    RejectedExecutionHandler handler,
    ThreadLocalChangeListener listener) {
    super(corePoolSize, maximumPoolSize, keepAliveTime, unit,
        workQueue, threadFactory, handler);
    this.listener = listener;
}
```

```
/* The interest bit of this class is below ... */

private static final ThreadLocal<ThreadLocalCleaner> local =
    new ThreadLocal<>();

protected void beforeExecute(Thread t, Runnable r) {
    assert t == Thread.currentThread();
    local.set(new ThreadLocalCleaner(listener));
}

protected void afterExecute(Runnable r, Throwable t) {
    ThreadLocalCleaner cleaner = local.get();
    local.remove();
    cleaner.cleanup();
}

}
```

You would use this just like a normal ThreadPoolExecutor, with the difference that this one would restore the state of the thread locals after each Runnable has been executed. You can also again attach a listener in case you need to debug your system. In our example here, you can see that we have attached a listener to dump the added thread locals to our LOG. Notice also that in Java 8, the `java.util.logging.Logger` methods now take a `Supplier<String>`, meaning that we do not need code guards anymore to make our logging performant.

```
import java.text.*;
import java.util.concurrent.*;
import java.util.logging.*;

public class ThreadPoolExecutorExtTest {
    private final static Logger LOG = Logger.getLogger(
        ThreadPoolExecutorExtTest.class.getName()
    );

    private static final ThreadLocal<DateFormat> df =
        new ThreadLocal<DateFormat>() {
            protected DateFormat initialValue() {
                return new SimpleDateFormat("yyyy-MM-dd");
            }
        };

    public static void main(String... args)
        throws InterruptedException {
        ThreadPoolExecutor tpe = new ThreadPoolExecutorExt(
```

```

    1, 1, 0, TimeUnit.SECONDS,
    new LinkedBlockingQueue<>(),
    (m, t, tl, v) -> {
        LOG.warning(
            () -> String.format(
                "Thread %s %s ThreadLocal %s with value %s%n",
                t, m, tl.getClass(), v)
        );
    }
);

for (int i = 0; i < 10; i++) {
    tpe.submit(() ->
        System.out.println(System.identityHashCode(df.get()));
        Thread.sleep(1000);
    }
    tpe.shutdown();
}
}
}

```

Output on my machine is something like this:

```

914524658
May 23, 2015 9:28:50 PM ThreadPoolExecutorExtTest lambda$main$1
WARNING: Thread Thread[pool-1-thread-1,5,main] \
    ADDED ThreadLocal class ThreadPoolExecutorExtTest$1 \
    with value java.text.SimpleDateFormat@f67a0200

957671209
May 23, 2015 9:28:51 PM ThreadPoolExecutorExtTest lambda$main$1
WARNING: Thread Thread[pool-1-thread-1,5,main] \
    ADDED ThreadLocal class ThreadPoolExecutorExtTest$1 \
    with value java.text.SimpleDateFormat@f67a0200

466968587
May 23, 2015 9:28:52 PM ThreadPoolExecutorExtTest lambda$main$1
WARNING: Thread Thread[pool-1-thread-1,5,main] \
    ADDED ThreadLocal class ThreadPoolExecutorExtTest$1 \
    with value java.text.SimpleDateFormat@f67a0200

```

That's it for now. This code has not been tried in the crucible of a production server, so

please use it with caution. Thank you for reading this newsletter and also for your support. I really appreciate it!

Kind regards

Heinz

Issue 228 - Extracting Real Task from FutureTask

Author: Dr. Heinz M. Kabutz

Date: 2015-04-30

Category: Tips and Tricks

Java Versions: Java 5,6,7,8,...

Abstract:

ExecutorService allows us to submit either Callable or Runnable. Internally, this is converted to a FutureTask, without the possibility of extracting the original task. In this newsletter we look at how we can dig out the information using reflection.

Welcome to the 228th edition of **The Java(tm) Specialists' Newsletter**. As you probably know, I live on an island in the Mediterranean Sea. And no, I don't own the entire island, just a large enough chunk so that my neighbours generally don't complain about our noise. This is a good thing, as my son started an alternative punk rock band three years ago with his friends. We have been collecting egg boxes from the whole village of Chorafakia to try contain the drumming. In January our little band came first in Greece at the **Global Battle of the Bands competition** and this past Sunday they competed in the world final in Oslo. Unfortunately I could not attend, but despite their #1 fan not being there, **they managed to place 4th in the world**. Here is **Pull the Curtain**, which I watched them produce in Athens. Well done my boy!

Extracting Real Task from FutureTask

One of the annoyances of the ThreadPoolExecutor and its subclass ScheduledThreadPoolExecutor, is that the tasks that we submit are wrapped with a FutureTask without a possibility of us getting the original task again. I wrote about this in **newsletter 154**, where I tackled the challenge of resubmitting failed timed tasks.

However, in this newsletter, I would like to show a small class that can extract the Runnable or Callable from the FutureTask that is returned from the ThreadPoolExecutor. Before we get into the details, here are the various ways I can think of where you might see a FutureTask wrapper:

1. When you call shutdownNow(), a Collection of Runnables is returned. These are not the Runnables that we submitted, but the FutureTask wrapper objects.
2. If you specify a RejectedExecutionHandler, the Runnable that is passed into the rejectedExecution() method is again the FutureTask wrapper.
3. The beforeExecute() and afterExecute() methods that you can override again provide you with the wrapper object.

The FutureTask always contains a Callable. If we submit a Runnable to the ThreadPoolExecutor, it wraps this with a Callable adapter using the Executors.callable(Runnable) method. I make certain assumptions in my JobDiscoverer that could certainly not be true on other implementations of the FutureTask. I assume that it contains a field called "callable". Furthermore I assume that if the type is the same as what we would get from calling Executors.callable(), that initially a Runnable has been passed into the ThreadPoolExecutor. I can think of quite a few scenarios in which this assumption is false.

Here is my JobDiscoverer class, which uses reflection to find out what the type is. Since the result can be either a Runnable or a Callable, I need to return Object. The code is relatively straightforward:

```

import java.lang.reflect.*;
import java.util.concurrent.*;

public class JobDiscoverer {
    private final static Field callableInFutureTask;
    private static final Class<? extends Callable> adapterClass;
    private static final Field runnableInAdapter;

    static {
        try {
            callableInFutureTask =
                FutureTask.class.getDeclaredField("callable");
            callableInFutureTask.setAccessible(true);
            adapterClass = Executors.callable(new Runnable() {
                public void run() { }
            }).getClass();
            runnableInAdapter =
                adapterClass.getDeclaredField("task");
            runnableInAdapter.setAccessible(true);
        } catch (NoSuchFieldException e) {
            throw new ExceptionInInitializerError(e);
        }
    }

    public static Object findRealTask(Runnable task) {
        if (task instanceof FutureTask) {
            try {
                Object callable = callableInFutureTask.get(task);
                if (adapterClass.isInstance(callable)) {
                    return runnableInAdapter.get(callable);
                } else {
                    return callable;
                }
            } catch (IllegalAccessException e) {
                throw new RuntimeException(e);
            }
        }
        return null;
    }
}

```

```
        }
    } catch (IllegalAccessException e) {
        throw new IllegalStateException(e);
    }
}
throw new ClassCastException("Not a FutureTask");
}
```

In my sample code, I submit ten jobs to an ExecutorService, both Runnable and Callable interleaved. Each of the tasks would block indefinitely. They also have overridden the `toString()` method to return the type of class they are. I then call `shutdownNow()` and first print out the values in the result list, followed by what our `JobDiscoverer` returns. Again, the code is fairly simple. The one thing that might be puzzling is that some of the threads show that they were interrupted, and others don't. I will leave that as an exercise to the reader to figure out :-) [It's also not difficult.]

```
import java.util.*;
import java.util.concurrent.*;

public class JobDiscovererTest {
    public static void main(String... args) {
        final CountDownLatch latch = new CountDownLatch(1);
        ExecutorService pool = Executors.newFixedThreadPool(3);
        for (int i = 0; i < 5; i++) {
            final int finalI = i;
            pool.submit(new Runnable() {
                public void run() {
                    try {
                        latch.await();
                    } catch (InterruptedException consumeAndExit) {
                        System.out.println(Thread.currentThread().getName() +
                                " was interrupted - exiting");
                    }
                }
            });
            pool.submit(new Callable<String>() {
                public String call() throws InterruptedException {
                    latch.await();
                }
            });
        }
    }
}
```

```

        return "success";
    }

    public String toString() {
        return "Callable: " + finalI;
    }
});

}

// Note: the Runnables returned from shutdownNow are NOT
// the same objects as we submitted to the pool!!!
List<Runnable> tasks = pool.shutdownNow();

System.out.println("Tasks from ThreadPool");
System.out.println("=====");
for (Runnable task : tasks) {
    System.out.println("Task from ThreadPool " + task);
}

System.out.println();
System.out.println("Using our JobDiscoverer");
System.out.println("=====");

for (Runnable task : tasks) {
    Object realTask = JobDiscoverer.findRealTask(task);
    System.out.println("Real task was actually " + realTask);
}
}
}

```

The output on my machine is the following:

```

Tasks from ThreadPool
=====
pool-1-thread-1 was interrupted - exiting
pool-1-thread-3 was interrupted - exiting
Task from ThreadPool java.util.concurrent.FutureTask@5a07e868
Task from ThreadPool java.util.concurrent.FutureTask@76ed5528
Task from ThreadPool java.util.concurrent.FutureTask@2c7b84de
Task from ThreadPool java.util.concurrent.FutureTask@3fee733d
Task from ThreadPool java.util.concurrent.FutureTask@5acf9800
Task from ThreadPool java.util.concurrent.FutureTask@4617c264
Task from ThreadPool java.util.concurrent.FutureTask@36baf30c

```

Using our JobDiscoverer

```
=====
Real task was actually Callable: 1
Real task was actually Runnable: 2
Real task was actually Callable: 2
Real task was actually Runnable: 3
Real task was actually Callable: 3
Real task was actually Runnable: 4
Real task was actually Callable: 4
```

Short and sweet newsletter I hope? :-) Sent from Athens International Airport.

Kind regards

Heinz

Issue 226 - Discovering Where Threads Are Being Constructed

Author: Dr. Heinz M. Kabutz

Date: 2015-02-24

Category: Tips and Tricks

Java Versions: Java 8

Abstract:

How can you discover all the places in your program where threads are being constructed?

In this newsletter we create our own little SecurityManager to keep an eye on thread creation.

Welcome to the 226th issue of **The Java(tm) Specialists' Newsletter**, sent to you from the beautiful island of Crete. The mountains of Crete are covered with snow at this time of year. From where I'm sitting right now in our conference room, I have a beautiful view of the Lefka Ori mountain range in the distance. My neighbour's sleepy vineyard is to my left and I can see Mr Kostas walking around and inspecting his property. He was just a wee lad when Zorba the Greek was filmed in our village, and he got the role of running around and throwing stones. Perfect for a young Cretan boy. I'm sure my son would've enjoyed that too. In front of me is the sea. Most days the sun is too bright on the azure water and I need to close the shutters. I installed a projector that can output 4200 lumen, which allows my students to easily see the slides off the high-reflective projector screen whilst the shutters can be left open to enjoy the beautiful views. Here is what the sea view looks like from within the conference room:



Discovering Where Threads Are Being Constructed

This newsletter was prompted by a question by one of my readers, Scott Morgan from [Adligo](#). He wanted to know whether there was a way in which we could somehow prevent threads from being created. Java does not have the concept of thread ownership. Once you

have started a thread, you have no further claim over at. It immediately has full rights, same as its creator. We do have thread groups. These were used initially with Java, but since Java 5, there has not been much need for them. We use to handle uncaught exceptions with thread groups in the early days of Java. But nowadays, we have `UncaughtExceptionHandler`. Furthermore, in modern code, threads are partitioned using thread pools rather than thread groups.

So how can we control the creation and starting of threads? If you look inside the source code for the constructor of `Thread`, you will notice a call to the security manager. This checks whether we are allowed to modify our current thread group. A thread automatically is created in the same thread group as its parent. Thus if we can check the permission, we could also determine whether the thread may be created or not.

The code I will present in this newsletter, would typically be used in a testing environment. It would not work if you already had a security manager installed. We present a `ThreadWatcher` security manager that will test a given predicate to determine whether an action should be executed. Let's start with the `ThreadWatcher`:

```
import java.security.*;
import java.util.function.*;

public class ThreadWatcher extends SecurityManager {
    private final Predicate<Thread> predicate;
    private final Consumer<Thread> action;

    public ThreadWatcher(Predicate<Thread> predicate,
                         Consumer<Thread> action) {
        this.predicate = predicate;
        this.action = action;
    }

    public void checkPermission(Permission perm) {
        // allow everything
    }

    public void checkPermission(Permission perm, Object context) {
        // allow everything
    }

    public void checkAccess(ThreadGroup g) {
        Thread creatingThread = Thread.currentThread();
        if (predicate.test(creatingThread)) {
            action.accept(creatingThread);
        }
    }
}
```

```
}
```

Let's say that we wanted to prevent threads within thread pools from creating new threads. This is an arbitrary rule, and just meant as an illustration. Typically, thread pool threads follow a particular naming convention, which we can find in the thread factory. It is of the form pool-#-thread-#, where the #'s would hopefully render the names unique. We could thus define a predicate that used a regular expression match on the thread name, such as a Lambda Predicate (Thread t) -> t.getName().matches("pool-\\d+-thread-\\d+")

We also define a simple job that prints "hello" plus the thread name and a consumer that throws a SecurityException if we try to create a thread for a context where the predicate matches. These are all provided by the DemoSupport interface:

```
import java.util.function.*;

public interface DemoSupport {
    static Predicate<Thread> createPredicate() {
        return (Thread t) ->
            t.getName().matches("pool-\\d+-thread-\\d+");
    }

    static Consumer<Thread> createConsumer() {
        return (Thread creator) -> {
            throw new SecurityException(creator +
                " tried to create a thread");
        };
    }

    static Runnable createHelloJob() {
        return () -> System.out.printf(
            "Hello from \"%s\"",
            Thread.currentThread());
    }
}
```

In our first example, we try to create a new thread from within our main thread. This should work:

```

import java.util.concurrent.*;

public class ThreadFromMainThread {
    public static void main(String... args)
        throws InterruptedException {
        System.setSecurityManager(
            new ThreadWatcher(
                DemoSupport.createPredicate(),
                DemoSupport.createConsumer()
            )
        );
        new Thread(DemoSupport.createHelloJob(),
            "This should work 1").start();

        System.setSecurityManager(null);
    }
}

```

Output is this:

```
Hello from "Thread[This should work 1,5,main]"
```

On the other hand, if we try to create a new thread from within a thread pool thread, then the name will match our regular expression and it will fail:

```

import java.util.concurrent.*;

public class ThreadFromThreadPool {
    public static void main(String... args)
        throws InterruptedException {
        System.setSecurityManager(
            new ThreadWatcher(
                DemoSupport.createPredicate(),
                DemoSupport.createConsumer()
            )
        );
}

```

```

        )
);

ExecutorService pool = Executors.newFixedThreadPool(10);
Future<?> future = pool.submit(() ->
    new Thread(DemoSupport.createHelloJob(),
               "This should print a warning 1")
);
try {
    future.get();
} catch (ExecutionException e) {
    e.getCause().printStackTrace();
}
pool.shutdown();

System.setSecurityManager(null);
}
}

```

The output is now:

```

java.lang.SecurityException: Thread[pool-1-thread-1,5,main] \
    tried to create a thread
at DemoSupport.lambda$createConsumer$1(DemoSupport.java:11)
at DemoSupport$$Lambda$2/558638686.accept(Unknown Source)
at ThreadWatcher.checkAccess(ThreadWatcher.java:25)
at java.lang.ThreadGroup.checkAccess(ThreadGroup.java:315)
at java.lang.Thread.init(Thread.java:391)
at java.lang.Thread.init(Thread.java:349)
at java.lang.Thread.<init>(Thread.java:548)
at ThreadFromThreadPool.lambda$main$0(ThreadFromThreadPool:15)
at ThreadFromThreadPool$$Lambda$3/1452126962.call
at java.util.concurrent.FutureTask.run(FutureTask.java:266)
at java.util.concurrent.ThreadPoolExecutor.runWorker
at java.util.concurrent.ThreadPoolExecutor$Worker.run
at java.lang.Thread.run(Thread.java:745)

```

We could also get a bit more fancy with how we manage the threads who are misbehaving. For example, we could put all the miscreants into a map, with functions `forEach()` and `toString()` implemented to view the elements, such as:

```

import java.util.*;
import java.util.concurrent.*;
import java.util.concurrent.atomic.*;
import java.util.function.*;
import java.util.stream.*;

public class ThreadAccumulator implements Consumer<Thread> {
    private final ConcurrentMap<Thread, LongAdder> miscreants =
        new ConcurrentHashMap<>();

    public void accept(Thread thread) {
        miscreants.computeIfAbsent(
            thread, t -> new LongAdder()).increment();
    }

    public int getNumberOfMisbehavingThreads() {
        return miscreants.size();
    }

    public void forEach(BiConsumer<Thread, Integer> action) {
        miscreants.entrySet()
            .forEach(e -> action.accept(
                e.getKey(),
                e.getValue().intValue()));
    }

    public String toString() {
        return miscreants.entrySet()
            .parallelStream()
            .map(ThreadAccumulator::format)
            .collect(Collectors.joining(", "));
    }

    private static String format(Map.Entry<Thread, LongAdder> e) {
        return String.format("%s created %d thread(s)",
            e.getKey().getName(),
            e.getValue().intValue());
    }
}

```

I must say, the new features in Java 8 are very nice. Have a look at how easy it is to convert the map of miscreants into a nice comma separated String in the `toString()` method! I also like the `computeIfAbsent()` method inside Map, which means that there is no more need for the `putIfAbsent` check.

Thanks for your feedback as always. If you'd like to say "hello", simply reply to this email or send me a note to heinz@javaspecialists.eu. I enjoy hearing from you :-)

Kind regards

Heinz

Issue 224 - Book Review: Mastering Lambdas: Java Programming in a Multicore World

Author: Dr. Heinz M. Kabutz

Date: 2014-12-27

Category: Book Review

Java Versions: Java 8

Abstract:

In his latest book, Maurice Naftalin takes us on a journey of discovery as we learn with him how Lambdas and Streams work in Java 8.

Welcome to the 224th issue of **The Java(tm) Specialists' Newsletter**, sent from the stunning Island of Crete. Last week I was in Paris with my partners Zenika, teaching their engineers how to present my **Design Patterns Course**. Thus if you would like it in French or even Italian, Zenika will be able to help you. On the Monday before the course, I got in slightly too late for lunch and too early for supper. In typical Parisian fashion, all the restaurants in my area were closed. I thus had a small snack and then went out for a run. If you've ever seen Forest Gump, you can imagine what happened next. It took me a while to get my bearings, but eventually I was crisscrossing the Seine river, completely mesmerized by the beauty of the place. After a while, I thought the Eiffel Tower didn't look too far away (hint: it is tall and it looks close from many places in Paris), so I kept on running. "Run Forest run!" Eventually I did a total of 15.6km, which for someone of my size felt like quite an achievement. By the time I got back and met up with my friend Kirk Pepperdine, yeah, the restaurants had stopped serving food. Gotta love the French!

Book Review: Mastering Lambdas: Java Programming in a Multicore World by Maurice Naftalin

Maurice Naftalin is a dear friend of mine. Even though my name does not appear anywhere in the text, he wrote "It's all your fault!" in the autographed copy that he gifted me over breakfast at Canary Wharf.

And he's not entirely wrong. A couple of years prior, Maurice told me that he was thinking of writing a 2nd edition of his **Java Generics and Collections** book to include lambdas and streams. Or perhaps a book on lambdas by themselves. He wasn't sure. I suggested that he should start by writing a Lambda FAQ in a similar vein to Angelika Langer's excellent **Generics FAQ**. This would have two purposes. First off, he could contribute something useful to society at large. This he certainly did. Due to his efforts, Maurice is now one of the newest **Oracle Java Champions**. Secondly, he could scope out the volume of material to see if there was enough in there to write a book.

Since I'm rather impetuous, I immediately registered the domain names

www.lambdafaq.org, www.lambdafaq.net and www.lambdafaq.com and put up a simple Wordpress site with the title "Maurice Naftalin's Lambda FAQ". I then pestered Maurice until he had a few questions and their answers on the website. Before too long, Oracle had linked to it from their main Lambda website. It has now evolved into a very useful tutorial to answer some of the tough questions that we encounter with lambdas.

When the time came to review his book, I was too busy with other things to even look at it. However, this past month I've given it a very detailed read and would like to share my findings with you. Besides a few small typos, real errors were annoyingly elusive. In other words, my contribution, besides kick-starting the website, was minuscule.

Before I continue with the review, I'd like to give a shout-out to two other friends who beat Maurice to the printing press: Dr Richard Warburton and almost-Dr Raoul-Gabriel Urma.

Dr Richard Warburton recently published [Java 8 Lambdas: Pragmatic Functional Programming](#). Whenever I meet "Sir" Richard (as I affectionately call him) at conferences, he is only too happy to try to answer some of the questions I'm battling with at the time. Usually he also does not know, but the good doctor always makes a point of finding the answer and getting back to me. I appreciate that :-)

Raoul-Gabriel Urma gave an excellent talk about Java 8 at the [jdk.io](#) conference in Denmark in 2014. Whilst enjoying an excellent dinner, I was telling him that I had bought my son size 52 shoes. He said to me: "Sorry, I don't know that system. What is that in European?" Um - that WAS European! Raoul wrote [Java 8 in Action: Lambdas, Streams, and functional-style programming](#). Raoul's book is very good indeed! He covers topics that are hard to find elsewhere, such as the CompletableFuture. Just that one chapter makes it worthwhile buying his book. He does not cover ManagedBlocker, but then neither do the others and you can always read [my newsletter to learn more](#).

Dr Richard Warburton, Raoul-Gabriel Urma and James Gough have put together what looks like a [nice little training course on Java 8](#). The outline looks promising and they even cover the new Date and Time API, largely produced by members of their London Java Community.

Maurice Naftalin is also busy putting together a kickass Java 8 course. It is based on his book and ETA is Q2 2015. Initially it will be available only as an in-house course. Please have a look at [Mastering Lambdas Course](#) for more information.

Lastly you could join me on my [Extreme Java - Concurrency Performance for Java 8](#) course. Whilst this is not a pure Java 8 Lambda course, you will pick up the essentials of lambdas and streams whilst broadening your mind with threading and performance. That particular course is also available as [self-study on Parleys](#).

Why Maurice's Book?

I am told that writing any book is a lot of work. But two years? Why so long, Maurice? After all, your book is rather short. I know authors that can churn that out in two weeks! I personally prefer short books. Even this book took me about a month to work through, next to all my other obligations. My bookshelf contains lots of books that I never finished because they were simply too voluminous. Maurice has done the thinking and refining for me. There is nothing in the book that I would say is superfluous. I thus save time studying his shorter book.

In addition, the book is focused. This is not a book about functional programming like the other two mentioned above. It is a book about Java's place in a new world where we need to utilize lots of cores. This is not surprising, as he had input from Brian Goetz, author of **Java Concurrency in Practice**. Maurice's chapter on performance is one of the best I've read in any book. I care about performance and so should you. Maurice teaches Kirk Pepperdine's **Java Performance Tuning Course** and all my **courses focused on concurrency and performance**. Besides knowing what he is talking about, he also knows who to ask for input into his writing. We thus see Aleksey Shipilev's **Java Microbenchmarking Harness** being employed, rather than some ad-hoc mechanism.

Some highlights from the book. To explain lambdas, Naftalin starts by showing a normal anonymous inner class.

```
pointList.forEach(new Consumer<Point>() {
    public void accept(Point p) {
        p.translate(1, 1);
    }
});
```

He then starts removing elements that are superfluous by greying them out. For example let's hide the "new Consumer" constructor call:

```
pointList.forEach(new Consumer<Point>() {
    public void accept(Point p) {
        p.translate(1, 1);
    }
});
```

He then greys out even more, specifically the single method inside the Consumer. Since

there is only one, it should be obvious what code is meant. The name of the method is no longer important:

```
pointList.forEach(new Consumer<Point>() {
    public void accept(Point p) {
        p.translate(1, 1);
    }
});
```

Even the fact that the parameter type is a "Point" can be deduced, so let's grey that out too:

```
pointList.forEach(new Consumer<Point>() {
    public void accept(Point p) {
        p.translate(1, 1);
    }
});
```

This can then be represented simply as a Java 8 lambda, using the `->` syntax:

```
pointList.forEach(p -> p.translate(1, 1));
```

Great explanation - just the way I would've done it!

Difference Between Lambda and Anonymous Inner Class

In the second chapter, we compare the old anonymous inner classes with lambdas. They are not the same. Anonymous classes always create a new object. Lambdas do not necessarily. In both cases, the object collection cost can be eliminated with escape analysis. Here is a small class that shows object identity:

```

public class IdentityAnonymousLambda {
    public static void main(String... args) {
        for (int i = 0; i < 2; i++) {
            showIdentity(() ->
                System.out.println("Lambda - no fields"));
            showIdentity(() ->
                System.out.println("Lambda - parameters - " + args));
            showIdentity(new Runnable() {
                public void run() {
                    System.out.println("anon - no fields");
                }
            });
            showIdentity(new Runnable() {
                public void run() {
                    System.out.println("anon - parameters - " + args);
                }
            });
            System.out.println();
        }
    }

    private static void showIdentity(Runnable runnable) {
        System.out.printf("%x ", System.identityHashCode(runnable));
        runnable.run();
    }
}

```

And here is the output. Note how the identity hash codes of the simple lambda are the same, since they are the same object both times we use the lambda. For the more complicated lambda, the hash codes are different.

```

404b9385 Lambda - no fields
58372a00 Lambda - parameters - [Ljava.lang.String;@4dd8dc3
6d03e736 anon - no fields
568db2f2 anon - parameters - [Ljava.lang.String;@4dd8dc3
404b9385 Lambda - no fields
378bf509 Lambda - parameters - [Ljava.lang.String;@4dd8dc3
5fd0d5ae anon - no fields
2d98a335 anon - parameters - [Ljava.lang.String;@4dd8dc3

```

The second big difference is in the scope of "this". Within the lambda it refers to the enclosing object, but within the anonymous class, it refers to the object instance of the anonymous inner class. To refer to the outer object, we would have to write OuterClass.this. For example, in the "Hello" class below, both lambdas will print "Hello, world!". Try it out yourself before reading any further:

```
/** @author Maurice Naftalin, from Mastering Lambdas */
public class Hello {
    Runnable r1 = () -> { System.out.println(this); };
    Runnable r2 = () -> { System.out.println(toString()); };
    public String toString() { return "Hello, world!"; }
    public static void main(String... args) {
        new Hello().r1.run();
        new Hello().r2.run();
    }
}
```

Heinz's approach to learning new language features: Delegate the grunt work to your IDE, in my case IntelliJ IDEA. Here's what I do (don't laugh):

1. Write the code using old-school anonymous inner classes.
2. Use refactoring tools in IntelliJ to convert these to lambda expressions semi-automatically. Use the "*Analyze*" -> "*Inspect*" -> "*Java Language Level migration aids*" -> "*Anonymous type can be replaced by lambda*".
3. When possible, replace lambda with a method reference.

Here are three small code snippets that get progressively improved with my technique:

```
import java.util.*;
import java.util.function.*;

public class HeinzLambdaTrainingWheels {
    public static void main(String... args) {
        // step 1. write using old-school anonymous inner classes
        Arrays.stream(args).map(new Function<String, String>() {
            public String apply(String s) {
                return s.toUpperCase();
            }
        })
    }
}
```

```

}).forEach(new Consumer<String>() {
    public void accept(String s) {
        System.out.println(s);
    }
});

// step 2. "Replace with lambda"
Arrays.stream(args).map(s -> s.toUpperCase()).
    forEach(s -> System.out.println(s));

// step 3. "Replace with Method Reference"
Arrays.stream(args).map(String::toUpperCase).
    forEach(System.out::println);
}
}

```

To you this might seem like a stupid thing to do, but it works for me. I shared "Heinz's approach" with Dr Warburton, from [Java 8 Lambdas](#). He didn't say anything, but his deprecating smile could have written a book. Over time, these training wheels will fall off. But in the meantime, I do find it useful to help me get the syntax right.

Overloaded Methods and Lambdas

Still in chapter 2, Maurice writes about the challenge of overloaded methods. Just to remind you, an overloaded method is when you have multiple methods with the same name, but with different parameters. For example Object.wait() is overloaded. This can go awry quite easily. For example, in my previous newsletter, I had gone wild with refactoring and produced the following (it compiles):

```

public class Interruptions {
    public static void saveForLater(InterruptibleAction action) {
        saveForLater(action::run);
    }

    public static <E> E saveForLater(
        InterruptibleTask<E> task) {
        boolean interrupted = Thread.interrupted(); // clears flag
        try {
            while (true) {
                try {
                    return task.run();
                } catch (InterruptedException e) {

```

```

        // flag would be cleared at this point too
        interrupted = true;
    }
}
} finally {
    if (interrupted) Thread.currentThread().interrupt();
}
}

@FunctionalInterface
public interface InterruptibleAction {
    public void run() throws InterruptedException;
}

@FunctionalInterface
public interface InterruptibleTask<E> {
    public E run() throws InterruptedException;
}
}

```

I was surprised when this resulted in a stack overflow error. Even though the code compiled, the resultant byte code was not what I had expected. Expanding the first saveForLater() method to an anonymous inner class demonstrates the issue:

```

public class Interruptions {
    public static void saveForLater(InterruptibleAction action) {
        saveForLater(new InterruptibleAction() {
            public void run() throws InterruptedException {
                action.run();
            }
        });
    }

    // ...
}

```

We can now see that we are calling ourselves recursively!

Technical Novel

Maurice has an excellent command of the English language. His book contains some of the most beautiful prose I have read in any technical book to date (and I have read many). When I was showing him how to teach my **Java Specialist Master Course**, he drove me quite nutty with his perfectionism. For example, he pointed out that the default locking mechanism wasn't "unfair". Instead it was "non-fair", which is the opposite of "fair". I have four beautiful children. I am not a "fair" father, rather I try to be "non-fair". If I was "fair", then when I bought my daughters a dress, I would also have to buy my son a dress. And when I bought my son size 52 basketball shoes, I would have to buy the same for my daughters. Instead, I buy things for the various children as and when they need them. I noticed that Maurice also mentioned the "non-fair" locking mechanism in his book :-)

This is the first technical book where I found myself re-reading a sentence a few times, simply because the words were so beautifully assembled. For example: "A 40-year trend of exponentially increasing processor speed had been halted by inescapable physical facts: signal leakage, inadequate heat dissipation, and the hard truth that, even at the speed of light, data cannot cross a chip quickly enough for further processor speed increases."

Pure art.

We now get into the more difficult parts of the book, where Maurice writes about streams and the different types of pipelines they can contain. Even though I studied those pages very carefully, I know I will have to go over that again if I want to apply them to real code. In chapter 4 he continues with streams, collection and reduction and just as everything started falling apart in my brain and the little neurons were about to go on hunger strike, he wacks me in the face with a "Worked Example". He makes us stop reading and start thinking. Those annoying horizontal lines tell us that we should not carry on reading until we have at least tried to come up with a solution. I must say that I was completely lost at the first worked example. However, I was amazed at the power and elegance of the solution presented in the book. He carried on with this approach of giving us some practical puzzles to solve. It worked for me. I found myself stopping what I was reading, and trying to solve it without his help. Usually I failed, but always I learned far more than if I had simply read his solution.

BigLineSpliterator

In chapter 5, Maurice presents an idea of a recursive grep using MappedByteBuffer. This LineSpliterator from the book is slated to be included in the JDK one of these days. We don't often have ideas from a book flowing into the JDK. However, there are some issues with the LineSpliterator in the book. First off, it only works with ByteBuffer. We all know that Java has an artificial limitation of Integer.MAX_VALUE for number of bytes. Thus we cannot create a MappedByteBuffer larger than that, even on a 64-bit machine.

Maurice used a clever trick to calculate the mid-point:

```
int mid = (lo + hi) >>> 1;
```

When I typed the code into my IDE, I changed that to the simpler `int mid = (lo + hi) / 2;` and was surprised when that overflowed with files close to 2 GB. The triple-shift `>>>` means an unsigned bit shift. `(lo + hi)` might very well be a negative number, but we'll shift the left-most bit one to the right, making it positive. Another way to have written that would be as `int mid = lo/2 + hi/2;`

Besides the artificial 2GB file size limit (do we really want to go back there?) there were also a few bugs in the code, which I found through unit testing. For example, the output would cut off the first character from each line.

A stream should be at least as fast when you run it in parallel as when you run it in serial. In the Fork/Join framework, we code that by having a threshold below which we do not fork further. We should do the same in the `trySplit()` method. In my code, I use the magic number 10000, based on observation in our small whitepaper on [When to use parallel streams](#). If the chunk of file is less than that, I do not split further.

Lastly, the `LineSpliterator` would fill the characters into a `StringBuilder` and would then convert that to a `String` before passing it on to a regular expression matcher. However, the `pattern.matcher()` method takes as parameter a `CharSequence`, thus there is no need to create the `Strings`. We could simply keep pointers to the original `StringBuilders`. This is a trick that not many programmers know about, but it made a measurable difference in my tests.

I spent far more time on this than I should have, but after removing a few bugs and improving the performance a bit, we now have something that can work with virtually any size file. Much to my surprise, even the non-parallel version was faster than the standard grep utility on Mac OS X. And the parallel version was about 4x the speed of the sequential, since I have four cores on my machine.

Here is the `DispLine` class, which counts the number of bytes where this particular `CharSequence` starts (similar to `grep -b`)

```
public class DispLine {
    private final long disp;
    private final CharSequence line;

    public DispLine(long disp, CharSequence line) {
        this.disp = disp;
        this.line = line;
    }
}
```

```

    }

    public CharSequence getLine() {
        return line;
    }

    public String toString() {
        return disp + ":" + line;
    }
}

```

And now my BigLineSpliterator. To understand how it works, you need to first understand Spliterators ([Maurice's book is a good place to start](#)). Secondly you would need to understand a bit about MappedByteBuffer and FileChannel. For that I can recommend my [Java Specialist Master Course](#) and then the musings of [Peter Lawrey](#).

I am willing to bet a fair wager that my code still contains several bugs. If you find one, please send me your unit test and I'll be happy to look at it. For example, this will not work properly with files that have the Windows format of \n\r. There are bugs, so please test extensively. This is more a learning exercise than production-ready code. You've been warned ... Now, without further delay, here is my BigLineSpliterator:

```

import java.io.*;
import java.nio.*;
import java.nio.channels.*;
import java.util.*;
import java.util.function.*;

import static java.nio.channels.FileChannel.MapMode.READ_ONLY;

/**
 * @author Dr Heinz M. Kabutz, Maurice Naftalin, based on the
 * LineSpliterator from the Mastering Lambdas book.
 */
public class BigLineSpliterator implements
    Spliterator<DispLine> {
    private static final int AVG_LINE_LENGTH = 40;
    private static final int CHUNK_SIZE = Integer.MAX_VALUE;
    private final ByteBuffer[] bbs;
    private long lo;
    private final long hi;
    private final long offset;

```

```

public BigLineSpliterator(ByteBuffer bb) {
    this(bb, 0, bb.limit());
}

public BigLineSpliterator(ByteBuffer bb, int lo, int hi) {
    this(new ByteBuffer[]{bb}, lo, hi, 0);
}

public BigLineSpliterator(FileChannel fc) throws IOException {
    this(fc, 0, fc.size());
}

public BigLineSpliterator(FileChannel fc, long lo, long hi)
    throws IOException {
    this(split(fc, lo, hi), lo, hi, lo);
}

private BigLineSpliterator(ByteBuffer[] bbs,
                           long lo, long hi, long offset) {
    this.bbs = bbs;
    this.lo = lo;
    this.hi = hi;
    this.offset = offset;
}

private static ByteBuffer[] split(FileChannel fc,
                                    long lo, long hi)
    throws IOException {
    int number_of_chunks = (int) Math.ceil(
        ((double) (hi - lo)) / CHUNK_SIZE);
    long remainingBytes = (hi - lo);
    ByteBuffer[] bbs = new ByteBuffer[number_of_chunks];
    for (int i = 0; i < bbs.length; i++) {
        long position = i * (long)CHUNK_SIZE + lo;
        long size = i < bbs.length - 1 ?
            CHUNK_SIZE : remainingBytes;
        remainingBytes -= CHUNK_SIZE;
        bbs[i] = fc.map(READ_ONLY, position, size);
    }

    long totalSize = 0;
    for (ByteBuffer bb : bbs) {
        totalSize += bb.limit();
    }
    if (totalSize != (hi - lo))
        throw new AssertionError("Split size does not match");
    return bbs;
}

```

```

}

public boolean tryAdvance(Consumer<? super DispLine> action) {
    long index = lo;
    StringBuilder sb = new StringBuilder();
    char next;
    while ((next = get(index++)) != '\n') {
        sb.append(next);
    }
    action.accept(new DispLine(lo, sb));
    lo = lo + sb.length() + 1;
    return lo < hi;
}

private char get(long pos) {
    long truePos = pos - offset;
    int chunk = (int) (truePos / CHUNK_SIZE);
    return (char) bbs[chunk].get((int) (truePos % CHUNK_SIZE));
}

private static final int SEQUENTIAL_THRESHOLD = 10000;

@Override
public Spliterator<DispLine> trySplit() {
    if (hi - lo < SEQUENTIAL_THRESHOLD) return null;
    long index = (lo + hi) >>> 1;
    while (get(index) != '\n') index++;
    BigLineSpliterator newSpliterator = null;
    if (index != hi) {
        newSpliterator = new BigLineSpliterator(
            bbs, lo, index, offset);
        lo = index + 1;
    }
    return newSpliterator;
}

@Override
public long estimateSize() {
    return (hi - lo) / AVG_LINE_LENGTH;
}

@Override
public int characteristics() {
    return ORDERED | IMMUTABLE | NONNULL;
}
}

```

I tested this on rather large files up to 60 GB in size. The results were the same as with the grep -b 12345* file.txt Mac OS X command, albeit significantly faster. However, I did not test whether it works with files, when we start at anything besides 0.

That's it for this month and this year. Thanks for all the feedback you send. If you read the book, Maurice has asked if you could please leave a review on Amazon, whether you liked it or not :-)

Kind regards

Heinz

Issue 223 - ManagedBlocker

Author: Dr. Heinz M. Kabutz

Date: 2014-11-27

Category: Language

Java Versions: Java 8

Abstract:

Blocking methods should not be called from within parallel streams in Java 8, otherwise the shared threads in the common ForkJoinPool will become inactive. In this newsletter we look at a technique to maintain a certain liveliness in the pool, even when some threads are blocked.

Welcome to the 223rd issue of **The Java(tm) Specialists' Newsletter**, sent to you from the Island of Crete. We have lived here since 2006 and could not help but accumulate some olive trees. Our yield is still low, but this is the first year that we have a little bit of an excess. It is delicious! For 2015, if you come to Crete to attend one of my courses, I will give you a personally autographed tin containing one liter of our liquid gold to take home with you, complements of JavaSpecialists.eu :-)

ManagedBlocker

Java 8 introduced the concept of a parallel stream, which gives us the ability to parallelize some of our work without very much effort on our part. All is needed is to add a call to parallel() into our streams. Some guidelines by Doug Lea et al on how to do parallelism in Java 8 [can be found here](#). I have already written about the issue with the availableProcessors() not always being accurate nor useful in [Issue 220](#) and [Issue 220b](#).

One of the issues that can happen is that someone might call a blocking method from within a parallel stream. Something simple like System.out::println could even end up blocking, for example:

```
package eu.javaspecialists.tjsn.examples.issue223;

import java.util.stream.*;

public class PrintlnFun {
    public static void main(String... args) {
        synchronized (System.out) {
            System.out.println("Hello World");
        }
    }
}
```

```

        IntStream.range(0, 4).parallel().
            forEach(System.out::println);
    }
}
}

```

Run the code without the call to parallel() and it outputs what you expect (Hello World, followed by 0,1,2,3). However, when we try run it in parallel, we deadlock the system. The main thread had acquired the lock to System.out, which meant that the worker threads from ForkJoinPool.commonPool() were blocked. The main thread could not complete until the worker threads are done, so we have our classic deadlock situation. The fun part is that the output is not going to be consistent. And even more fun is that a thread dump does not detect this as a Java deadlock.

My point with picking on System.out::println is that almost all examples that I've seen of parallel streams have used that call at some point, but this could also be a blocking call. Obviously we should avoid blocking calls from within parallel streams, but sometimes it is not so obvious that it might happen.

One of the ways to deal with blocking methods in parallel streams is by using the ForkJoinPool.ManagedBlocker interface. Instead of blocking directly, we instead call the ForkJoinPool.managedBlock(<instance of ManagedBlocker>) method. I spoke about this at a short [Nighthacking interview at Devoxx](#). I had heard about ManagedBlocker from [Paul Sandoz](#) a few minutes before the interview started, with the result that this really was a *hacking* interview. I made a silly mistake, but eventually we got it working. I knew that Phaser "played nice" with ForkJoinPool, but was not aware that we could also write our own blocking methods that could get along with that framework.

This week, I decided to try to write a ReentrantLock that would be compatible with parallel streams. Ideally we should never call a blocking operation from within a parallel stream. But if we need to, this will allow us to have some element of liveness in our shared pool that we otherwise would forfeit.

Interruptions

Before we get into the subtleties of ManagedBlocker, one of the parts of Java that programmers seem to struggle with a lot is how to deal with InterruptedException. A mechanism that is used in Lock.lock(), Condition.awaitUninterruptibly(), Semaphore.acquireUninterruptibly() and many others, is to save the interrupt until the action has been completed and to then return with a thread that is in the interrupted state. (If you don't understand interruptions, I can strongly recommend my [new Java 8 concurrency course available on Parleys](#)). Here is the code to save interruptions for later, which would typically be called using lambdas:

```

package eu.javaspecialists.tjsn.concurrency.util;

public class Interruptions {
    public static void saveForLater(InterruptibleAction action) {
        saveForLaterTask(() -> {
            action.run();
            return null;
        });
    }

    public static <E> E saveForLaterTask(
        InterruptibleTask<E> task) {
        boolean interrupted = Thread.interrupted(); // clears flag
        try {
            while (true) {
                try {
                    return task.run();
                } catch (InterruptedException e) {
                    // flag would be cleared at this point too
                    interrupted = true;
                }
            }
        } finally {
            if (interrupted) Thread.currentThread().interrupt();
        }
    }

    @FunctionalInterface
    public interface InterruptibleAction {
        public void run() throws InterruptedException;
    }

    @FunctionalInterface
    public interface InterruptibleTask<E> {
        public E run() throws InterruptedException;
    }
}

```

We can use this to manage interruptions if we do not want to be interruptible. You will see it inside our ManagedReentrantLock in a moment.

ManagedReentrantLock

Perhaps one day the AbstractQueuedSynchronizer, on which most of the concurrency constructs in Java are based, will be converted to support ManagedBlocker. This would solve the problem where some of the threads in the ForkJoinPool are blocked by using a concurrency synchronizer. Since Phaser already works with ManagedBlocker, we can use that instead of CountDownLatch and CyclicBarrier.

In my experiment, I tried to adapt the ReentrantLock. I think my code works correctly. I have written several unit tests to check whether it is at least compatible with ReentrantLock's behaviour and also "plays nicely" with ForkJoinPool and therefore with parallel streams. However, I do not give any guarantees that it really works, nor that it is a good idea to use blocking calls inside parallel streams. In fact, I'd like to state that in general it is a really bad idea to use blocking calls in thread pools :-)

In a moment we will look at the code for my ManagedReentrantLock. It is fairly simple. Instead of calling lock.lock() directly, we delegate the call to the ForkJoinPool.managedBlock(ManagedBlocker). The first thing managedBlock() does is check whether we need to even bother blocking. It does this by calling the blocker.isReleasable() method. If that returns true, then no further action is necessary. If it is false, it calls blocker.block(). If that is true, then the block was successful and we can return from the managedBlock() method. The method effectively does something like this:

```
while (!blocker.isReleasable() && !blocker.block()) {}
```

We should always try to achieve our goal within the isReleasable() method without blocking. In most cases, a lock would be available, so we would not even need to consider compensating the FJ pool with another thread to keep it live.

The code is a bit tough to digest, so I would recommend you work through it one piece at a time. Start at the top and figure out how lock.lockInterruptibly() and lock.lock() work. For unlock() we are simply using the parent's method. The most tricky part is the conditions, which also have to be managed. However, things go awry when we pass the condition objects back into the lock, for example with the hasWaiters(Condition) method. We thus need to first map back to the original Condition object produced by the superclass.

```
package eu.javaspecialists.tjsn.concurrency.locks;

import eu.javaspecialists.tjsn.concurrency.util.*;
import java.util.*;
```

```

import java.util.concurrent.*;
import java.util.concurrent.locks.*;

/**
 * The ManagedReentrantLock is a lock implementation that is
 * compatible with the Fork/Join framework, and therefore also
 * with the Java 8 parallel streams. Instead of just blocking
 * when the lock is held by another thread, and thereby removing
 * one of the active threads from the Fork/Join pool, we instead
 * use a ManagedBlocker to manage it.
 * <p>
 * The ManagedReentrantLock subclasses ReentrantLock, which means
 * we can use it as a drop-in replacement. See also the
 * ManagedBlockers facade, which can adapt several known
 * synchronizers with our new ManagedReentrantLock.
 *
 * @author Heinz Kabutz
 * @see eu.javaspecialists.tjsn.concurrency.util.ManagedBlockers
 */
public class ManagedReentrantLock extends ReentrantLock {
    public ManagedReentrantLock() {
    }

    public ManagedReentrantLock(boolean fair) {
        super(fair);
    }

    public void lockInterruptibly() throws InterruptedException {
        ForkJoinPool.managedBlock(new DoLockInterruptibly());
    }

    public void lock() {
        DoLock locker = new DoLock(); // we want to create this
        // before passing it into the lambda, to prevent it from
        // being created again if the thread is interrupted for some
        // reason
        Interruptions.saveForLater(
            () -> ForkJoinPool.managedBlock(locker));
    }

    public boolean tryLock(long time, TimeUnit unit)
        throws InterruptedException {
        // If we already have the lock, then the TryLocker will
        // immediately acquire the lock due to reentrancy. We do not
        // really care whether we had a timeout inside the TryLocker,
        // but only want to return whether or not we hold the lock
        // at the end of the method.
        ForkJoinPool.managedBlock(new TryLocker(time, unit));
    }
}

```

```

    return isHeldByCurrentThread();
}

public Condition newCondition() {
    return new ManagedCondition(super.newCondition());
}

public boolean hasWaiters(Condition c) {
    return super.hasWaiters(getRealCondition(c));
}

public int getWaitQueueLength(Condition c) {
    return super.getWaitQueueLength(getRealCondition(c));
}

protected Collection<Thread> getWaitingThreads(Condition c) {
    return super.getWaitingThreads(getRealCondition(c));
}

//////// Helper functions and inner classes //////////

private Condition getRealCondition(Condition c) {
    if (!(c instanceof ManagedCondition))
        throw new IllegalArgumentException("not owner");
    return ((ManagedCondition) c).condition;
}

private class ManagedCondition implements Condition {
    private final Condition condition;

    private ManagedCondition(Condition condition) {
        this.condition = condition;
    }

    public void await() throws InterruptedException {
        managedBlock(() -> condition.await());
    }

    public void awaitUninterruptibly() {
        Interruptions.saveForLater(
            () -> managedBlock(
                () -> condition.awaitUninterruptibly()
            );
    }

    public long awaitNanos(long nanosTimeout)
        throws InterruptedException {
        long[] result = {nanosTimeout};

```

```

managedBlock(
    () -> result[0] = condition.awaitNanos(nanosTimeout));
return result[0];
}

public boolean await(long time, TimeUnit unit)
throws InterruptedException {
boolean[] result = {false};
managedBlock(
    () -> result[0] = condition.await(time, unit));
return result[0];
}

public boolean awaitUntil(Date deadline)
throws InterruptedException {
boolean[] result = {false};
managedBlock(
    () -> result[0] = condition.awaitUntil(deadline));
return result[0];
}

public void signal() {
condition.signal();
}

public void signalAll() {
condition.signalAll();
}
}

private static void managedBlock(
AlwaysBlockingManagedBlocker blocker)
throws InterruptedException {
ForkJoinPool.managedBlock(blocker);
}

// we should always try to achieve our goal within the
// isReleasable() method instead of block(). This avoids
// trying to compensate the loss of a thread by creating
// a new one.

private abstract class AbstractLockAction
implements ForkJoinPool.ManagedBlocker {
private boolean hasLock = false;

public final boolean isReleasable() {
return hasLock || (hasLock = tryLock());
}
}
}

```

```

private class DoLockInterruptibly extends AbstractLockAction {
    public boolean block() throws InterruptedException {
        if (isReleasable()) return true;
        ManagedReentrantLock.super.lockInterruptibly();
        return true;
    }
}

private class DoLock extends AbstractLockAction {
    public boolean block() {
        if (isReleasable()) return true;
        ManagedReentrantLock.super.lock();
        return true;
    }
}

private class TryLocker extends AbstractLockAction {
    private final long time;
    private final TimeUnit unit;

    private TryLocker(long time, TimeUnit unit) {
        this.time = time;
        this.unit = unit;
    }

    public boolean block() throws InterruptedException {
        if (isReleasable()) return true;
        ManagedReentrantLock.super.tryLock(time, unit);
        return true;
    }
}

@FunctionalInterface
private interface AlwaysBlockingManagedBlocker
    extends ForkJoinPool.ManagedBlocker {
    default boolean isReleasable() {
        return false;
    }

    default boolean block() throws InterruptedException {
        doBlock();
        return true;
    }

    void doBlock() throws InterruptedException;
}
}

```

We can now use this ManagedReentrantLock just like we would use an ordinary ReentrantLock, with the only difference being that if it were held inside a ForkJoinPool, we would be compensated with additional threads to keep the system alive.

Here is a demo that suspends the threads, thereby jamming up the common ForkJoinPool:

```
package eu.javaspecialists.tjsn.examples.issue223;

import eu.javaspecialists.tjsn.concurrency.locks.*;

import java.util.concurrent.locks.*;
import java.util.stream.*;

public class ManagedReentrantLockDemo {
    public static void main(String... args) {
        ReentrantLock lock = new ReentrantLock();
//        ReentrantLock lock = new ManagedReentrantLock();
        Condition condition = lock.newCondition();
        int upto = Runtime.getRuntime().availableProcessors() * 10;
        IntStream.range(0, upto).parallel().forEach(
            i -> {
                lock.lock();
                try {
                    System.out.println(i + ": Got lock, now waiting - " +
                        Thread.currentThread().getName());
                    condition.awaitUninterruptibly();
                } finally {
                    lock.unlock();
                }
            }
        );
    }
}
```

On my MacBook Pro with 8 hardware threads, I see the following output when I use an ordinary ReentrantLock:

```
52: Got lock, now waiting - main
72: Got lock, now waiting - ForkJoinPool.commonPool-worker-2
65: Got lock, now waiting - ForkJoinPool.commonPool-worker-4
5: Got lock, now waiting - ForkJoinPool.commonPool-worker-6
35: Got lock, now waiting - ForkJoinPool.commonPool-worker-7
12: Got lock, now waiting - ForkJoinPool.commonPool-worker-3
25: Got lock, now waiting - ForkJoinPool.commonPool-worker-1
32: Got lock, now waiting - ForkJoinPool.commonPool-worker-5
```

However, when I run it with the ManagedReentrantLock, I see this output instead:

```
5: Got lock, now waiting - ForkJoinPool.commonPool-worker-6
65: Got lock, now waiting - ForkJoinPool.commonPool-worker-4
22: Got lock, now waiting - ForkJoinPool.commonPool-worker-7
35: Got lock, now waiting - ForkJoinPool.commonPool-worker-5
72: Got lock, now waiting - ForkJoinPool.commonPool-worker-2
25: Got lock, now waiting - ForkJoinPool.commonPool-worker-1
12: Got lock, now waiting - ForkJoinPool.commonPool-worker-3
52: Got lock, now waiting - main
27: Got lock, now waiting - ForkJoinPool.commonPool-worker-12
62: Got lock, now waiting - ForkJoinPool.commonPool-worker-0
2: Got lock, now waiting - ForkJoinPool.commonPool-worker-14
28: Got lock, now waiting - ForkJoinPool.commonPool-worker-11
60: Got lock, now waiting - ForkJoinPool.commonPool-worker-10
17: Got lock, now waiting - ForkJoinPool.commonPool-worker-9
63: Got lock, now waiting - ForkJoinPool.commonPool-worker-13
0: Got lock, now waiting - ForkJoinPool.commonPool-worker-15
15: Got lock, now waiting - ForkJoinPool.commonPool-worker-8
32: Got lock, now waiting - ForkJoinPool.commonPool-worker-18
77: Got lock, now waiting - ForkJoinPool.commonPool-worker-19
70: Got lock, now waiting - ForkJoinPool.commonPool-worker-27
73: Got lock, now waiting - ForkJoinPool.commonPool-worker-20
10: Got lock, now waiting - ForkJoinPool.commonPool-worker-30
75: Got lock, now waiting - ForkJoinPool.commonPool-worker-23
30: Got lock, now waiting - ForkJoinPool.commonPool-worker-31
3: Got lock, now waiting - ForkJoinPool.commonPool-worker-24
18: Got lock, now waiting - ForkJoinPool.commonPool-worker-17
38: Got lock, now waiting - ForkJoinPool.commonPool-worker-29
13: Got lock, now waiting - ForkJoinPool.commonPool-worker-22
45: Got lock, now waiting - ForkJoinPool.commonPool-worker-28
42: Got lock, now waiting - ForkJoinPool.commonPool-worker-21
```

```

48: Got lock, now waiting - ForkJoinPool.commonPool-worker-16
40: Got lock, now waiting - ForkJoinPool.commonPool-worker-26
47: Got lock, now waiting - ForkJoinPool.commonPool-worker-25
57: Got lock, now waiting - ForkJoinPool.commonPool-worker-42
33: Got lock, now waiting - ForkJoinPool.commonPool-worker-51
55: Got lock, now waiting - ForkJoinPool.commonPool-worker-44
50: Got lock, now waiting - ForkJoinPool.commonPool-worker-37
58: Got lock, now waiting - ForkJoinPool.commonPool-worker-61
53: Got lock, now waiting - ForkJoinPool.commonPool-worker-54
37: Got lock, now waiting - ForkJoinPool.commonPool-worker-47
68: Got lock, now waiting - ForkJoinPool.commonPool-worker-40
78: Got lock, now waiting - ForkJoinPool.commonPool-worker-33
8: Got lock, now waiting - ForkJoinPool.commonPool-worker-59
43: Got lock, now waiting - ForkJoinPool.commonPool-worker-52
67: Got lock, now waiting - ForkJoinPool.commonPool-worker-45
20: Got lock, now waiting - ForkJoinPool.commonPool-worker-38
7: Got lock, now waiting - ForkJoinPool.commonPool-worker-62
23: Got lock, now waiting - ForkJoinPool.commonPool-worker-55

```

We don't get to 80 threads, but that is because the number of leaves are limited inside the `java.util.stream.AbstractTask` class.

Note that the purpose of the managed blockers is not to necessarily create an unbounded number of threads, nor to maintain exactly the same number of active threads as desired parallelism in the `ForkJoinPool`, but rather to improve the liveliness of the `ForkJoinPool` when faced with potentially blocking tasks.

ManagedBlockers Facade

Since we subclassed `ReentrantLock` in our `ManagedReentrantLock` solution, we can use it anywhere that we currently have a `ReentrantLock`. In the next piece of code, I use reflection to replace the locks inside some well-known classes, specifically `LinkedBlockingQueue`, `ArrayBlockingQueue` and `PriorityBlockingQueue`. Thus if you want to use a `LinkedBlockingQueue` from within parallel streams, but are concerned about using blocking operations (rightly so), you could instead make it managed with our `ManagedBlockers` class. Just make sure that you call the `makeManaged()` method prior to publishing the constructs!

```

package eu.javaspecialists.tjsn.concurrency.util;

import eu.javaspecialists.tjsn.concurrency.locks.*;

```

```

import java.lang.reflect.*;
import java.util.concurrent.*;
import java.util.concurrent.locks.*;

public class ManagedBlockers {
    public static <E> ArrayBlockingQueue<E> makeManaged(
        ArrayBlockingQueue<E> queue) {
        Class<?> clazz = ArrayBlockingQueue.class;

        try {
            Field lockField = clazz.getDeclaredField("lock");
            lockField.setAccessible(true);
            ReentrantLock old = (ReentrantLock) lockField.get(queue);
            boolean fair = old.isFair();
            ReentrantLock lock = new ManagedReentrantLock(fair);
            lockField.set(queue, lock);

            replace(queue, clazz, "notEmpty", lock.newCondition());
            replace(queue, clazz, "notFull", lock.newCondition());

            return queue;
        } catch (IllegalAccessException | NoSuchFieldException e) {
            throw new IllegalStateException(e);
        }
    }

    public static <E> LinkedBlockingQueue<E> makeManaged(
        LinkedBlockingQueue<E> queue) {
        Class<?> clazz = LinkedBlockingQueue.class;

        ReentrantLock takeLock = new ManagedReentrantLock();
        ReentrantLock putLock = new ManagedReentrantLock();

        try {
            replace(queue, clazz, "takeLock", takeLock);
            replace(queue, clazz, "notEmpty", takeLock.newCondition());
            replace(queue, clazz, "putLock", putLock);
            replace(queue, clazz, "notFull", putLock.newCondition());

            return queue;
        } catch (IllegalAccessException | NoSuchFieldException e) {
            throw new IllegalStateException(e);
        }
    }

    public static <E> PriorityBlockingQueue<E> makeManaged(
        PriorityBlockingQueue<E> queue) {
        Class<?> clazz = PriorityBlockingQueue.class;
    }
}

```

```

ReentrantLock lock = new ManagedReentrantLock();

try {
    replace(queue, clazz, "lock", lock);
    replace(queue, clazz, "notEmpty", lock.newCondition());

    return queue;
} catch (IllegalAccessException | NoSuchFieldException e) {
    throw new IllegalStateException(e);
}
}

private static void replace(Object owner,
                           Class<?> clazz, String fieldName,
                           Object fieldValue)
throws NoSuchFieldException, IllegalAccessException {
Field field = clazz.getDeclaredField(fieldName);
field.setAccessible(true);
field.set(owner, fieldValue);
}
}
}

```

Here is a small class that uses it to make the ForkJoinPool a bit more lively:

```

package eu.javaspecialists.tjsn.examples.issue223;

import eu.javaspecialists.tjsn.concurrency.util.*;

import java.util.concurrent.*;
import java.util.concurrent.locks.*;
import java.util.stream.*;

public class ManagedLivenessQueueDemo {
    private final static LinkedBlockingQueue<Integer> numbers =
        new LinkedBlockingQueue<>();

    public static void main(String... args) {
        ManagedBlockers.makeManaged(numbers);
        Thread jamThread = makeJamThread();
        LockSupport.parkNanos(TimeUnit.MILLISECONDS.toNanos(100));
        for (int i = 0; i < 100; i++) {
            numbers.add(i);
        }
    }
}

```

```
        }
    }

private static Thread makeJamThread() {
    Thread jamup = new Thread(() -> {
        int par = Runtime.getRuntime().availableProcessors() * 4;
        IntStream.range(0, par).parallel().forEach(
            i -> {
                System.out.println(i + ": Waiting for number " +
                    Thread.currentThread().getName());
                int num = Interruptions.saveForLaterTask(
                    () -> numbers.take());
                System.out.println("Got number: " + num);
            }
        );
    });
    jamup.start();
    return jamup;
}
```

Again, you will see different output if you use a plain unmanaged LinkedBlockingQueue.

I hope you enjoyed this newsletter. I know that reading the ManagedReentrantLock class is a bit hard, but I hope that you learned something new. I certainly did!

Kind regards from a chilly Crete

Heinz

Issue 222 - Identity Crisis

Author: Dr. Heinz M. Kabutz

Date: 2014-09-09

Category: Language

Java Versions: Java 6,7,8

Abstract:

The JavaDocs for method Object.hashCode() seems to suggest that the value is somehow related to memory location. However, in this newsletter we discover that there are several different algorithms and that Java 8 has new defaults to give better threading result if we call hashCode a lot.

Welcome to the 222nd issue of **The Java(tm) Specialists' Newsletter**. A few years ago, I bought myself a little birthday present, which I named "Java" (or TZABA in Greek letters). Only fair, considering that it was with income from JAVA that I bought her. Since she has 9x more HP than can be driven without a skipper's license, I verified that my South African permit was valid in Greece. "Yes", the port police said on two occasions. But this year, they changed the fellows working there and their answer became "Yes, but ...". The "but" was rather complicated - I would have to have it translated by the South African embassy and get a letter from them proving its authenticity. Only catch is - our embassy does not do that. Eventually, the only option for me was to write the Greek skipper's license - in GREEK! So how can someone who hardly speaks the language pass a written exam, you might ask? I seriously considered selling TZABA, but her namesake helped me out.



Enter Java to save TZABA. In order to pass my exam, my instructor gave me a list of 104 sample questions - in Greek. I ran them through Google Translate in order to get a very basic idea of what they meant. I then wrote a Java program to quiz me on sections of the questions. The program would present a question to me, with the three possible answers and an option to show my English translation. Initially I relied a lot on the translation, but eventually I was able to answer all 104 questions correctly in Greek. (In fact, I just tried again, 5 weeks after writing the exam, and I could still get 100% right.) In some cases it meant recognizing certain patterns. For example, in most of the questions, the longest answer was the correct one! Greek is rather verbose, so to state something clearly takes a lot of long words, which might explain why they speak so fast here. In some cases, I only really recognized one out of 42 words, for example: "orthi", meaning, at right angles (this is how vessels are supposed to cross a traffic corridor). After preparing myself, I also spent two weeks in the evenings sitting in a skipper's class (in Greek) with the best teacher I've ever had, Pavlos Fourakis, who managed to explain everything despite my poor language skills.

The exam was fun. I had done what I could to prepare myself. I knew the subject very well, having already done the license in South Africa. During the exam, I asked the examiner what a simple Greek word meant. From my grasp of the Greek language, she realized that I would definitely fail. Fortunately, about 70% of the questions were directly from the sample questions and the rest were similar enough that I was able to guess the correct answer. The examiners were astonished when I scored 100% correct in the written exam (as was I) :-) So you see, my dear readers in **135 countries**, Java does have its practical uses for day-to-day activities.

Identity Crisis

You have probably at some point of your coding career made the classic mistake of forgetting to implement the hashCode() method when writing equals(), with the result that you can insert your object into a HashMap, but finding it again can only be done by iterating over it. Something like this:

```
public final class JavaChampion {
    private final String name;

    public JavaChampion(String name) {
        this.name = name;
    }

    public boolean equals(Object o) { // simplified equals()
        if (!(o instanceof JavaChampion)) return false;
        return name.equals(((JavaChampion) o).name);
    }
}
```

And here is a little test with three Java Champions in its map:

```
import java.util.*;

public class JavaChampionTest {
    public static void main(String... args) {
        Map<JavaChampion, String> urls = new HashMap<>();
        urls.put(new JavaChampion("Jack"), "fasterj.com");
        urls.put(new JavaChampion("Kirk"), "kodewerk.com");
        urls.put(new JavaChampion("Heinz"), "javaspecialists.eu");
    }
}
```

```

urls.forEach((p,u) -> System.out.println(u)); // Java 8

System.out.println("URL for Kirk: " +
    urls.get(new JavaChampion("Kirk"))); // url == null
}
}

```

Most Java programmers can probably guess what the answer is going to be:

```

heinz$ java JavaChampionTest
javaspecialists.eu
fasterj.com
kodewerk.com
URL for Kirk: null

```

But is it guaranteed to always give this answer? Let's run this again, but with the `-XX:hashCode=2` JVM parameter:

```

heinz$ java -XX:hashCode=2 JavaChampionTest
javaspecialists.eu
fasterj.com
kodewerk.com
URL for Kirk: kodewerk.com

```

Oh wow, did you see that coming? What is `-XX:hashCode`? (BTW, I first asked the question on Twitter - if you are interested in hearing the latest Java gossip, follow me on [heinzkabutz](#).)

Since we did not explicitly write a `hashCode()` method in our class, it defaulted to `Object.hashCode()`. In the JavaDocs it states that "*As much as is reasonably practical, the hashCode method defined by class Object does return distinct integers for distinct objects. (This is typically implemented by converting the internal address of the object into an integer, but this implementation technique is not required by the Java(tm) programming language.*"

Thanks to the comment in the JavaDocs, I always just assumed that the identity hash code, also obtained with `System.identityHashCode(object)`, had something to do with the memory location of the object or at least to be reasonably unique. However, since on a modern machine we have a 64-bit address space, the possibility has to exist that with a 32-bit identity hash code we sometimes have clashes.

This identity hash code is used in **Java Concurrency in Practice** to avoid a lock-ordering deadlock. However, in the book Goetz makes provision for the possibility that two distinct objects can have the same identity hash code. I spent several years puzzling with the question: how do you test that? After all, identity hash codes did look quite unique most of the time. For example:

```
public class UniqueNumbers {
    public static void main(String... args) {
        for (int i = 0; i < 10; i++) {
            System.out.printf("%,d%n", new Object().hashCode());
        }
    }
}
```

Please run it a few times on Java 7 and then a few times on Java 8. Spot any difference?

Here's my output from Java 7:

```
2,102,755,048
1,766,475,321
189,300,272
1,146,390,297
158,440,795
34,719,285
1,558,954,658
2,050,443,606
1,135,602,633
1,386,281,942
```

And now from Java 8:

```
1,950,409,828  
1,995,265,320  
746,292,446  
1,072,591,677  
1,523,554,304  
1,175,962,212  
918,221,580  
2,055,281,021  
1,554,547,125  
617,901,222
```

And just for good measure, Java 7, but in 32-bit:

```
17,547,117  
23,818,829  
18,693,832  
17,102,985  
22,615,283  
24,024,212  
14,455,742  
23,584,771  
10,749,831  
15,359,367
```

Do you notice how much smaller the 32-bit numbers are? And did you see that the identity hash code for 64-bit Java was never negative? And how some numbers always seem to be there, no matter how often you run the class?

First a few words about 32-bit vs 64-bit and then we will examine Java 7 vs Java 8 a bit further.

If you look at the object header in the **markOop.hpp** file, you will notice that for a 32-bit JVM, we have just 25 bits to store the identity hash code. Thus the highest number we could possibly have is 33,554,431 on a 32-bit JVM. For a 64-bit JVM, we have 31 bits of storage space, which explains why the identity hash codes are always positive.

Next we will examine differences between Java 7 and Java 8. The numbers both look like

they have roughly the same maximum of 31 bits, but they are just different. We would expect that in a way. But how are they calculated? Are they memory locations as suggested in the Object.hashCode() method? When we run Java 7 and 8 with the -XX:+PrintFlagsFinal flag and grep for "hashCode" we see that Java 7 has hashCode flag of 0 and Java 8 has value 5.

To see the various options that we can use, please have a look at the get_next_hash() method in [vm/runtime/synchronizer.cpp](#):

HashCode==0: Simply returns random numbers with no relation to where in memory the object is found. As far as I can make out, the global read-write of the seed is not optimal for systems with lots of processors.

HashCode==1: Counts up the hash code values, not sure at what value they start, but it seems quite high.

HashCode==2: Always returns the exact same identity hash code of 1. This can be used to test code that relies on object identity. The reason why JavaChampionTest returned Kirk's URL in the example above is that all objects were returning the same hash code.

HashCode==3: Counts up the hash code values, starting from zero. It does not look to be thread safe, so multiple threads could generate objects with the same hash code.

HashCode==4: This seems to have some relation to the memory location at which the object was created.

HashCode>=5: This is the default algorithm for Java 8 and has a per-thread seed. It uses Marsaglia's xor-shift scheme to produce pseudo-random numbers.

Even though -XX:hashCode=5 seems to scale better on machines with lots of processors, it does suffer more acutely from "twins", which is, two distinct objects with the same identity hash code. Let's try find twins with my class FindTwins, where we create one object and then keep on creating new objects until we find its twin:

```
public class FindTwins {
    public static void main(String... args) {
        Object obj = new Object();
        Object twin = findTwin(obj);
        System.out.printf("found twin: %s and %s, but == is %b%n",
            obj, twin, obj == twin);
    }
}
```

```

private static Object findTwin(Object obj) {
    int hash = obj.hashCode();
    Object twin;
    long created = 0;
    do {
        twin = new Object();
        if ((++created & 0xffffffff) == 0) {
            System.out.printf("%,d created%n", created);
        }
    } while (twin.hashCode() != obj.hashCode());
    System.out.printf("We had to create %,d objects%n", created);
    return twin;
}
}

```

Here you see the output from running it in Java 7:

```

268,435,456
536,870,912
805,306,368
1,073,741,824
1,342,177,280
1,610,612,736
1,879,048,192
We had to create 2,147,479,519 objects
found twin: Object@45486b51 and Object@45486b51, but == is false

```

As you can see, we had to create almost 2^{31} objects to find a twin. Now let's look at the Java 8 output, which uses the new identity hash code algorithm:

```

268,435,456
536,870,91
805,306,36
1,073,741,824
1,342,177,280
We had to create 1,608,293,922 objects
found twin: Object@5a07e868 and Object@5a07e868, but == is false

```

There were a lot less objects needed to get back to our object with Marsaglia's xor-shift algorithm.

Let's look at a different way of discovering twins, using the principles of the **Birthday Paradox**. Instead of looking for the twin of our first object, we look for a twin of any object. As we create objects, we put them into a `HashMap` with the key being the identity hash code and the value the object. If the key was already in the map, we know that we have found a twin. This should hopefully find a twin faster, if we have enough memory to hold the existing objects:

```
import java.util.concurrent.*;

public class FindAnyTwin {
    public static void main(String... args) {
        ConcurrentMap<Integer, Object> all =
            new ConcurrentHashMap<>();
        int created = 0;
        Object obj, twin;
        do {
            obj = new Object();
            twin = all.putIfAbsent(obj.hashCode(), obj);
            if ((++created & 0xffffffff) == 0) {
                System.out.printf("%d created%n", created);
            }
        } while (twin == null);
        System.out.println("found twin: " + obj +
            " and " + twin + " but == is " + (obj == twin));
        System.out.println("Size of map is " + all.size());
    }
}
```

I ran this for a while with Java 7, after allocating 14 GB of memory, but could not find a twin:

```
16,777,216
33,554,432
50,331,648
```

```
67,108,864  
83,886,080  
100,663,296  
117,440,512  
134,217,728  
Exception in thread "main" OutOfMemoryError: Java heap space
```

However, in Java 8 with the new identity hash code algorithm, I found a twin almost immediately:

```
found twin: Object@7f385cbe and Object@7f385cbe but == is false  
Size of map is 105786
```

I asked on my [twitter feed](#) whether anybody knew how to make the JavaChampionTest produce a non-null URL for Kirk, but besides some nasty code by Peter Lawrey involving overwriting the object header, no one produced the answer I was looking for.

One last little gotcha to do with the identity hash code. It is assigned lazily. Thus if you call hashCode() on Object, it is calculated and written into the object header. Unfortunately space is at a premium in the object header and there is not enough space for the biased locking bits and the object identity. Thus as far as I can tell, objects that have had their identity hash code set, also automatically lose their ability to be part of biased locking. If they already have locks biased towards a thread, then calling System.identityHashCode(obj) will revoke the bias. You can read more about biased locking in [Dave Dice's blog](#).

Kind regards from Hinxton, home of the stunning European Bioinformatics Institute.

Heinz

Issue 220 - Runtime.getRuntime().availableProcessors()

Author: Dr. Heinz M. Kabutz

Date: 2014-05-28

Category: Java 8

Java Versions: Java 8

Abstract:

Java 8 magically makes all our code run in parallel. If you believe that, I've got a tower in Paris that I'm trying to sell. Really good price. In this newsletter we look at how the parallelism is determined and how we can influence it.

Welcome to the 220th issue of **The Java(tm) Specialists' Newsletter**, sent to you from the island of the Minotaur. Before Crete, I lived in South Africa for 35 years. I grew up in the surreal system of "apartheid". Everybody was grouped according to the pigmentation of their skin (paper bag test) and the curliness of their hair (pencil test). Low melanin levels made you "European". Otherwise you were "Non-European". Visitors from the USA would get confused and join the much longer "Non-European" queue. Since my family have lived in Africa since 1870, I never qualified for European citizenship. I am African, even though I look and sound German. Even German passport officials find it hard to believe that I am not German. And now I am Greek, thanks to the amazing generosity of the Hellenic State. Similarly, people have a hard time believing that I am, even when I show them my Greek ID. The look on their faces is priceless when I start speaking their language :-) If you are thinking of emigrating to Greece - I also did a PhD in Computer Science and that was easier and took less time.

Last week we did our **Java Specialist Master Course** here on Crete with some physical and some virtual attendees (see the TV on the back wall). I know that some managers think all we do here on the island courses is lie on the beach and drink ouzo. We do that too. But we also work hard. Here you can see my students trying to solve an exercise that I had told them would be easy. Yeah, easy if you know the tricks :-) The funniest is the look of despair on Giannis' face at the back in the middle. He looked at the question and thought he'd solve it in about 15 seconds. But 15 minutes later, it still did not work. The exercises are designed in such a way that you have a great "aha" moment at the end :-)



Runtime.getRuntime().availableProcessors()

Java 8 has been with us for two months and I'm getting requests from companies to deliver training on how to program with the new constructs. On my question, how soon they would be writing production code in Java 8, the answer is universally "not yet, we're just looking at it for now". Whenever a new major version is released, it takes a while for good coding idioms to be established. It was the same with Java 5 generics. Initially, programmers tried all sorts of very complicated things. I am guilty myself of doing things that were beyond the original design (see [Strategy Pattern with Generics](#)). But over the years we have started using them more sparingly and they are now most commonly used only to make collections a bit safer.

I believe we will have the same experience with Java 8, especially some of the cooler features, such as lambdas and parallel streams. The promise is that our code will magically run faster. The stream will automatically flow through the Fork/Join pool and be executed in parallel. I have heard a few talks already on the subject on Java 8 and they all contained mistakes on this very important topic. I will address parallel streams in more detail in a later newsletter, once I've had a chance to do some proper analysis of it. In this issue, I would like to ask a very simple question, but one that is really important because so much hinges on it. And the question is this: **Where do the threads for this parallel magic come from?**

In Java 8, we have a common Fork/Join pool, which we can access with `ForkJoinPool.commonPool()`. This is used for parallel streams, parallel sorting, `CompletableFuture`, etc. When you construct a Fork/Join pool, you do not specify the maximum number of threads. You instead specify a desired parallelism, which says how many active threads you would like to run at the same time. When a thread blocks on a phaser, another thread is created to keep the pool at the required active thread count. The phaser is the only synchronizer that will cause this behaviour. A Fork/Join pool has a maximum number of threads 32767, but most operating systems will fail with an `OutOfMemoryError` long before this number is reached. In this sample code, I fork new

RecursiveActions until we reach the first phase (after 200 threads have arrived). If we increase the phases to a larger number, say for example to 100_000, then this code will fail.

```

import java.util.concurrent.*;

public class PhaserForkJoin {
    public static void main(String... args) {
        ForkJoinPool common = ForkJoinPool.commonPool();
        Phaser phaser = new Phaser(200);
        common.invoke(new PhaserWaiter(phaser));
    }

    private static class PhaserWaiter extends RecursiveAction {
        private final Phaser phaser;

        private PhaserWaiter(Phaser phaser) {
            this.phaser = phaser;
            System.out.println(ForkJoinPool.commonPool().getPoolSize());
        }

        protected void compute() {
            if (phaser.getPhase() > 0) return; // we've passed first phase
            PhaserWaiter p1 = new PhaserWaiter(phaser);
            p1.fork();
            phaser.arriveAndAwaitAdvance();
            p1.join();
        }
    }
}

```

The Fork/Join pool thus does not have a practical maximum number of threads, only the desired parallelism, which shows us how many concurrently active threads we should allow.

Having a common pool is great, because it means that we can share the same pool for different types of jobs, without exceeding the total desired parallelism of the machine that the code is running on. Of course, if one of the threads blocks in any way besides with a Phaser, then this common pool will not perform nearly as well as was hoped.

What is the Default Desired Parallelism of the Common FJ Pool?

The default value for the desired parallelism of the common FJ pool is `Runtime.getRuntime().availableProcessors() - 1`. Thus if you take a dual-core

machine and try to run parallel sort with `Arrays.parallelSort()`, it will default to the ordinary `Arrays.sort()` method. Despite what you might have been promised during Oracle presentations, you would not see any speedup at all on a dual-core machine. (Even mentioned on [Developer.com](#) by someone who evidently didn't try it out himself.)

However, a bigger issue is that `Runtime.getRuntime().availableProcessors()` does not always return the value that you expect. For example, on my dual-core 1-2-1 machine, it returns the value 2, which is what I would expect. But on my 1-4-2 machine, meaning one socket, four cores and two hyperthreads per core, this method returns the value 8. However, I only have 4 cores and if the code is bottlenecked on CPU, will have 7 threads competing for the CPU cycles instead of a more reasonable 4. If my bottleneck is on memory, then I might get a 7x speedup on the test.

But that's not all! One of our fellow [Java Champions](#) found a case where he had a 16-4-2 machine (thus with 16 sockets, each with four cores and 2 hyperthreads per core) return the value 16! Based on the results on my i7 MacBook Pro, I would have expected the value to be $16 * 4 * 2 = 128$. Run Java 8 on this machine, and it will configure the common Fork/Join pool to have a parallelism of only 15. As Brian Goetz pointed out on the list, "The VM doesn't really have an opinion about what a processor is; it just asks the OS for a number. Similarly, the OS usually doesn't care either, it asks the hardware. The hardware responds with a number, usually the number of "hardware threads". The OS believes the hardware. The VM believes the OS."

Fortunately there is a workaround. On startup, you can specify the common pool parallelism with the system property

`java.util.concurrent.ForkJoinPool.common.parallelism`. Thus we could start this code with

`-Djava.util.concurrent.ForkJoinPool.common.parallelism=128` and it would show us that our parallelism is now 128:

```
import java.util.concurrent.*;

public class ForkJoinPoolCommon {
    public static void main(String... args) {
        System.out.println(ForkJoinPool.commonPool());
    }
}
```

We have two additional system properties for controlling the common pool. If you would like to handle uncaught exceptions, you could specify the handler class with `java.util.concurrent.ForkJoinPool.common.exceptionHandler`. And if you would prefer to have your own thread factory, that is configured with

`java.util.concurrent.ForkJoinPool.common.threadFactory`. The default thread factory for Fork/Join pool uses daemon threads, which you might want to avoid in your application. Be careful if you do that - you cannot shut down the common pool!

Whilst I was writing this newsletter, the article "**What's Wrong in Java 8, Part III: Streams and Parallel Streams**" by **Pierre-yves Saumont** arrived in my inbox. In it, he writes: "... *by default, all streams will use the same ForkJoinPool, configured to use as many threads as there are cores in the computer on which the program is running.*" However, as we saw in this newsletter, the default number is typically the number of hardware threads minus one, not cores! In addition, it might sometimes instead be the number of sockets minus one. He does point out an issue, in that once the pool has reached the desired parallelism level, it will no longer create new threads, even if there are lots of tasks waiting. Imagine for example a Java 8 application server that extensively uses parallel streams. They would all share the same common pool, potentially causing a bottleneck in the application server!

It is still early days for Java 8 adoption and I'm in no great hurry to rush a course out the door. You might find this hard to believe, but some of my customers still use Java 1.4.2. I even have someone coding Java 1.1. We are seeing the more adventurous customers slowly moving over to Java 7 now. It is a pity, as I really like some of the syntactic sugar of Java 8 and will be writing some newsletters soon about my findings.

Kind regards from Crete

Heinz

Issue 218 - Thread Confinement

Author: Dr. Heinz M. Kabutz

Date: 2014-02-18

Category: Concurrency

Java Versions: Java 7 and 8

Abstract:

One of the techniques we use to ensure that a non-threadsafe class can still be used by multiple threads is to give each thread its own instance. We call this "thread confinement". In this newsletter we look at some of the issues that can happen when this instance leaks.

Welcome to the 218th issue of **The Java(tm) Specialists' Newsletter**. If you missed accepting your invitation to our **4th annual JCrete unconference**, then I have sad news: we are completely full and have started a waiting list. As in previous years, the combination of sunshine, great food and deep intellectual stimulation proved too tempting to our fellow Java geeks. This year we have participants from 25 nations! Here is again the link to the **promotional video, showcasing what JCrete is all about, from interesting discussions about Java to hurling ourselves off treacherous cliffs**. Plus delicious Cretan food every day!

I wrote most of this newsletter in my little Suzuki Jimny at Kalathas beach, a bit more inspiring than sitting in a cubicle in a cold grey building. If I was retired, I would probably have spent my morning exactly the way I did. BTW, we call this "winter":



Thread Confinement

After studying Brian Goetz's excellent **Java Concurrency in Practice** book over a period of about 10 months in order to write my **Concurrency Specialist Course**, I came to the conclusion that two of the most useful tricks for ensuring thread safety are "stack confinement" and "thread confinement". In "stack confinement", we ensure that an object never escapes from a method. In "thread confinement", we only ever see a particular object from a single thread. Even if the object is not threadsafe, it now does not matter, since it is unshared.

Unfortunately there is no language support for enforcing "thread confined" objects. Let's start with a classic example of the `SimpleDateFormat`. It is one of the most tempting objects in the JDK. Typically, we would begin by having it "stack confined", such as:

```
public String stackConfined(Date date) {
    DateFormat df = new SimpleDateFormat("yyyy-MM-dd");
    return df.format(date);
}
```

After calling this method many times, we might discover that it is quite expensive to construct a new `SimpleDateFormat` object every time and thus be tempted to do the following:

```
private static final DateFormat df =
    new SimpleDateFormat("yyyy-MM-dd");
public String broken(Date date) {
    return df.format(date);
}
```

Of course, now the object is no longer "stack confined" and we will get all sorts of weird dates being returned to us. See also my [newsletter on "Wonky Dating"](#). A common trick is to make the object "thread confined", like so:

```
private static final ThreadLocal<DateFormat> tdf =
    new ThreadLocal<DateFormat>() {
```

```

protected DateFormat initialValue() {
    return new SimpleDateFormat("yyyy-MM-dd");
}
};

public String threadConfined(Date date) {
    return tdf.get().format(date);
}
}

```

This technique can cause memory leaks with managed threads. [See my newsletter 164](#) for more on ThreadLocal.

ThreadLocalRandom Puzzle

I recently sent this puzzle to the [Concurrency-interest mailing list](#). Before carrying on reading, please try answer the questions: What does this program do in Java 7? What does it do in Java 8?

```

import java.util.concurrent.*;
import java.util.concurrent.atomic.*;

public class MagicMirror {
    private static final ThreadLocalRandom tlr =
        ThreadLocalRandom.current();

    public boolean amIPretty() {
        return tlr.nextBoolean();
    }

    public static void main(String... args) {
        final AtomicBoolean vanity = new AtomicBoolean(true);
        while (vanity.get()) {
            new Thread(new Runnable() {
                public void run() {
                    MagicMirror mirrorOnTheWall = new MagicMirror();
                    boolean beauty = mirrorOnTheWall.amIPretty();
                    if (!beauty) vanity.set(false);
                }
            }).start();
        }
        System.out.println("Oh no, now I am depressed!");
    }
}

```

Before you carry on reading, please make sure that you have thought a bit about the code above. Maybe run it in both Java 7 and 8 to see if it matches your expectations.

Next puzzle. What is the output when we run this code that generates a random message? Try running it a few times in Java 7 and then run it in Java 8:

```
import javax.swing.*;
import java.awt.*;
import java.util.concurrent.*;

import static javax.swing.WindowConstants.*;

public class MagicMessage {
    private static final ThreadLocalRandom tlr =
        ThreadLocalRandom.current();

    public static void main(String... args) {
        SwingUtilities.invokeLater(new Runnable() {
            public void run() {
                JFrame frame = new JFrame();
                JLabel label = new JLabel(generateRandomString(),
                    SwingConstants.CENTER);
                frame.add(label, BorderLayout.NORTH);
                frame.setSize(300, 100);
                frame.setVisible(true);
                frame.setDefaultCloseOperation(EXIT_ON_CLOSE);
            }
        });
    }

    private static String generateRandomString() {
        char[] randomText = "HVTia\u0000ED1ciP".toCharArray();
        for (int i = 0; i < randomText.length; i++) {
            randomText[i] += tlr.nextInt(26);
        }
        return new String(randomText);
    }
}
```

Please try it out before you carry on reading ... :-) It will be worth it :-)

A Word About Random

We would imagine to sometimes get the same value twice in a row when generating random values. For example, when I play Tavli (Greek backgammon, which involve a lot of cheating, tsikoudia and laughter) with my synteknos, we sometimes throw the same dice three or four times in a row. As humans we usually attribute this to something other than chance. If we were asked to manually generate a sequence of random dice, we would probably not have the same two dice appear even twice in sequence. But in real life, this does happen.

I have [written before about how slow Math.random\(\) is](#), especially in a multi-threaded context, and that we should rather use ThreadLocalRandom since Java 7. I also mentioned that the correct way to use it was like so: `ThreadLocalRandom.current().nextInt();`

However, in Java 7, the random seeds for ThreadLocalRandom were stored inside a ThreadLocal field. Every time we called `ThreadLocalRandom.current()`, we did a hash map lookup until we found the correct seed value. This meant that if we needed to generate a lot of random values, it was tempting to store the instance somewhere, either inside a method, or perhaps in a field. If we stored it inside a method, then it would be "stack confined" and thus not available to other threads. But if we stored it in a field, then the reference would leak and thus no longer be "thread confined". It would be a mistake to do this.

Let's say we wanted to generate a bunch of random values and fill an array with these, such as this:

```
import java.util.concurrent.*;

public class RandomArrayFiller {
    public static void main(String... args) {
        int[] numbers = new int[100_000_000];

        for (int i = 0; i < 5; i++) {
            // thread confined
            long time = System.currentTimeMillis();
            fillRandomThreadConfined(numbers);
            time = System.currentTimeMillis() - time;
            System.out.println("Thread Confined took " + time + " ms");

            // stack confined
            time = System.currentTimeMillis();
            fillRandomStackConfined(numbers);
            time = System.currentTimeMillis() - time;
        }
    }

    private static void fillRandomThreadConfined(int[] numbers) {
        ThreadLocalRandom.current().ints(0, 100).asInts()
            .mapToObj(i -> i)
            .limit(100_000_000)
            .forEach(numbers::add);
    }

    private static void fillRandomStackConfined(int[] numbers) {
        ThreadLocalRandom.current().ints(0, 100).asInts()
            .mapToObj(i -> i)
            .limit(100_000_000)
            .forEach(numbers::add);
    }
}
```

```

System.out.println("Stack Confined took " + time + " ms");

// leaked
time = System.currentTimeMillis();
fillRandomLeaked(numbers);
time = System.currentTimeMillis() - time;
System.out.println("Leaked took " + time + " ms");
}

}

public static void fillRandomThreadConfined(int[] numbers) {
    for (int i = 0; i < numbers.length; i++) {
        numbers[i] = ThreadLocalRandom.current().nextInt();
    }
}

public static void fillRandomStackConfined(int[] numbers) {
    ThreadLocalRandom tlr = ThreadLocalRandom.current();
    for (int i = 0; i < numbers.length; i++) {
        numbers[i] = tlr.nextInt();
    }
}

static ThreadLocalRandom tlr = ThreadLocalRandom.current();

public static void fillRandomLeaked(int[] numbers) {
    for (int i = 0; i < numbers.length; i++) {
        numbers[i] = tlr.nextInt();
    }
}
}

```

In Java 7, the `fillRandomThreadConfined()` is a lot slower than the other two methods. In Java 8, they are closer in speed, but the `fillRandomThreadConfined()` is still a bit slower. If we ran this `fillRandomLeaked()` code from multiple threads, we would occasionally get the same value as race conditions occurred. Since we are working out random values anyway, this should not disturb us too much. But in Java 8, the seed values are stored inside the Thread and are initialized lazily when we call the `current()` method on `ThreadLocalRandom`. However, if we never call `current` on the thread, then the seeds stay at the default value, which means each thread would see the same sequence of random values. This explains the strange behaviour in my puzzles above.

I pointed this out to the concurrency interest mailing list, but was told that the `fillRandomLeaked()` method had broken the contract by storing a `ThreadLocalRandom` instance and thus deserved to be punished. In a way they are correct, but in my opinion lack an understanding of how programmers work in the real world. To leak the instance into a field

is a mistake that is easy to make, since there is no language construct to ensure thread confinement. I understand that they did not want to have too many dependencies between Thread and ThreadLocalRandom, but since the fields are now in Thread anyway, it would make more sense to me to simply do an eager initialization of the seed values. All it would require is an atomic long update. Thread is such an expensive resource to create anyway, that another couple of instructions at start up won't break the camel's back. I am pretty sure that the reasoning to do the initialization lazily was based on managing dependencies, rather than performance.

I tried to persuade the authors of the Java 8 ThreadLocalRandom to either:

- Initialize the seed values eagerly
- or, check that the seed has been set in the next() method, rather than in current()
- or, fail in next(), via exception or assertion, if the seed has not been set.

For now, everything will stay exactly as it is. So please, don't ever leak a ThreadLocalRandom instance, either by storing it in a field or passing it to a method or having it leak accidentally to an anonymous class. Here is an example that will only work in Java 8, as local variables can be "effectively final". In Java 7, it could be more obvious, in that you *have* to make local variables final if you want to use them from inside an anonymous class (or as in this case, from inside a class defined inside the method). And if you are one of those programmers who have the **incredibly annoying habit of marking ALL your local variables and parameters "final"**, then even in Java 7 it would be hard to see that tlr was being leaked into the instance of RandomFillerTask.

```
public static void fillRandomParallel(int[] numbers) {
    ThreadLocalRandom tlr = ThreadLocalRandom.current();

    class RandomFillerTask extends RecursiveAction {
        private static final int THRESHOLD = 100_000;
        private final int[] numbers;
        private final int offset;
        private final int length;

        RandomFillerTask(int[] numbers, int offset, int length) {
            this.numbers = numbers;
            this.offset = offset;
            this.length = length;
        }

        protected void compute() {
            if (length < THRESHOLD) {
                for (int i = offset; i < offset + length; i++) {

```

```
        numbers[i] = tlr.nextInt();
    }
} else {
    int offsetLeft = offset;
    int lengthLeft = length / 2;
    int offsetRight = offset + lengthLeft;
    int lengthRight = length - lengthLeft;
    RandomFillerTask left = new RandomFillerTask(
        numbers, offsetLeft, lengthLeft
    );
    RandomFillerTask right = new RandomFillerTask(
        numbers, offsetRight, lengthRight
    );
    invokeAll(left, right);
}
}

ForkJoinPool fjp = new ForkJoinPool();
fjp.invoke(new RandomFillerTask(numbers, 0, numbers.length));
fjp.shutdown();
}
```

Again, I would like to point out that the code above has a bug - we are letting the ThreadLocalRandom instance leak into other threads. But you can hopefully see that this is a mistake that is incredibly easy to make. Imagine every time you pressed the accelerator and break pedals at the same time, your motor exploded? Well, in your car's manual, there might be a warning to never do that. Unless you are a rally driver, why would you want to? But blowing up your motor seems a bit extreme, as does what happens here. Incidentally, no one on the list guessed the correct answer for the MagicMirror class.

Kind regards from Crete - hope to see you on one of our courses here, so we can enjoy it together :-)

Heinz

Issue 217b - Parallel Parking (Follow-up)

Author: Dr. Heinz M. Kabutz

Date: 2014-02-11

Category: Performance

Java Versions: Java 8

Abstract:

Heavily contended concurrent constructs may benefit from a small pause after a CAS failure, but this may lead to worse performance if we do more work in between each update attempt.

In this follow-up newsletter, we show how adding CPU consumption can change our performance characteristics.

I do not usually send a follow-up newsletter, but this one is too important to keep waiting for a long time. In my [last newsletter](#), I showed how parking on CAS failure could improve our performance on heavily contended CAS code. I also pointed out *twice* how one would need to carefully tune and choose the delay according to specific performance characteristics of one's system. But the message still did not come through. As a result, I decided to send a follow-up email to my [previous newsletter](#).

Parallel Parking (Follow-up)

Martin Thompson kindly pointed out that instead of using the park function, we really should use the "86 PAUSE instruction inside spin loops to avoid speculative execution that cause latency in such tests and reduce the waste of CPU resource when spinning." Unfortunately this function is not available in Java. Maybe in future we can have a Thread.pause(). Thread.yield() is not the same, as it is OS dependent. Here is the C++ code from the file jvm.cpp:

```
JVM_ENTRY(void, JVM_Yield(JNIEnv *env, jclass threadClass))
JVMWrapper("JVM_Yield");
if (os::dont_yield()) return;
#ifndef USDT2
HS_DTRACE_PROBE0(hotspot, thread__yield);
#else /* USDT2 */
HOTSPOT_THREAD_YIELD();
#endif /* USDT2 */
// When ConvertYieldToSleep is off (default), this matches the
// classic VM use of yield.
// Critical for similar threading behaviour
if (ConvertYieldToSleep) {
```

```

        os::sleep(thread, MinSleepInterval, false);
    } else {
        os::yield();
    }
JVM_END

```

The flag `-XX:+DontYieldALot` effectively turns the `Thread.yield()` into a no-op on all the platforms I looked at.

In order to demonstrate the danger of having a delay that is too long in a less contended lock, I modified the benchmark to add some busy CPU work in between each of the update methods using the JMH `BlackHole.consumeCPU()` method. Here is my new Benchmarking class. The larger the `TOKENS` environment variable, the more CPU work is done:

```

package eu.javaspecialists.microbenchmarks.numberrange;

import org.openjdk.jmh.annotations.*;
import org.openjdk.jmh.logic.*;

import static java.util.concurrent.TimeUnit.*;

@BenchmarkMode(Mode.AverageTime)
@OutputTimeUnit(NANOSECONDS)
@Warmup(iterations = 5, time = 1, timeUnit = SECONDS)
@Measurement(iterations = 10, time = 1, timeUnit = SECONDS)
@Fork(5)
@State(Scope.Benchmark)
@Threads(8)
public class Microbenchmark0 {
    public static final int TOKENS = Integer.getInteger("TOKENS");
    private final NumberRangeSynchronized nrSync =
        new NumberRangeSynchronized();
    private final NumberRangeAtomic nrAtomic =
        new NumberRangeAtomic();
    private final NumberRangeAtomicWithPark nrAtomicPark =
        new NumberRangeAtomicWithPark();

    @GenerateMicroBenchmark
    public void test_0_1_synchronized() {
        nrSync.setUpper(100);
        BlackHole.consumeCPU(TOKENS);
        nrSync.setLower(10);
    }
}

```

```

    BlackHole.consumeCPU(TOKENS);
    nrSync.setLower(50);
    BlackHole.consumeCPU(TOKENS);
    nrSync.setUpper(200);
    BlackHole.consumeCPU(TOKENS);
    nrSync.setLower(30);
    BlackHole.consumeCPU(TOKENS);
}

@GenerateMicroBenchmark
public void test_0_2_atomic() {
    nrAtomic.setUpper(100);
    BlackHole.consumeCPU(TOKENS);
    nrAtomic.setLower(10);
    BlackHole.consumeCPU(TOKENS);
    nrAtomic.setLower(50);
    BlackHole.consumeCPU(TOKENS);
    nrAtomic.setUpper(200);
    BlackHole.consumeCPU(TOKENS);
    nrAtomic.setLower(30);
    BlackHole.consumeCPU(TOKENS);
}

@GenerateMicroBenchmark
public void test_0_3_atomic_park() {
    nrAtomicPark.setUpper(100);
    BlackHole.consumeCPU(TOKENS);
    nrAtomicPark.setLower(10);
    BlackHole.consumeCPU(TOKENS);
    nrAtomicPark.setLower(50);
    BlackHole.consumeCPU(TOKENS);
    nrAtomicPark.setUpper(200);
    BlackHole.consumeCPU(TOKENS);
    nrAtomicPark.setLower(30);
    BlackHole.consumeCPU(TOKENS);
}
}

```

In order to show the data without too much clutter, I've left out the "mean error" values, but all the raw data is available on demand. These results are when we use 180 microseconds as a backoff delay.

| | | | | | | | | | | |
|----------|---|----|----|----|----|-----|-----|-----|------|------|
| -DTOKEN= | 0 | 10 | 20 | 40 | 80 | 160 | 320 | 640 | 1280 | 2560 |
|----------|---|----|----|----|----|-----|-----|-----|------|------|

```
synchronized 1623 2626 3444 5278 5924 6182 5429 8146 15067 29407
atomic 4229 3412 3509 3295 2733 2053 3663 7160 13947 27854
atomic_park 485 844 1498 3507 6504 9917 14317 18814 27844 38242
```

The atomic parking version is faster when the update method is extremely highly contended. However, as the contention lessens, the parking starts to take over as a main denominator. We see that in some cases, the parking version is about 4x slower than the plain CAS version.

I also ran the same experiment with less threads than 8. Here are the results with 4 threads on my i7 machine:

```
-DTOKEN= 0 10 20 40 80 160 320 640 1280 2560
synchronized 804 1227 1737 2593 2706 2333 3517 6775 13233 26155
atomic 1604 1515 1484 1190 1013 1749 3314 6490 12878 25539
atomic_park 282 476 753 1857 2260 2060 5457 7307 13847 27159
```

And here are the results with 2 threads on my i7 machine:

```
-DTOKEN= 0 10 20 40 80 160 320 640 1280 2560
synchronized 390 605 555 1027 1011 1797 3346 6479 12617 24929
atomic 615 583 692 472 870 1651 3162 6268 12353 24670
atomic_park 126 197 338 697 1093 2202 3531 7303 12723 25001
```

Running the same experiment with 8 threads on my 2-4-1 server (total of 8 cores) gave the following results:

```
-DTOKEN= 0 10 20 40 80 160 320 640 1280 2560
synchronized 6346 8019 9027 7760 7202 8517 6562 12632 24941 49250
atomic 1430 1129 1147 1237 1589 2956 5758 11238 22410 44647
atomic_park 795 2007 3760 6229 7140 7015 7565 13423 25214 49557
```

Thus the technique for pausing for a short while after a CAS failure would only help you if the update code is very heavily contended, otherwise it would be counterproductive.

As with most benchmarks like this, we are experiencing lots of interesting effects as a result of cache miss costs. Please don't add parks to all your classes and expect your system to magically become 10x faster. As always, we want to tune our identified and proven bottlenecks. If you'd like to know more on how to do that, please have a look at our [Java Specialist Master Course](#).

Kind regards from Crete

Heinz

Issue 217 - Parallel Parking

Author: Dr. Heinz M. Kabutz

Date: 2014-02-07

Category: Performance

Java Versions: Java 8

Abstract:

What is faster? Synchronized or Atomic compare-and-set? In this newsletter we look at some surprising results when we run Java on the new i7 chip.

Welcome to the 217th issue of **The Java(tm) Specialists' Newsletter**. In my [last newsletter](#), I mentioned we'd be heading off in search of snow on the Island of Crete. It took a bit of effort, but we were successful! Have a look at a [short clip of our adventure](#) (make sure to watch in HD).

From the 25-29 August 2014, we are running the 4th Java Specialists' Symposium aka [JCrete 2014](#). [Note that the venue has changed to the Kolymbari area of Chania - more information coming soon]. Here are some comments from previous attendees:

"In conferences with huge numbers of attendees you'll get great canned talks and lots of merchandise from the sponsors. You might even get a good meal at the hotel restaurant. If that's what you like don't go to JCrete. In JCrete it's extremely, and sunny, you only get 3 talks a day and wifi is poor. You talk with other attendees, get to know them while having a salad at a small taverna by the beach (sometimes a taverna _in_ the beach). You get to meet the great people like I met Maurice Naftalin, Heinz Kabutz or Martin Thompson while drinking home-made Raki. You get to spend all-nighters discussing last Vimeo hit by Bret Victor with people worried about our future as developer/computer scientists. That and great small-group sessions on garbage collecting, language abusing, concurrency, java byte code and many other subjects is what you get at JCrete. But if you are the type that enjoys the Martinis at the Oracle Evening Party at QCon then JCrete is not for you." - *Ignasi Marimon-Clos i Sunyol (Spain)*

And another one by a regular attendee:

"JCrete is the best combination of multiple worlds. In the morning you are engaged in interesting topics with highly intelligent professionals, often learning new stuff or getting interesting views on existing technology. In the afternoon you spend time with your family or fellow geeks, snorkeling, hiking or just relaxing. Later you eventually gather with the rest of the hard-cores in the foo-bar discussing java, politics, railways or whatever. And at all times you are supported by beautiful surroundings, Greek hospitality and great local food!" - *Jesper Uddy (Denmark)*

Oh and last, but not least, **a short video to demonstrate the types of things we do at our Unconference**. It is very different to other more traditional conferences. We might only have three sessions per day, but the discussions carry on whilst we are swimming in the sea or enjoying a delicious meal. Places are limited to 70 attendees, **so please put your name down on the list ASAP**. There are no conference fees, but you need to cover your own travel expenses (flight, food, hotel, entertainment).

Parallel Parking

After waiting far too long, I finally upgraded to a quad-core i7. Proud to show off my shiny new hardware and hack programming skills to some **Concurrency Specialist Course** students, I fired it up and waited for them to be dazzled by the awesome speed. But let me go back a few steps. We ended day one of the concurrency course with various versions of NumberRange. The idea was inspired by Brian Goetz's excellent book **Java Concurrency in Practice**. We start with an interface:

```
package eu.javaspecialists.microbenchmarks.numberrange;

public interface NumberRange {
    boolean setLower(int newLower);
    boolean setUpper(int newUpper);
    boolean isInRange(int number);
    boolean checkCorrectRange();
}
```

In **JCIP**, Brian explains that this version of NumberRange is broken, because even though the lower and upper objects are in themselves threadsafe, we have an invariant across both fields that is not protected through some form of synchronization:

```
package eu.javaspecialists.microbenchmarks.numberrange;

import java.util.concurrent.atomic.*;

// does not work correctly - we could get upper < lower!
public class NumberRangeAtomicBroken implements NumberRange {
    private final AtomicInteger lower = new AtomicInteger(0);
    private final AtomicInteger upper = new AtomicInteger(0);

    public boolean setLower(int newLower) {
        if (newLower > upper.get()) return false;
    }
}
```

```

        lower.set(newLower);
        return true;
    }

    public boolean setUpper(int newUpper) {
        if (newUpper < lower.get()) return false;
        upper.set(newUpper);
        return true;
    }

    public boolean isInRange(int number) {
        return number >= lower.get() && number <= upper.get();
    }

    public boolean checkCorrectRange() {
        int lower = this.lower.get();
        int upper = this.upper.get();
        return lower <= upper;
    }
}

```

It is easy to reason why this does not work, but here is a unit test that demonstrates the problem (if you still have questions, read the book :-)):

```

private void testConcurrently(NumberRange range)
    throws InterruptedException {
    range.setLower(0);
    range.setUpper(200);
    ExecutorService pool = Executors.newCachedThreadPool();
    pool.submit(() -> {
        for (int i = 0; i < 100_000_000; i++) {
            range.setLower(100);
            range.setLower(0);
        }
    });
    pool.submit(() -> {
        for (int i = 0; i < 100_000_000; i++) {
            range.setUpper(40);
            range.setUpper(200);
        }
    });
    Future<Boolean> checker = pool.submit(() -> {
        for (int i = 0; i < 100_000_000; i++) {

```

```

        if (!range.checkCorrectRange()) return false;
    }
    return true;
});
try {
    assertTrue(checker.get());
} catch (ExecutionException e) {
    fail(e.getCause().toString());
}
pool.shutdown();
}
}

```

There is a very easy fix and that is to use synchronized instead of AtomicInteger fields. For example:

```

package eu.javaspecialists.microbenchmarks.numberrange;

public class NumberRangeSynchronized implements NumberRange {
    // INVARIANT: lower <= upper
    private int lower = 0;
    private int upper = 0;

    public synchronized boolean setLower(int newLower) {
        if (newLower > upper) return false;
        lower = newLower;
        return true;
    }

    public synchronized boolean setUpper(int newUpper) {
        if (newUpper < lower) return false;
        upper = newUpper;
        return true;
    }

    public synchronized boolean isInRange(int number) {
        return number >= lower && number <= upper;
    }

    public synchronized boolean checkCorrectRange() {
        return lower <= upper;
    }
}

```

However, this means that we have synchronized methods and so the threads will only be able to call these one at a time.

I then proceeded to show another much cleverer solution that used long and bit masking, together with a compare-and-swap (CAS) to set upper and lower in a single operation. Here it is:

```
package eu.javaspecialists.microbenchmarks.numberrange;

import java.util.concurrent.atomic.*;
import java.util.concurrent.locks.*;

public class NumberRangeAtomic implements NumberRange {
    private final AtomicLong range = new AtomicLong(0);

    public boolean setLower(int newLower) {
        while (true) {
            long current = range.get();
            int upper = getUpper(current);
            int lower = getLower(current);
            if (newLower > upper) return false;
            long next = combine(newLower, upper);
            if (range.compareAndSet(current, next)) return true;
        }
    }

    public boolean setUpper(int newUpper) {
        while (true) {
            long current = range.get();
            int upper = getUpper(current);
            int lower = getLower(current);
            if (newUpper < lower) return false;
            long next = combine(lower, newUpper);
            if (range.compareAndSet(current, next)) return true;
        }
    }

    public boolean isInRange(int number) {
        long current = range.get();
        int upper = getUpper(current);
        int lower = getLower(current);
        return number >= lower && number <= upper;
    }
}
```

```

private long combine(long lower, long upper) {
    return upper | (lower << 32);
}

private int getLower(long range) {
    return (int) (range >>> 32);
}

private int getUpper(long range) {
    return (int) (range);
}

public boolean checkCorrectRange() {
    long range = this.range.get();
    int lower = getLower(range);
    int upper = getUpper(range);
    return lower <= upper;
}
}

```

We finished for the day and said our virtual goodbyes. My students were in Brazil(1), Luxembourg(1), USA(2), England(1), Austria(1), Romania(1), Sweden(1) and I was at home in Greece(1). So for some of them, the day had just begun and for others, it was beer time. These virtual classes are fun and we all learn a lot. We keep numbers small so that everyone gets a chance to ask their questions. Have a look at our [Concurrency](#), [Master](#) and [Patterns](#) Courses.

Each morning, I like to start by giving my students an opportunity to ask questions that they had come up with in the night. Our brain needs a while to reset itself and so taking a complete break often brings doubts to the surface that they did not realize they had. Sometimes this Q&A session goes on for over an hour as we go freestyle off the script and answer the questions that students have.

In our particular Q&A session, James S. asked what the point of that bit shifting code was. It seemed so much more complicated to him and he could not justify writing code that had such a high chance of bugs. I explained that this type of programming style would be typically used for identified bottlenecks. It would be much faster though, I confidently asserted. To prove it, I whipped up a quick microbenchmark. Now I've written hundreds of microbenchmarks and they usually give me the results I want pretty quickly. However, in this case, it showed me that the NumberRangeSynchronized class was actually several times *faster* than the far more complicated NumberRangeAtomic. I had seen similar results on other fast i7 laptops, but since I had only had my machine for a couple of days, was not able to investigate exactly why. I muttered something about "cache synchronization" and "data moving between cores" and "CAS being a known weakness", but was not satisfied with my answer. I also told them that there was something funny going on and that I would investigate

this further.

Instead of showing you the slapped-together benchmark that I coded live in class, I'd rather show you how to do it with [OpenJDK's excellent Java Microbenchmarking Harness \(JMH\) tool](#). JMH takes care of all the benching things, such as statistics, starting and stopping threads, forking the processes, etc. One thing I really like is that JMH typically measures the amount of work done within a time interval, rather than measuring how long a certain number of steps takes. This is also my preferred way of measuring code. Of course, even JMH can sometimes be subject to weird Hotspot optimizations, so please check whatever results you get and see whether they make sense. If unsure, inspect the generated assembler code.

```
package eu.javaspecialists.microbenchmarks.numberrange;

import org.openjdk.jmh.annotations.*;
import static java.util.concurrent.TimeUnit.*;

@BenchmarkMode(Mode.AverageTime)
@OutputTimeUnit(NANOSECONDS)
@Warmup(iterations = 10, time = 1, timeUnit = SECONDS)
@Measurement(iterations = 20, time = 1, timeUnit = SECONDS)
@Fork(5)
@State(Scope.Benchmark)
@Threads(8)
public class Microbenchmark1 {
    private final NumberRangeSynchronized nrSync =
        new NumberRangeSynchronized();
    private final NumberRangeAtomic nrAtomic =
        new NumberRangeAtomic();

    @GenerateMicroBenchmark
    public void test_1_1_synchronized() {
        nrSync.setUpper(100);
        nrSync.setLower(10);
        nrSync.setLower(50);
        nrSync.setUpper(200);
        nrSync.setLower(30);
    }

    @GenerateMicroBenchmark
    public void test_1_2_atomic() {
        nrAtomic.setUpper(100);
        nrAtomic.setLower(10);
        nrAtomic.setLower(50);
        nrAtomic.setUpper(200);
    }
}
```

```

        nrAtomic.setLower(30);
    }
}

```

I will leave the exercise of getting this to run on your machine as an exercise to the reader. It's really simple though - just follow the instructions on the [JMH website](#). JMH could not have made it easier and I am embarrassed that I have so far been too lazy to bother learning it. Anyway, here are the results:

| Benchmark | Mean | Mean error | Units |
|---------------------------------------|----------|------------|-------|
| Microbenchmark1.test_1_1_synchronized | 1552.081 | 15.408 | ns/op |
| Microbenchmark1.test_1_2_atomic | 3855.470 | 77.010 | ns/op |

As we can see in this run, our fancy NumberRangeAtomic was more than twice as slow as the NumberRangeSynchronized! All that effort for nothing!

However, the answer came in the form of [Jack Shirazi's Java Performance Tuning Newsletter](#). In his latest bulletin, he explains that locking can outperform compare-and-swap (CAS), especially if there are a lot of repeats. The trick is that if we have a CAS failure to let the losing thread sleep a little bit in order to give the other threads a chance to go through more often. Even a constant time sleep could improve the situation.

After trying different settings, I settled on a figure of 0.18 ms as working well. But this could be completely different on your machine so it would be best to make this figure tunable or maybe auto-adjust at runtime. Thus we only need to add LockSupport.parkNanos(180_000) after the compareAndSet method has returned false, like so:

```

public boolean setLower(int newLower) {
    while (true) {
        long current = range.get();
        int upper = getUpper(current);
        int lower = getLower(current);
        if (newLower > upper) return false;
        long next = combine(newLower, upper);
        if (range.compareAndSet(current, next)) return true;
        // now we sleep a constant time
    }
}

```

```

        LockSupport.parkNanos(180_000);
    }
}

public boolean setUpper(int newUpper) {
    while (true) {
        long current = range.get();
        int upper = getUpper(current);
        int lower = getLower(current);
        if (newUpper < lower) return false;
        long next = combine(lower, newUpper);
        if (range.compareAndSet(current, next)) return true;
        // now we sleep a constant time
        LockSupport.parkNanos(180_000);
    }
}

```

Let's run another benchmark, this time with the new NumberRangeAtomicWithPark class that parks after the failure:

```

package eu.javaspecialists.microbenchmarks.numberrange;

import org.openjdk.jmh.annotations.*;
import static java.util.concurrent.TimeUnit.*;

@BenchmarkMode(Mode.AverageTime)
@OutputTimeUnit(NANOSECONDS)
@Warmup(iterations = 10, time = 1, timeUnit = SECONDS)
@Measurement(iterations = 20, time = 1, timeUnit = SECONDS)
@Fork(5)
@State(Scope.Benchmark)
@Threads(8)
public class Microbenchmark2 {
    private final NumberRangeAtomicWithPark nrAtomicPark =
        new NumberRangeAtomicWithPark();

    @GenerateMicroBenchmark
    public void test_2_1_atomic_park() {
        nrAtomicPark.setUpper(100);
        nrAtomicPark.setLower(10);
        nrAtomicPark.setLower(50);
        nrAtomicPark.setUpper(200);
        nrAtomicPark.setLower(30);
    }
}

```

```

    }
}

```

Here are the results with the parking atomic included:

| Benchmark | Mean | Mean error | Units |
|---------------------------------------|----------|------------|-------|
| Microbenchmark1.test_1_1_synchronized | 1552.081 | 15.408 | ns/op |
| Microbenchmark1.test_1_2_atomic | 3855.470 | 77.010 | ns/op |
| Microbenchmark2.test_2_1_atomic_park | 430.649 | 3.125 | ns/op |

We see how the performance has increased almost 10x, simply by parking when we have a CAS failure due to parallel threads trying to write at the same time. So sleeping can actually make your program run faster?

We also see from those results that the NumberRangeAtomicWithPark class is now about 4x faster than the NumberRangeSynchronized. Since I have 4 cores, these results are believable. The CAS code *should* be faster than the exclusive pessimistic code.

A word of warning though. The 180 microseconds seemed to give good results on my machine, but may be quite bad on yours. Making it configurable would allow you to fine tune it for your target machine.

Please note that I've also [published a follow-up to this article](#), which shows what happens when we do some CPU work in between calls to the update methods.

Kind regards from Crete

Heinz

Issue 215 - StampedLock Idioms

Author: Dr. Heinz M. Kabutz

Date: 2013-12-02

Category: Java 8

Java Versions: Java 8

Abstract:

Java 8 includes a new synchronization mechanism called StampedLock. It differentiates between exclusive and non-exclusive locks, similar to the ReentrantReadWriteLock.

However, it also allows for optimistic reads, which is not supported by the ReentrantReadWriteLock. In this newsletter, we look at some idioms on how to use this new synchronizer.

Welcome to the 215th issue of **The Java(tm) Specialists' Newsletter**, sent from the Island of Crete. As you would probably have heard by now, Greece is in a state of crisis. Lots of my friends have lost jobs or have had their hours reduced. On one of my flights, I learned about a great new program called "**Boroume**" - meaning "we can". It connects those with excess food (bakeries, hotels, restaurants, supermarkets) with those that have a need for food (orphanages, old-age homes, shelters). Thus instead of the perfectly good yesterday's bread being thrown away, it can now be consumed by someone in need. They seem to run a lean operation, meaning they do not need lots of funds to get a lot done. I sent a little bit and was overwhelmed by their response. I like clever projects like this, that maximize the available resources we have in our country.

Please note that I will present a free webinar about this newsletter on Friday the 6th December 2013 at 08:00 UTC. I expect it to take about 30 minutes, but it might take longer if we get lots of interesting questions. Usually this talk results in fascinating discussions. I will make a recording available afterwards. Please join me if you'd like to hear more or would like to ask some questions. **Simply click here**, fill in your name and email address and you will automatically be registered. You will also be invited to the next two free webinars about the StampedLock.

StampedLock

As you should know by now, I have a keen interest in concurrency. Even before I wrote the **Concurrency Specialist Course**, I was doing research for this newsletter. For example, I showed in **2008 how the ReadWriteLock could starve certain threads**. In Java 5, the writers were starved and in Java 6, a single reader with many writers could go hungry for CPU cycles. The Java 5 flaw can easily be explained, but the reader starvation was not so obvious and could only be seen with careful experimentation.

The idioms for the ReentrantLock and ReentrantReadWriteLock are quite similar.

```

Lock lock = new ReentrantLock();

...

lock.lock();
try {
    // do something that needs exclusive access
} finally {
    lock.unlock();
}

```

And here is the ReentrantReadWriteLock:

```

ReadWriteLock rwlock = new ReentrantReadWriteLock();

...

rwlock.writeLock().lock();
try {
    // do something that needs exclusive access
} finally {
    rwlock.writeLock().unlock();
}

```

Even though these idioms were straightforward, a lot of programmers still got them wrong. Even books and articles had incorrect idioms. For example, some wrote the lock() method call into the try block. Or they did not have a finally block to do the unlocking. Using synchronized was so much easier, as the locking/unlocking was ensured by the syntax of the language.

The ReentrantReadWriteLock had a lot of shortcomings: It suffered from starvation. You could not upgrade a read lock into a write lock. There was no support for optimistic reads. Programmers "in the know" mostly avoided using them.

Doug Lea's new Java 8 StampedLock addresses all these shortcomings. With some clever code idioms we can also get better performance. Instead of the usual locking, it returns a long number whenever a lock is granted. This stamp number is then used to unlock again. For example, here is how the code above would look:

```

StampedLock sl = new StampedLock();

...

long stamp = sl.writeLock();
try {
    // do something that needs exclusive access
} finally {
    sl.unlockWrite(stamp);
}

```

This still begs the question - why do we need this in the first place? Let's take a slightly larger example and write it with different approaches: synchronized, synchronized/volatile, ReentrantLock, ReentrantReadWriteLock and StampedLock. Lastly we will show how the immutable approach.

BankAccount Without Any Synchronization

Our first BankAccount has no synchronization at all. It also does not check that the amounts being deposited and withdrawn are positive. We will leave out argument checking in our code.

```

public class BankAccount {
    private long balance;

    public BankAccount(long balance) {
        this.balance = balance;
    }

    public void deposit(long amount) {
        balance += amount;
    }

    public void withdraw(long amount) {
        balance -= amount;
    }

    public long getBalance() {

```

```

    return balance;
}
}

```

Synchronized BankAccount

The second version uses the **synchronized** keyword to protect the methods from being called simultaneously by multiple threads on the same BankAccount object. We could either synchronize on "this" or on a private field. For our example, this would not make a difference. In this approach a thread would not be able to read the balance whilst another thread is depositing money.

```

public class BankAccountSynchronized {
    private long balance;

    public BankAccountSynchronized(long balance) {
        this.balance = balance;
    }

    public synchronized void deposit(long amount) {
        balance += amount;
    }

    public synchronized void withdraw(long amount) {
        balance -= amount;
    }

    public synchronized long getBalance() {
        return balance;
    }
}

```

Synchronized Volatile BankAccount

The third version uses a combination of the **volatile** and **synchronized** keywords. Only one thread at a time can update the balance, but another thread could read the balance in the middle of an update. The volatile modifier is absolutely necessary, since the value might get cached by a thread and so they would not necessarily see the latest value. Also, since it is a 64-bit value, it could get set in two operations on a 32-bit JVM, leading to impossible values. However, this is a very good solution, as it gives us fast reads and is easy to write

and understand.

```
public class BankAccountSynchronizedVolatile {
    private volatile long balance;

    public BankAccountSynchronizedVolatile(long balance) {
        this.balance = balance;
    }

    public synchronized void deposit(long amount) {
        balance += amount;
    }

    public synchronized void withdraw(long amount) {
        balance -= amount;
    }

    public long getBalance() {
        return balance;
    }
}
```

ReentrantLock BankAccount

The fourth version is similar to the BankAccountSynchronized, except that we are using explicit locks. So what is "better" - ReentrantLock or synchronized? As you can see below, it is a lot more effort to code the ReentrantLock. In Java 5, the performance of *contended* ReentrantLocks was a lot better than synchronized. However, synchronized was greatly improved for Java 6, so that now the difference is minor. In addition, *uncontended* synchronized locks can sometimes be automatically optimized away at runtime, leading to great performance improvements over ReentrantLock. The only reason to use ReentrantLock nowadays is if you'd like to use the more advanced features such as better timed waits, tryLock, etc. Performance is no longer a good reason.

```
import java.util.concurrent.locks.*;

public class BankAccountReentrantLock {
    private final Lock lock = new ReentrantLock();
    private long balance;
```

```

public BankAccountReentrantLock(long balance) {
    this.balance = balance;
}

public void deposit(long amount) {
    lock.lock();
    try {
        balance += amount;
    } finally {
        lock.unlock();
    }
}

public void withdraw(long amount) {
    lock.lock();
    try {
        balance -= amount;
    } finally {
        lock.unlock();
    }
}

public long getBalance() {
    lock.lock();
    try {
        return balance;
    } finally {
        lock.unlock();
    }
}
}

```

ReentrantReadWriteLock BankAccount

The fifth version uses the ReentrantReadWriteLock, which differentiates between *exclusive* and *non-exclusive* locks. In both cases, the locks are pessimistic. This means that if a thread is currently holding the write lock, then any reader thread will get suspended until the write lock is released again. It is thus different to the BankAccountSynchronizedVolatile version, which would allow us to read the balance whilst we were busy updating it. However, the overhead of using a ReentrantReadWriteLock is substantial. As a ballpark figure, we need the read lock to execute for about 2000 clock cycles in order to win back the cost of using it. In our case the getBalance() method does substantially less, so we would probably be better off just using a normal ReentrantLock.

```

import java.util.concurrent.locks.*;

public class BankAccountReentrantReadWriteLock {
    private final ReadWriteLock lock = new ReentrantReadWriteLock();
    private long balance;

    public BankAccountReentrantReadWriteLock(long balance) {
        this.balance = balance;
    }

    public void deposit(long amount) {
        lock.writeLock().lock();
        try {
            balance += amount;
        } finally {
            lock.writeLock().unlock();
        }
    }

    public void withdraw(long amount) {
        lock.writeLock().lock();
        try {
            balance -= amount;
        } finally {
            lock.writeLock().unlock();
        }
    }

    public long getBalance() {
        lock.readLock().lock();
        try {
            return balance;
        } finally {
            lock.readLock().unlock();
        }
    }
}

```

StampedLock BankAccount

Our sixth version uses StampedLock. I have written two getBalance() methods. The first uses pessimistic read locks, the other optimistic. In our case, since there are no invariants on the fields that would somehow restrict the values, we never need to have a pessimistic lock. Thus the optimistic read is only to ensure memory visibility, much like the BankAccountSynchronizedVolatile approach.

```
import java.util.concurrent.locks.*;

public class BankAccountStampedLock {
    private final StampedLock sl = new StampedLock();
    private long balance;

    public BankAccountStampedLock(long balance) {
        this.balance = balance;
    }

    public void deposit(long amount) {
        long stamp = sl.writeLock();
        try {
            balance += amount;
        } finally {
            sl.unlockWrite(stamp);
        }
    }

    public void withdraw(long amount) {
        long stamp = sl.writeLock();
        try {
            balance -= amount;
        } finally {
            sl.unlockWrite(stamp);
        }
    }

    public long getBalance() {
        long stamp = sl.readLock();
        try {
            return balance;
        } finally {
            sl.unlockRead(stamp);
        }
    }

    public long getBalanceOptimisticRead() {
        long stamp = sl.tryOptimisticRead();
        long balance = this.balance;
        if (!sl.validate(stamp)) {
            stamp = sl.readLock();
            try {
                balance = this.balance;
            } finally {
                sl.unlockRead(stamp);
            }
        }
        return balance;
    }
}
```

```

        }
    }
    return balance;
}
}

```

In our `getBalanceOptimisticRead()`, we could retry several times. However, as I said before, if memory visibility is all we care about, then `StampedLock` is overkill.

Immutable BankAccount

Our seventh version has an immutable `BankAccount`. Whenever we need to change the balance, we create a new account. Most Java programmers have a knee-jerk reaction to this: "Ah, but this will create too many objects!" This might be true. However, contended `ReentrantLocks` also create objects. Thus this argument is not always valid. You might be better off using a non-blocking algorithm that simply creates a new account and writes it into an `AtomicReference` using a `CompareAndSwap` (CAS).

```

public class BankAccountImmutable {
    private final long balance;

    public BankAccountImmutable(long balance) {
        this.balance = balance;
    }

    public BankAccountImmutable deposit(long amount) {
        return new BankAccountImmutable(balance + amount);
    }

    public BankAccountImmutable withdraw(long amount) {
        return new BankAccountImmutable(balance - amount);
    }

    public long getBalance() {
        return balance;
    }
}

```

Atomic BankAccount

We need an eighth version, just to satisfy the naysayers who want to see an atomic solution. Here we could either store the balance inside an AtomicLong or inside a volatile long and then use an AtomicLongFieldUpdater or Unsafe to set the field using a CAS.

```
import java.util.concurrent.atomic.*;

public class BankAccountAtomic {
    private final AtomicLong balance;

    public BankAccountAtomic(long balance) {
        this.balance = new AtomicLong(balance);
    }

    public void deposit(long amount) {
        balance.addAndGet(amount);
    }

    public void withdraw(long amount) {
        balance.addAndGet(-amount);
    }

    public long getBalance() {
        return balance.get();
    }
}
```

W H Y ???

We have many ways to write similar type of code. In our very simple BankAccount example, the BankAccountSynchronizedVolatile and BankAccountAtomic solutions are both simple and work very well. However, if we had multiple fields and/or an invariant over the field, we would need a slightly stronger mechanism. Let's take for example a point on a plane, consisting of an "x" and a "y". If we move in a diagonal line, we want to update the x and y in an atomic operation. Thus a moveBy(10,10) should never expose a state where x has moved by 10, but y is still at the old point. The fully synchronized approach would work, as would the ReentrantLock and ReentrantReadWriteLock. However, all of these are pessimistic. How can we read state in an optimistic approach, expecting to see the correct values?

Let's start by defining a simple Point class that is synchronized and has three methods for reading and manipulating the state:

```

public class Point {
    private int x, y;

    public Point(int x, int y) {
        this.x = x;
        this.y = y;
    }

    public synchronized void move(int deltaX, int deltaY) {
        x += deltaX;
        y += deltaY;
    }

    public synchronized double distanceFromOrigin() {
        return Math.hypot(x, y);
    }

    public synchronized boolean moveIfAt(int oldX, int oldY,
                                         int newX, int newY) {
        if (x == oldX && y == oldY) {
            x = newX;
            y = newY;
            return true;
        } else {
            return false;
        }
    }
}

```

If we use a StampedLock, our move() method look similar to the BankAccountStampedLock.deposit():

```

public void move(int deltaX, int deltaY) {
    long stamp = sl.writeLock();
    try {
        x += deltaX;
        y += deltaY;
    } finally {
        sl.unlockWrite(stamp);
    }
}

```

However, the `distanceFromOrigin()` method could be rewritten to use an optimistic read, for example:

```
public double distanceFromOrigin() {
    long stamp = sl.tryOptimisticRead();
    int localX = x, localY = y;
    if (!sl.validate(stamp)) {
        stamp = sl.readLock();
        try {
            localX = x;
            localY = y;
        } finally {
            sl.unlockRead(stamp);
        }
    }
    return Math.hypot(localX, localY);
}
```

In our optimistic `distanceFromOrigin()`, we first try to get an optimistic read stamp. This might come back as zero if a writeLock stamp has been issued and has not been unlocked yet. However, we assume that it is non-zero and continue reading the fields into local variables `localX` and `localY`. After we have read `x` and `y`, we validate the stamp. Two things could make this fail: 1. The `sl.tryOptimisticRead()` method might have come back as zero initially or 2. After we obtained our optimistic lock, another thread requested a `writeLock()`. We don't know whether this means our copies are invalid, but we need to be safe, rather than sorry. In this version we only try this once and if we do not succeed we immediately move over to the pessimistic read version. Depending on the system, we could get significant performance gains by spinning for a while, hoping to do a successful optimistic read. In our experiments, we also found that a shorter code path between `tryOptimisticRead()` and `validate()` leads to a higher chance of success in the optimistic read case.

Here is another idiom that retries a number of times before defaulting to the pessimistic read version. It uses the trick in Java where we break out to a label, thus jumping out of a code block. We could have also solved this with a local boolean variable, but to me this is a bit clearer:

```

private static final int RETRIES = 5;

public double distanceFromOrigin() {
    int localX, localY;
    out:
    {
        // try a few times to do an optimistic read
        for (int i = 0; i < RETRIES; i++) {
            long stamp = sl.tryOptimisticRead();
            localX = x;
            localY = y;
            if (sl.validate(stamp)) {
                break out;
            }
        }
        // pessimistic read
        long stamp = sl.readLock();
        try {
            localX = x;
            localY = y;
        } finally {
            sl.unlockRead(stamp);
        }
    }
    return Math.hypot(localX, localY);
}

```

Conditional Write

The last idiom to demonstrate is the conditional write. We first read the current state. If it is what we expect, then we try to upgrade the read lock to a write lock. If we succeed, then we update the state and exit, otherwise we unlock the read lock and then ask for a write lock. The code is a bit harder to understand, so look it over carefully:

```

public boolean moveIfAt(int oldX, int oldY,
                        int newX, int newY) {
    long stamp = sl.readLock();
    try {
        while (x == oldX && y == oldY) {
            long writeStamp = sl.tryConvertToWriteLock(stamp);
            if (writeStamp != 0) {
                stamp = writeStamp;
                x = newX;
            }
        }
    } finally {
        sl.unlockRead(stamp);
    }
}

```

```
    y = newY;
    return true;
} else {
    sl.unlockRead(stamp);
    stamp = sl.writeLock();
}
}
return false;
} finally {
    sl.unlock(stamp); // could be read or write lock
}
}
```

Whilst this idiom looks clever, I doubt it is really that useful. Initial tests have shown that it does not perform that well. In addition, it is difficult to write and understand.

This is the first in a three-part series on the StampedLock. The next one will examine how we could write the same class using non-blocking code and how the idioms could be made simpler by means of Java 8 Lambdas. The last newsletter will be dedicated to looking at how it performs against other mechanisms.

Please remember to **register for our free webinar on StampedLock Idioms on Friday the 6th December at 08:00 UTC**. This will allow you to ask questions and discuss the StampedLock idioms. A recording will be available afterwards.

Kind regards

Heinz

P.S. Which do you think is the most popular Java library: junit, slf4j or log4j? **Click here to find out if you are correct!**

Issue 214 - CountingCompletionService

Author: Dr. Heinz M. Kabutz

Date: 2013-10-10

Category: Concurrency

Java Versions: Java 8

Abstract:

CompletionService queues finished tasks, making it easier to retrieve Futures in order of completion. But it lacks some basic functionality, such as a count of how many tasks have been submitted.

Welcome to the 214th issue of **The Java(tm) Specialists' Newsletter**, sent from sunny Crete. As a proud owner of a polytekno card (meaning I have produced four new Greek taxpayers), I now get discounts almost everywhere I go in Greece. All forms of public transport, ferries, shops. Spar gives me 3% off everything, including beer. That's fair, I think. Beer is important.

The dates for our fourth **Java Specialists Symposium** are from the 25-29 August 2014. In 2013 we almost reached our maximum capacity. Thus the secret for getting a place for next year is to put your name down early. If you think you'd like to attend, **please send me** a short motivation explaining why you should be invited. I will pass on your details to the "disorganizers" who will make the final decision.

I assembled a **list of airlines** that flew directly to Chania Airport (CHQ) in August 2013. Once you have seen what is on offer, you can find further information by **Googling for "flights to CHQ"**. I hope this will make it easier for you to visit our island, either for the unconference or just for fun :-)

CountingCompletionService

ExecutorService provides a convenient way to submit tasks that can then be executed asynchronously by threads provided by the service. Instead of paying the price of thread start latency, the job can immediately be started by an available thread. The decrease in thread start latency however only gives us a performance advantage if a thread is currently idle waiting for work. If that is not the case, then either a thread will have to be started, as happens with the cached thread pool, or the thread will have to be put in a waiting queue, as you would find in the fixed thread pool. In either case, our latency can be greater than if we had just started a thread directly. As Kirk says: "When last did you see a queue and say to yourself - yay, that will go fast!"

Let's think about queuing a bit, based on our experience in real life. Most of us have been to airports. Imagine you are walking towards the check-in queue. At the same time, you see a

large family walking towards the queue, but from the other side. What do you do? You subtly up your pace a bit, hoping to get there first. Once you are in the queue, it does not really matter what is happening around you as you have to wait anyway. Until you get near the front, that is. For some reason, the person directly in front of me always seems fast asleep. The agent at the open counter is articulating wildly with her arms, but our hero is obviously studying his fingernails. Even though the single queue with multiple server system theoretically gives us the shortest average wait time, we get contention at the head and tail of the single queue. The same contention happens with the traditional thread pool design and is one of the reasons why the ForkJoinPool employs several queues and work stealing. Tasks may sometimes be executed out-of-order, but because tasks are supposed to be independent, this should not cause issues.

Usually when you submit a `Callable<V>` to an `ExecutorService`, you need to manage the `Future<V>` that is returned by the `submit()` method. However, since each task may take a different time to complete, you could quite easily block on a future whilst another future might already be available. Here is an example (using Java 8 lambdas - see [Maurice Naftalin's Lambda FAQ](#)):

```
import java.util.*;
import java.util.concurrent.*;

public class ExecutorServiceExample {
    public static void main(String... args) throws Exception {
        try (DotPrinter dp = new DotPrinter()) {
            ExecutorService pool = Executors.newCachedThreadPool();
            Collection<Future<Integer>> futures = new ArrayList<>();
            for (int i = 0; i < 10; i++) {
                int sleeptime = 5 - i % 5;
                int order = i;
                futures.add(pool.submit(() -> {
                    TimeUnit.SECONDS.sleep(sleeptime);
                    return order;
                }));
            }

            for (Future<Integer> future : futures) {
                System.out.printf("Job %d is done%n", future.get());
            }
            pool.shutdown();
        }
    }
}
```

Here is the code for our DotPrinter, whose entire purpose in life is to keep our attention occupied by printing a meaningless dot once every second:

```
import java.util.concurrent.*;

public class DotPrinter implements AutoCloseable {
    private final ScheduledExecutorService timer =
        Executors.newSingleThreadScheduledExecutor();

    public DotPrinter() {
        timer.scheduleAtFixedRate(() -> {
            System.out.print(".");
            System.out.flush();
        }, 1, 1, TimeUnit.SECONDS);
    }

    public void close() {
        timer.shutdown();
    }
}
```

Output would be the following:

```
.....Job 0 is done
Job 1 is done
Job 2 is done
Job 3 is done
Job 4 is done
Job 5 is done
Job 6 is done
Job 7 is done
Job 8 is done
Job 9 is done
```

As we see in the output, as we iterate through the futures in the order we submitted them, we get the results in order of submission. However, because the first job takes the longest, we have to wait until that is done before seeing the results of jobs that have been available for some time. In order to solve that, Java also has an ExecutorCompletionService. It is a

very simple class that just contains a `BlockingQueue` of completed tasks and specialized `Futures` that enqueue themselves when the `done()` method is called.

The `ExecutorCompletionService` lacks some basic functionality, such as a way to find out how many tasks have been submitted. In our next example, we show how we could use the `CompletionService` to improve the `ExecutorServiceExample` above.

```
import java.util.concurrent.*;

public class CompletionServiceExample {
    public static void main(String... args) throws Exception {
        try (DotPrinter dp = new DotPrinter()) {
            ExecutorService pool = Executors.newCachedThreadPool();
            CompletionService<Integer> service =
                new ExecutorCompletionService<>(pool);
            for (int i = 0; i < 10; i++) {
                int sleeptime = 5 - i % 5;
                int order = i;
                service.submit(() -> { // time to get used to lambdas?
                    TimeUnit.SECONDS.sleep(sleeptime);
                    return order;
                });
            }

            for (int i = 0; i < 10; i++) {
                System.out.printf("Job %d is done%n", service.take().get());
            }
            pool.shutdown();
        }
    }
}
```

The output now comes to:

```
.Job 4 is done
Job 9 is done
.Job 3 is done
Job 8 is done
.Job 2 is done
```

```
Job 7 is done
.Job 1 is done
Job 6 is done
.Job 0 is done
Job 5 is done
```

Nice. However, we do need to remember how many tasks we added to the CompletionService. Instead, we could extend the ExecutorCompletionService and add that functionality:

```
import java.util.concurrent.*;
import java.util.concurrent.atomic.*;

public class CountingCompletionService<V>
    extends ExecutorCompletionService<V> {
    private final AtomicLong submittedTasks = new AtomicLong();
    private final AtomicLong completedTasks = new AtomicLong();

    public CountingCompletionService(Executor executor) {
        super(executor);
    }

    public CountingCompletionService(
        Executor executor, BlockingQueue<Future<V>> queue) {
        super(executor, queue);
    }

    public Future<V> submit(Callable<V> task) {
        Future<V> future = super.submit(task);
        submittedTasks.incrementAndGet();
        return future;
    }

    public Future<V> submit(Runnable task, V result) {
        Future<V> future = super.submit(task, result);
        submittedTasks.incrementAndGet();
        return future;
    }

    public Future<V> take() throws InterruptedException {
        Future<V> future = super.take();
        completedTasks.incrementAndGet();
        return future;
    }
}
```

```

}

public Future<V> poll() {
    Future<V> future = super.poll();
    if (future != null) completedTasks.incrementAndGet();
    return future;
}

public Future<V> poll(long timeout, TimeUnit unit)
    throws InterruptedException {
    Future<V> future = super.poll(timeout, unit);
    if (future != null) completedTasks.incrementAndGet();
    return future;
}

public long getNumberOfCompletedTasks() {
    return completedTasks.get();
}

public long getNumberOfSubmittedTasks() {
    return submittedTasks.get();
}

public boolean hasUncompletedTasks() {
    return completedTasks.get() < submittedTasks.get();
}
}

```

We can thus replace the result iterating loop like this:

```

for (int i = 0; i < service.getNumberOfSubmittedTasks(); i++) {
    System.out.printf("Job %d is done%n", service.take().get());
}

```

Of course, an even nicer solution would be to build an iterable CompletionService. The loop above could then be replaced with simply:

```

for (Future<Integer> future : service) {
    System.out.printf("Job %d is done%n", future.get());
}

```

One of the exercises in my **Concurrency Specialist Course** is writing such an Iterable. It looks really simple, but there are a few gotchas that you need to be aware of. If you'd like to test your skill, here is an outline of what you need to do:

```

import java.util.*;

public class CompletionServiceIterable<V>
    implements Iterable<Future<V>> {
    public CompletionServiceIterable() { // TODO
    }

    public void submit(Callable<V> task) { // TODO
    }

    public Iterator<Future<V>> iterator() { // TODO
    }

    public void shutdown() { // TODO
    }

    public boolean isTerminated() { // TODO
    }
}

```

If you **send me your solution**, I'll run it against my unit tests and then give you my personal feedback. You should solve it without using any external libraries, just the JDK (6, 7 or 8) and your own code. Good luck :-)

Kind regards

Heinz

Issue 213 - Livelocks from wait/notify

Author: Dr. Heinz M. Kabutz

Date: 2013-09-20

Category: Language

Java Versions: Java 1 to 8

Abstract:

When a thread is interrupted, we need to be careful to not create a livelock in our code by re-interrupting without returning from the method.

Welcome to the 213th issue of **The Java(tm) Specialists' Newsletter**, sent to you from the wonderful Island of Crete. In August we had another amazing Java Specialist Symposium here in Crete. Our semi-professional team photographer David Gomez did a **spectacular job of capturing the spirit of what we did**. For our final evening, we had a barbecue and live music with famous Greek singer **Manolis Kontaros** at our house. It was a perfect ending to an inspiring time.

In my **last newsletter**, I mentioned that I had gone for a Greek citizenship interview. I heard this week that I passed, which means that in a few months time, I will be Greek. They generously overlooked that my Greek language ability is still not perfect and instead concentrated on my knowledge of Greek culture, my family and my integration into local life. I am very grateful for their kindness.

Livelocks from wait/notify

A few years ago, one of my friends sent me some classes that contained the following code:

```
public void assign(Container container) {
    synchronized (lock) {
        PooledThread thread = null;
        do {
            while (this.idle.isEmpty()) {
                try {
                    lock.wait();
                } catch (InterruptedException ie) {
                    Thread.currentThread().interrupt();
                }
            }
            if (!this.idle.isEmpty())

```

```

        thread = this.idle.getFirst();
    } while ((thread==null) || (!thread.isRunning()));
    this.moveToRunning(thread);
    thread.start(container);
    lock.notify();
}
}

```

Can you spot the problem?

When I looked at the code those many years ago, I recognized that there was an issue with the way that the thread was re-interrupted, *without* leaving the method. However, it was only a few weeks ago that I realized just how bad this was.

A thread goes through several states. It would typically be in the RUNNABLE state, but if it needed to get a monitor lock with synchronized, it could go into the BLOCKED state if that lock was not available. And if it was suspended due to a wait(), it would go into the WAITING or TIMED_WAITING state, after first releasing the lock. After being released from the wait(), it would have to reacquire the lock. The key is that usually, when we wait(), the lock is also released. However, if the thread is currently interrupted, then the wait() would immediately throw the InterruptedException, without first releasing and then reacquiring the lock.

Here is another example to illustrate this situation:

```

public class WaitNotifyLivelock {
    private boolean state = false;
    private final Object lock = new Object();
    public static volatile Thread waitingThread = null;

    public void waitFor() {
        synchronized (lock) {
            waitingThread = Thread.currentThread();
            while (!state) {
                try {
                    lock.wait();
                } catch (InterruptedException e) {
                    // In this context, re-interrupting is a mistake
                    Thread.currentThread().interrupt();
                }
            }
        }
    }
}

```

```

    }

    public void notifyIt() {
        synchronized (lock) {
            state = true;
            lock.notifyAll();
        }
    }
}

```

In our test program, we have three threads at play. The first calls the `waitFor()` method. A short while later, the main thread interrupts the first thread. After that, a third thread tries to call `notifyIt()`. Since the first thread never releases the lock as part of the `wait()` method call, it is impossible for the third thread to get the lock in order to send the notify and change the state.

In order to make this a bit more interesting, I have used the Java 8 Lambda syntax, including the Java 8 method reference `wnll::waitFor`. The equivalent Java 7 code is in the comments.

```

import java.util.concurrent.*;

public class WaitNotifyLiveLockTest {
    public static void main(String[] args) throws Exception {
        // Local variables and parameters accessed from inner classes
        // in Java 8 do not need to be explicitly declared as final!
        // They are implicitly final.
        WaitNotifyLiveLock wnll = new WaitNotifyLiveLock();
        ExecutorService pool = Executors.newCachedThreadPool();
        Future<?> waitForFuture = pool.submit(wnll::waitFor);
        // "wnll::waitFor" is a Java 8 method reference and is
        // roughly equivalent to:
        // pool.submit(new Runnable() {
        //     public void run() {
        //         wnll.waitFor();
        //     }
        // });
        while (WaitNotifyLiveLock.waitingThread == null) {
            Thread.sleep(10);
        }
        // now we interrupt the thread waiting for the signal
        WaitNotifyLiveLock.waitingThread.interrupt();
    }
}

```

```

Future<?> notifyFuture = pool.submit(wnll::notifyIt);

try {
    notifyFuture.get(1, TimeUnit.SECONDS);
} catch (TimeoutException e) {
    System.err.println("notifyFuture could not complete");
}

try {
    waitForFuture.get(1, TimeUnit.SECONDS);
} catch (TimeoutException e) {
    System.err.println("waitForFuture could not complete");
    System.out.println("Waiting thread state: " +
        WaitNotifyLivelock.waitingThread.getState());
}
}
}
}

```

The waiting thread gets into an infinite loop looking at the `state` field. However, since the `notifyIt` thread cannot get the lock, it is also not able to change the state. We see the following output:

```

notifyFuture could not complete
waitForFuture could not complete
Waiting thread state: WAITING

```

However, it is also possible for the "Waiting thread" to be in the RUNNABLE state, as we can see from this thread dump:

```

"pool-1-thread-2" waiting for monitor entry
java.lang.Thread.State: BLOCKED (on object monitor)
  at WaitNotifyLivelock.notifyIt(WaitNotifyLivelock.java:22)
  - waiting to lock <0x000000010d402700> (a java.lang.Object)
  at WaitNotifyLiveLockTest$$Lambda$2.run(Unknown Source)
  ...

```

```

"pool-1-thread-1" runnable

```

```
java.lang.Thread.State: RUNNABLE
at java.lang.Object.wait(Object.java:502)
at WaitNotifyLiveLock.waitFor(WaitNotifyLiveLock.java:11)
- locked <0x000000010d402700> (a java.lang.Object)
at WaitNotifyLiveLockTest$$Lambda$1.run(Unknown Source)
...
...
```

However, it never lets go of the lock, thus also not allowing the notifying thread from entering the critical section.

Next time you re-interrupt a thread, make sure that your surrounding code is also correct.

Would you like to know more about concurrency? Then take our [Concurrency Specialist Course](#).

Kind regards

Heinz

Issue 212 - Creating Sets from Maps

Author: Dr. Heinz M. Kabutz

Date: 2013-07-19

Category: Language

Java Versions: Java 6,7,8

Abstract:

Maps and Sets in Java have some similarities. In this newsletter we show a nice little trick for converting a map class into a set.

Welcome to the 212th issue of **The Java(tm) Specialists' Newsletter**, sent to you from the wonderful Island of Crete. Three weeks ago, I had the great privilege of being interviewed for Greek citizenship. The interview was in Greek, which meant I had to be able to both understand the question and then formulate an answer in at least passable Greek. I also had to be able to read and write Greek. I started learning a bit of Greek to impress my then future father-in-law in 1991. I have also lived in Greece for the last seven years. So if I had any brain cells at all, I really should have aced the interview. Unfortunately my worst subjects at school were history and geography, which most of the questions were about. As of writing, I do not know whether I passed or failed. All I know is that I enjoyed the panel interview with five examiners. And I think they also had fun, since they kept me there for 50 minutes. The other candidates were done in just 25. We ended up talking about Java, our **Cretan Java Unconference** and many other topics that were not really in their set of questions. If I failed, I'll be invited back next year again for another fun session of discussing Greek history, politics and geography.

Creating Sets from Maps

Seeing that it is summer here in Crete, this will be a short newsletter. Just one small tip that I picked up a few months ago and that has surprised quite a few people.

You probably know that the HashSet is just a wrapper around the HashMap, with the Set methods exposed. Similarly, the TreeSet is a wrapper around the TreeMap. The Map does not allow us to iterate over it directly. Thus the following would not work:

```
Map<String, Double> salaries = new HashMap<>();
for(double salary : salaries) { // does not compile
}
```

We can iterate over the set of keys, over the collection of values or over the set of entries. Values can have duplicates, which is why a collection is returned. Keys can never have duplicates, so both the keys and the entries can be returned as sets.

Thus if we wanted to iterate over just the values, we could do this:

```
Map<String, Double> salaries = new HashMap<>();
for (double salary : salaries.values()) {
}
```

Or over the keys would be:

```
Map<String, Double> salaries = new HashMap<>();
for (String name : salaries.keySet()) {
}
```

And lastly, over the entries would be:

```
Map<String, Double> salaries = new HashMap<>();
for (Map.Entry<String, Double> entry : salaries.entrySet()) {
    String name = entry.getKey();
    double salary = entry.getValue();
}
```

An approach I have often seen is when programmers get the keySet and then iterate through the set to find the values with get(), such as:

```
Map<String, Double> salaries = new HashMap<>();
```

```

for (String name : salaries.keySet()) { // less efficient way to
    double salary = salaries.get(name); // iterate over entries
}

```

Intuitively, it seems to me that the approach of iterating over the entry set should be much faster. The iteration is O(n). However, if the HashMap is well balanced, then we should get O(1) performance on the lookup call to get(), thus the complexity of the two approaches is the same. There might be a small difference, but they should both scale linearly. This is not true with other types of maps, such as the TreeMap. The lookup complexity is O(log n), thus the complexity of the second loop would be O(n x log n).

There are lots of maps available and for some of them, as we saw already with TreeMap and HashMap, we have corresponding set implementations. These sets are thin wrappers around the Map and thus have the same complexity and concurrency behaviour.

Now for the tip. In one of the exercises for my **Concurrency Specialist Course** I needed a fast, thread-safe HashSet. Initially, I used a ConcurrentHashMap with a dummy value. However, in Java 6, a method newSetFromMap() was added to the java.util.Collections class for creating new sets based on maps. The generic type parameter K of the map should be the element type inside the set and the V should be a Boolean, used to confirm the presence of the element.

```

import java.util.*;
import java.util.concurrent.*;

public class ConcurrentSetTest {
    public static void main(String[] args) {
        Set<String> names = Collections.newSetFromMap(
            new ConcurrentHashMap<String, Boolean>()
        );
        names.add("Brian Goetz");
        names.add("Victor Grazi");
        names.add("Heinz Kabutz");
        names.add("Brian Goetz");
        System.out.println("names = " + names);
    }
}

```

The newSetFromMap() method obviously only exposes the methods that are in Set. Thus if your map has a richer interface than the standard Map, you would need to still write your own

wrapper for it.

I hope you enjoyed this little snippet of information. If you ever looked for the ConcurrentHashSet, now you know how to get one.

Kind regards

Heinz

Issue 207 - Final Parameters and Local Variables

Author: Dr. Heinz M. Kabutz

Date: 2012-12-27

Category: Language

Java Versions: Java 1.1 - 8

Abstract:

The trend of marking parameters and local variables as "final" does not really enhance your code, nor does it make it more secure.

Welcome to the 207th issue of **The Java(tm) Specialists' Newsletter**, sent from the beautiful Island of Crete. Yesterday we went for a hike in the Stavros mountains, where the final crash scene of Zorba the Greek was filmed. We clambered up a hill and were able to see Milos, over a hundred kilometers away. We only recently started exploring the hills behind our house and have already found quite a few nice places with fantastic views. Very few people walk there, so we pretty much have the place to ourselves. Maybe it will be more busy in summer.

Final Parameters and Local Variables

Java 1.1 introduced the inner class, including the anonymous version. Since they did not really want to change the class format, they compiled these inner classes as normal classes with accessor methods for the privately accessed fields and methods. Thus the following class:

```
public class Foo {
    public class Bar {
    }
}
```

Would result in two class files, Foo.class and Foo\$Bar.class:

```
public class Foo {
    public Foo() {
```

```

    }
}

public class Foo$Bar {
    final Foo this$0;
    public Foo$Bar(Foo foo) {
        this$0 = foo;
        super();
    }
}

```

We could also define new classes inside methods, for example:

```

public class Foo {
    public void baz() {
        class Bar {
            public String toString() {
                return "I am a bar!";
            }
        }
        Bar bar = new Bar();
        System.out.println(bar);
    }

    public static void main(String[] args) {
        Foo foo = new Foo();
        foo.baz();
    }
}

```

This would be compiled to again two classes, this time to something like Foo.class and Foo\$1Bar.class. The naming convention could change by compiler, since Bar is only visible inside the method. However, Bar is *not* an anonymous class in this case. Instead, it is a **local class**, meaning that it has an enclosing method, which you can determine with the `getEnclosingMethod()` call. If it was defined inside a constructor, you could get that with the `getEnclosingConstructor()` call. Local classes are rare. In the JDK 1.7.0_09, I counted only 23, whereas there were 12987 normal classes, 2449 anonymous inner classes and 4341 non-anonymous inner classes. I would wager a beer that in most bodies of code, you would also find the local class to be a bit of a rarity.

Here is what the compiler would typically generate:

```

class Foo$1Bar {
    final Foo this$0;

    Foo$1Bar(Foo foo) {
        this$0 = foo;
        super();
    }

    public String toString() {
        return "I am a bar!";
    }
}

public class Foo {
    public Foo() {
    }

    public void baz() {
        Foo$1Bar bar = new Foo$1Bar(this);
        System.out.println(bar);
    }

    public static void main(String[] args) {
        Foo foo = new Foo();
        foo.baz();
    }
}

```

If we instead defined an anonymous inner class, it would look like this:

```

public class Foo {
    public void baz() {
        Object bar = new Object() {
            public String toString() {
                return "I am an anonymous bar!";
            }
        };
        System.out.println(bar);
    }
}

```

```

}

public static void main(String[] args) {
    Foo foo = new Foo();
    foo.baz();
}
}

```

The generated classes would now typically be called Foo.class and Foo\$1.class:

```

class Foo$1 {
    final Foo this$0;

    Foo$1(Foo foo) {
        this$0 = foo;
        super();
    }

    public String toString() {
        return "I am an anonymous bar!";
    }
}

public class Foo {
    public Foo() {
    }

    public void baz() {
        Object bar = new Foo$1(this);
        System.out.println(bar);
    }

    public static void main(String[] args) {
        Foo foo = new Foo();
        foo.baz();
    }
}

```

In Java, parameters are always passed by value. When I pass a primitive type to a method, this method is able to change its copy without affecting the variable that was passed in. If we pass in an object reference, we are able to change the reference inside the method without

the outside reference changing. In a few other languages such as Pascal you could pass in parameters by reference. In C you could pass in a pointer. In Java, the following would not have any effect:

```
public void increment(int i) {
    i++;
}
```

Now let's get back to the inner class construct. When we define an anonymous inner class, the parameters and local variables that it is allowed to see have to be defined as **final**. Thus the following would not compile:

```
public void show(int i) {
    new Runnable() {
        public void run() {
            System.out.println(i);
        }
    }.run();
}
```

Instead, we would need to make the parameter **i final**, which would make it visible *and* stop us from accidentally modifying it in the inner class. Once we have done that, the anonymous inner class would look something like this:

```
class PassByValue$1 implements Runnable {
    final int val$i;
    final PassByValue this$0;

    PassByValue$1(PassByValue passbyvalue, int i) {
        this$0 = passbyvalue;
        val$i = i;
        super();
    }

    public void run() {
```

```

        System.out.println(val$i);
    }
}

```

So this is **the** reason why we have the option since Java 1.1 to mark local variables and parameters as final.

A few years ago, someone started the trend of marking *all* their local variables and parameters as final. The first time one of my customers insisted on this was end of 2004. Even though I thought it was an idiotic coding standard, it took me about five minutes to apply using [IntelliJ IDEA's](#) Analyze/Inspect Code function. In retrospect, I should perhaps have punished them with punitive billing by charging them for how much this would have taken me with a lesser IDE. After I applied the changes and handed over the code, I found another 100+ places in *their* code where they had not marked local variables and parameters as final.

I like to compare this trend to tattoos. When I was a kid, gangsters, prisoners and seamen sported tattoos. As with most fashions, it suddenly became the cool thing to have a tattoo as it demonstrated a streak of rebellion and going against the social order. However, this became so pervasive that anybody *without* a tattoo can today be considered a rebel. Now if you are one of those that feel the need to decorate your skin with paint, please don't let me cramp your style. Go ahead, it's the fashion and everyone else is also doing it. But I don't like it. I personally don't feel that it is an enhancement.

Similarly, marking all your local variables and parameters as **final** is a trend that does not enhance your code in my opinion, but just clutters it with unnecessary text. I would prefer to read:

```
public void printNames(List<String> names, int maxNames) ...
```

rather than:

```
public void printNames(final List<String> names,
                      final int maxNames) ...
```

As a user of your library, I don't want to know or care that you have decided to not change the parameters inside the method call. Since the arguments are passed by value anyway, it won't affect my life one little bit if you decided to use maxNames as a counter variable rather than define your own.

A recent book on Java concurrency used final parameters for all their code samples and it was rather unreadable as a result. I won't mention the title of the book as I didn't find it particularly good and would not want to promote it.

I am of the opinion that you should not generally not change parameters inside a method. But I don't need the compiler to tell me this. That is how I code anyway. Similarly, I try to not change local variables. In fact, I avoid using local variables where I can.

I understand that you might feel differently about this.

Note that I am not against marking fields final, as this has some useful concurrency semantics. I am also not against making methods and classes final if that is going to help to guide users of your classes on how they should be extended. This is only a rant against marking local variables and parameters as final, when it is not necessary to do so.

Kind regards

Heinz

Issue 206 - Striped Executor Service

Author: Dr. Heinz M. Kabutz

Date: 2012-11-13

Category: Concurrency

Java Versions: Java 7

Abstract:

We present a new type of ExecutorService that allows users to "stripe" their execution in such a way that all tasks belonging to one stripe are executed in-order.

Welcome to the 206th issue of **The Java(tm) Specialists' Newsletter**, which I started writing en route to **W-JAX**, followed by a short visit to a customer in Amsterdam. Next week I'm off to Vienna for a fun-filled week of **Java Design Patterns**. Then we are running our new **Concurrency Course** in Düsseldorf Germany. Lots of flying at the moment. I might miss the olive oil harvest this year unfortunately. We have about 150 trees and we should get quite a few hundred liters of oil from them this year. Sadly the olive oil price has dropped significantly, to where it is only marginally more precious than heating oil.

If you are on my email distribution list, you would have received an invite to the 3rd Java Specialists Symposium on Crete in August 2013. If you did not get it, [please look here](#).

Striped Executor Service

A few months ago, Glenn McGregor sent an email to the **Concurrency Interest** mailing list, asking whether any of us knew of a thread pool with the ability to execute jobs in-order by stripe. Each Runnable would implement something like his StripedRunner and then get sorted according to its stripeClass.

```
interface StripedRunner {
    Object getStripeClass();
    void run();
}
```

This magical thread pool would ensure that all Runnables with the same stripeClass would be executed in the order they were submitted, but StripedRunners with different stripedClasses could still execute independently. He wanted to use a relatively small thread pool to service a large number of Java NIO clients, but in such a way that the runnables

would still be executed in-order.

Several suggestions were made, such as having a SingleThreadExecutor for each stripeClass. However, that would not satisfy the requirement that we could share the threads between connections.

My first attempt to solve this challenge was to create a special StripedBlockingQueue that would release objects in order by stripe, and that also would not release new objects from a particular stripe until the previous runnable had been completed. It almost worked, but unfortunately jobs are not always enqueued before being delivered to the worker threads. If there are workers available, they are handed over immediately. Thus this approach could not guarantee in-order execution by runnables within a stripeClass.

In my second attempt, I instead wrote a StripedExecutorService that can run striped tasks in-order. To keep things simple, I decided to use Java 5 locks and conditions to communicate state between threads. I know that this is not as hip as lock-free, non-blocking algorithms, but I always start with a basic design that works and then later refactor it if the need arises. Usually a lock-free algorithms is several factors more effort to get right and does not always guarantee better performance.

In my design I use the SerialExecutor mentioned in the [JavaDocs of Executor](#), kindly pointed out by Joe Bowbeer.

The SerialExecutor unfortunately misses an important component: shutdown. Ask most parents and they will tell you that conceiving a new life is surprisingly effortless. It is easy starting new things, but not so easy winding them down. For example, it took me about two weeks to start Maximum Solutions (Pty) Ltd in South Africa, a company that I ran for 10 years from 1998 until 2008. When it came time to shut down the company, I discovered that this took many years. It has been lying dormant for four years, but still exists, I think. The shutdownNow() method exists, but it involves unpleasantries such as tax audits that would cost me a lot in accounting fees to manage. When we founded our company in Greece, the accountant asked us for how long we wanted our company to run. You can actually decide up front for how many years you want to carry on. I found this a rather curious question. Don't most businessmen want their company to run for a hundred years?

Before we examine the striped executor, we should study the SerialExecutor and understand how it works. Here is the code, with a minor modification in that I call tasks.add() instead of tasks.offer(). If offer() for some reason did not work, it would cause a silent failure, whereas it would be better to see an exception.

```
public class SerialExecutor implements Executor {
    private final Queue<Runnable> tasks = new ArrayDeque<>();
    private final Executor executor;
```

```

private Runnable active;

public SerialExecutor(Executor executor) {
    this.executor = executor;
}

public synchronized void execute(final Runnable r) {
    tasks.add(new Runnable() {
        public void run() {
            try {
                r.run();
            } finally {
                scheduleNext();
            }
        }
    });
    if (active == null) {
        scheduleNext();
    }
}

protected synchronized void scheduleNext() {
    if ((active = tasks.poll()) != null) {
        executor.execute(active);
    }
}
}

```

Here is how it works. Let's say that SubmitterThread passes in Runnables R1, R2, R3 and R4, one after the other. When R1 is passed in, it is wrapped with a new Runnable that will call the R1.run() method and then allow the next Runnable in the tasks queue to be executed. Let's call the new Runnable R1*. Since there is no other Runnable currently active (thus active==null), the SubmitterThread also calls scheduleNext(), which pulls R1* out of the queue and submits it to the wrapped Executor. The SubmitterThread can now call execute() with R2, R3 and R4, which will wrap these Runnables also with our special Runnable and submit them to our tasks queue.

Inside R1*, once R1 is completed, the scheduleNext() method is invoked. This will pull R2* from the tasks list and submit it to the wrapped Executor. This will continue until R4*, which will call scheduleNext(), but that will simply set active to null.

The SerialExecutor implements only the Executor interface, which means we cannot submit Callables, nor can we shut it down cleanly. In this newsletter, we will expand this mechanism to be fronted by an ExecutorService that will be more flexible in its design and also allow us to shut it down cleanly.

Here is a short example of the SerialExecutor at work. We use the Phaser to signal when the jobs have been completed, otherwise we do not know when to return from the test() method. Before we add a new Runnable to the executor, we register an additional party to the phaser. Each time one of the Runnables is done, it sends a message to the phaser with the "arrive()" method. The phaser.arriveAndAwaitAdvance() will do so uninterruptibly.

```

public class SerialExecutorExample {
    private static final int UPTO = 10;

    public static void main(String[] args) {
        ExecutorService cached = Executors.newCachedThreadPool();
        test(new SerialExecutor(cached));
        test(cached);
        cached.shutdown();
    }

    private static void test(Executor executor) {
        final Vector<Integer> call_sequence = new Vector<>();
        final Phaser phaser = new Phaser(1);
        for(int i=0; i < UPTO; i++) {
            phaser.register();
            final int tempI = i;
            executor.execute(new Runnable() {
                public void run() {
                    try {
                        TimeUnit.MILLISECONDS.sleep(
                            ThreadLocalRandom.current().nextInt(2, 10)
                        );
                    } catch (InterruptedException e) {
                        Thread.currentThread().interrupt();
                    }
                    call_sequence.add(tempI);
                    phaser.arrive();
                }
            });
        }
        // we need to wait until all the jobs are done
        phaser.arriveAndAwaitAdvance();
        System.out.println(call_sequence);
    }
}

```

When I run the SerialExecutorExample, I get results such as these:

```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
[2, 4, 1, 7, 0, 8, 5, 9, 6, 3]
```

StripedExecutor Solution

In our solution, instead of having to maintain one SerialExecutor per stripe of execution, we allow users to submit objects that fall into a certain stripe. We accept Runnable and Callable.

First we have three interfaces that we can implement in order to indicate which stripe our runnable belongs to, starting with the StripedObject. We decide on the stripe by identity, rather than the actual hash code of the object.

```
public interface StripedObject {
    Object getStripe();
}
```

In my implementation, you can mix Callable and Runnable objects. As long as they belong to the same stripe, they will be executed in-order.

```
public interface StripedRunnable extends Runnable, StripedObject{}
```

And of course we can also submit Callables to our pool:

```
public interface StripedCallable<V> extends Callable<V>,
    StripedObject {
```

```
}
```

The StripedExecutorService accepts Runnable/Callable objects, which may also implement the StripedObject interface. If they do, then they will be executed in order for their stripe. If they do not, that is, they are ordinary Runnable/Callable tasks, we immediately pass them on to the wrapped Executor Service.

As mentioned earlier, we use the **SerialExecutor** to invoke all the tasks for a particular stripe in-order. The stripe association with the SerialExecutor is maintained in an IdentityHashMap. In order to avoid a memory leak, we remove the SerialExecutor as soon as all tasks for that stripe are completed.

```
/**
 * The StripedExecutorService accepts Runnable/Callable objects
 * that also implement the StripedObject interface. It executes
 * all the tasks for a single "stripe" consecutively.
 *
 * In this version, submitted tasks do not necessarily have to
 * implement the StripedObject interface. If they do not, then
 * they will simply be passed onto the wrapped ExecutorService
 * directly.
 *
 * Idea inspired by Glenn McGregor on the Concurrency-interest
 * mailing list and using the SerialExecutor presented in the
 * Executor interface's JavaDocs.
 *
 * http://cs.oswego.edu/mailman/listinfo/concurrency-interest
 *
 * @author Dr Heinz M. Kabutz
 */
public class StripedExecutorService extends AbstractExecutorService {
    /**
     * The wrapped ExecutorService that will actually execute our
     * tasks.
     */
    private final ExecutorService executor;

    /**
     * The lock prevents shutdown from being called in the middle
     * of a submit. It also guards the executors IdentityHashMap.
     */
    private final ReentrantLock lock = new ReentrantLock();
```

```

/**
 * This condition allows us to cleanly terminate this executor
 * service.
 */
private final Condition terminating = lock.newCondition();

/**
 * Whenever a new StripedObject is submitted to the pool, it
 * is added to this IdentityHashMap. As soon as the
 * SerialExecutor is empty, the entry is removed from the map,
 * in order to avoid a memory leak.
 */
private final Map<Object, SerialExecutor> executors =
    new IdentityHashMap<>();

/**
 * The default submit() method creates a new FutureTask and
 * wraps our StripedRunnable with it. We thus need to
 * remember the stripe object somewhere. In our case, we will
 * do this inside the ThreadLocal "stripes". Before the
 * thread returns from submitting the runnable, it will always
 * remove the thread local entry.
 */
private final static ThreadLocal<Object> stripes =
    new ThreadLocal<>();

/**
 * Valid states are RUNNING and SHUTDOWN. We rely on the
 * underlying executor service for the remaining states.
 */
private State state = State.RUNNING;

private static enum State {
    RUNNING, SHUTDOWN
}

/**
 * The constructor taking executors is private, since we do
 * not want users to shutdown their executors directly,
 * otherwise jobs might get stuck in our queues.
 *
 * @param executor the executor service that we use to execute
 *                  the tasks
 */
private StripedExecutorService(ExecutorService executor) {
    this.executor = executor;
}

```

```

/**
 * This constructs a StripedExecutorService that wraps a
 * cached thread pool.
 */
public StripedExecutorService() {
    this(Executors.newCachedThreadPool());
}

/**
 * This constructs a StripedExecutorService that wraps a fixed
 * thread pool with the given number of threads.
 */
public StripedExecutorService(int numberOfThreads) {
    this(Executors.newFixedThreadPool(numberOfThreads));
}

/**
 * If the runnable also implements StripedObject, we store the
 * stripe object in a thread local, since the actual runnable
 * will be wrapped with a FutureTask.
 */
protected <T> RunnableFuture<T> newTaskFor(
    Runnable runnable, T value) {
    saveStripedObject(runnable);
    return super.newTaskFor(runnable, value);
}

/**
 * If the callable also implements StripedObject, we store the
 * stripe object in a thread local, since the actual callable
 * will be wrapped with a FutureTask.
 */
protected <T> RunnableFuture<T> newTaskFor(
    Callable<T> callable) {
    saveStripedObject(callable);
    return super.newTaskFor(callable);
}

/**
 * Saves the stripe in a ThreadLocal until we can use it to
 * schedule the task into our pool.
 */
private void saveStripedObject(Object task) {
    if (isStripedObject(task)) {
        stripes.set(((StripedObject) task).getStripe());
    }
}

```

```

/**
 * Returns true if the object implements the StripedObject
 * interface.
 */
private static boolean isStripedObject(Object o) {
    return o instanceof StripedObject;
}

/**
 * Delegates the call to submit(task, null).
 */
public Future<?> submit(Runnable task) {
    return submit(task, null);
}

/**
 * If the task is a StripedObject, we execute it in-order by
 * its stripe, otherwise we submit it directly to the wrapped
 * executor. If the pool is not running, we throw a
 * RejectedExecutionException.
 */
public <T> Future<T> submit(Runnable task, T result) {
    lock.lock();
    try {
        checkPoolIsRunning();
        if (isStripedObject(task)) {
            return super.submit(task, result);
        } else { // bypass the serial executors
            return executor.submit(task, result);
        }
    } finally {
        lock.unlock();
    }
}

/**
 * If the task is a StripedObject, we execute it in-order by
 * its stripe, otherwise we submit it directly to the wrapped
 * executor. If the pool is not running, we throw a
 * RejectedExecutionException.
 */
public <T> Future<T> submit(Callable<T> task) {
    lock.lock();
    try {
        checkPoolIsRunning();
        if (isStripedObject(task)) {
            return super.submit(task);
        }
    } finally {
        lock.unlock();
    }
}

```

```

} else { // bypass the serial executors
    return executor.submit(task);
}
} finally {
    lock.unlock();
}
}

/**
 * Throws a RejectedExecutionException if the state is not
 * RUNNING.
 */
private void checkPoolIsRunning() {
    assert lock.isHeldByCurrentThread();
    if (state != State.RUNNING) {
        throw new RejectedExecutionException(
            "executor not running");
    }
}

/**
 * Executes the command. If command implements StripedObject,
 * we execute it with a SerialExecutor. This method can be
 * called directly by clients or it may be called by the
 * AbstractExecutorService's submit() methods. In that case,
 * we check whether the stripes thread local has been set. If
 * it is, we remove it and use it to determine the
 * StripedObject and execute it with a SerialExecutor. If no
 * StripedObject is set, we instead pass the command to the
 * wrapped ExecutorService directly.
 */
public void execute(Runnable command) {
    lock.lock();
    try {
        checkPoolIsRunning();
        Object stripe = getStripe(command);
        if (stripe != null) {
            SerialExecutor ser_exec = executors.get(stripe);
            if (ser_exec == null) {
                executors.put(stripe, ser_exec =
                    new SerialExecutor(stripe));
            }
            ser_exec.execute(command);
        } else {
            executor.execute(command);
        }
    } finally {
        lock.unlock();
    }
}

```

```

        }

    }

    /**
     * We get the stripe object either from the Runnable if it
     * also implements StripedObject, or otherwise from the thread
     * local temporary storage. Result may be null.
     */
    private Object getStripe(Runnable command) {
        Object stripe;
        if (command instanceof StripedObject) {
            stripe = (((StripedObject) command).getStripe());
        } else {
            stripe = stripes.get();
        }
        stripes.remove();
        return stripe;
    }

    /**
     * Shuts down the StripedExecutorService. No more tasks will
     * be submitted. If the map of SerialExecutors is empty, we
     * shut down the wrapped executor.
     */
    public void shutdown() {
        lock.lock();
        try {
            state = State.SHUTDOWN;
            if (executors.isEmpty()) {
                executor.shutdown();
            }
        } finally {
            lock.unlock();
        }
    }

    /**
     * All the tasks in each of the SerialExecutors are drained
     * to a list, as well as the tasks inside the wrapped
     * ExecutorService. This is then returned to the user. Also,
     * the shutdownNow method of the wrapped executor is called.
     */
    public List<Runnable> shutdownNow() {
        lock.lock();
        try {
            shutdown();
            List<Runnable> result = new ArrayList<>();
            for (SerialExecutor ser_ex : executors.values()) {

```

```

        ser_ex.tasks.drainTo(result);
    }
    result.addAll(executor.shutdownNow());
    return result;
} finally {
    lock.unlock();
}
}

/**
 * Returns true if shutdown() or shutdownNow() have been
 * called; false otherwise.
*/
public boolean isShutdown() {
    lock.lock();
    try {
        return state == State.SHUTDOWN;
    } finally {
        lock.unlock();
    }
}

/**
 * Returns true if this pool has been terminated, that is, all
 * the SerialExecutors are empty and the wrapped
 * ExecutorService has been terminated.
*/
public boolean isTerminated() {
    lock.lock();
    try {
        if (state == State.RUNNING) return false;
        for (SerialExecutor executor : executors.values()) {
            if (!executor.isEmpty()) return false;
        }
        return executor.isTerminated();
    } finally {
        lock.unlock();
    }
}

/**
 * Returns true if the wrapped ExecutorService terminates
 * within the allotted amount of time.
*/
public boolean awaitTermination(long timeout, TimeUnit unit)
    throws InterruptedException {
    lock.lock();
    try {

```

```

long waitUntil = System.nanoTime() + unit.toNanos(timeout);
long remainingTime;
while ((remainingTime = waitUntil - System.nanoTime()) > 0
    && !executors.isEmpty()) {
    terminating.awaitNanos(remainingTime);
}
if (remainingTime <= 0) return false;
if (executors.isEmpty()) {
    return executor.awaitTermination(
        remainingTime, TimeUnit.NANOSECONDS);
}
return false;
} finally {
    lock.unlock();
}
}

private void removeEmptySerialExecutor(Object stripe,
                                         SerialExecutor ser_ex) {
    assert ser_ex == executors.get(stripe);
    assert lock.isHeldByCurrentThread();
    assert ser_ex.isEmpty();

    executors.remove(stripe);
    terminating.signalAll();
    if (state == State.SHUTDOWN && executors.isEmpty()) {
        executor.shutdown();
    }
}

private static boolean DEBUG = false;



```

```

* also removes itself automatically once the queue is empty.
*/
private class SerialExecutor implements Executor {
    /**
     * The queue of unexecuted tasks.
     */
    private final BlockingQueue<Runnable> tasks =
        new LinkedBlockingQueue<>();
    /**
     * The runnable that we are currently busy with.
     */
    private Runnable active;
    /**
     * The stripe that this SerialExecutor was defined for. It
     * is needed so that we can remove this executor from the
     * map once it is empty.
     */
    private final Object stripe;

    /**
     * Creates a SerialExecutor for a particular stripe.
     */
    private SerialExecutor(Object stripe) {
        this.stripe = stripe;
        if (DEBUG) {
            System.out.println("SerialExecutor created " + stripe);
        }
    }

    /**
     * We use finalize() only for debugging purposes. If
     * DEBUG==false, the body of the method will be compiled
     * away, thus rendering it a trivial finalize() method,
     * which means that the object will not incur any overhead
     * since it won't be registered with the Finalizer.
     */
    protected void finalize() throws Throwable {
        if (DEBUG) {
            System.out.println("SerialExecutor finalized " + stripe);
            super.finalize();
        }
    }

    /**
     * For every task that is executed, we add() a wrapper to
     * the queue of tasks that will run the current task and
     * then schedule the next task in the queue.
     */
}

```

```

public void execute(final Runnable r) {
    lock.lock();
    try {
        tasks.add(new Runnable() {
            public void run() {
                try {
                    r.run();
                } finally {
                    scheduleNext();
                }
            }
        });
        if (active == null) {
            scheduleNext();
        }
    } finally {
        lock.unlock();
    }
}

/**
 * Schedules the next task for this stripe. Should only be
 * called if active == null or if we are finished executing
 * the currently active task.
 */
private void scheduleNext() {
    lock.lock();
    try {
        if ((active = tasks.poll()) != null) {
            executor.execute(active);
            terminating.signalAll();
        } else {
            removeEmptySerialExecutor(stripe, this);
        }
    } finally {
        lock.unlock();
    }
}

/**
 * Returns true if the list is empty and there is no task
 * currently executing.
 */
public boolean isEmpty() {
    lock.lock();
    try {
        return active == null && tasks.isEmpty();
    } finally {

```

```

        lock.unlock();
    }
}
}
}
```

A full test suite is available in my git repository in [StripedExecutorServiceTest](#). However, here is a small example that shows how the StripedExecutorService works:

```

public class StripedExecutorServiceExample {
    private static final int UPTO = 10;

    public static void main(String[] args)
        throws InterruptedException {
        test(new StripedExecutorService());
        test(Executors.newCachedThreadPool());
    }

    private static void test(ExecutorService pool)
        throws InterruptedException {
        final Vector<Integer> call_sequence = new Vector<>();
        for (int i = 0; i < UPTO; i++) {
            final int tempI = i;
            pool.submit(new StripedCallable<Void>() {
                public Void call() throws Exception {
                    TimeUnit.MILLISECONDS.sleep(
                        ThreadLocalRandom.current().nextInt(2, 10)
                    );
                    call_sequence.add(tempI);
                    return null;
                }

                public Object getStripe() {
                    return call_sequence;
                }
            });
        }
        pool.shutdown();
        while (!pool.awaitTermination(1, TimeUnit.SECONDS)) ;
        System.out.println(call_sequence);
    }
}
```

We could now have several "stripes" of execution inside one ExecutorService. In addition, we are able to shut it down cleanly.

Kind regards

Heinz

Issue 204 - Book Review: The Well-Grounded Java Developer

Author: Dr. Heinz M. Kabutz

Date: 2012-08-06

Category: Book Review

Java Versions: Java 7

Abstract:

Ben Evans and Martijn Verburg explain to us in their new book what it takes to be a well-grounded Java developer. The book contains a section on the new Java 7 features and also vital techniques that we use for producing robust and performant systems.

Welcome to the 204th issue of **The Java(tm) Specialists' Newsletter** sent to you from a warm Crete. We spoke to a French art dealer in the sea at Kalathas today and asked him why he came to Crete on holiday. We have a lot of Italians, French and Russians here this year, plus of course thousands of Scandinavians. He told us that countries like Croatia had become overrun with visitors and also quite expensive for holidays. In comparison to other popular destinations, Crete offered excellent value for money. It is true. The prices in restaurants have not changed much since 2006. I can get a delicious freshly squeezed orange juice in Chorafakia for just 2 Euro at Pantelis' cafeteria. And nothing beats the cooking of Irene's next door. The only group of holiday makers that is missing this year are my fellow Germans. Angsthasen ;-)

A few weeks ago, we went to apply for a Greek ID card for my 14 year old son. When we came to Greece, my name was converted into the Greek alphabet as XAINZ KAMPOYTZ. Greek does not have an "H" sound, so they used "X", which is pronounced as "CH". The "U" sound is made up of the diphthong "OY". Unfortunately someone had the bright idea of automatically reverse engineering Latin from Greek names. So the computer was fired up somewhere in Athens and converted me to CHAINZ KAMPOUTZ. Knowing how much trouble incorrect names can cause, I asked them to fix it. This turned out to be rather difficult for them. After all, who can argue with a computer translation? At one point, the policewoman tried to convince me that *their* system was correct and that I had it wrong. Gosh, 40 years of being called Heinz (like ketchup) Kabutz and only now I find out that it was wrong all the time? Must let my mom know!

The Well-Grounded Java Developer

Ben Evans and Martijn Verburg are both well known Java experts who consult in the financial industry in London. They have many years of experience as well-grounded Java developers. Together they wrote **The Well-Grounded Java Developer: Vital techniques of Java 7 and polyglot programming**.

They kindly asked me to write the foreword, which you can read [on Manning's website](#). Hope you enjoy it. It is a bit different. And yes, the fact that I wrote the foreword is an endorsement for the book. I certainly would not have agreed if I did not like the book. I had the great privilege of getting a sneak peak at the book and also to meet Ben and Marty in person when they came to Crete in April.

Cost of Change in Java

In the book they explain why it is so expensive to add new features to the JVM. Adding new library extensions such as fork/join or syntactic sugar like switch-on-string is relatively easy, but a JVM instruction such as invokedynamic is very costly to add. This is why we have not seen many changes to Java's fundamental infrastructure since Java 1.0. I was always wondering why change flowed so slowly in the Java environment. This is all described in chapter 1 of the book, which you can [download as a sample chapter](#).

Binary Literals

One of the new features in Java 7 are binary literals. We can now write numbers as 0b101010101. Unfortunately you are also allowed to write long binary numbers with a lower case L, such as: 0b1111101110111011011011011. This is quite confusing to readers of the code, as they can easily mistake the lower case L at the end of the number for a one. It is much clearer to write 0b111110111011101101101101L. I would have welcomed it if they had decided to not allow the lower case L for binary numbers, but they probably wanted to stay consistent with other primitive number representations.

Better Exceptions

In previous versions of Java, if we caught the general "Exception" and then re-threw that, we needed to declare that our method throws "Exception" as well:

```
public void foo() throws Exception {
    try {
        doSomethingWhichMightThrowIOException();
        doSomethingElseWhichMightThrowSQLException();
    } catch (Exception e) {
        // do something with e ...
        throw e;
    }
}
```

In Java 7, the compiler is clever enough to figure out that only the checked exceptions need

to be declared. Thus we can write:

```
public void foo() throws IOException, SQLException {
    try {
        doSomethingWhichMightThrowIOException();
        doSomethingElseWhichMightThrowSQLException();
    } catch (Exception e) {
        // do something with e ...
        throw e;
    }
}
```

In the book, Ben and Martijn recommend that you mark the Exception as "final". The compiler does not insist on this, so it is just a convention that they use to signal their intention. In my opinion, this is not necessary, since a lot of code already would have the exception marked as final.

Try-With-Resource

They make an important point about try-with-resource. We need to declare each of the objects that we want to have automatically closed in the try section. For example, this would not be correct:

```
try (
    ObjectInputStream in = new ObjectInputStream(
        new BufferedInputStream(
            new FileInputStream("someFile.bin")));
) {
    // use the ObjectInputStream
}
```

If the FileInputStream construction succeeds (because the file does exist) but the ObjectInputStream construction fails (because the file is corrupt) or the BufferedInputStream fails (because of an OutOfMemoryError), then the FileInputStream will not be closed automatically.

The correct way to write the code is like this:

```

try {
    FileInputStream fis = new FileInputStream("someFile.bin");
    BufferedInputStream bis = new BufferedInputStream(fis);
    ObjectInputStream in = new ObjectInputStream(bis);
} {
    // use the ObjectInputStream
}

```

Now, if any part of the construction fails, the previously declared and constructed objects will be automatically closed.

The "javap" Tool

The book contains a nice discussion on the javap tool and how we can use it to analyse what is going on in our code. I have mentioned the javap tool in 15 newsletters already ([042](#), [064](#), [066](#), [068](#), [069](#), [083](#), [091](#), [105](#), [109](#), [115](#), [129](#), [136](#), [137](#), [147](#) and [174](#)). As you can imagine, it is a technique that I often employ to understand what the byte code looks like. However, I do not recall seeing it written about in any Java book to date, at least at the level that Ben and Martijn did. Jack Shirazi mentioned javap very briefly in his excellent book, [Java Performance Tuning](#). Warning: Even though Shirazi's book is fantastic, it is quite dated. As always with clever performance tricks, you need to measure that the tricks works for you. Some parts of his book, such as his methodologies for tuning performance, are still very relevant today.

Bottleneck on Caches

One of the most surprising classes was their CacheTester. I have seen a number of benchmarks that try to show how fast Fork/Join is by iterating over a very large array in parallel. For example, the code might try to find the largest int inside the array.

Usually the benchmark bottlenecks on memory, thus incorrectly proving that fork/join does not give any performance gains. In the CacheTester, Ben and Marty show how iterating over the array one element at the time is not much slower than looking at every 16th element. Here is their CacheTester:

```

public class CacheTester {
    private final int ARR_SIZE = 1 * 1024 * 1024;

```

```

private final int[] arr = new int[ARR_SIZE];
private void doLoop2() {
    for (int i=0; i<arr.length; i++) arr[i]++;
}
private void doLoop1() {
    for (int i=0; i<arr.length; i += 16) arr[i]++;
}
private void run() {
    for (int i=0; i<10000; i++) {
        doLoop1();
        doLoop2();
    }
    for (int i=0; i<100; i++) {
        long t0 = System.nanoTime();
        doLoop1();
        long t1 = System.nanoTime();
        doLoop2();
        long t2 = System.nanoTime();
        long el = t1 - t0;
        long el2 = t2 - t1;
        System.out.println("Loop1: "+ el +" nanos ; Loop2: "+ el2);
    }
}
public static void main(String[] args) {
    CacheTester ct = new CacheTester();
    ct.run();
}
}

```

I ran their code on my 8-core server and got the following results in microseconds:

| | Average | Variance |
|-------|---------|----------|
| Loop1 | 239 | 12 |
| Loop2 | 549 | 48 |

We can thus see that even though we are reading 16x as many array elements, it is only 2.3 times slower to do that.

Even though the results are good in that the variance is not too high, they could be better if we changed a couple of things. First off, to measure time at the nanosecond granularity

invites slight abberations in the system to have an influence on our variance. In my CacheTester, I repeat the iteration 1000 times, thus getting the results in milliseconds. Secondly, I usually try to produce output that I can then copy and paste directly into a spreadsheet. Comma separated values seem to work nicely. Thirdly, the number 10000 in the CacheTester is significant. Typically, after you have called a method 10000 times, the HotSpot compiler *starts* profiling and optimizing the code. However, it may be a while before the new optimized code is available. Thus we sleep for a second after the 10000 warm-up calls in order to immediately have the fastest times:

```

public class CacheTester {
    private final int ARR_SIZE = 1 * 1024 * 1024;
    private final int[] arr = new int[ARR_SIZE];
    private static final int REPEATS = 1000;

    private void doLoop2() {
        for (int i = 0; i < arr.length; i++) arr[i]++;
    }

    private void doLoop1() {
        for (int i = 0; i < arr.length; i += 16) arr[i]++;
    }

    private void run() throws InterruptedException {
        for (int i = 0; i < 10000; i++) {
            doLoop1();
            doLoop2();
        }
        Thread.sleep(1000); // allow the hotspot compiler to work
        System.out.println("Loop1,Loop2");
        for (int i = 0; i < 100; i++) {
            long t0 = System.currentTimeMillis();
            for (int j = 0; j < REPEATS; j++) doLoop1();
            long t1 = System.currentTimeMillis();
            for (int j = 0; j < REPEATS; j++) doLoop2();
            long t2 = System.currentTimeMillis();
            long el = t1 - t0;
            long el2 = t2 - t1;
            System.out.println(el + "," + el2);
        }
    }

    public static void main(String[] args)
        throws InterruptedException {
        CacheTester ct = new CacheTester();
        ct.run();
    }
}

```

```
}
```

Here are the results of my CacheTester, which show almost no variance at all:

| | Average | Variance |
|-------|---------|----------|
| Loop1 | 238 | 0.3 |
| Loop2 | 546 | 1.8 |

When I ran the code on my MacBook Pro with an Intel Core 2 Duo, I had the following results with my benchmark:

| | Average | Variance |
|-------|---------|----------|
| Loop1 | 168 | 17 |
| Loop2 | 580 | 37 |

You can see that the variance was again quite high, because my laptop had too many other programs running on it. On my MacBook Pro hardware, iterating through every element in the array was 3.4 times slower.

Concurrency

Another sample chapter [you can download](#) is the one on concurrency. Both Ben and Martijn are certified to present my [concurrency course](#). Ben has a lot of experience in the subject, which led to many interesting discussions when they came here in April.

Just one minor gripe. In the book they use `Math.random() * 10` in order to calculate a random delay. Since Java 7, we should rather use `ThreadLocalRandom.current().nextInt(10)`. This has several benefits. First off, `ThreadLocalRandom` keeps a `Random` instance per thread, so that we do not have any contention on the random seed. Secondly, the random distribution is fairer with the `nextInt(10)` method call. The fairness is a minor point, but the contention is not. `Math.random()` shares an instance of `Random` and the seed is protected by compare and swap. Thus if a lot of threads call this at the same time, they will need to repeat a lot of expensive calculations to eventually update the random seed.

The book is filled with many other interesting tidbits, and is definitely on my "recommended reading" list for the Java specialist.

Kind regards

Heinz

P.S. Hot off the press: Martin Thompson just published an article on the cost of memory access, showing different approaches to traversing the elements. This is closely related to the CacheTester in Evans and Verburg. [Click here](#) to read it.

Issue 202 - Distributed Performance Tuning

Author: Dr. Heinz M. Kabutz

Date: 2012-05-28

Category: Concurrency

Java Versions: Java 7

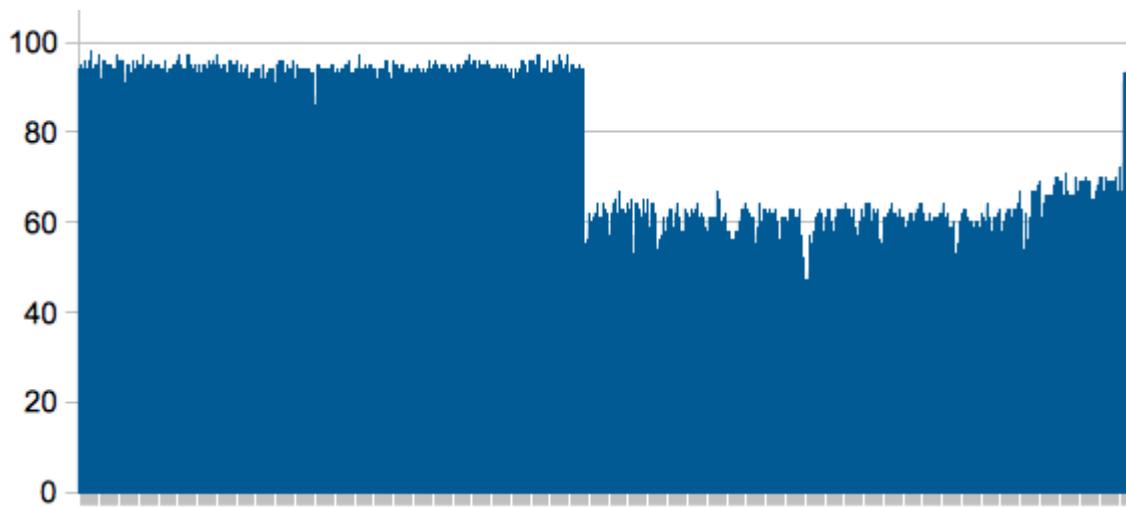
Abstract:

In this newsletter, it is up to you to figure out how we improved the performance of our previous Fibonacci newsletter by 25%.

Welcome to the 202nd issue of **The Java(tm) Specialists' Newsletter** sent to you en route to Lausanne in Switzerland to visit one of my favourite customers. We had a fantastic time last week with Kirk and his students on the Island of Crete. We ended up with some delicious Cretan lamb on the spit, as we usually do.

Distributed Performance Tuning

After I sent the [last newsletter about the Fibonacci calculation with fork/join](#), I produced a graph of what the CPU looked like on my 8-core server. The graph is of the user time; the system time is negligible. The y-axis is percentage CPU on my 8-core system. The x-axis is the time. This particular run completed in 31.5 minutes.



My question to you is: "At the half-way point, the CPUs drops down to an average of about 60% utilization. What can we do to keep the CPUs busy?"

Whilst the system was 60% utilized, I did a stack dump. [Click here](#) to download the vmstat output, with the stack dump near the end.

Now it is over to you. You need to come up with an explanation of what is keeping the CPUs from being busy and then give us a solution of how to fix it. Hint: I wrote about this behaviour in one of my newsletters, about 7 months before I moved from South Africa to Greece.

There is one more output that you may request. I won't tell you what it is, but if you ask for it specifically I will send you a link to where you can download it. You need this other piece of the puzzle to come up with a good solution.

I showed this exercise to Kirk's performance tuning class last week and we solved it together in about 10 minutes. This one change got the time down from 31.5 minutes to 23.63, representing a noticeable 25% improvement. That is impressive for a single change!

Kind regards

Heinz

Issue 201 - Fork/Join With Fibonacci and Karatsuba

Author: Dr. Heinz M. Kabutz

Date: 2012-05-24

Category: Concurrency

Java Versions: Java 7

Abstract:

The new Java 7 Fork/Join Framework allows us to define our algorithms using recursion and then to easily parallelize them. In this newsletter we describe how that works using a fast Fibonacci algorithm that uses the sum of the squares rather than brute force. We also present a faster algorithm for multiplying two large BigInteger numbers, using the Fork/Join Framework and the Karatsuba algorithm.

Welcome to the 201st issue of **The Java(tm) Specialists' Newsletter** sent to you from the Island of Crete. Kirk Pepperdine is running his Java **Performance Tuning Course** at our **conference room** this week. Helene had the brilliant idea that they should shift their day by a few hours today. So instead of starting sharp at 9am, we went to **a secret beach for a swim** and then began with the lessons at 11:30am. Kirk will catch up with the material tonight. I think we will do this on all our Cretan courses in future. A quick jump in our fresh-water pool was a perfect way to get me inspired to write this newsletter (You might enjoy this **short video my son and his buddy made of our pool** - a friend gave us some old playground slides that we have hooked up to our pool pump to make a lovely water slide - the video might be blocked in your region due to the sound track.)

So, now that you're completely annoyed with my introduction and are sitting there behind your desk fuming with envy, there are two opportunities that you could take advantage of to visit our beautiful island :-) First off, I am running my **Java Specialist Master Course** here in 11 days time, from the 4-7 June 2012. It is the perfect course for a Java programmer who has used Java for at least 2 years and who would like to grow in their knowledge of Java. Money-back guarantee if you are not fully satisfied! Flights to Greece are cheap this year and Crete is perfectly safe. Secondly, we are running our second **Java Specialist Symposium** from the 10-13 September 2012. We doubled the attendance fees from last year (\$0) and will keep on doing that every year for as long as we do the symposium. Space is limited though, first-come-first-served.

One last interesting bit of news from Crete, before we start with the newsletter. One of the main tourist areas, Platanias, is also a breeding ground for the loggerhead turtle, called Caretta-Caretta. Why they allowed hotels and beach umbrellas in that area boggles the mind. Last month, I took some friends out for a drive in my boat and almost ran over a turtle that was swimming in the middle of the sea. It was the first time I saw one of these critters in the wild and it was a completely amazing experience watching and filming it. My son was kind enough to turn our recordings into a **short documentary about the Caretta-Caretta**. Some viewers in Canada had errors when they tried to play it. If that happens, please just try again. I think you will enjoy seeing the turtle as much as I did.

Fork/Join With Fibonacci

Figuring out new language features can be daunting. The JavaDocs are not always that helpful. Here is an example shown in the [RecursiveTask JavaDocs](#) to demonstrate how the fork/join framework should be used:

```
class Fibonacci extends RecursiveTask<Integer> {
    final int n;
    Fibonacci(int n) { this.n = n; }
    Integer compute() {
        if (n <= 1)
            return n;
        Fibonacci f1 = new Fibonacci(n - 1);
        f1.fork();
        Fibonacci f2 = new Fibonacci(n - 2);
        return f2.compute() + f1.join();
    }
}
```

What is wrong with this example? A minor issue is that it contains a fairly basic Java syntax error. The `compute()` method is implementing a protected abstract method defined in the `RecursiveTask` class. In the example, the method is package access. It should either be protected or public. We can widen the access to an overridden method, but we cannot narrow it. Stylistically also, the field "n" should be private.

However, the syntax error will be semi-automatically repaired by any decent IDE. The real problem is one of computational complexity. The algorithm for Fibonacci presented here is exponential. Solving $f(n+1)$ requires approximately twice as many steps as $f(n)$. Thus if we manage to solve $f(n)$ with a single core within some time, then with a 1024 processor machine, we will only be able to solve the $f(n+10)$ in the same amount of time.

The idea of using Fibonacci as an example is not bad, but the choice of an exponential algorithm was unfortunate. I tried to produce a fast Java implementation of the famous function. Exactly three months ago I [posted a challenge on java.net to try to find Fibonacci of one billion](#). Alexey Solodovnikov managed to solve the problem in 36 seconds using gcc 4.5.2 + gmp with a 3 GHz single core machine. His Java version took about 2.5 hours. Rexy Young solved it using Dijkstra's famous sum of squares algorithm (more later) in 55 minutes. The code I will present in this newsletter solved the problem without any 3rd party libraries in 34 minutes in Java on my 8-core server.

Something else I have done is start uploading the code from my newsletters onto GitHub. You can grab it from <git://github.com/kabutz/javaspecialists.git> if you like and then build it with Maven. It will still be changing a lot until I have decided how to arrange the various packages. I also still have to pull the source code from over 200 Java newsletters into my project tree. This will take a while, but I think it's good that my friend Alvaro pushed me into this direction. Thanks Alvaro! The license will probably be Apache 2.0, unless there are any compelling reasons why that won't do.

The first class we have is Fibonacci. It contains caching behaviour enabled by default, since a lot of Fibonacci algorithms do much better with it turned on. You might notice that the calculate() and doActualCalculate() methods both declare that they throw the InterruptedException. We can use interrupts to cancel long calculations.

```
import java.math.*;

public abstract class Fibonacci {
    private final FibonacciCache cache;

    protected Fibonacci(FibonacciCache cache) {
        this.cache = cache;
    }

    public Fibonacci() {
        this(null);
    }

    public BigInteger calculate(int n)
        throws InterruptedException {
        if (cache == null) return doActualCalculate(n);

        BigInteger result = cache.get(n);
        if (result == null) {
            cache.put(n, result = doActualCalculate(n));
        }
        return result;
    }

    protected abstract BigInteger doActualCalculate(int n)
        throws InterruptedException;
}
```

Here is an implementation that uses the caching feature of the Fibonacci class. The algorithm is recursive, but thanks to the cache, we can do all the calculations in linear time.

However, the BigInteger.add() method is also linear, which means that overall, this function ends up being quadratic. This means that if we double the size of n, it will take four times as long to solve. In addition, because it is still recursive in nature, the stack depth is a limiting factor in which numbers we can calculate.

```
import java.math.*;

public class FibonacciRecursive extends Fibonacci {
    public BigInteger doActualCalculate(int n)
        throws InterruptedException {
        if (Thread.interrupted()) throw new InterruptedException();
        if (n < 0) throw new IllegalArgumentException();
        if (n == 0) return BigInteger.ZERO;
        if (n == 1) return BigInteger.ONE;
        return calculate(n - 1).add(calculate(n - 2));
    }
}
```

At this point, we should probably show the special FibonacciCache class. It contains the optimization that if we request "n" and the cache does not contain that, we see whether "n-1" and "n-2" are contained. If they are, then we can use those to quickly calculate the Fibonacci value for "n". Similarly, we can look for "n+1" and "n+2" or "n+1" and "n-1" to find "n" quickly. It also tries to avoid calculating the same value twice through an elaborate reserved caching scheme (see cacheReservation and solutionArrived fields). Unfortunately this solution does not increase the number of worker threads in the ForkJoinPool if one of the threads is blocked waiting for another thread to finish its work on a number.

One of the things that kept on happening was that several threads would request the same number at the same time. Since neither of them found it, they would both go and work out the value. This wasted a lot of computing power. Thus, before we return "null" from this cache, we reserve the right to calculate the number by adding the value of "n" into the cacheReservation set. If another thread now tries to calculate the same fibonacci number, it is suspended until the first thread finds the answer. The speedup with this optimization was quite dramatic, but required that the number of threads in the ForkJoinPool exceeded the number of cores.

```
import java.math.*;
import java.util.*;
import java.util.concurrent.*;
import java.util.concurrent.locks.*;
```

```

class FibonacciCache {
    private final ConcurrentHashMap<Integer, BigInteger> cache =
        new ConcurrentHashMap<>();

    private final Lock lock = new ReentrantLock();
    private final Condition solutionArrived = lock.newCondition();
    private final Set<Integer> cacheReservation = new HashSet<>();

    public BigInteger get(int n) throws InterruptedException {
        lock.lock();
        try {
            while (cacheReservation.contains(n)) {
                // we now want to wait until the answer is in the cache
                solutionArrived.await();
            }
            BigInteger result = cache.get(n);
            if (result != null) {
                return result;
            }

            BigInteger nMinusOne = cache.get(n - 1);
            BigInteger nMinusTwo = cache.get(n - 2);
            if (nMinusOne != null && nMinusTwo != null) {
                result = nMinusOne.add(nMinusTwo);
                put(n, result);
                return result;
            }

            BigInteger nPlusOne = cache.get(n + 1);
            BigInteger nPlusTwo = cache.get(n + 2);
            if (nPlusOne != null && nPlusTwo != null) {
                result = nPlusTwo.subtract(nPlusOne);
                put(n, result);
                return result;
            }

            if (nPlusOne != null && nMinusOne != null) {
                result = nPlusOne.subtract(nMinusOne);
                put(n, result);
                return result;
            }
            cacheReservation.add(n);
            return null;
        } finally {
            lock.unlock();
        }
    }

    public void put(int n, BigInteger value) {

```

```

        lock.lock();
    try {
        solutionArrived.signalAll();
        cacheReservation.remove(n);
        cache.putIfAbsent(n, value);
    } finally {
        lock.unlock();
    }
}
}

```

Some of the algorithms do not need caching. For example, if we solve the problem with iteration, then we will never try to find the same Fibonacci number more than once. Our NonCachingFibonacci abstract base class redefines the calculate() method as abstract and stops us from overriding doActualCalculate():

```

import java.math.*;

public abstract class NonCachingFibonacci extends Fibonacci {
    protected NonCachingFibonacci() {
        super(null);
    }

    public final BigInteger doActualCalculate(int n)
        throws InterruptedException {
        throw new UnsupportedOperationException();
    }

    public abstract BigInteger calculate(int n)
        throws InterruptedException;
}

```

The first example that uses the NonCachingFibonacci is the iterative solution. If we ran this with "long" values, it would be very fast. However, we are slowed down by the BigInteger method being linear, which again makes this function quadratic, similar to the FibonacciRecursive class shown earlier. Since it is not recursive, this function does not run out of stack space as would happen with FibonacciRecursive.

```

import java.math.*;

public class FibonacciIterative extends NonCachingFibonacci {
    public BigInteger calculate(int n)
        throws InterruptedException {
        if (n < 0) throw new IllegalArgumentException();
        BigInteger n0 = BigInteger.ZERO;
        BigInteger n1 = BigInteger.ONE;
        for (int i = 0; i < n; i++) {
            if (Thread.interrupted()) throw new InterruptedException();
            BigInteger temp = n1;
            n1 = n1.add(n0);
            n0 = temp;
        }
        return n0;
    }
}

```

The classical Fibonacci algorithm as described in the RecursiveTask JavaDocs is exponential. The only way of making it a bit performant is to cache the previous values. In this version we do *not* cache, making it incredibly slow. We can thus only calculate for very small values of n.

```

import java.math.*;

public class FibonacciRecursiveNonCaching
    extends NonCachingFibonacci {
    public BigInteger calculate(int n)
        throws InterruptedException {
        if (Thread.interrupted()) throw new InterruptedException();
        if (n < 0) throw new IllegalArgumentException();
        if (n == 0) return BigInteger.ZERO;
        if (n == 1) return BigInteger.ONE;
        return calculate(n - 1).add(calculate(n - 2));
    }
}

```

We have shown simple iterative and recursive solutions. However, there is also a formula that we can use to calculate a specific Fibonacci number, without having to go back and calculate the previous numbers. This formula depends on the square root of 5. With "long" and "double", we can work out an accurate value of Fibonacci up to n=71.

Let $\phi = (1 + \sqrt{5}) / 2$ and $\psi = (1 - \sqrt{5}) / 2$. $Fibonacci(n) = (\phi^n - \psi^n) / (\phi - \psi)$, where " n " means "to the power of". Or we could simply say $Fibonacci(n) = (\phi^n - \psi^n) / \sqrt{5}$.

```
import java.math.*;

public class FibonacciFormulaLong extends NonCachingFibonacci {
    private static final double root5 = Math.sqrt(5);
    private static final double PHI = (1 + root5) / 2;
    private static final double PSI = (1 - root5) / 2;
    private static final int MAXIMUM_PRECISE_NUMBER = 71;

    public BigInteger calculate(int n)
        throws InterruptedException {
        if (Thread.interrupted()) throw new InterruptedException();
        if (n < 0) throw new IllegalArgumentException();
        if (n > MAXIMUM_PRECISE_NUMBER)
            throw new IllegalArgumentException(
                "Precision loss after " + MAXIMUM_PRECISE_NUMBER);
        return new BigInteger(Long.toString(fibWithFormula(n)));
    }

    private static long fibWithFormula(int n) {
        return (long)((Math.pow(PHI, n) - Math.pow(PSI, n)) / root5);
    }
}
```

If we want to be able to work out $Fibonacci(1000)$ with this formula, we are forced to use `BigDecimal`. Unfortunately it is a rather slow class. I used a value of `root5` that would give me an accurate Fibonacci number up to $n=1000$. The more accurate we make `root5`, the slower this algorithm takes. The biggest time sink is the call to `divide(root5)` at the end of the calculation.

```
import java.math.*;

public class FibonacciFormulaBigInteger extends NonCachingFibonacci {
    private static final BigDecimal root5 = new BigDecimal(
        "2.23606797749978969640917366873127623544061835961152572" +
```

```

"4270897245410520925637804899414414408378782274969508176" +
"1507737835042532677244470738635863601215334527088667781" +
"7319187916581127664532263985658053576135041753378");
private static final BigDecimal PHI = root5.add(
    new BigDecimal(1)).divide(new BigDecimal(2));
private static final BigDecimal PSI = root5.subtract(
    new BigDecimal(1)).divide(new BigDecimal(2));
private static final int MAXIMUM_PRECISE_NUMBER = 1000;

public BigInteger calculate(int n) throws InterruptedException {
    if (Thread.interrupted()) throw new InterruptedException();
    if (n < 0) throw new IllegalArgumentException();
    if (n > MAXIMUM_PRECISE_NUMBER)
        throw new IllegalArgumentException(
            "Precision loss after " + MAXIMUM_PRECISE_NUMBER);

    BigDecimal phiToTheN = PHI.pow(n);
    if (Thread.interrupted()) throw new InterruptedException();
    BigDecimal psiToTheN = PSI.pow(n);
    if (Thread.interrupted()) throw new InterruptedException();
    BigDecimal phiMinusPsi = phiToTheN.subtract(psiToTheN);
    BigDecimal result = phiMinusPsi.divide(
        root5, 0, RoundingMode.UP);
    return result.toBigInteger();
}
}

```

So far, the best we have achieved is a linear algorithm, but with the addition of linear `BigInteger.add()`, we ended up with quadratic performance. Can we do better?

One of the algorithms that is popular is Dijkstra's sum of the squares is also mentioned on [R.Knott's website](#).

My implementation goes like this: if n is odd, then $f(2n-1) = f(n-1)^2 + f(n)^2$; otherwise $f(2n) = (2 * f(n-1) + f(n)) * f(n)$.

```

import java.math.*;

public class FibonacciRecursiveDijkstra extends Fibonacci {
    public BigInteger doActualCalculate(int n)
        throws InterruptedException {

```

```

if (Thread.interrupted()) throw new InterruptedException();
if (n == 0) return BigInteger.ZERO;
if (n == 1) return BigInteger.ONE;
if (n % 2 == 1) {
    // f(2n-1) = f(n-1)^2 + f(n)^2
    int left = (n + 1) / 2;
    int right = (n + 1) / 2 - 1;
    return square(calculate(left)).add(square(calculate(right)));
} else {
    // f(2n) = (2 * f(n-1) + f(n)) * f(n)
    int n_ = n / 2;
    BigInteger fn = calculate(n_);
    BigInteger fn_1 = calculate(n_ - 1);
    return (fn_1.add(fn_1).add(fn)).multiply(fn);
}
}

protected BigInteger multiply(BigInteger bi0, BigInteger bi1) {
    return bi0.multiply(bi1);
}

protected BigInteger square(BigInteger num) {
    return multiply(num, num);
}
}

```

The algorithm now is logarithmic, meaning that we can solve Fibonacci(1000) with about 10 calculations. I expected this to be really fast and to outperform all the other algorithms. After all, the iterative and simple recursive solutions both are quadratic. However, it turns out that the bottleneck in the above code is the BigInteger.multiply() method, which is quadratic. So we now have $O(n^2 * \log n)$ performance. Good enough to find Fibonacci for a thousand but not for a billion.

After some research, I stumbled across the algorithm by **Karatsuba**. Instead of quadratic performance, it can achieve $3n^{1.585}$ performance. There are other algorithms one can use as your numbers get larger, but I tried to build this solver without using third-party libraries. Examples of third-party libraries are **JScience's LargeInteger**, which internally uses Karatsuba and also a parallel version. However, I really wanted a solution that could cooperate with an existing ForkJoinPool rather than start new threads. Another option would be to simply use a Java wrapper to **GMP**, which should give us the same speed as a C wrapper to GMP, since all we are doing is delegate the Fibonacci function to GMP.

For large numbers, Karatsuba's multiplication algorithm, is a lot faster than ordinary BigInteger multiply. We give two implementations, the BasicKaratsuba for single-threaded calculations and the ParallelKaratsuba that utilizes the Fork/Join Framework. Another place

that mentions Karatsuba is in Sedgewick and Wayne's book **Introduction to Programming in Java: An Interdisciplinary Approach**.

```
import java.math.*;  
  
public interface Karatsuba {  
    BigInteger multiply(BigInteger x, BigInteger y);  
}
```

Our Karatsuba calculations use some utility functions for adding and splitting BigIntegers.

```
import java.math.*;  
  
class BigIntegerUtils {  
    public static BigInteger add(BigInteger... ints) {  
        BigInteger sum = ints[0];  
        for (int i = 1; i < ints.length; i++) {  
            sum = sum.add(ints[i]);  
        }  
        return sum;  
    }  
  
    public static BigInteger[] split(BigInteger x, int m) {  
        BigInteger left = x.shiftRight(m);  
        BigInteger right = x.subtract(left.shiftLeft(m));  
        return new BigInteger[]{left, right};  
    }  
}
```

The BasicKaratsuba contains a threshold value. If either of the numbers being multiplied is less than the threshold, then we use the standard BigInteger.multiply().

```
import java.math.*;  
import static eu.javaspecialists.tjsn.math.numbers.BigIntegerUtils.*;
```

```

public class BasicKaratsuba implements Karatsuba {
    public static final String THRESHOLD_PROPERTY_NAME =
        "eu.javaspecialists.tjsn.math.numbers.BasicKaratsubaThreshold";
    private static final int THRESHOLD = Integer.getInteger(
        THRESHOLD_PROPERTY_NAME, 1000);

    public BigInteger multiply(BigInteger x, BigInteger y) {
        int m = java.lang.Math.min(x.bitLength(), y.bitLength())/2;
        if (m <= THRESHOLD)
            return x.multiply(y);

        // x = x1 * 2^m + x0
        // y = y1 * 2^m + y0
        BigInteger[] xs = BigIntegerUtils.split(x, m);
        BigInteger[] ys = BigIntegerUtils.split(y, m);

        // xy = (x1*2^m + x0)(y1*2^m + y0) = z2*2^2m + z1*2^m + z0
        // where:
        // z2 = x1 * y1
        // z0 = x0 * y0
        // z1 = x1 * y0 + x0 * y1 = (x1 + x0)(y1 + y0) - z2 - z0
        BigInteger z2 = multiply(xs[0], ys[0]);
        BigInteger z0 = multiply(xs[1], ys[1]);
        BigInteger z1 = multiply(add(xs), add(ys)).
            subtract(z2).subtract(z0);

        // result = z2 * 2^2m + z1 * 2^m + z0
        return z2.shiftLeft(2 * m).add(z1.shiftLeft(m)).add(z0);
    }
}

```

The ParallelKaratsuba is almost identical to the BasicKaratsuba, except that it distributes the individual tasks to the ForkJoinPool. We can thus utilize the available hardware to multiply the numbers faster. Note how similar it looks to the sequential version in BasicKaratsuba. As long as your algorithm is recursive, it is very easy to employ the Fork/Join framework to speed up your work.

```

import java.math.*;
import java.util.concurrent.*;

import static eu.javaspecialists.tjsn.math.numbers.BigIntegerUtils.*;

public class ParallelKaratsuba implements Karatsuba {

```

```

public static final String THRESHOLD_PROPERTY_NAME =
    "eu.javaspecialists.tjsn.math.numbers.ParallelKaratsubaThreshold";
private static final int THRESHOLD = Integer.getInteger(
    THRESHOLD_PROPERTY_NAME, 1000);
private final ForkJoinPool pool;

public ParallelKaratsuba(ForkJoinPool pool) {
    this.pool = pool;
}

public BigInteger multiply(BigInteger x, BigInteger y) {
    return pool.invoke(new KaratsubaTask(x, y));
}

private static class KaratsubaTask
    extends RecursiveTask<BigInteger> {
    private final BigInteger x, y;

    public KaratsubaTask(BigInteger x, BigInteger y) {
        this.x = x;
        this.y = y;
    }

    protected BigInteger compute() {
        int m = (Math.min(x.bitLength(), y.bitLength()) / 2);
        if (m <= THRESHOLD) {
            return x.multiply(y);
        }

        BigInteger[] xs = split(x, m);
        BigInteger[] ys = split(y, m);

        KaratsubaTask z2task = new KaratsubaTask(xs[0], ys[0]);
        KaratsubaTask z0task = new KaratsubaTask(xs[1], ys[1]);
        KaratsubaTask z1task = new KaratsubaTask(add(xs), add(ys));

        z0task.fork();
        z2task.fork();
        BigInteger z0, z2;
        BigInteger z1 = z1task.invoke().subtract(
            z2 = z2task.join()).subtract(z0 = z0task.join());

        return z2.shiftLeft(2*m).add(z1.shiftLeft(m)).add(z0);
    }
}
}

```

Karatsuba makes a big difference in the performance, especially if we are able to utilize the available cores. We can do the same thing with Dijkstra's Fibonacci algorithm where we distribute the various Fibonacci calculations to the various cores. However, the biggest speedup is probably from the Karatsuba multiply, since that is where most of the time is spent. Our parallel solution uses its own internal FibonacciCache.

```

import java.math.*;
import java.util.concurrent.*;

public class FibonacciRecursiveParallelDijkstraKaratsuba
    extends NonCachingFibonacci {
    public static final String SEQUENTIAL_THRESHOLD_PROPERTY_NAME =
        "eu.javaspecialists.tjsn.math.fibonacci.SequentialThreshold";
    private final static int SEQUENTIAL_THRESHOLD =
        Integer.getInteger(SEQUENTIAL_THRESHOLD_PROPERTY_NAME, 10000);
    private final FibonacciCache cache = new FibonacciCache();
    private final Fibonacci sequential =
        new FibonacciRecursiveDijkstraKaratsuba();
    private final ForkJoinPool pool;
    private final Karatsuba karatsuba;

    public FibonacciRecursiveParallelDijkstraKaratsuba(
        ForkJoinPool pool) {
        this.pool = pool;
        karatsuba = new ParallelKaratsuba(pool);
    }

    public BigInteger calculate(int n) throws InterruptedException {
        if (Thread.interrupted()) throw new InterruptedException();
        BigInteger result = pool.invoke(new FibonacciTask(n));
        if (result == null) throw new InterruptedException();
        return result;
    }

    private class FibonacciTask extends RecursiveTask<BigInteger> {
        private final int n;

        private FibonacciTask(int n) {
            this.n = n;
        }

        protected BigInteger compute() {
            try {
                BigInteger result = cache.get(n);
                if (result != null) {
                    return result;
                }
            }
        }
    }
}

```

The FibonacciGenerator uses the given Fibonacci function to work out the result as a BigDecimal. However, the conversion of BigDecimal to a String is again, you guessed it, quadratic. Instead, what we do is print out the first and last 10 bytes and an Adler32

checksum.

```

import java.math.*;
import java.util.zip.*;

public class FibonacciGenerator {
    private final Fibonacci fib;

    public FibonacciGenerator(Fibonacci fib) {
        this.fib = fib;
    }

    public void findFib(int n) throws InterruptedException {
        System.out.printf("Searching for Fibonacci(%d)%n", n);
        long time = System.currentTimeMillis();
        BigInteger num = fib.calculate(n);
        time = System.currentTimeMillis() - time;
        printProof(num);
        System.out.printf(" Time to calculate %d ms%n", time);
    }

    private void printProof(BigInteger num) {
        System.out.printf(" Number of bits: %d%n", num.bitLength());
        byte[] numHex = num.toByteArray();
        System.out.print(" First 10 bytes: ");
        for (int i = 0; i < 10; i++) {
            System.out.printf(" %02x", numHex[i]);
        }
        System.out.println();

        System.out.print(" Last 10 bytes: ");
        for (int i = numHex.length - 10; i < numHex.length; i++) {
            System.out.printf(" %02x", numHex[i]);
        }
        System.out.println();
    }

    Checksum ck = new Adler32();
    ck.update(numHex, 0, numHex.length);
    System.out.printf(" Adler32 Checksum: 0x%016x%n",
        ck.getValue());
}
}

```

The FibonacciGeneratorExample finds Fibonacci of one billion in about 34 minutes on my

8-core server.

```

import java.util.concurrent.*;

public class FibonacciGeneratorExample {
    private static ForkJoinPool pool = new ForkJoinPool(
        Runtime.getRuntime().availableProcessors() * 4);

    public static void main(String[] args)
        throws InterruptedException {
        int[] ns;
        if (args.length != 0) {
            ns = new int[args.length];
            for (int i = 0; i < ns.length; i++) {
                ns[i] = Integer.parseInt(args[i]);
            }
        } else {
            ns = new int[]{
                1_000_000,
                10_000_000,
                100_000_000, // takes a bit long
                1_000_000_000, // takes a bit long
            };
        }
        test(new FibonacciRecursiveParallelDijkstraKaratsuba(pool), ns);
    }

    private static void test(Fibonacci fib, int... ns)
        throws InterruptedException {
        for (int n : ns) {
            FibonacciGenerator fibgen = new FibonacciGenerator(fib);
            fibgen.findFib(n);
            System.out.println(pool);
        }
    }
}

```

I would love to hear if you can come up with an algorithm in Java to solve Fibonacci one billion faster on my hardware than my 34.58 minutes. Fastest solution gets a whopping 34.58% discount on the **Java Symposium entrance fees!** The rules are: you need to write the code yourself, but you can use existing algorithms.

Kind regards

Heinz

P.S. Did you know that the Fibonacci series occurs in nature? For example, the shells of a turtle are arranged according to the series. That's something to think about as you **watch this big fellow swimming around the Gulf of Chania.**

Issue 200 - On Learning Concurrency

Author: Dr. Heinz M. Kabutz

Date: 2012-03-26

Category: Inspirational

Java Versions: All Software

Abstract:

Every Java programmer I have met knows that they should know more about concurrency. But it is a topic that is quite hard to learn. In this newsletter I give some tips on how you can become proficient in concurrency.

Welcome to the 200th issue of **The Java(tm) Specialists' Newsletter!** Thank you for your amazing support. It has been fun. As a token of appreciation, I am presenting a gratis three-hour online workshop on Thursday 29th March 2012 from 6-9pm GMT. Topic is **Avoiding Liveness Hazards**, which incidentally was also the topic of my first newsletter. **Please register here**. In the first 1.5 hours I will lecture on the subject and then it will be up to you to find liveness hazards in some example code using the techniques presented. We will also have some time for questions and discussions.

I had the privilege of being invited to speak at the **codemotion conference** in Spain on the weekend, which was also broadcast in Rome. I expected to see about 150 people, but it was completely packed with over 1000 attendees. That is a very good turnout for a first programming conference. Well done to **David Gómez García** who organized the event in Madrid.

One of my hobbies is food. When I was a child, I wanted to own a restaurant. I built my own little taverna underneath a palm tree in our garden, where my poor family was treated to the most horrible meals, which they pretended to enjoy. As a child, I frequently had stomach problems, maybe as a result of my passion with food. Every time my mother took me to the doctor, she complained about how expensive he was. I quickly changed my career prospects to doctor and told everybody "because then I can help people". Fortunately I discovered the joys of programming in my teens and a hobby became my career.

Madrid is a place to visit if you like good food. I had one of my best steaks ever at the **Casa Paco**. Despite what the reviews say, prices are reasonable for what you are getting. I did not get suckered into trying their various starters, since there are better places for tapas. The meat was old ox, very tender and with lots of flavour. I think I would have done a better job of grilling it, but the quality of meat made up for it. I also had a fantastic Mexican meal at the Taqueria del Alamillo with my friend and Java Specialist instructor for Latin America, Alvaro Hernandez. We decided to go there on the spur of the moment and were very lucky to get a table after a long wait. Great food, but I would have preferred it a bit more spicy.

On Learning Concurrency

"Hi, I'm Anton." A large red-haired fellow shook my hand at the special maths 106 class for gifted students. In the previous year, half the class had failed and the University of Cape Town had considered cancelling this course. You were only allowed to take the class if you had achieved a high maths mark in high school. Since this was a small class, there were only three students with the exact same selection of classes: Anton de Swardt, John Green and myself. I had taken the special maths class because I was under the illusion that I was quite smart in the subject, after coming 11th in the maths olympiad of Cape Town, a city of three million people. Boy was I in for a shock. I scraped through the subject, whilst John got 99%.

From then onwards, the three of us were inseparable. Whenever there was a group project to be done in computer science, it was at least John, Anton and Heinz, with the occasional addition of another of our friends, Zach. John managed to get the class medal for computer science three years in a row, the only person to ever manage that. He always beat Anton and me in tests and exams. To say that he is a genius is not giving him enough credit. After all, he came second in all the schools of his race, despite having to stay at home for the entire grade 10 due to rioting. The 80s in South Africa were tumultuous. Thus super-genius John always had better marks than me, except one single time.

Of all the modules that I chose, the one that I found the most interesting was on parallel and distributed computing. For some reason, I took to the course like a duck to water. John took the same course and was hardly seen in class, having also signed up for a bunch of additional extramural activities, such as organizing the university choir and learning ballroom dancing. When the big day of the exam came, I was extremely well prepared like a well-oiled machine. John had barely opened the book. We wrote the exam and afterwards John asked me which two questions I had decided to answer. We were supposed to answer three of the given four questions. Accidentally, John had misread the instructions and had only answered two! He ended up with 65% and I with 96%. It was the only shot I had at beating him academically and at the same time was John's worst mark of his entire university career.

Parallel and concurrent programming is something that every Java programmer realizes that they *should* know more about. Talks on multi-threading are always well attended at conferences. Threading was usually the subject where programmers lost points in the Java Programmer Certification. When I wrote the Java 5 certification, I also lost a point for threading, which I mentioned in [newsletter 120](#). But it was not my fault. The question had a mistake in it and there was no correct answer. After I mentioned this in my newsletter, someone from Sun Microsystems found the question and removed it from the examination. Thus even the inventors of Java made mistakes with this tricky subject.

As I said already, every Java programmer I speak to is aware that they need to know more about concurrency. Even if they are not always exposed to the innards due to the frameworks they are using, the beast of concurrency still leaks through. It might raise its ugly head due to their systems not scaling due to contention or they could see deadlocks or race conditions.

Last Friday I gave a deadlock problem to two groups of Java programmers to solve. Most of

them immediately started snooping around in the source code. The exercise consisted of 15000 lines of ugly Java code. With a bit of luck, they could possibly find the problem in a couple of hours. However, if it were 15 million lines of code, then there would be no chance at all of discovering the deadlock by trawling through the source. The correct approach would have been to look at the stack trace and check what the threads were up to. By process of elimination, they would pretty quickly find three or four candidates and the stack trace would then have taken them to the exact line of code that was at fault.

I heard about a project where over 1000 methods were marked as synchronized. Did they really need that many methods to be single-threaded? Some of the methods were 2000 lines long. Safety is important, but marking all your methods as synchronized could lead to deadlocks and poor performance. I am sure that the programmers had good intentions. They might even have read a tutorial or a book on the subject. Knowing just a little bit can be more dangerous than knowing nothing at all.

So how did I learn about Java concurrency? First off, I had the privilege of getting a good foundation in distributed and parallel programming with Prof Ken MacGregor at the University of Cape Town. This was further expanded on with a PhD on analysing the performance of communicating concurrent systems. As those with a PhD know, the degree is more about persistence than brains. Also, the information I learned and discovered was not that useful for real world Java programs. Again, this is typical of a post-graduate degree. My education at university was thus a small help, but I would have also learned threading without that.

I started coding Java in 1997 and a lot of the work I did involved threading. I coded several concurrency bugs in my program. Usually we learn best by making mistakes. Fortunately none of my errors caused a serious loss of income for my company. I did manage to write race conditions and deadlocks, sometimes in the most unusual way.

In 1999 I wrote my first Sun Java Programmer Certification. To prepare myself, I studied threading in far more detail and realized that a lot of my understanding was incorrect. Certifications are useful in that they highlight the areas where we need to pay more attention.

In 2000, I studied both the [Java Virtual Machine Specification](#) and Doug Lea's [Concurrent Programming in Java](#). Both books explained a lot about the inner workings of Java. They are a bit dated by now, but lay a good foundation. The key verb here is "studied" rather than "read". It is never enough to simply "read" a book, you have to internalize it by working through it carefully and spending time trying out the code examples and possibly coming up with better ones.

By the time [Java Concurrency in Practice](#) was released in 2006, I was already well versed in the subject matter, which is why I had the honour of being allowed to review the book prior to publication. What I like about Goetz's book is that it covers the essentials of concurrency that every Java programmer needs to know. As Brian points out, concurrency is not some "advanced subject" that you can avoid learning about. Every Java programmer should understand threading.

I have a confession to make. I have never been on a Java course. I almost went once in early 1998, but then decided at the last moment that I probably knew more than the instructor. Arrogant, I know. It was true though. My colleagues afterwards told me that the instructor did not know the difference between `notify()` and `notifyAll()`. In the meantime, we were using that in production code. I learned later through Doug Lea's book that our production code had in fact been incorrect, but it did work for many years until another team found some serious errors in it. We had been lucky. However, we would not have learned of our errors from that course, since the instructor also did not know. Having a well-trained and knowledgeable instructor is what you are paying for in a course.

I spent about twenty weeks in the last twelve months studying Java Concurrency in Practice in great detail and producing my new **Concurrency Specialist Course**. When you write a course on such a complicated subject, you need to make sure that everything you write is correct. This means running your own experiments to verify that you see the same results. Sometimes new Java versions caused differences in behaviour. At other times there were better, more efficient approaches. Also, I wanted to use all the latest Java 7 utilities, such as Phaser and Fork/Join.

Another thing I did was join the **Concurrency Interest Mailing List**. This contains a lot of clever questions and allowed me to bounce a few of my ideas to the best brains in the industry. It's a list that is worthwhile joining, even just as an observer.

Here are some tips to follow if you would like to learn concurrency:

1. Recognize that concurrency is a topic that you *should* understand. Despite the excellent frameworks such as **Akka** now available for Java, you should still understand what is going on.
2. Get hold of some good books on the subject and study them. Spend at least 40 hours per chapter, trying out all the code examples and making sure that you understand everything.
3. Start writing some threaded code and try it out on lots of different hardware. Remember that if it works on your machine, you might have just been lucky.
4. Join the concurrency interest mailing list and read their archives. Try to just observe for the first few months.

It is not necessary to attend a course. I didn't. However, it does save you a lot of time and is a lot easier than trying to figure out everything yourself. But if you have lots of free time, then Goetz's book, coupled with the other resources I mentioned, will be enough to help you understand concurrency. If you do not have a lot of free time or would like a guide to take you on this fascinating journey, then please make sure that the guide understands the difference between `notify()` and `notifyAll()` :-)

Kind regards from Crete

Heinz

Issue 198 - Pushing the Limits in Java's Random

Author: Dr. Heinz M. Kabutz and Dr. Wolfgang Laun

Date: 2012-02-13

Category: Language

Java Versions: Java 7

Abstract:

What is the largest double that could in theory be produced by Math.random()? In this newsletter, we look at ways to calculate this based on the 48-bit random generator available in standard Java. We also prove why in a single-threaded program, $(int)(\text{Random.nextDouble()} + 1)$ can never be rounded up to 2.

Welcome to the 198th issue of **The Java(tm) Specialists' Newsletter** sent to you from Chania in Greece. My readers in colder climates frequently get annoyed when I tell them that "life's a beach". They imagine us having constant sun here. It's not true. We do sometimes have torrential rains and cold weather. It even snows here occasionally. In fact, the delivery truck for my heating oil arrived as I was writing this.

If you are looking for a great Java book to read, try pre-ordering **The Well-Grounded Java Developer**, by Ben Evans and Martijn Verburg. A book review will be published shortly.

We switched to a new email delivery system, which I believe will have a better success rate at getting this newsletter into your inbox. Our new system also allows new subscribers to specify their country. It is with great pleasure that I welcome: Djessy Menard from Haiti, Halle Johnson from Antigua and Barbuda, and Shaik Abdul Gafoor from Qatar. This brings the number of *known* subscriber **countries to 124**.

Pushing the Limits in Java's Random

My **previous newsletter** caused quite a stir. I saw lots of wrong answers and some excellent explanations. Quite a few thought that Math.random() did indeed sometimes return 1.0. It does not, but the value is sometimes close enough to 1.0 that we lose precision in the rounding.

Here is what the nextDouble() calculation looks like in java.util.Random:

```
public double nextDouble() {
    return (((long)(next(26)) << 27) + next(27))
```

```

    / (double)(1L << 53);
}

```

One of our most thorough readers, Dr. Wolfgang Laun, went on a voyage of discovery to find out what the highest number was that could possibly be returned by Math.random(). This newsletter is a summary of his findings, plus some other tidbits that I discovered. Dr. Laun lives in Vienna in Austria and works for Thales as their local software guru. I've had the pleasure of meeting him personally and been invited for dinner at his house. He is probably the most intelligent human being I have met. And he also has a great sense of humour and a keen curiosity of how the world works. I am honoured to count him as a friend.

The first question that Wolfgang tried to answer is: Can (int)(Math.random() + 1) ever return 2? I thought it could. The calculation for producing the double is to make up a 53 bit number and to then divide it by 2^{53} . This means that the maximum possible number would be $((2^{53}-1)/(2^{53}))$, or 0.9999999999999999. You can calculate this easily:

```
System.out.println(((1L << 53) - 1)) / (double) (1L << 53));
```

If we use the standard Random calculation as shown in makeDouble(), if we add 1 and cast it to an int, the result will be 2. For example:

```

public class CloseToOne {
    public static double makeDouble(long first, long second) {
        return ((first << 27) + second) / (double) (1L << 53);
    }

    public static void main(String[] args) {
        long first = (1 << 26) - 1;
        long second = (1 << 27) - 1;

        System.out.println(makeDouble(first, second));
        System.out.println((int)(makeDouble(first, second)+1));

        second--;
        System.out.println(makeDouble(first, second));
        System.out.println((int)(makeDouble(first, second)+1));
    }
}

```

```
}
```

In our previous newsletter, "meaning" was sometimes incremented by two, when Math.random was close enough to 1 and "meaning" was large enough. For example, if "meaning" is 100_000_000, we don't need to be as close to 1 as if "meaning" is 1.

```
public class CloseToOne2 {
    public static void main(String[] args) {
        double d = 0.999999993;
        System.out.println("d = " + d);
        int i = (int) (1 + d);
        System.out.println("i = " + i);
        int j = (int) (100_000_000 + d);
        System.out.println("j = " + j);
    }
}
```

The question is, can Math.random() return a significantly large number that is less than 1.0, but that makes `(int)(Math.random() + 1)` return 2? I thought so, but Wolfgang proved me wrong.

Java's random calculation is based on the 48-bit seed, linear congruential formula described in [The Art of Computer Programming, Volume 2](#) by Donald Knuth in Section 3.2.1. This is also described on [Wikipedia](#).

The point to note is that it is based on a 48-bit seed. In order for `(int)(Math.random() + 1)` to be 2, all the bits would have to be set on the left. If it is just one less, then the double would equal 0.9999999999999998 and this would not round up to 2.

Since the next random value is based on the current value, all we would need to check is whether there exists a 48-bit number with the lower 26 bits set, such that the next number has the lower 27 bits set. If we find such a number, then we can conclude that at least in theory, it is possible for Math.random() to return 0.9999999999999999. We would still need to establish for what random seed we could get this number. If we did not find such a number, then we could conclude that it was not possible.

Wolfgang wrote the code to calculate the largest number that could be returned by Math.random():

```

/**
 * @author Wolfgang Laun
 */

public class FindRandom {
    private final static long multiplier = 0x5DEECE66DL;
    private final static long addend = 0xBL;
    private final static long mask = (1L << 48) - 1;

    public static double makeDouble(long pre, long post) {
        return ((pre << 27) + post) / (double) (1L << 53);
    }

    private static long setbits(int len, int off) {
        long res = (1L << len) - 1;
        return res << off;
    }

    /**
     * A random double is composed from two successive random
     * integers.
     * The most significant 26 bits of the first one are taken and
     * concatenated with the most significant 27 bits of the second
     * one.
     * (ms(ril,26)) << 27 + (ms(ri2,27))
     * This is divided by (double)(1L << 53) to obtain a
     * result in [0.0, 1.0].
     * To find the maximum random double, we assume that
     * (ms(ril,m26))
     * is a maximum (all 1b) and vary the remaining 22 bits from
     * 0 to m22, inclusive. Assuming this to be ril, we perform the
     * calculation according to Random.next() to obtain its
     * successor, our ri2. The maximum of the most significant 27
     * bits of all ri2 would then be the second part of the maximum
     * 53-bit integer used for a double random's mantissa.
    */
    private static void findMaxDouble() {
        long ones = setbits(26, 22);
        System.out.println("ones: " + Long.toHexString(ones));
        long maxpost = setbits(22, 0);
        System.out.println("maxpost: " + Long.toHexString(maxpost));
        long maxintw = 0;
        for (long post = 0; post <= maxpost; post++) {
            long oldseed = ones + post;
            long nextseed = (oldseed * multiplier + addend) & mask;
        }
    }
}

```

```

long intw = nextseed >>> (48 - 27);
if (intw > maxintw) {
    maxintw = intw;
}
}
System.out.println("maxintw: " + Long.toHexString(maxintw));
long b26 = setbits(26, 0);
System.out.println("b26: " + Long.toHexString(b26));
double d = makeDouble(b26, maxintw);
System.out.println("max. double: " +
    Double.toHexString(d) + " = " + d);
}

private static void findMinDouble() {
    long b26 = 0L;
    long zeroes = 0L;
    long maxpost = setbits(22, 0);
    long minintw = 0x7fffffffffffffL;
    for (int post = 0; post < maxpost; post++) {
        long oldseed = zeroes + post;
        long nextseed = (oldseed * multiplier + addend) & mask;
        long intw = nextseed >>> (48 - 27);
        if (intw < minintw) {
            minintw = intw;
        }
    }
    System.out.println("minintw: " + minintw);
    double d = makeDouble(b26, minintw);
    System.out.println("min. double: " +
        Double.toHexString(d) + " = " + d);
}

public static void main(String[] args) {
    findMaxDouble();
    double belowOne = Math.nextAfter(1.0, 0.0);
    System.out.println("Biggest double below 1.0 is: " +
        Double.toHexString(belowOne) + " = " + belowOne);
    findMinDouble();
}
}

```

From this, we can calculate that the highest number that can be returned by Math.random() is 0.9999999999999996, whereas the highest double less than 1.0 is 0.9999999999999999, a difference of .0000000000000039.

Concurrency

We could stop here, were it not for concurrency. However, the `Math.random()` method delegates to a shared mutable instance of `Random`. Since they use atomics, rather than locking, it is impossible for the compound `nextDouble()` method to be atomic. Thus it is possible that in between the two calls to `next(27)` and `next(26)`, other threads call `next()`. Remember the code for `Random.nextDouble()` from earlier:

```
public double nextDouble() {
    return (((long)(next(26)) << 27) + next(27))
        / (double)(1L << 53);
}
```

In theory, if you had thousands of threads calling `Math.random()` at the same time, your thread could be swapped out for long enough, so that `(int)(Math.random() + 1)` could return 2. The probability of this happening might be greater than zero, but it is infinitesimally small. For example, in my experiments I did find a value of `next(26)` for which, if it was swapped out long enough and other threads had called `next(27)` 105 times, `next(27)` would return a number with all bits set. As Wolfgang told me, I'm rapidly entering into the realms of the metaphysical here.

Java 7 ThreadLocalRandom

As of Java 7, we should never again use `Math.random()`. Java 7 gives us a new `ThreadLocalRandom` class that has one unshared `Random` instance per thread. Since it is an unshared object, they do not need to have any synchronization to prevent data races. As a result it is blazingly fast.

We can use it like this: `ThreadLocalRandom.current().nextInt(3000);`

In this example, we construct 20 buttons and let them change color at a random interval using `ThreadLocalRandom`. I will go over this code again in another newsletter, as it also demonstrates the new Java 7 `Phaser` class.

```
import javax.swing.*;
import java.awt.*;
import java.util.*;
import java.util.concurrent.*;
```

```

/**
 * @author Heinz Kabutz
 */
public class Blinker extends JFrame {
    public Blinker() {
        setLayout(new GridLayout(0, 4));
    }

    private void addButtons(int buttons, final int blinks) {
        final Phaser phaser = new Phaser(buttons) {
            protected boolean onAdvance(int phase, int parties) {
                return phase >= blinks - 1 || parties == 0;
            }
        };
        for (int i = 0; i < buttons; i++) {
            final JComponent comp = new JButton("Button " + i);
            comp.setOpaque(true);
            final Color defaultColor = comp.getBackground();
            changeColor(comp, defaultColor);
            add(comp);
            new Thread() {
                public void run() {
                    Random rand = ThreadLocalRandom.current();
                    try {
                        Thread.sleep(1000);
                        do {
                            Color newColor = new Color(rand.nextInt());
                            changeColor(comp, newColor);
                            Thread.sleep(500 + rand.nextInt(3000));
                            changeColor(comp, defaultColor);
                            Toolkit.getDefaultToolkit().beep();
                            Thread.sleep(2000);
                            phaser.arriveAndAwaitAdvance();
                        } while (!phaser.isTerminated());
                    } catch (InterruptedException e) {
                        Thread.currentThread().interrupt();
                    }
                }
            }.start();
        }
    }

    private void changeColor(
        final JComponent comp, final Color color) {
        SwingUtilities.invokeLater(new Runnable() {
            public void run() {
                comp.setBackground(color);
            }
        });
    }
}

```

```

        invalidate();
        repaint();
    }
});

}

public static void main(String[] args) {
    SwingUtilities.invokeLater(new Runnable() {
        public void run() {
            Blinker blinker = new Blinker();
            blinker.addButtons(20, 3);
            blinker.pack();
            blinker.setVisible(true);
            blinker.setDefaultCloseOperation(
                WindowConstants.EXIT_ON_CLOSE);
        }
    });
}
}

```

If you run the Blinker with a version of Java 7 older than 1.7.0_02, you will notice that all the colors and delays are always the same. This is due to a bug in the constructor of `java.util.Random`, which failed to set the seed correctly on the subclass. Time to upgrade to the latest version of **Java 7**. Note that if you are using Mac OS X Snow Leopard or Leopard, the **Mac OS X Port installer** will tell you that you need to upgrade to Lion. Instead, I used **Pacifist** to extract the DMG file. I am not ready to install Lion on my work machines yet.

(int)(Math.random() * some_int)

A common code idiom is to multiply the result of `Math.random()` with some integer value. There are several reasons *not* to do that anymore. First off, `Math.random()` is using a shared mutable instance of `Random`, thus it needs to be synchronized. It does use atomics to eliminate locking, but with lots of threads calling `Math.random()`, you might still need to redo your work several times until you emerge as the winner.

The second reason is that `Math.random()` calls `nextDouble()`, which as we have seen, will call `next(bits)` twice.

The third reason is that `nextDouble() * int` is more biased. We will get a fairer distribution if we call `nextInt(upto)`.

One Last Thing

Concurrency allows us to do some strange things. For example, we can set the seed on a shared mutable Random instance so that eventually, `(int)(random.nextDouble() + 1)` will return 2. Here is the code:

```
public class MathTeaser {
    public static void main(String[] args) {
        final Random random = new Random();
        Thread seeder = new Thread() {
            public void run() {
                while(true) {
                    random.setSeed(51102269); // causes 2^26-1 as next(26)
                    random.setSeed(223209395); // causes 2^27-1 as next(27)
                }
            }
        };
        seeder.setDaemon(true);
        seeder.start();

        while(true) {
            double num = random.nextDouble();
            if ((int)(num + 1) == 2) {
                System.out.println("Yes, random.nextDouble() can: " + num);
                break;
            }
        }
    }
}
```

Thanks to Dr. Wolfgang Laun for the fascinating discussions of how Random really works and for your contribution in this newsletter.

Kind regards

Heinz

Issue 195 - Performance Puzzler With a Stack Trace

Author: Kirk Pepperdine

Date: 2011-09-15

Category: Performance

Java Versions: Sun Java 6

Abstract:

In this newsletter, we present a little performance puzzler, written by Kirk Pepperdine. What is happening with this system? There is only one explanation and it can be discovered by just looking at the stack trace.

Welcome to the 195th issue of **The Java(tm) Specialists' Newsletter** sent to you from the beautiful Island of Crete. The symposium two weeks ago went amazingly well. We had a total of 41 attendees this year, which was small enough to have animated discussions rather than just listen to presentations. At first, the delegates were shocked by the thought that it was up to them to self-organize. Once they get used to it, the discussions flowed much better than at traditional conferences. I certainly learned a lot, and the small group encourages participation. We will probably allow a maximum of 50 attendees in 2012.

Imagine a conference where the major focus is fun! Instead of being organized, we were deliberately disorganized. Kim and Kate Tucker did a marvelous job of eliciting ideas from us and generally keeping our un-conference flowing. Besides the technical discussions, we also went to some interesting places, such as Pirate Bay, a little cove that I discovered on Kirk's last course. The walk down is treacherous due to very thorny bushes. Having done the walk a few times before, I knew how to avoid the spikes, but some of our attendees still bear the mark of this "horror hike" as Helene calls it.

Being an un-conference, we did not even plan our after hours activities. With agility, we un-planned a barbecue at my house with 40 people. We only have 10 chairs. The butcher thought we were crazy when we ordered food for between 20 and 50 people. "How many are you exactly?", he said to us. "We have no idea", was a response. In the end, most of our guests sat on the floor, but without fail we all had a fantastic time. Next year, we will probably plan our activities a bit better. Restaurants can generally not cope with the sudden influx of 50 guests.

It gives me great pleasure to send you a puzzle written by Kirk Pepperdine. I eventually figured it out, but not before looking at the wrong thing initially. If you think you know the answer, please e-mail Kirk directly at kirk@kodewerk.com. Without any further ado, over to Mr. Pepperdine.

Performance Puzzler With a Stack Trace

This performance puzzler showed up in my inbox and it's kind of interesting and so I asked the owner of this data if I might share it with you all. Here is some background. The JVM is running Tomcat with BlueDragon, a CFML engine installed. The application was previously calm but since the move to OS X Lion, the Java process would ramp up to 135% CPU utilization and then just sit there. My immediate reaction was to request the thread dump that follows. The question is; what is this thread dump telling you and what would you do to confirm. I would only ask that you email me with your answer rather than spoiling other people's fun by posting it here.

[Heinz: In order to make the stack trace fit without line wrapping, I split up the first line of each thread and replaced java.lang with "j.l" and java.util.concurrent with "j.u.c"]

```
2011-09-06 19:37:43
Full thread dump Java HotSpot(TM) 64-Bit Server VM
(20.1-b02-383 mixed mode):

"Attach Listener"
    daemon
    prio=9
    tid=7f846ba62800
    nid=0x113e15000
    waiting on condition [00000000]
j.l.Thread.State: RUNNABLE

"0595F2FF-4284-4CBF-BAC941472F23EE14"
    daemon
    prio=5
    tid=7f846a810000
    nid=0x117bbf000
    waiting on condition [117bbe000]
j.l.Thread.State: WAITING (parking)
at sun.misc.Unsafe.park(Native Method)
- parking to wait for <7a0edb9f8>
    (a j.u.c.locks.AbstractQueuedSynchronizer$ConditionObject)
at j.u.c.locks.LockSupport.park(LockSupport.java:158)
at j.u.c.locks.AbstractQueuedSynchronizer$ConditionObject.await
    (AbstractQueuedSynchronizer.java:1987)
at j.u.c.LinkedBlockingQueue.take(LinkedBlockingQueue.java:399)
at org.apache.tomcat.util.threads.TaskQueue.take
    (TaskQueue.java:104)
at org.apache.tomcat.util.threads.TaskQueue.take
    (TaskQueue.java:32)
at j.u.c.ThreadPoolExecutor.getTask(ThreadPoolExecutor.java:947)
at j.u.c.ThreadPoolExecutor$Worker.run
    (ThreadPoolExecutor.java:907)
```

```
at j.l.Thread.run(Thread.java:680)

"C2F18A1A-8893-4FD8-B0B2703665071215"
    daemon
    prio=5
    tid=7f846b43f000
    nid=0x117abc000
    waiting on condition [117abb000]
j.l.Thread.State: WAITING (parking)
at sun.misc.Unsafe.park(Native Method)
- parking to wait for <7a0edb9f8>
    (a locks.AbstractQueuedSynchronizer$ConditionObject)
at j.u.c.locks.LockSupport.park(LockSupport.java:158)
at j.u.c.locks.AbstractQueuedSynchronizer$ConditionObject.await
    (AbstractQueuedSynchronizer.java:1987)
at j.u.c.LinkedBlockingQueue.take(LinkedBlockingQueue.java:399)
at org.apache.tomcat.util.threads.TaskQueue.take
    (TaskQueue.java:104)
at org.apache.tomcat.util.threads.TaskQueue.take
    (TaskQueue.java:32)
at j.u.c.ThreadPoolExecutor.getTask(ThreadPoolExecutor.java:947)
at j.u.c.ThreadPoolExecutor$Worker.run
    (ThreadPoolExecutor.java:907)
at j.l.Thread.run(Thread.java:680)

"399842CB-BEA7-498E-A9E3C27C8F945844"
    daemon
    prio=5
    tid=7f846c109000
    nid=0x1179b9000
    waiting on condition [1179b8000]
j.l.Thread.State: WAITING (parking)
at sun.misc.Unsafe.park(Native Method)
- parking to wait for <7a0edb9f8>
    (a j.u.c.locks.AbstractQueuedSynchronizer$ConditionObject)
at j.u.c.locks.LockSupport.park(LockSupport.java:158)
at j.u.c.locks.AbstractQueuedSynchronizer$ConditionObject.await
    (AbstractQueuedSynchronizer.java:1987)
at j.u.c.LinkedBlockingQueue.take(LinkedBlockingQueue.java:399)
at org.apache.tomcat.util.threads.TaskQueue.take
    (TaskQueue.java:104)
at org.apache.tomcat.util.threads.TaskQueue.take
    (TaskQueue.java:32)
at j.u.c.ThreadPoolExecutor.getTask(ThreadPoolExecutor.java:947)
at j.u.c.ThreadPoolExecutor$Worker.run
    (ThreadPoolExecutor.java:907)
at j.l.Thread.run(Thread.java:680)
```

```
"8C28E94D-6C2A-4EED-89C0BB9AA1A8BCCC"
    daemon
    prio=5
    tid=7f846b43e800
    nid=0x1178b6000
    waiting on condition [1178b5000]
j.l.Thread.State: WAITING (parking)
at sun.misc.Unsafe.park(Native Method)
- parking to wait for <7a0edb9f8>
    (a j.u.c.locks.AbstractQueuedSynchronizer$ConditionObject)
at j.u.c.locks.LockSupport.park(LockSupport.java:158)
at j.u.c.locks.AbstractQueuedSynchronizer$ConditionObject.await
    (AbstractQueuedSynchronizer.java:1987)
at j.u.c.LinkedBlockingQueue.take(LinkedBlockingQueue.java:399)
at org.apache.tomcat.util.threads.TaskQueue.take
    (TaskQueue.java:104)
at org.apache.tomcat.util.threads.TaskQueue.take
    (TaskQueue.java:32)
at j.u.c.ThreadPoolExecutor.getTask(ThreadPoolExecutor.java:947)
at j.u.c.ThreadPoolExecutor$Worker.run
    (ThreadPoolExecutor.java:907)
at j.l.Thread.run(Thread.java:680)
```

```
"05ED2DC5-849A-4EA4-806567122BEB6192"
    daemon
    prio=5
    tid=7f846c159800
    nid=0x1177b3000
    waiting on condition [1177b2000]
j.l.Thread.State: WAITING (parking)
at sun.misc.Unsafe.park(Native Method)
- parking to wait for <7a0edb9f8>
    (a j.u.c.locks.AbstractQueuedSynchronizer$ConditionObject)
at j.u.c.locks.LockSupport.park(LockSupport.java:158)
at j.u.c.locks.AbstractQueuedSynchronizer$ConditionObject.await
    (AbstractQueuedSynchronizer.java:1987)
at j.u.c.LinkedBlockingQueue.take(LinkedBlockingQueue.java:399)
at org.apache.tomcat.util.threads.TaskQueue.take
    (TaskQueue.java:104)
at org.apache.tomcat.util.threads.TaskQueue.take
    (TaskQueue.java:32)
at j.u.c.ThreadPoolExecutor.getTask(ThreadPoolExecutor.java:947)
at j.u.c.ThreadPoolExecutor$Worker.run
    (ThreadPoolExecutor.java:907)
at j.l.Thread.run(Thread.java:680)
```

```
"BA98F79A-50E1-4849-98A29B148EB006CE"
```

```
    daemon
```

```

prio=5
tid=7f846b43d800
nid=0x11756e000
waiting on condition [11756d000]
j.l.Thread.State: WAITING (parking)
at sun.misc.Unsafe.park(Native Method)
- parking to wait for <7a0edb9f8>
    (a j.u.c.locks.AbstractQueuedSynchronizer$ConditionObject)
at j.u.c.locks.LockSupport.park(LockSupport.java:158)
at j.u.c.locks.AbstractQueuedSynchronizer$ConditionObject.await
    (AbstractQueuedSynchronizer.java:1987)
at j.u.c.LinkedBlockingQueue.take(LinkedBlockingQueue.java:399)
at org.apache.tomcat.util.threads.TaskQueue.take
    (TaskQueue.java:104)
at org.apache.tomcat.util.threads.TaskQueue.take
    (TaskQueue.java:32)
at j.u.c.ThreadPoolExecutor.getTask(ThreadPoolExecutor.java:947)
at j.u.c.ThreadPoolExecutor$Worker.run
    (ThreadPoolExecutor.java:907)
at j.l.Thread.run(Thread.java:680)

```

"F5066395-F9D9-4F98-A6727B13979B46FA"

```

daemon
prio=5
tid=7f846acf8000
nid=0x11746b000
waiting on condition [11746a000]
j.l.Thread.State: WAITING (parking)
at sun.misc.Unsafe.park(Native Method)
- parking to wait for <7a0edb9f8>
    (a j.u.c.locks.AbstractQueuedSynchronizer$ConditionObject)
at j.u.c.locks.LockSupport.park(LockSupport.java:158)
at j.u.c.locks.AbstractQueuedSynchronizer$ConditionObject.await
    (AbstractQueuedSynchronizer.java:1987)
at j.u.c.LinkedBlockingQueue.take(LinkedBlockingQueue.java:399)
at org.apache.tomcat.util.threads.TaskQueue.take
    (TaskQueue.java:104)
at org.apache.tomcat.util.threads.TaskQueue.take
    (TaskQueue.java:32)
at j.u.c.ThreadPoolExecutor.getTask(ThreadPoolExecutor.java:947)
at j.u.c.ThreadPoolExecutor$Worker.run
    (ThreadPoolExecutor.java:907)
at j.l.Thread.run(Thread.java:680)

```

"DAB951C0-F38D-44C9-8CB87B3145FD8360"

```

daemon
prio=5
tid=7f846ab2e800

```

```

nid=0x116ace000
waiting on condition [116acd000]
j.l.Thread.State: WAITING (parking)
at sun.misc.Unsafe.park(Native Method)
- parking to wait for <7a0edb9f8>
(a j.u.c.locks.AbstractQueuedSynchronizer$ConditionObject)
at j.u.c.locks.LockSupport.park(LockSupport.java:158)
at j.u.c.locks.AbstractQueuedSynchronizer$ConditionObject.await
(AbstractQueuedSynchronizer.java:1987)
at j.u.c.LinkedBlockingQueue.take(LinkedBlockingQueue.java:399)
at org.apache.tomcat.util.threads.TaskQueue.take
(TaskQueue.java:104)
at org.apache.tomcat.util.threads.TaskQueue.take
(TaskQueue.java:32)
at j.u.c.ThreadPoolExecutor.getTask(ThreadPoolExecutor.java:947)
at j.u.c.ThreadPoolExecutor$Worker.run
(ThreadPoolExecutor.java:907)
at j.l.Thread.run(Thread.java:680)

"org.apache.commons.vfs.cache.SoftRefFilesCache$SoftRefReleaseThread"
daemon
prio=5
tid=7f8463239000
nid=0x1176b0000
in Object.wait() [1176af000]
j.l.Thread.State: TIMED_WAITING (on object monitor)
at j.l.Object.wait(Native Method)
- waiting on <7dfbb99d8> (a j.l.ref.ReferenceQueue$Lock)
at j.l.ref.ReferenceQueue.remove(ReferenceQueue.java:118)
- locked <7dfbb99d8> (a j.l.ref.ReferenceQueue$Lock)
at SoftRefFilesCache$SoftRefReleaseThread.run
(SoftRefFilesCache.java:79)

"35B3A1DD-CA13-41DB-95902A6A2204FADE"
daemon
prio=5
tid=7f8466ad6800
nid=0x116991000
waiting on condition [116990000]
j.l.Thread.State: WAITING (parking)
at sun.misc.Unsafe.park(Native Method)
- parking to wait for <7a0edb9f8>
(a j.u.c.locks.AbstractQueuedSynchronizer$ConditionObject)
at j.u.c.locks.LockSupport.park(LockSupport.java:158)
at j.u.c.locks.AbstractQueuedSynchronizer$ConditionObject.await
(AbstractQueuedSynchronizer.java:1987)
at j.u.c.LinkedBlockingQueue.take(LinkedBlockingQueue.java:399)
at org.apache.tomcat.util.threads.TaskQueue.take

```

```

        (TaskQueue.java:104)
at org.apache.tomcat.util.threads.TaskQueue.take
        (TaskQueue.java:32)
at j.u.c.ThreadPoolExecutor.getTask(ThreadPoolExecutor.java:947)
at j.u.c.ThreadPoolExecutor$Worker.run
        (ThreadPoolExecutor.java:907)
at j.l.Thread.run(Thread.java:680)

"B8DBA591-39F2-4B41-933934D9DE84D963"
        daemon
        prio=5
        tid=7f8466ad6000
        nid=0x11688e000
        waiting on condition [11688d000]
j.l.Thread.State: WAITING (parking)
at sun.misc.Unsafe.park(Native Method)
- parking to wait for <7a0edb9f8>
        (a j.u.c.locks.AbstractQueuedSynchronizer$ConditionObject)
at j.u.c.locks.LockSupport.park(LockSupport.java:158)
at j.u.c.locks.AbstractQueuedSynchronizer$ConditionObject.await
        (AbstractQueuedSynchronizer.java:1987)
at j.u.c.LinkedBlockingQueue.take(LinkedBlockingQueue.java:399)
at org.apache.tomcat.util.threads.TaskQueue.take
        (TaskQueue.java:104)
at org.apache.tomcat.util.threads.TaskQueue.take
        (TaskQueue.java:32)
at j.u.c.ThreadPoolExecutor.getTask(ThreadPoolExecutor.java:947)
at j.u.c.ThreadPoolExecutor$Worker.run
        (ThreadPoolExecutor.java:907)
at j.l.Thread.run(Thread.java:680)

""ajp-bio-8009"-AsyncTimeout"
        daemon
        prio=5
        tid=7f84688a3800
        nid=0x116707000
        waiting on condition [116706000]
j.l.Thread.State: TIMED_WAITING (sleeping)
at j.l.Thread.sleep(Native Method)
at org.apache.tomcat.util.net.JIoEndpoint$AsyncTimeout.run
        (JIoEndpoint.java:143)
at j.l.Thread.run(Thread.java:680)

""ajp-bio-8009"-Acceptor-0"
        daemon
        prio=5
        tid=7f84689ce800
        nid=0x116604000

```

```

        runnable [116603000]
j.l.Thread.State: RUNNABLE
at java.net.PlainSocketImpl.socketAccept(Native Method)
at java.net.PlainSocketImpl.accept(PlainSocketImpl.java:408)
- locked <7dfbba628> (a java.net.SocksSocketImpl)
at java.net.ServerSocket.implAccept(ServerSocket.java:462)
at java.net.ServerSocket.accept(ServerSocket.java:430)
at tomcat.util.net.DefaultServerSocketFactory.acceptSocket
    (DefaultServerSocketFactory.java:59)
at org.apache.tomcat.util.net.JIoEndpoint$Acceptor.run
    (JIoEndpoint.java:211)
at j.l.Thread.run(Thread.java:680)

""http-bio-8080"-AsyncTimeout"
daemon
prio=5
tid=7f8468d19000
nid=0x116501000
waiting on condition [116500000]
j.l.Thread.State: TIMED_WAITING (sleeping)
at j.l.Thread.sleep(Native Method)
at org.apache.tomcat.util.net.JIoEndpoint$AsyncTimeout.run
    (JIoEndpoint.java:143)
at j.l.Thread.run(Thread.java:680)

""http-bio-8080"-Acceptor-0"
daemon
prio=5
tid=7f84689cf800
nid=0x1163fe000
runnable [1163fd000]
j.l.Thread.State: RUNNABLE
at java.net.PlainSocketImpl.socketAccept(Native Method)
at java.net.PlainSocketImpl.accept(PlainSocketImpl.java:408)
- locked <7dfbb9d28> (a java.net.SocksSocketImpl)
at java.net.ServerSocket.implAccept(ServerSocket.java:462)
at java.net.ServerSocket.accept(ServerSocket.java:430)
at apache.tomcat.util.net.DefaultServerSocketFactory.acceptSocket
    (DefaultServerSocketFactory.java:59)
at org.apache.tomcat.util.net.JIoEndpoint$Acceptor.run
    (JIoEndpoint.java:211)
at j.l.Thread.run(Thread.java:680)

"ContainerBackgroundProcessor[StandardEngine[Catalina]]"
daemon
prio=5
tid=7f8469fe4000
nid=0x1160a4000

```

```

        waiting on condition [1160a3000]
j.l.Thread.State: TIMED_WAITING (sleeping)
at j.l.Thread.sleep(Native Method)
at catalina.core.ContainerBase$ContainerBackgroundProcessor.run
(ContainerBase.java:1369)
at j.l.Thread.run(Thread.java:680)

"CFQUERY Backgrounder"
daemon
prio=1
tid=7f846503a800
nid=0x115f1b000
waiting on condition [115f1a000]
j.l.Thread.State: TIMED_WAITING (sleeping)
at j.l.Thread.sleep(Native Method)
at com.naryx.tagfusion.cfm.sql.platform.java.queryBatchServer.run
(Unknown Source)

"BlueDragon MailSender"
daemon
prio=1
tid=7f846680a000
nid=0x1162fb000
in Object.wait() [1162fa000]
j.l.Thread.State: TIMED_WAITING (on object monitor)
at j.l.Object.wait(Native Method)
- waiting on <7a0edbf80> (a j.l.Object)
at com.bluedragon.platform.java.smtp.OutgoingMailServer.getMail
(Unknown Source)
- locked <7a0edbf80> (a j.l.Object)
at com.bluedragon.platform.java.smtp.OutgoingMailServer.run
(Unknown Source)

"OpenBDAlarmManager"
daemon
prio=5
tid=7f84652df800
nid=0x1161f8000
waiting on condition [1161f7000]
j.l.Thread.State: TIMED_WAITING (sleeping)
at j.l.Thread.sleep(Native Method)
at com.nary.util.AlarmManager.run(Unknown Source)
at j.l.Thread.run(Thread.java:680)

"BlueDragon MailSender"
daemon
prio=1
tid=7f84651b6000

```

```

nid=0x115d4e000
in Object.wait() [115d4d000]
j.l.Thread.State: WAITING (on object monitor)
at j.l.Object.wait(Native Method)
- waiting on <7a0edc008> (a j.l.Object)
at j.l.Object.wait(Object.java:485)
at com.naryx.tagfusion.cfm.mail.mailServer$MailSender.getMail
(Unknown Source)
- locked <7a0edc008> (a j.l.Object)
at com.naryx.tagfusion.cfm.mail.mailServer$MailSender.run
(Unknown Source)

"CFQUERY Backgrounder"
daemon
prio=1
tid=7f846699e800
nid=0x115c4b000
waiting on condition [115c4a000]
j.l.Thread.State: TIMED_WAITING (sleeping)
at j.l.Thread.sleep(Native Method)
at com.naryx.tagfusion.cfm.sql.platform.java.queryBatchServer.run
(Unknown Source)

"BlueDragon AlarmManager"
daemon
prio=5
tid=7f8469acc800
nid=0x1156f5000
waiting on condition [1156f4000]
j.l.Thread.State: TIMED_WAITING (sleeping)
at j.l.Thread.sleep(Native Method)
at com.naryx.util.AlarmManager.run(Unknown Source)

"BlueDragon MailSender"
daemon
prio=1
tid=7f846880a000
nid=0x1159fe000
in Object.wait() [1159fd000]
j.l.Thread.State: WAITING (on object monitor)
at j.l.Object.wait(Native Method)
- waiting on <7dfbba4f0> (a j.l.Object)
at j.l.Object.wait(Object.java:485)
at com.naryx.tagfusion.cfm.mail.mailServer$MailSender.getMail
(Unknown Source)
- locked <7dfbba4f0> (a j.l.Object)
at com.naryx.tagfusion.cfm.mail.mailServer$MailSender.run
(Unknown Source)

```

```
"BlueDragon CFQUERY Backgrounder"
    daemon
    prio=1
    tid=7f8469964800
    nid=0x1158fb000
    in Object.wait() [1158fa000]
j.l.Thread.State: TIMED_WAITING (on object monitor)
at j.l.Object.wait(Native Method)
- waiting on <7dfbbada8>
    (a com.naryx.tagfusion.cfm.sql.queryBatchServer)
at com.naryx.tagfusion.cfm.sql.queryBatchServer.run
    (Unknown Source)
- locked <7dfbbada8>
    (a com.naryx.tagfusion.cfm.sql.queryBatchServer)

"BlueDragon AlarmManager"
    daemon
    prio=5
    tid=7f846995f800
    nid=0x1157f8000
    waiting on condition [1157f7000]
j.l.Thread.State: TIMED_WAITING (sleeping)
at j.l.Thread.sleep(Native Method)
at com.nary.util.AlarmManager.run(Unknown Source)

"BlueDragon MailSender"
    daemon
    prio=1
    tid=7f84669a5000
    nid=0x115311000
    in Object.wait() [115310000]
j.l.Thread.State: WAITING (on object monitor)
at j.l.Object.wait(Native Method)
- waiting on <7dfbba5b0> (a j.l.Object)
at j.l.Object.wait(Object.java:485)
at com.naryx.tagfusion.cfm.mail.mailServer$MailSender.getMail
    (Unknown Source)
- locked <7dfbba5b0> (a j.l.Object)
at com.naryx.tagfusion.cfm.mail.mailServer$MailSender.run
    (Unknown Source)

"CFQUERY Backgrounder"
    daemon
    prio=1
    tid=7f84669a4000
    nid=0x1149c5000
    waiting on condition [1149c4000]
```

```
j.l.Thread.State: TIMED_WAITING (sleeping)
at j.l.Thread.sleep(Native Method)
at com.naryx.tagfusion.cfm.sql.platform.java.queryBatchServer.run
(Unknown Source)

"BlueDragon AlarmManager"
daemon
prio=5
tid=7f84669a3800
nid=0x1141a7000
waiting on condition [1141a6000]
j.l.Thread.State: TIMED_WAITING (sleeping)
at j.l.Thread.sleep(Native Method)
at com.nary.util.AlarmManager.run(Unknown Source)

"CFQUERY Backgrounder"
daemon
prio=1
tid=7f84651fb000
nid=0x115b48000
waiting on condition [115b47000]
j.l.Thread.State: TIMED_WAITING (sleeping)
at j.l.Thread.sleep(Native Method)
at com.naryx.tagfusion.cfm.sql.platform.java.queryBatchServer.run
(Unknown Source)

"BlueDragon MailSender"
daemon
prio=1
tid=7f8463bba800
nid=0x115151000
in Object.wait() [115150000]
j.l.Thread.State: TIMED_WAITING (on object monitor)
at j.l.Object.wait(Native Method)
- waiting on <7dfbb9ac8> (a j.l.Object)
at com.bluedragon.platform.java.smtp.OutgoingMailServer.getMail
(Unknown Source)
- locked <7dfbb9ac8> (a j.l.Object)
at com.bluedragon.platform.java.smtp.OutgoingMailServer.run
(Unknown Source)

"OpenBDAlarmManager"
daemon
prio=5
tid=7f8466bca800
nid=0x11504e000
waiting on condition [11504d000]
j.l.Thread.State: TIMED_WAITING (sleeping)
```

```
at j.l.Thread.sleep(Native Method)
at com.nary.util.AlarmManager.run(Unknown Source)
at j.l.Thread.run(Thread.java:680)

"CFQUERY Backgrounder"
    daemon
    prio=1
    tid=7f8463281800
    nid=0x114734000
    waiting on condition [114733000]
j.l.Thread.State: TIMED_WAITING (sleeping)
at j.l.Thread.sleep(Native Method)
at com.naryx.tagfusion.cfm.sql.platform.java.queryBatchServer.run
(Unknown Source)

"AWT-AppKit"
    daemon
    prio=5
    tid=7f846325e800
    nid=0x7fff70b3e960
    runnable [00000000]
j.l.Thread.State: RUNNABLE

"BlueDragon MailSender"
    daemon
    prio=1
    tid=7f846325d800
    nid=0x1143ad000
    in Object.wait() [1143ac000]
j.l.Thread.State: TIMED_WAITING (on object monitor)
at j.l.Object.wait(Native Method)
- waiting on <7dfbb9a50> (a j.l.Object)
at com.bluedragon.platform.java.smtp.OutgoingMailServer.getMail
(Unknown Source)
- locked <7dfbb9a50> (a j.l.Object)
at com.bluedragon.platform.java.smtp.OutgoingMailServer.run
(Unknown Source)

"OpenBDAlarmManager"
    daemon
    prio=5
    tid=7f846392a800
    nid=0x1142aa000
    waiting on condition [1142a9000]
j.l.Thread.State: TIMED_WAITING (sleeping)
at j.l.Thread.sleep(Native Method)
at com.nary.util.AlarmManager.run(Unknown Source)
at j.l.Thread.run(Thread.java:680)
```

```
"Poller SunPKCS11-Darwin"
    daemon
    prio=1
    tid=7f84638cf800
    nid=0x1140a4000
    runnable [1140a3000]
j.l.Thread.State: RUNNABLE
at sun.security.pkcs11.wrapper.PKCS11.C_GetSlotInfo
(Native Method)
at sun.security.pkcs11.SunPKCS11.initToken(SunPKCS11.java:767)
at sun.security.pkcs11.SunPKCS11.access$100(SunPKCS11.java:42)
at sun.security.pkcs11.SunPKCS11$TokenPoller.run
(SunPKCS11.java:700)
at j.l.Thread.run(Thread.java:680)

"GC Daemon"
    daemon
    prio=2
    tid=7f8463b14800
    nid=0x113d12000
    in Object.wait() [113d11000]
j.l.Thread.State: TIMED_WAITING (on object monitor)
at j.l.Object.wait(Native Method)
- waiting on <7dfbb9780> (a sun.misc.GC$LatencyLock)
at sun.misc.GC$Daemon.run(GC.java:100)
- locked <7dfbb9780> (a sun.misc.GC$LatencyLock)

"Low Memory Detector"
    daemon
    prio=5
    tid=7f8463094000
    nid=0x113538000
    runnable [00000000]
j.l.Thread.State: RUNNABLE

"C2 CompilerThread1"
    daemon
    prio=9
    tid=7f8463093800
    nid=0x113435000
    waiting on condition [00000000]
j.l.Thread.State: RUNNABLE

"C2 CompilerThread0"
    daemon
    prio=9
    tid=7f8463092800
```

```
    nid=0x113332000
    waiting on condition [00000000]
j.l.Thread.State: RUNNABLE

"Signal Dispatcher"
    daemon
    prio=9
    tid=7f8463092000
    nid=0x11322f000
    runnable [00000000]
j.l.Thread.State: RUNNABLE

"Surrogate Locker Thread (Concurrent GC)"
    daemon
    prio=5
    tid=7f8463091000
    nid=0x11312c000
    waiting on condition [00000000]
j.l.Thread.State: RUNNABLE

"Finalizer"
    daemon
    prio=8
    tid=7f846380b800
    nid=0x112e68000
    in Object.wait() [112e67000]
j.l.Thread.State: WAITING (on object monitor)
at j.l.Object.wait(Native Method)
- waiting on <7dfa00cf8> (a j.l.ref.ReferenceQueue$Lock)
at j.l.ref.ReferenceQueue.remove(ReferenceQueue.java:118)
- locked <7dfa00cf8> (a j.l.ref.ReferenceQueue$Lock)
at j.l.ref.ReferenceQueue.remove(ReferenceQueue.java:134)
at j.l.ref.Finalizer$FinalizerThread.run(Finalizer.java:159)

"Reference Handler"
    daemon
    prio=10
    tid=7f846380b000
    nid=0x112d65000
    in Object.wait() [112d64000]
j.l.Thread.State: WAITING (on object monitor)
at j.l.Object.wait(Native Method)
- waiting on <7dfa00bc0> (a j.l.ref.Reference$Lock)
at j.l.Object.wait(Object.java:485)
at j.l.ref.Reference$ReferenceHandler.run(Reference.java:116)
- locked <7dfa00bc0> (a j.l.ref.Reference$Lock)

"main"
```

```

prio=5
tid=7f8463001000
nid=0x10b110000
runnable [10b10e000]
j.l.Thread.State: RUNNABLE
at java.net.PlainSocketImpl.socketAccept(Native Method)
at java.net.PlainSocketImpl.accept(PlainSocketImpl.java:408)
- locked <7a1061e30> (a java.net.SocksSocketImpl)
at java.net.ServerSocket.implAccept(ServerSocket.java:462)
at java.net.ServerSocket.accept(ServerSocket.java:430)
at org.apache.catalina.core.StandardServer.await
    (StandardServer.java:447)
at org.apache.catalina.startup.Catalina.await(Catalina.java:707)
at org.apache.catalina.startup.Catalina.start(Catalina.java:653)
at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
at sun.reflect.NativeMethodAccessorImpl.invoke
    (NativeMethodAccessorImpl.java:39)
at sun.reflect.DelegatingMethodAccessorImpl.invoke
    (DelegatingMethodAccessorImpl.java:25)
at j.l.reflect.Method.invoke(Method.java:597)
at org.apache.catalina.startup.Bootstrap.start
    (Bootstrap.java:303)
at org.apache.catalina.startup.Bootstrap.main(Bootstrap.java:431)

"VM Thread"
prio=9
tid=7f8463806000
nid=0x112c62000
runnable

"Gang worker#0 (Parallel GC Threads)"
prio=9
tid=7f8463002800
nid=0x10e44a000
runnable

"Gang worker#1 (Parallel GC Threads)"
prio=9
tid=7f8463003000
nid=0x10e54d000
runnable

"Gang worker#2 (Parallel GC Threads)"
prio=9
tid=7f8463003800
nid=0x10e650000
runnable

```

```
"Gang worker#3 (Parallel GC Threads)"
    prio=9
    tid=7f8463004000
    nid=0x10e753000
    runnable

"Concurrent Mark-Sweep GC Thread"
    prio=9
    tid=7f846307f800
    nid=0x11249a000
    runnable

"VM Periodic Task Thread"
    prio=10
    tid=7f84630a5800
    nid=0x11363b000
    waiting on condition

"Exception Catcher Thread"
    prio=10
    tid=7f8463001800
    nid=0x10b272000
    runnable
```

Kind regards from Crete

Kirk Pepperdine

We now have a [Facebook page](#) for the Java Specialists. If you've enjoyed reading this newsletter, please take a moment to "like" our group.

Issue 194 - trySynchronize

Author: Dr. Heinz M. Kabutz

Date: 2011-08-27

Category: Concurrency

Java Versions: Sun Java 6

Abstract:

Did you know that it possible to "try" to synchronize a monitor? In this newsletter we demonstrate how this can be used to avoid deadlocks and to keep the wine coming.

Welcome to the 194th issue of **The Java(tm) Specialists' Newsletter**. This coming Monday we are running our first **Java Specialists Symposium** here on Crete. 40 Java experts and enthusiasts from all around the world from as far afield as Canada are honouring our little island with their presence. We will spend 4 intense days discussing Java with the theme "Making Java Fun Again". Not just will we talk Java, but we will walk the mountains and swim the seas. We will try to record as much as possible of the discussions, but nothing will replace coming down to Crete yourself.

This open spaces conference was first named the Java Specialist Roundup. But since we are in Greece, John Kostaras suggested "Symposium" would be a better name. As in previous newsletters, a quick Greek lesson for the Philistines amongst us: The word symposium is made up out of two words. "Sym" comes from "syn" and means "together", as you will find in "symphony", "symbiosis", etc. The word "posium" comes from "poto", meaning drink. Thus the literal and historical meaning is a drinking party for philosophers.

trySynchronize

Crete produces so much food, that it is impossible to eat and drink it all. Since the transport system does not work well, we often have to simply throw our produce away. For example, we planted water melons and even after harvesting several tons, still had about 50 that we kept at home. Water melon becomes delicious juice in a blender, pips and all. You can even throw a bit of the rind in. Even en route to our conference hotel, you will drive past fields with hundreds of abandoned water melons.

Another excess is the village wine. It is an interesting drink that takes some getting used to. It is more like dry sherry than red wine. For the South Africans amongst our readership, think Sedgwick's Old Brown. The locals have so much that they simply cannot drink it all, so we are given an abundant supply, far more than would be good for me. I have about 30 liters that need to be finished next week at the Symposium. My friends are harvesting in September and I will probably be inundated with last year's stock!

This brought me to this idea. Instead of the classical "eating philosophers" we will have

"drinking philosophers". And instead of forks, we have two cups, because around this table you have to drink with both hands.

The lock for our "symposium" is the class Krasi, Greek for "wine".

```
public class Krasi { }
```

Our first "Thinker" has a deadlock, because if everyone picks up the right cup at the same time, then it forms a ring and they cannot pick up the left cup. Thus they end up in a state of limbo and the symposium deadlocks.

```
import java.util.concurrent.*;

public class Thinker implements Callable<String> {
    private final int id;
    private final Krasi left, right;

    public Thinker(int id, Krasi left, Krasi right) {
        this.id = id;
        this.left = left;
        this.right = right;
    }

    public String call() throws Exception {
        for (int i = 0; i < 1000; i++) {
            drink();
            think();
        }
        return "Java is fun";
    }

    public void drink() {
        synchronized (left) {
            synchronized (right) {
                System.out.printf("(%d) Drinking%n", id);
            }
        }
    }

    public void think() {
```

```

        System.out.printf("(%d) Thinking%n", id);
    }
}

```

It is fairly easy to prove that the system can deadlock. We simply construct a bunch of Thinkers and make their locks form a circle. In order to not cause an early escape of "this" by starting threads in the constructor, we only start the symposium once it has been constructed.

```

import java.util.concurrent.*;

public class Symposium {
    private final Krasi[] cups;
    private final Thinker[] thinkers;

    public Symposium(int delegates) {
        cups = new Krasi[delegates];
        thinkers = new Thinker[delegates];
        for (int i = 0; i < cups.length; i++) {
            cups[i] = new Krasi();
        }
        for (int i = 0; i < delegates; i++) {
            Krasi left = cups[i];
            Krasi right = cups[(i + 1) % delegates];
            thinkers[i] = new Thinker(i, left, right);
        }
    }

    public void run() throws InterruptedException {
        // do this after we created the symposium, so that we do not
        // let the reference to the Symposium escape.
        ExecutorService exec = Executors.newCachedThreadPool();
        CompletionService<String> results =
            new ExecutorCompletionService<String>(exec);
        for (Thinker thinker : thinkers) {
            results.submit(thinker);
        }
        System.out.println("Waiting for results");
        for (int i = 0; i < thinkers.length; i++) {
            try {
                System.out.println(results.take().get());
            } catch (ExecutionException e) {
                e.getCause().printStackTrace();
            }
        }
    }
}

```

```

        }
    }
    exec.shutdown();
}
}

```

We can create a Symposium with 5 thinkers and very quickly we will see that there is a deadlock.

```

public class JavaSpecialistsSymposium2011Crete {
    public static void main(String[] args)
        throws InterruptedException {
        Symposium symposium = new Symposium(5);
        symposium.run();
    }
}

```

Here is some output we might see:

```

(0) Drinking
(0) Thinking
Waiting for results
(2) Drinking
(2) Thinking
(2) Drinking
(2) Thinking

```

The jstack program also verifies that we have a deadlock:

```

Found one Java-level deadlock:
=====
"pool-1-thread-5":

```

```

waiting to lock monitor 10086e908 (object 7f319c300, a Krasi),
which is held by "pool-1-thread-1"
"pool-1-thread-1":
waiting to lock monitor 10080d360 (object 7f319c310, a Krasi),
which is held by "pool-1-thread-2"
"pool-1-thread-2":
waiting to lock monitor 10080d2b8 (object 7f319c320, a Krasi),
which is held by "pool-1-thread-3"
"pool-1-thread-3":
waiting to lock monitor 10086d408 (object 7f319c330, a Krasi),
which is held by "pool-1-thread-4"
"pool-1-thread-4":
waiting to lock monitor 10086d360 (object 7f319c340, a Krasi),
which is held by "pool-1-thread-5"

```

There are several ways of avoiding deadlocks. One is to do lock ordering, where we guarantee to always synchronize the same lock first. The last thinker would thus first lock right and then left, whereas all the others would be the other way round. Another approach is to use Java 5 locks which have a tryLock() method. Effectively you could do something like:

```

while (true) {
    if (Thread.interrupted()) throw new InterruptedException();
    if (left.tryLock()) {
        try {
            if (right.tryLock()) {
                try {
                    System.out.printf("(%d) Drinking%n", id);
                    return;
                } finally {
                    right.unlock();
                }
            }
        } finally {
            left.unlock();
        }
    }
    if (timeoutExceeded()) throw new TimeoutException();
    sleepRandomTime();
}

```

You have probably seen code like that before. However, did you know that it is also possible

to *try to synchronize*?

In a recent email, one of my youngest readers, 17 year old Mr S.Perlov from the Ukraine, suggested that I tell you about the class sun.misc.Unsafe. Up to now I have avoided writing about it, as it is a class that should be avoided. Here are two reasons: #1 it is "unsafe" and lets us do all the nasty things that we had in C, such as pointer arithmetic or modifying memory directly. #2 it is a sun.misc.* class. You do not know when that might be renamed to oracle.misc.Unsafe or whether you will even run your program on a Sun JVM. By binding yourself to a specific implementation of the JVM, you are limiting the application of your code.

Two reasons to *not* use Unsafe. I have personally never used Unsafe in production code. Some experts do use it to write directly to memory. Dangerous stuff!

Unsafe allows us to manually lock and unlock monitors, with the monitorEnter() and monitorExit() methods. We can also "try to lock" with the tryMonitorEnter() method. Here is a wrapper class that we can use to do the dirty work for us:

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

public class MonitorUtils {
    private static Unsafe unsafe = getUnsafe();

    public static boolean trySynchronize(Object monitor) {
        return unsafe.tryMonitorEnter(monitor);
    }

    public static void unsynchronize(Object monitor) {
        unsafe.monitorExit(monitor);
    }

    private static Unsafe getUnsafe() {
        try {
            for (Field field : Unsafe.class.getDeclaredFields()) {
                if (Modifier.isStatic(field.getModifiers())) {
                    if (field.getType() == Unsafe.class) {
                        field.setAccessible(true);
                        return (Unsafe) field.get(null);
                    }
                }
            }
        } catch (Exception e) {
            throw new IllegalStateException("Unsafe field not found");
        }
    }
}
```

```
        } catch (Exception e) {
            throw new IllegalStateException(
                "Could not initialize unsafe", e);
        }
    }
}
```

We can now change our "drink()" method to

```
while (true) {
    if (Thread.interrupted()) throw new InterruptedException();
    if (MonitorUtils.trySynchronize(left)) {
        try {
            if (MonitorUtils.trySynchronize(right)) {
                try {
                    System.out.printf("(%d) Drinking%n", id);
                    return;
                } finally {
                    MonitorUtils.unsyncrhonize(right);
                }
            }
        } finally {
            MonitorUtils.unsyncrhonize(left);
        }
    }
    if (timeoutExceeded()) throw new TimeoutException();
    sleepRandomTime();
}
```

Why use this when we have `Lock.tryLock()`? You might want to change code that is already implemented with monitor locking. Instead of modifying everything to use `Lock`, you could get the same functionality with this `trySynchronize()`.

Even better is to avoid locking yourself by using thread-safe classes. Sadly this is not always an option.

Kind regards from Crete

Heinz

We now have a [Facebook page](#) for the Java Specialists. If you've enjoyed reading this newsletter, please take a moment to "like" our group.

Issue 192b - How Does "this" Escape?

Author: Dr. Heinz M. Kabutz

Date: 2011-06-01

Category: Concurrency

Java Versions: Java 5.0-7.0

Abstract:

A quick follow-up to the previous newsletter, to show how the ThisEscape class is compiled, causing the "this" pointer to leak.

When I posted the [previous newsletter](#), I did not explicitly show how the "this" reference escapes implicitly. I have been showing these mysteries in my Java courses since 1999 and I forgot that there might be some mortals who have never seen a decompiled inner class ;-)

How Does "this" Escape?

When you construct a inner class within a non-static context, such as in a non-static method, a constructor or an initializer block, the class *always* has a pointer to the outer object.

Here again is our class ThisEscape:

```
import java.util.*;

public class ThisEscape {
    private final int num;

    public ThisEscape(EventSource source) {
        source.registerListener(
            new EventListener() {
                public void onEvent(Event e) {
                    doSomething(e);
                }
            });
        num = 42;
    }

    private void doSomething(Event e) {
        if (num != 42) {
            System.out.println("Race condition detected at " +

```

```
    new Date());  
}  
}  
}
```

When it gets compiled, javac generates two classes. The outer class looks like this:

```
import java.util.*;  
  
public class ThisEscape {  
    private final int num;  
  
    public ThisEscape(EventSource source) {  
        source.registerListener(new ThisEscape$1(this));  
        num = 42;  
    }  
  
    private void doSomething(Event e) {  
        if (num != 42)  
            System.out.println(  
                "Race condition detected at " + new Date());  
    }  
  
    static void access$000(ThisEscape _this, Event event) {  
        _this.doSomething(event);  
    }  
}
```

Note how the compiler added the static, package access method `access$000()`. Note also how it passed a pointer to `this` into the constructor for the anonymous inner class `ThisEscape$1`.

Next we have the anonymous inner class. It is package access, with a package access constructor. Internally it maintains a reference to the ThisEscape object. Note: the anonymous class sets the `this$0` field *before* calling `super()`. This is the only place where this is allowed in Java.

```
class ThisEscape$1 implements EventListener {
    final ThisEscape this$0;

    ThisEscape$1(ThisEscape thisescape) {
        this$0 = thisescape;
        super();
    }

    public void onEvent(Event e) {
        ThisEscape.access$000(this$0, e);
    }
}
```

Hopefully it is now clearer how the "this" pointer is implicitly leaked.

Kind regards from Crete

Heinz

Issue 192 - Implicit Escape of "this"

Author: Dr. Heinz M. Kabutz

Date: 2011-05-31

Category: Concurrency

Java Versions: Java 5.0-7.0

Abstract:

We should never allow references to our objects to escape before all the final fields have been set, otherwise we can cause race conditions. In this newsletter we explore how this is possible to do.

Welcome to the 192nd issue of **The Java(tm) Specialists' Newsletter**, which I am writing underneath some plane trees on the 1821 square of Chania. Last week Kirk Pepperdine ran his performance course at our conference room, so we had our hands full showing the students Cretan hospitality. I managed to convince them to join me on a short walk to a beach that no one I knew had ever been to. After about an hour of scrambling down a ravine, over boulders and squeezing through dense vegetation, we realised that the light was fading and decided to rather head back again. It was a great adventure for my three kids, aged 4 to 12. I am going back this Friday with **brother-in-law Cam** to scout the place out properly.

We are running an **open spaces conference here in Crete** at the end of August. Please let us know if you are interested in attending. We have space for about another five people. The conference is free, but of course you need to pay your own transport here.

Also, our **conference venue is now available for hire**. Crete is the perfect place for a European "bosberaad", a South African word meaning "bush summit". Companies often go "into the bush" (literally) to plan for the future and to train their employees. It's very relaxing to be away in nature and is a great enabler for learning.

Implicit Escape of "this"

A few weeks ago, one of my friends sent me a question about section 3.2 of **Java Concurrency in Practice**. He could not understand how a method in Class A with a reference to an inner class in class B could obtain a reference to the outer class B. My friend thought that Brian might have meant that you could access the object via reflection.

For those of you who do not own the book yet, here is Brian's statement from Section 3.2 (this is one book that you really want to own though):

Java Concurrency in Practice, Section 3.2: A final [heinz: Brian means "final" as in "last", not as in Java **final] mechanism by which an object or its internal state can be published is**

to publish an inner class instance, as shown in ThisEscape in Listing 3.7. When ThisEscape publishes the EventListener, it implicitly publishes the enclosing ThisEscape instance as well, because inner class instances contain a hidden reference to the enclosing instance.

```
//-----Listing 3.7-----
public class ThisEscape {
    public ThisEscape(EventSource source) {
        source.registerListener(
            new EventListener() {
                public void onEvent(Event e) {
                    doSomething(e);
                }
            });
    }
}
```

Since my good friend was puzzling about this, I decided to expand the ThisEscape class with its own fields and doSomething() method and some additional code to demonstrate a possible data race.

In my class, I have a final field num that is initialized in the constructor. However, before it is set to 42, we register an anonymous inner class, which also leaks a pointer to the enclosing object.

```
import java.util.*;

public class ThisEscape {
    private final int num;

    public ThisEscape(EventSource source) {
        source.registerListener(
            new EventListener() {
                public void onEvent(Event e) {
                    doSomething(e);
                }
            });
        num = 42;
    }

    private void doSomething(Event e) {
```

```

if (num != 42) {
    System.out.println("Race condition detected at " +
        new Date());
}
}
}

```

In my example, the Event and EventListener classes are kept as simple as possible:

```

public class Event { }
public interface EventListener {
    public void onEvent(Event e);
}

```

The EventSource is more complicated. In our case it is a Thread that repeatedly sends events to its latest listeners. Since we are trying to produce the race condition, we only ever send an event to a listener once. Thus the registerListener() method appends it to the end of the listeners queue and it is then taken off by the take() method call within the run() method of the thread.

```

import java.util.concurrent.*;

public class EventSource extends Thread {
    private final BlockingQueue<EventListener> listeners =
        new LinkedBlockingQueue<EventListener>();

    public void run() {
        while (true) {
            try {
                listeners.take().onEvent(null);
            } catch (InterruptedException e) {
                break;
            }
        }
    }

    public void registerListener(EventListener eventListener) {

```

```

        listeners.add(eventListener);
    }
}

```

All that is left is to construct a lot of ThisEscape objects in a row and watch the wheels come off:

```

public class ThisEscapeTest {
    public static void main(String[] args) {
        EventSource es = new EventSource();
        es.start();
        while(true) {
            new ThisEscape(es);
        }
    }
}

```

On my machine, I get race conditions immediately. What is interesting is that they stopped for a while and then started up again. However, I think this was probably a capacity problem with using a queue for the listeners. It would maybe be better to have a queue of capacity 1, in which case we could use an AtomicReference instead:

```

import java.util.concurrent.atomic.*;

public class EventSource2 extends Thread {
    private final AtomicReference<EventListener> listeners =
        new AtomicReference<EventListener>();

    public void run() {
        while (true) {
            EventListener listener = listeners.getAndSet(null);
            if (listener != null) {
                listener.onEvent(null);
            }
        }
    }
}

```

```
public void registerListener(EventListener eventListener) {  
    listeners.set(eventListener);  
}  
}
```

Now we see the race conditions immediately and they do not stop when the queue gets too long.

Our lesson to learn is: We should never allow references to our objects to escape before all the final fields have been set, otherwise we can cause race conditions.

Heinz

P.S. One of our readers, Ulf, decided to log a bug against this behavior. He thinks that the compiler should detect and warn about such race conditions. Feel free to [vote on this issue](#) if you like.

Issue 191 - Delaying Garbage Collection Costs

Author: Dr. Heinz M. Kabutz

Date: 2011-04-27

Category: Performance

Java Versions: Java 6

Abstract:

The garbage collector can cause our program to slow down at inappropriate times. In this newsletter we look at how we can delay the GC to never run and thus produce reliable results. Then, instead of actually collecting the dead objects, we simply shut down and start again.

Welcome to the 191st issue of **The Java(tm) Specialists' Newsletter**, sent to you from the stunning Island of Crete. A Greek friend remarked on Sunday how surprised he was at my recent progress with the Greek language. Michaelis had been encouraging me for the last 4 years to knuckle down and put in a bit of an effort.

Learning a new language requires a good memory, which has never been my strong point. I discovered in grade 8 history that I hardly had any short-term memory at all. That, coupled with an illegible scrawl, were serious handicaps in school. As a result, my grades were never particularly good. I was also overconfident and seldom read questions properly before writing my answer. The only time I ever got a distinction in school was for my final exam. It is funny that the rest of my 12 years of school do not count for anything. Nobody was ever interested in how many PT classes I bunked or that I failed "hand writing" in grade 5, all they wanted to know was my final score.

Greek is not the first language I learned. In high school, I decided to learn an African language called **isiXhosa**, spoken by our former president Nelson Mandela. This was in the years of apartheid, when it was extremely unfashionable for a melanin deprived individual to learn an African language. In my class, I was the most fluent speaker. My secret? I would walk home from school and speak to anybody who cared to listen. I made a lot of mistakes and most of the time I spoke about the same things. "Where are you from? Where are you going? It is hot today". This loosened my tongue and gave me confidence. I do the same with Greek.

My second secret was that I wrote a computer program in GWBasic to help me remember the vocabulary. The program would show me a word and then I would have to write it in isiXhosa. If I made a mistake, I would get punished by having to type it three times. I wrote a similar Java program for Greek, which was challenging because of the strange character set. What surprises me is how this program flashes the words into my medium term memory, without first going into the short-term memory. Even though I don't actively remember the word, it is there in my brain and I can recall it, as long as I do not think too hard. Another surprise is that I am finding it as difficult (or as easy) to learn Greek as I found it to learn

isiXhosa, when I was 13 years old. Some expats tell me that they are too old to learn. But it is just as hard for me to remember anything as when I was 13. It is a major effort for me to recall even a single word.

So, in conclusion of this very long opening, if you are finding it hard learning new languages, computer or natural, you are in good company. It is hard for me too. The trick is to *use* the language, then eventually something will stick :-) (Oh and just for interest, "use" in Greek is xrhsimopoiw :-))

Delaying Garbage Collection Costs

A few weeks ago, our French **Java Specialist Master Course** instructor sent us an interesting puzzle from his brilliant **The Coder's Breakfast** blog. I love the name, as it evokes good French coffee and a croissant whilst puzzling over the latest Java quiz.

I will rephrase his question a bit, because I think something might have gotten lost in the translation from French into English. With minimal Java code changes, how can you make the Quiz44 class run at least 10x faster?

```
public class Quiz44 {
    private static final int NB_VALUES = 10000000;
    private static final int MAX_VALUE = 1000;
    private static final int NB_RUNS = 10;

    public static void main(String[] args) {
        Integer[] boxedValues = new Integer[NB_VALUES];
        int[] values = initValues();

        System.out.println("Benchmarking...");
        for (int run = 1; run <= NB_RUNS; run++) {
            long t1 = System.currentTimeMillis();
            for (int i = 0; i < NB_VALUES; i++) {
                boxedValues[i] = values[i];
            }
            long t2 = System.currentTimeMillis();
            System.out.printf("Run %2d : %4dms%n", run, t2 - t1);
        }
    }

    /**
     * Nothing important here, just values init.
     */
    private static int[] initValues() {
        System.out.println("Generating values...");
    }
}
```

```

int[] values = new int[NB_VALUES];
Random random = new Random();
for (int i = 0; i < values.length; i++) {
    values[i] = random.nextInt(MAX_VALUE);
}
return values;
}
}

```

When I run this on my MacBook Pro, with -Xmx500m, I see the following results:

```

Generating values...
Benchmarking...
Run 1 : 1657ms
Run 2 : 2879ms
Run 3 : 2265ms
Run 4 : 2217ms
Run 5 : 2211ms
Run 6 : 2216ms
Run 7 : 930ms
Run 8 : 2216ms
Run 9 : 2241ms
Run 10 : 2216ms

```

The average is 2105, with a variance of 254,000, caused by Run 7.

There are some obvious changes to the code that would make it complete faster. For example, we could reduce the number of NB_VALUES or MAX_VALUE fields, but I think Olivier would consider that as cheating. However, there is also a non-obvious change we could make.

For example, we could add a `System.gc()` before the first call to `System.currentTimeMillis()`. In that case, the test runs with the following values:

```

Generating values...
Benchmarking...

```

```

Run 1 : 1720ms
Run 2 : 2645ms
Run 3 : 2240ms
Run 4 : 927ms
Run 5 : 918ms
Run 6 : 930ms
Run 7 : 965ms
Run 8 : 914ms
Run 9 : 903ms
Run 10 : 913ms

```

This time the average is 1308 with a variance of 430,000. Not a bad speedup at all. After all, it is now 38% faster than before. It should be obvious that a large proportion of time is wasted with collecting all these unnecessary objects. If we run the code again with -verbose:gc on, we can see the costs involved:

```

Generating values...
[GC 40783K->39473K(83008K), 0.0041315 secs]
[Full GC 39473K->39465K(83008K), 0.0696936 secs]
[GC 78528K(123980K), 0.0002081 secs]
[GC 79209K(123980K), 0.0002467 secs]
Benchmarking...
[Full GC 79197K->78518K(150000K), 0.0828044 secs]
[GC 78518K(150000K), 0.0001254 secs]
[GC 78995K(150000K), 0.0006032 secs]
[GC 95542K->103000K(150000K), 0.3685923 secs]
[GC 103000K(150000K), 0.0021857 secs]
[GC 120024K->129617K(150000K), 0.1627059 secs]
[GC 146641K->155227K(172464K), 0.1826291 secs]
[GC 172251K->180831K(198000K), 0.1499428 secs]
[GC 197855K->206365K(223536K), 0.1794985 secs]
[GC 223389K->231900K(249072K), 0.1751786 secs]
[GC 248924K->257435K(274800K), 0.1594760 secs]
[GC 274459K->282969K(300144K), 0.2015765 secs]
Run 1 : 1774ms
[Full GC 283309K->282255K(300144K), 0.8866413 secs]
[GC 315119K->330066K(506684K), 0.3946753 secs]
[GC 364178K->398754K(506684K), 0.3282639 secs]
[GC 398754K(506684K), 0.0043726 secs]
[GC 432866K->449971K(506684K), 0.3649566 secs]
[Full GC 484083K->282879K(506684K), 1.1812640 secs]
Run 2 : 2708ms
[Full GC 284935K->282881K(507776K), 1.0651874 secs]

```

```

[GC 316993K->345922K(507776K), 0.2532635 secs]
[GC 380034K->399611K(507776K), 0.2922708 secs]
[GC 400297K(507776K), 0.0042360 secs]
[GC 433723K->450807K(507776K), 0.3415709 secs]
[Full GC 484919K->282884K(507776K), 1.0057979 secs]
Run 3 : 2324ms
[Full GC 283571K->282884K(507776K), 0.8885789 secs]
[GC 316996K->347295K(507776K), 0.2598463 secs]
[GC 381407K->400631K(507776K), 0.3051485 secs]
[GC 401318K(507776K), 0.0042195 secs]
[GC 434743K->451850K(507776K), 0.3479674 secs]
Run 4 : 1024ms
[Full GC 485962K->282884K(507776K), 1.0040985 secs]
[GC 316996K->347295K(507776K), 0.2591832 secs]
[GC 381407K->400631K(507776K), 0.2888177 secs]
[GC 401318K(507776K), 0.0042504 secs]
[GC 434743K->451850K(507776K), 0.3348886 secs]
Run 5 : 994ms
[Full GC 485962K->282884K(507776K), 1.0128758 secs]
[GC 316996K->347295K(507776K), 0.2580010 secs]
[GC 381407K->400631K(507776K), 0.2884526 secs]
[GC 402692K(507776K), 0.0060617 secs]
[GC 434743K->451848K(507776K), 0.3290486 secs]
Run 6 : 1004ms
[Full GC 485960K->282884K(507776K), 1.0040235 secs]
[GC 316996K->347295K(507776K), 0.2596790 secs]
[GC 381407K->400631K(507776K), 0.2851338 secs]
[GC 401318K(507776K), 0.0042191 secs]
[GC 434743K->451840K(507776K), 0.3340752 secs]
Run 7 : 989ms
[Full GC 485952K->282884K(507776K), 1.0022637 secs]
[GC 316996K->347295K(507776K), 0.2612456 secs]
[GC 381407K->400631K(507776K), 0.2933666 secs]
[GC 401318K(507776K), 0.0043201 secs]
[GC 434743K->451842K(507776K), 0.3280430 secs]
Run 8 : 997ms
[Full GC 485954K->282884K(507776K), 1.0126833 secs]
[GC 316996K->347295K(507776K), 0.2569432 secs]
[GC 381407K->400631K(507776K), 0.2866691 secs]
[GC 401318K(507776K), 0.0042206 secs]
[GC 434743K->451842K(507776K), 0.3418867 secs]
Run 9 : 1000ms
[Full GC 485954K->282884K(507776K), 1.0074827 secs]
[GC 316996K->347295K(507776K), 0.2594386 secs]
[GC 381407K->400631K(507776K), 0.2966164 secs]
[GC 401318K(507776K), 0.0042397 secs]
[GC 434743K->451840K(507776K), 0.3391696 secs]
Run 10 : 1009ms

```

[Full GC 485952K->400K(507776K), 0.3041337 secs]

The Full GC's in *italics* are the artificial GCs that we introduced to make our test run faster. If we exclude them from the timing, we can work out how much of each run is spent collecting garbage:

| Run | GC | Program | Total |
|-----|------|---------|-------|
| 1 | 1583 | 191 | 1774 |
| 2 | 2274 | 434 | 2708 |
| 3 | 1897 | 427 | 2324 |
| 4 | 917 | 107 | 1024 |
| 5 | 887 | 107 | 994 |
| 6 | 882 | 122 | 1004 |
| 7 | 883 | 106 | 989 |
| 8 | 887 | 110 | 997 |
| 9 | 890 | 110 | 1000 |
| 10 | 899 | 101 | 1009 |

We can get even better figures from our program if instead of elapsed time (System.currentTimeMillis()), we use user CPU time (ThreadMXBean.getCurrentThreadUserTime()). Here are the run times:

```
Generating values...
Benchmarking...
Run 1 : 1667ms 106ms
Run 2 : 2576ms 114ms
Run 3 : 2199ms 109ms
Run 4 : 897ms 109ms
Run 5 : 899ms 109ms
Run 6 : 899ms 109ms
Run 7 : 903ms 109ms
Run 8 : 889ms 109ms
Run 9 : 892ms 109ms
Run 10 : 896ms 108ms
```

Our average user CPU time is now 109ms with a variance of 4.

Let's get back to the problem. How do we make the test run 10 times faster with minimal code changes? My first approach was to simply delay the cost of GC until after the test has run through. Since I have 8GB of memory in my laptop, I can beef up the initial and maximum size of the heap to a huge amount. I also set the NewSize to a crazy amount, so that the GC never needs to run whilst the test is running.

```
$ java -showversion -Xmx10g -Xms10g -XX:NewSize=9g Quiz44

java version "1.6.0_24"
Java(TM) SE Runtime Environment (build 1.6.0_24-b07-334-10M3326)
Java HotSpot(TM) 64-Bit Server VM (build 19.1-b02-334, mixed mode)

Generating values...
Benchmarking...
Run 1 : 198ms
Run 2 : 203ms
Run 3 : 179ms
Run 4 : 192ms
Run 5 : 179ms
Run 6 : 191ms
Run 7 : 183ms
Run 8 : 192ms
Run 9 : 180ms
Run 10 : 197ms
```

This gave me an average of 189ms with a slightly high variance of 75. Since my initial results had an average of 2105, I achieved Olivier's target of being at least 10x faster. In case you think what I've done is really stupid, I know of companies who use this trick to avoid all GC pauses. They try to never construct objects. They make the young generation very large. After running for several hours, they quit the application and start again. This gives them very predictable run times. So it's not as harebrained an idea as it appears at first.

In our case, things start to go wrong when we increase the number of runs, for example, let's change NB_RUNS to 100:

```
java version "1.6.0_24"
Java(TM) SE Runtime Environment (build 1.6.0_24-b07-334-10M3326)
```

```
Java HotSpot(TM) 64-Bit Server VM (build 19.1-b02-334, mixed mode)

Generating values...
Benchmarking...
Run  1 : 188ms
Run  2 : 214ms
Run  3 : 234ms
Run  4 : 191ms
Run  5 : 203ms
Run  6 : 193ms
Run  7 : 207ms
Run  8 : 188ms
Run  9 : 208ms
Run 10 : 183ms
Run 11 : 199ms
Run 12 : 183ms
Run 13 : 200ms
Run 14 : 186ms
Run 15 : 202ms
Run 16 : 192ms
Run 17 : 216ms
Run 18 : 191ms
Run 19 : 211ms
Run 20 : 188ms
Run 21 : 210ms
Run 22 : 190ms
Run 23 : 210ms
Run 24 : 187ms
Run 25 : 211ms
Run 26 : 186ms
Run 27 : 208ms
Run 28 : 188ms
Run 29 : 205ms
Run 30 : 188ms
Run 31 : 216ms
Run 32 : 189ms
Run 33 : 210ms
Run 34 : 186ms
Run 35 : 215ms
Run 36 : 190ms
Run 37 : 222ms
Run 38 : 191ms
Run 39 : 209ms
Run 40 : 186ms
Run 41 : 1068ms
Run 42 : 7281ms
Run 43 : 8705ms
Run 44 : 8583ms
```

```
Run 45 : 7675ms
Run 46 : 6346ms
Run 47 : 1625ms
Run 48 : 896ms
Run 49 : 834ms
Run 50 : 948ms
Run 51 : 554ms
Run 52 : 901ms
Run 53 : 904ms
Run 54 : 4253ms
Run 55 : 815ms
Run 56 : 111ms
Run 57 : 127ms
Run 58 : 143ms
Run 59 : 159ms
Run 60 : 117ms
Run 61 : 138ms
Run 62 : 136ms
Run 63 : 140ms
Run 64 : 128ms
Run 65 : 131ms
Run 66 : 136ms
Run 67 : 147ms
Run 68 : 122ms
Run 69 : 127ms
Run 70 : 135ms
Run 71 : 135ms
Run 72 : 135ms
Run 73 : 134ms
Run 74 : 126ms
Run 75 : 125ms
Run 76 : 217ms
Run 77 : 765ms
Run 78 : 2973ms
Run 79 : 2459ms
Run 80 : 2180ms
Run 81 : 4733ms
Run 82 : 7484ms
Run 83 : 3177ms
Run 84 : 6209ms
Run 85 : 5129ms
Run 86 : 1051ms
Run 87 : 5177ms
Run 88 : 5515ms
Run 89 : 6217ms
Run 90 : 8865ms
Run 91 : 7981ms
Run 92 : 3578ms
```

```
Run 93 : 3472ms
Run 94 : 3645ms
Run 95 : 4006ms
Run 96 : 3933ms
Run 97 : 4211ms
Run 98 : 3127ms
Run 99 : 3714ms
Run 100 : 3811ms
```

This gives us an average of 1656 and variance of 6,000,000. There are also times when the system is almost unusable because of the time wasted in GC.

One of our **Java Specialist Club** members, Marek Bickos, quickly discovered a flag that made this code run much faster. By simply invoking the code with `-XX:+AggressiveOpts`, it caused no GC at all:

```
Generating values...
Benchmarking...
Run 1 : 56ms
Run 2 : 45ms
Run 3 : 39ms
Run 4 : 50ms
Run 5 : 66ms
Run 6 : 39ms
Run 7 : 39ms
Run 8 : 40ms
Run 9 : 46ms
Run 10 : 46ms
```

A bit of experimentation explained why this was so much faster. In modern JVMs, the autoboxing cache size is configurable. It used to be that all values from -128 to 127 were cached, but nowadays we can specify a different upper bound. When we run the code with `-XX:+AggressiveOpts`, it simply increases this value. We can try it out with this little program:

```
public class FindAutoboxingUpperBound {
    public static void main(String[] args) {
        int i=0;
```

```

while(isSame(i,i)) {
    i++;
}
System.out.println("Upper bound is " + (i - 1));
i = 0;
while(isSame(i,i)) {
    i--;
}
System.out.println("Lower bound is " + (i + 1));
}

private static boolean isSame(Integer i0, Integer i1) {
    return i0 == i1;
}
}

```

If we run this without any flags, it comes back as:

```

Upper bound is 127
Lower bound is -128

```

With `-XX:+AggressiveOpts`, we get these results:

```

Upper bound is 20000
Lower bound is -128

```

It is thus pretty obvious why it is so much faster. Instead of constructing new objects, it is simply getting them from the Integer cache.

If we increase the `MAX_VALUE` to 100000, and run the program with `-XX:+AggressiveOpts` and `-Xmx500m`, we get these results:

```

Generating values...
Benchmarking...
Run 1 : 1622ms
Run 2 : 1351ms
Run 3 : 2675ms
Run 4 : 2037ms
Run 5 : 2322ms
Run 6 : 2325ms
Run 7 : 2195ms
Run 8 : 2394ms
Run 9 : 2264ms
Run 10 : 2109ms

```

However, with my flags of "-Xmx10g -Xms10g -XX:NewSize=9g", it is still as fast as before.

The question is thus, which is the better solution? I think it depends very much on how this is going to be used in the real world. My solution would work with any objects that are being made, not just Integers. However, we also saw the serious downside to my approach. Once we do fill up our young space, the JVM will get very slow, at least for a while.

The solution Olivier was looking for was to set the maximum number of values in the integer cache using the system parameter `java.lang.Integer.IntegerCache.high`, for example:

```

java -Djava.lang.Integer.IntegerCache.high=1000 Quiz44

Generating values...
Benchmarking...
Run 1 : 40ms
Run 2 : 54ms
Run 3 : 48ms
Run 4 : 49ms
Run 5 : 66ms
Run 6 : 48ms
Run 7 : 48ms
Run 8 : 49ms
Run 9 : 48ms
Run 10 : 66ms

```

The `-XX:AutoBoxCacheMax=1000` flag would do the same thing. Whilst it is possible to specify the maximum Integer in the cache, no doubt to satisfy some silly microbenchmark to prove that Java is at least as fast as Scala, it is currently not possible to set the minimum Integer.

We talk about this and many other tricks of the trade in our [Java Specialists Master Course](#) on the Island of Crete on the 7th of June 2011...

Heinz

Issue 190b - Automatically Unlocking with Java 7 - Follow-up

Author: Dr. Heinz M. Kabutz

Date: 2011-04-15

Category: Language

Java Versions: Java 7

Abstract:

In this newsletter we discuss why we unfortunately will not be able to use the try-with-resource mechanism to automatically unlock in Java 7.

Welcome to the follow-up newsletter to the 190th issue of **The Java(tm) Specialists' Newsletter**, sent to you from the beautiful Island of Crete. Greek Easter is around the corner, so the lambs are already starting to get mildly concerned. Everything works in a rhythm in Crete. We have certain festivals, parades, religious holidays, then the 14 week summer holiday. On the 15th of August, there is a nationwide party, after which a lot of people go back to work. We then have a big day on the 28th October as the entire country celebrates my wife's birthday. At least I *think* that is what is going on. Then of course we have Christmas, New Year, the throwing of the cross into the harbour, a party before lent, then lent, and then we get back to where we are today. It almost seems like we are caught in an infinite loop as many of these festivals have been celebrated for thousands of years. And at the end of lent, the lambs are set upon by ten million ravenous and meat deprived Greeks. Poor critters. The Greek word to describe them is "hostimo".

Automatically Unlocking with Java 7 - Follow-up

In this newsletter we look at some of the objections raised regarding my previous [Automatically Unlocking with Java 7](#) article.

The first person to write was Brian Goetz, pointing out that we will probably not be able to use the expression form with try-with-resource. It is possible to fudge it by using a declaration, but the result is so ugly that I'd rather not show it. Bruce Chapman then sent me a link to the [discussion on Joe Darcy's blog](#) where he talks about this. I will explain in a bit why they disallowed expressions.

Several people contacted me to let me know that you *can* now have a semicolon after the **try** declaration. I have an older Java 7 build on my Mac that did not allow this.

After pointing to my newsletter on the coin-dev mailing list, several people pointed out that the cost of constructing the additional lock objects would be unacceptable with critical code. However, based on my initial results, escape analysis in the Server Hotspot compiler seems

to eradicate that cost completely. Worrying about the costs of object construction is thus a premature optimization.

Programmers are notoriously bad at coding concurrency correctly. We should thus rather encourage them to use higher-level, thread-safe classes, rather than trying to use locks explicitly.

Reinier Zwitserloot published a nice explanation why we cannot allow expressions in the new try-with-resource construct. He kindly gave permission for me to republish it here:

To summarize the log, you can't tell the difference between generics on the type of a variable declaration and the lesser than operator:

```
try (x < y ? resourceA : resourceB) {}
```

vs:

```
try (x<y> z = resourceA) {}
```

An LL(k) parser can't tell the difference until its too late. A packrat/rd parser could, but both javac and ecj are LL(k) parsers so changing this would require both to be rewritten from scratch.

Re-introducing the ability to use just an expression here would require some sort of keyword or symbol that is unambiguous. This could work:

```
try (= expression) {}
```

but it breaks the principle of least surprise, in that without knowing a heck of a lot about the capabilities of LL(k) parsers, the need for the = sign is a mystery.

I gather from this discussion as well as the reaction to the fine work by Heinz Kabutz that the primary use case for expressions in ARM statements is locks, and this use case is not good enough, even if it is actually fast enough at least on server mode VMs.

Thank you very much Reinier for that excellent explanation. It is easy to forget the reasons why computer languages need to have a certain syntax.

Heinz

Issue 190 - Automatically Unlocking with Java 7

Author: Dr. Heinz M. Kabutz

Date: 2011-02-27

Category: Language

Java Versions: Java 7

Abstract:

In this newsletter we explore my favourite new Java 7 feature "try-with-resource" and see how we can use this mechanism to automatically unlock Java 5 locks.

Welcome to the 190th issue of **The Java(tm) Specialists' Newsletter**, sent to you from the beautiful Island of Crete. I started Greek lessons last week. With 8 hours under my belt, I headed off to a Greek dinner party last night. For the first time, I could understand the conversations going on around me and respond relatively correctly. Watch this space for some funny stories as I learn this confusing language, where a "t'axi" is a classroom (emphasis on "a") and a "tax'i" that smelly vehicle you get driven to the airport in. Where a professor is "kathiyitis" and a janitor a "katharistis". My job description is "programmatistis ypologiston kai kathiyitis pliroforikis" (computer programmer and computer science instructor). Maybe I will just use "anergos" (unemployed) in future - that is so much easier to remember ... :-) [besides, that is how banks view self-employed people like me anyway ;-]

Talking of Greece, we are doing our **Masters course** from the 7-10 June here in Chania. [Let me know](#) if you need some nice warm sunny weather and great food to make 4 intense days of learning more bearable.

Automatically Unlocking with Java 7

One of the quirks of Java is how difficult it is to correctly close resources. For example, if you open a file, it is easy to forget to close it. Your code might even work on some operating systems. But on others, you will run out of file handles.

To simplify things, Java 7 introduced the new `try-with-resource` statement, which automatically closes any `AutoCloseable` resources referenced in the `try` statement. For example, instead of manually closing streams ourselves, we can simply do this:

```
import java.io.*;

public class AutoClosingFiles {
    public static void main(String[] args) throws IOException {
```

```

try {
    PrintStream out = new PrintStream (
        new BufferedOutputStream(
            new FileOutputStream("file.txt")))
    ) {
        out.print("This would normally not be written!");
    }
}
}

```

Note that there is never a semicolon at the end of the "try ()" declaration, even when you specify several resources that must be closed individually. **Update:** Thanks to Marco Hunsicker for pointing out that they are working on allowing us to have tailing semicolons in try-with-resource declarations. [Here is a link](#). Since I'm a Mac OS X user, I had to compile the OpenJDK myself.

```

import java.io.*;

public class AutoClosingFiles2 {
    public static void main(String[] args) throws IOException {
        try {
            FileOutputStream fout = new FileOutputStream("file.txt");
            BufferedOutputStream bout = new BufferedOutputStream(fout);
            PrintStream out = new PrintStream (fout)
        ) {
            out.print("This would normally not be written!");
        }
    }
}

```

There are two new features in Java 7 that allow this. First we have the `java.lang.AutoCloseable` interface, implemented by a whopping 553 classes in the JDK! To put this in perspective, `java.io.Serializable` is implemented 3919 times, `java.lang.Cloneable` 1297 times and `java.lang.Comparable` 759 times. `java.lang.Runnable` is only implemented 186 times and `java.lang.Iterable` 252 times.

The second new feature is the method `Throwable.addSuppressed(Throwable)`. This method has some strange semantics. It is only ever called by the try-with-resources code construct. Here is how it works:

1. If addSuppressed is called the first time with **null**, then it will never have suppressed throwables attached.
2. If it is subsequently called with a non-null element, we will see a NullPointerException.
3. If addSuppressed is called with a non-null throwable, then it will contain a collection of throwables.
4. If it is subsequently called with a null element, we will see a NullPointerException.

These weird semantics are not meant to be understandable, but rather support the new try-with-resource mechanism. In a future newsletter, I will explore this in more detail. For now it will suffice to know that the try-with-resource guarantees that created objects will be closed again, with exceptions managed correctly.

When I woke up yesterday, my mind wandered to Java 5 locks and I was trying to figure out why they were not also AutoCloseable. After all, over 500 other classes were. In my half-sleeping state, I figured out why and also worked out a way to make it work.

The try-with-resource works well with objects that are constructed and then closed immediately again. It will not work with resources that need to be kept alive. Locks would fall into this latter category. We construct locks once and then use them for as long as we need to protect the critical section. Thus we cannot autoclose them when they go out of scope. Problem is, they won't go out of scope at the same time that we want to unlock them.

However, we can write a wrapper that automatically unlocks the lock for us. We call lock() in the constructor and unlock() in the close() method overridden from AutoCloseable:

```
import java.util.concurrent.locks.*;

public class AutoLockSimple implements AutoCloseable {
    private final Lock lock;

    public AutoLockSimple(Lock lock) {
        this.lock = lock;
        lock.lock();
    }

    public void close() {
        lock.unlock();
    }
}
```

Disclaimer: I have not used Java 7 in a production environment yet. Thus I do not know if

there are any issues with my idea of AutoLockSimple. It seems good to me, but I give no guarantees. Please let me know if you think of anything. One of the issues that could be a problem is that we will make new objects every time we lock. This unnecessary object creation could end up straining the GC. However, from my initial tests, it seems that escape analysis takes care of the object construction cost.

Here is how we use it in our code. We use the handle to the ReentrantLock so that we can call the isHeldByCurrentThread() method. This way we can determine whether we are locked or not.

```
import java.util.concurrent.locks.*;

public class AutoLockSimpleTest {
    private final static ReentrantLock lock = new ReentrantLock();

    public static void main(String[] args) {
        try (new AutoLockSimple(lock)) {
            printLockStatus();
        }
        printLockStatus();
    }

    private static void printLockStatus() {
        System.out.println("We are locked: " + lock.isHeldByCurrentThread());
    }
}
```

Output is simply this:

```
We are locked: true
We are locked: false
```

This is nice, but we can make it a bit less obvious that we have to construct an object by using static factory methods. In addition, we can then also implement a lockInterruptibly() mechanism. In this code, I define the various locking approaches as static inner classes. The

```

package eu.javaspecialists.concurrent;

import java.util.concurrent.locks.*;

public class AutoLock implements AutoCloseable {
    public static AutoLock lock(Lock lock) {
        return new AutoLockNormal(lock);
    }

    public static AutoLock lockInterruptibly(Lock lock) throws InterruptedException {
        return new AutoLockInterruptibly(lock);
    }

    private final Lock lock;

    public void close() {
        lock.unlock();
    }

    private AutoLock(Lock lock) {
        this.lock = lock;
    }

    private static class AutoLockNormal extends AutoLock {
        public AutoLockNormal(Lock lock) {
            super(lock);
            lock.lock();
        }
    }

    private static class AutoLockInterruptibly extends AutoLock {
        public AutoLockInterruptibly(Lock lock) throws InterruptedException {
            super(lock);
            lock.lockInterruptibly();
        }
    }
}

```

This in combination with static imports is far more readable than the try-finally approach commonly used for Java 5 locks. Here is how you would call the factory methods:

```

try (lock(lock)) {

```

```

    printLockStatus();
}

```

And if you want to be interruptible, then we do it like this:

```

try (lockInterruptibly(lock)) {
    printLockStatus();
}

```

It will be challenging to implement tryLock(), because we only want to unlock() if we were successful in our tryLock(). We also only want to enter the critical section if we were able to lock.

Here is a complete test class that demonstrates how the lock() method can be used in combination with static imports:

```

import java.util.concurrent.locks.*;

import static eu.javaspecialists.concurrent.AutoLock.lock;

public class AutoLockTest {
    private final static ReentrantLock lock = new ReentrantLock();
    public static void main(String[] args) {
        // The old way - far more verbose
        lock.lock();
        try {
            printLockStatus();
        } finally {
            lock.unlock();
        }

        // Heinz's new way
        try (lock(lock)) {
            printLockStatus();
        }
        printLockStatus();
    }
}

```

```

private static void printLockStatus() {
    System.out.println( "We are locked: " + lock.isHeldByCurrentThread());
}
}

```

Output from our code is this:

```

We are locked: true
We are locked: true
We are locked: false

```

The test code for the interruptible lock is a bit more involved, since we need to interrupt the testing thread.

```

import java.util.concurrent.locks.*;
import static eu.javaspecialists.concurrent.AutoLock.*;

public class AutoLockInterruptiblyTest {
    private static final ReentrantLock lock = new ReentrantLock();

    public static void main(String[] args) throws InterruptedException {
        testLock();
        Thread.currentThread().interrupt();
        try {
            testLock();
        } catch (InterruptedException ex) {
            System.out.println(ex);
        }
    }

    public static void testLock() throws InterruptedException {
        try (lockInterruptibly(lock)) {
            printLockStatus();
        }
        printLockStatus();
    }
}

```

```
private static void printLockStatus() {  
    System.out.println( "We are locked: " + lock.isHeldByCurrentThread());  
}  
}
```

Here is the output from the test program:

```
We are locked: true  
We are locked: false  
java.lang.InterruptedException
```

Hopefully Java 7 will give me lots of new material for newsletters. It was becoming difficult to find new things in Java 6. In another newsletter I will show how my use of try-with-resources is actually more correct than the traditional try-finally. But that will have to wait for another day.

Heinz

Update: A more complete set of tests is available in [AutoLockTest](#).

Issue 188 - Interlocker - Interleaving Threads

Author: Dr. Heinz M. Kabutz

Date: 2010-10-31

Category: Concurrency

Java Versions: Java 6

Abstract:

In this newsletter, we explore a question of how to call a method interleaved from two threads. We show the merits of lock-free busy wait, versus explicit locking. We also discuss an "unbreakable hard spin" that can cause the JVM to hang up.

Welcome to the 188th issue of **The Java(tm) Specialists' Newsletter**, sent to you from the **K Trade Fair**, an International Trade Fair for Plastics and Rubber. About a quarter of a million people have descended on Dusseldorf to see what is going on in the plastics industry. Since I am a director of a drinking straw factory in South Africa, it seemed like a good place to meet up with our general manager. We have seen a huge number of very interesting products. One of the most interesting was a three dimensional printer. You enter a model in the computer and it constructs this by applying layer after layer of molten plastic. Eventually you have a little plastic shape that accurately represents the model on the computer. One of the things they have not figured out yet is anti-aliasing, so the models have steps in between the levels. Maybe one day they will even solve that problem. Also, the current 3D printers can only apply one colour per layer of plastic.

On Monday I am off to New York, where I will be speaking at the **NYC Java Meetup** on November 4th 2010.

Interlocker - Interleaving Threads

A few months ago, one of my subscribers, Mohammed Irfanulla S., sent me an interesting question. How can you alternately call a method from two threads? Imagine you have two threads, first thread 1 calls a method, then thread 2, then thread 1 again, and so on. One thread should never call the method twice in a row.

Mohammed sent me several different solutions to this puzzle. My first step was to make our testing a bit more robust, so that we could catch any mistakes early. I also spent a while renaming the classes, to make our intentions clearer. Lastly, I marked up the classes with the new **jpatterns annotations** to indicate where we were using which design patterns. The jpatterns.org annotations are a new project started by **The Java Specialist Club**. You are welcome to participate in the project if you would like to.

Disclaimer: I cannot think of a practical application of where such an "interlocker" would be useful. However, some of the discoveries might be useful in highly communicative systems.

such as the lock-free approach to solving the problem. Only use if you know exactly what you are doing :-)

As a first class, we define Interlocker, which uses the template method to start threads that will call the Runnables. The execute() method will block until all the threads have finished. The Runnable objects returned by the getRunnable() method should guarantee that the InterlockTask is called interleaved by the threads. They do not have to guarantee which thread starts, but the total call count must be correct.

```
import org.jpatterns.gof.*;

/*
 * This special executor guarantees that the call() method of the
 * task parameter is invoked in turns by two threads. There is
 * probably no practical application for this class, except as a
 * learning experience.
 */

@TemplateMethodPattern.AbstractClass
public abstract class Interlocker {

    @TemplateMethodPattern.PrimitiveOperation
    protected abstract Runnable[] getRunnables(InterlockTask task);

    @TemplateMethodPattern.TemplateMethod
    public final <T> T execute(InterlockTask<T> task)
        throws InterruptedException {
        Runnable[] jobs = getRunnables(task);
        Thread[] threads = new Thread[jobs.length];
        for (int i = 0; i < threads.length; i++) {
            threads[i] = new Thread(jobs[i]);
            threads[i].start();
        }
        for (Thread thread : threads) {
            thread.join();
        }
        return task.get();
    }
}
```

Before we look at some possible solutions, let us view the InterlockTask interface. It is self explanatory.

```

import org.jpatterns.gof.*;

@StrategyPattern.Strategy
public interface InterlockTask<T> {
    boolean isDone();

    /**
     * The call() method is called interleaved by the threads
     * in a round-robin fashion.
     */
    void call();

    /**
     * Returns the result after all the call()'s have completed.
     */
    T get();

    void reset();
}

```

In the first test case, we want to simply increment a value. Since we are accessing the field from multiple threads, we need to declare it as volatile, but since only one thread is invoking the call() method, we do not need to synchronize. Since this test does very little, we can also use it to measure the overhead of our thread coordination mechanism.

```

import org.jpatterns.gof.*;

@StrategyPattern.ConcreteStrategy
public class EmptyInterlockTask implements
    InterlockTask<Integer> {
    public final int upto;
    private volatile int count;

    public EmptyInterlockTask(int upto) {
        this.upto = upto;
    }

    public boolean isDone() {
        return count >= upto;
    }

    public void call() {
        count++;
    }
}

```

```

    }

    public Integer get() {
        return count;
    }

    public void reset() {
        count = 0;
    }
}

```

The next test verifies that the call() method was invoked by alternating threads. We do this by inserting the current call number into a LinkedHashMap, with the number as key and the thread as a value. Afterwards, when we call get(), we verify that the result is correct. This is returned in the VerifyResult object.

```

import org.jpatterns.gof.*;

import java.util.*;
import java.util.concurrent.atomic.*;

@StrategyPattern.ConcreteStrategy
public class InterleavedNumberTestingStrategy implements
    InterlockTask<VerifyResult> {
    public final int upto;
    private final Map<Integer, Thread> numbers =
        new LinkedHashMap<Integer, Thread>();
    private final AtomicInteger count = new AtomicInteger(0);

    public InterleavedNumberTestingStrategy(int upto) {
        this.upto = upto;
    }

    public boolean isDone() {
        return count.get() >= upto;
    }

    public void call() {
        int next = count.getAndIncrement();
        numbers.put(next, Thread.currentThread());
    }

    public VerifyResult get() {

```

```

if (numbers.size() < upto) {
    return new VerifyResult("Only " + numbers.size() +
        " numbers were entered");
}
Object previous = null;
int i = 0;
for (Map.Entry<Integer, Thread> entry : numbers.entrySet()) {
    if (i != entry.getKey()) {
        return new VerifyResult("numbers out of sequence");
    }
    if (entry.getValue() == previous) {
        return new VerifyResult("Did not alternate threads");
    }
    previous = entry.getValue();
    i++;
}
Set<Thread> values = new HashSet<Thread>(numbers.values());
if (values.size() != 2) {
    return new VerifyResult(
        "More than two threads were inserting values");
}
return new VerifyResult();
}

public void reset() {
    numbers.clear();
    count.set(0);
}
}

```

```

public class VerifyResult {
    private final boolean success;
    private final String failReason;

    private VerifyResult(boolean success, String failReason) {
        this.success = success;
        this.failReason = failReason;
    }

    public VerifyResult(String failReason) {
        this(false, failReason);
    }

    public VerifyResult() {
        this(true, null);
    }
}

```

```

    }

    public boolean isSuccess() {
        return success;
    }

    public String getFailReason() {
        return failReason;
    }

    public String toString() {
        return success ? "Success" : "Failure - " + failReason;
    }
}

```

Another task could print the threads, for example:

```

// *snip*
public boolean isDone() {
    return row.get() >= upto;
}

public void call() {
    System.out.println(Thread.currentThread().getName());
    row.incrementAndGet();
}

```

Interlocker Solutions

The easiest solution is to use semaphores. We start with two semaphores. The first has 1 barrier, the second zero. From thread 1, we acquire semaphore 1, do the call(), then release semaphore 2. From thread 2, we acquire semaphore 2, do the call(), then release semaphore 1. The reason this works is that you can release a semaphore from a thread that has not acquired it. This is quite different to locks, where only the thread that acquired the lock can release it.

```

import java.util.concurrent.*;

```

```

public class SemaphoreInterlocker extends Interlocker {
    private static class Job implements Runnable {
        private final InterlockTask task;
        private final Semaphore first;
        private final Semaphore second;

        public Job(InterlockTask task,
                   Semaphore first, Semaphore second) {
            this.task = task;
            this.first = first;
            this.second = second;
        }

        public void run() {
            while (!task.isDone()) {
                first.acquireUninterruptibly();
                if (task.isDone()) return;
                task.call();
                second.release();
            }
        }
    }

    protected Runnable[] getRunnables(InterlockTask task) {
        Semaphore even = new Semaphore(1);
        Semaphore odd = new Semaphore(0);
        return new Runnable[]{
            new Job(task, even, odd),
            new Job(task, odd, even)
        };
    }
}

```

When running this code, I noticed that it was rather slow. For example, to increment an int one million times took 4 seconds on my machine, of which most was context switching:

```

InterlockTask task = new EmptyInterlockTask(1000 * 1000);
Interlocker generator = new SemaphoreInterlocker();
long time = System.currentTimeMillis();
generator.execute(task);
time = System.currentTimeMillis() - time;
System.out.println(time);

```

The other solutions that we developed, using wait-notify and Java 5 Condition, were similar in performance characteristic. We will leave those Interlockers as an exercise for the reader :-)

If you want to attempt them, remember the effects that spurious wakeups can have on your Object.wait() and Condition.await().

Lock Free Interlocker

I wanted to write a fast Interlocker implementation, ideally not locking at all and using a volatile field as a communication mechanism. Here is what I did:

```
public class LockFreeInterlocker extends Interlocker {
    private volatile boolean evenHasNextTurn = true;

    private class Job implements Runnable {
        private final InterlockTask task;
        private final boolean even;

        private Job(InterlockTask task, boolean even) {
            this.task = task;
            this.even = even;
        }

        public void run() {
            while (!task.isDone()) {
                while (even ^ evenHasNextTurn);
                if (task.isDone()) {
                    return;
                }
                task.call();
                evenHasNextTurn = !even;
            }
        }
    }

    protected Runnable[] getRunnables(InterlockTask task) {
        return new Runnable[]{
            new Job(task, true), new Job(task, false)
        };
    }
}
```

This approach to thread communication is faster when the call() method completes quickly. Since we are doing a *busy wait*, we will burn up CPU cycles in the waiting thread. For a long call() method, one of your CPUs would be running at 100% polling a field value. This is not a good idea unless you have lots of spare CPUs that you do not know what to do with. However, if you want to just do a very task inside call(), then this approach is substantially faster.

Where this might be handy is if you have a lot of thread communication that you need to speed up. Some financial applications might find this information useful.

Caveat: Livelock (or "unbreakable hard spin")

If we modify the LockFreeInterlocker slightly to use a boolean XOR instead of bitwise, then we can cause the JVM to hang up. I called this a livelock, for want of a better term. When I demonstrated this to Doug Lea, he named it an "unbreakable hard spin". Cliff Click also had a look at it and thinks it might be because the server hotspot compiler is not inserting a safepoint at the right place. This problem occurs in OpenJDK / Sun JDK server HotSpot compilers running on at least 2 physical cores and from JDK 1.6.0_14 onwards. Code available on request.

Heinz

Issue 187 - Cost of Causing Exceptions

Author: Dr. Heinz M. Kabutz

Date: 2010-08-31

Category: Performance

Java Versions: Java 6

Abstract:

Many years ago, when hordes of C++ programmers ventured to the greener pastures of Java, some strange myths were introduced into the language. It was said that a "try" was cheaper than an "if" - when nothing went wrong.

Welcome to the 187th issue of **The Java(tm) Specialists' Newsletter**. Summer is rapidly ending here in Crete. We have not had a drop of rain since June, but the temperature is now a chilly 28 degrees Celsius. I will need to soon check that my central heating system is operational. Our kids are still on holiday for a couple of weeks, having broken up for their summer break in the middle of June. If I ever had a chance be a child again, I would choose Greece as the place to grow up. Sun, sea and every three years on holiday for one (on average, not counting strikes by teachers or pupils).

I spent my afternoon reading through our **club forum**. I wanted to see where I could add to the discussions. What I found interesting was how civil and polite all the posts were. Not a single emotional outburst or personal attack. The questions were also of a high technical standard. It is great to see that the club is becoming a valuable resource for our members. We have now added an annual payment option for those who would like to sign up. It works out at \$49.95 per month, rather than \$69.95. (I had considered offering the annual payment option when I first started, but was not sure whether the club would ever take off. Now that it is firmly established, I am happy to give a yearly payment option with a nice discount.)

Cost of Causing Exceptions

A few weeks ago, **Herman Lintvelt** sent me a discussion he had with some of the Java programmers he was mentoring. One of the programmers had read that a try/catch was for free and so he was doing things like:

```
try {
    return policy.calculateInterest();
} catch(NullPointerException ex) {
    // policy object was null
    return 0.0f;
```

```
}
```

instead of using an if-else construct:

```
if (policy != null) {  
    return policy.calculateInterest();  
} else {  
    // policy object was null  
    return 0.0f;  
}
```

This led to Java code that was hard to understand. However, it was equally difficult to persuade Herman's prodigy that we should not write such code. After all, he kept pointing out that **try** was faster than **if**.

In this newsletter, we will try to persuade you that it is a bad coding practice to cause exceptions unnecessarily. We will then explain why these two approaches take almost the same amount of time.

Causing Exceptions: Style

Here are some reasons why it is better to check conditions rather than cause exceptions:

1. Exceptions should be used to indicate an exceptional or error condition. We should not use them to change the control flow of our program. When we use them to avoid an if-else for null, we are effectively using them to steer the control flow of our program. This makes it difficult to understand the code.
2. It is not at all clear that the `NullPointerException` is happening due to `policy` being null. What if the method `calculateInterest()` causes a `NullPointerException` internally due to a programming bug? All we would see is that we calculate zero interest, hiding a potentially serious error.
3. Debuggers are often set up to pause whenever an exception is generated. By causing exceptions in our code, we make debugging of our code almost impossible.

This list is by no means complete. There are many other reasons why we should not code like this. In fact, it would be better to use the Null Object Pattern instead of testing for null, that way the behaviour for when policy is null is well defined. Something like:

```
return policy.calculateInterest();
```

Where policy might be an instance of NullPolicy:

```
public class NullPolicy implements Policy {
    public float calculateInterest() {
        return 0.0f;
    }
}
```

Performance

The first time I saw this strange construct was in 1999, whilst mentoring C++ programmers on this new wonder called "Java". In order to make Java run faster, they wrote equals() methods that assumed everything would be fine, returning **false** when a RuntimeException occurred.

For many years, it was faster to just do a try, rather than an if statement (unless of course an exception occurred, in which case it was much much slower). Even though the code was ugly, it did perform a tiny little bit faster.

Constructing exceptions is very expensive, we know that. For most objects, the cost of construction in terms of time complexity is constant. With exceptions, it is related to the call stack depth. Thus it is more expensive to generate exceptions deep down in the hierarchy.

However, a few months ago, I tried to demonstrate to some Java Master students how costly it really was to construct exceptions. For some reason, my test did not work. It in fact demonstrated that it is extremely fast to cause exceptions. We had run out of time for that day, so I was unable to investigate the exact reasons for our weird results.

Herman also tried to write some code that clearly demonstrated that it was much slower to

occasionally cause exceptions, rather than use an if-else statement all the time. In his code, he had however used Math.random() to decide whether an object would be null or not. The call to Math.random() swamped the rest of his results, so that he could not get clear results.

I rewrote his test slightly by creating an array of random objects and then calling the methods on the objects. According to some factor, I would have a certain percentage of **null** values in the array, which would occasionally cause NullPointerException. I expected the cost of creating the exceptions to be much greater than the cost of the if statements. However, it turns out that the speed for both was roughly the same.

After causing the same NullPointerException a number of times, Java starts returning the same exception instance to us. Here is a piece of code that demonstrates this nicely:

```
import java.util.*;

public class DuplicateExceptionChecker {
    private final IdentityHashMap<Exception, Boolean> previous =
        new IdentityHashMap<Exception, Boolean>();

    public void handleException(Exception e) {
        checkForDuplicates(e);
    }

    private void checkForDuplicates(Exception e) {
        Boolean hadPrevious = previous.get(e);
        if (hadPrevious == null) {
            previous.put(e, false);
        } else if (!hadPrevious) {
            notifyOfDuplicate(e);
            previous.put(e, true);
        }
    }

    public void notifyOfDuplicate(Exception e) {
        System.err.println("Duplicate Exception: " + e.getClass());
        System.err.println("count = " + count(e));
        e.printStackTrace();
    }

    private int count(Exception e) {
        int count = 0;
        Class exceptionType = e.getClass();
        for (Exception exception : previous.keySet()) {
            if (exception.getClass() == exceptionType) {
                count++;
            }
        }
    }
}
```

```
        }  
    }  
    return count;  
}  
}
```

We would expect to eventually run out of memory, as obviously all new exceptions would be brand new objects? I was surprised to discover that this was not true for the server HotSpot compiler. After a relatively short while, it began returning the same exception instance, with an empty stack trace. You might have seen empty stack traces in your logs. This is why they occur. Too many exceptions are happening too closely together and eventually the server HotSpot compiler optimizes the code to deliver a single instance back to the user.

Here is a class that uses the `DuplicateExceptionChecker` to check for `NullPointerException` duplicates. What we do is occasionally set the array element to `null`, which causes the `NullPointerException` when we call `randomObjects[i].toString()`.

```
import java.util.*;  
  
public class NullPointerTest extends DuplicateExceptionChecker {  
    private final Object[] randomObjects =  
        new Object[1000 * 1000];  
  
    private final String[] randomStrings =  
        new String[1000 * 1000];  
  
    public static void main(String[] args) {  
        NullPointerTest npt = new NullPointerTest();  
        npt.fillArrays(0.01);  
        npt.test();  
    }  
  
    public void notifyOfDuplicate(Exception e) {  
        super.notifyOfDuplicate(e);  
        System.exit(1);  
    }  
  
    private void fillArrays(double probabilityObjectIsNull) {  
        Random random = new Random(0);  
        for (int i = 0; i < randomObjects.length; i++) {  
            if (random.nextDouble() < probabilityObjectIsNull) {  
                randomObjects[i] = null;  
            }  
        }  
    }  
}
```

```
        } else {
            randomObjects[i] = new Integer(i);
        }
    }
    Arrays.fill(randomStrings, null);
}

private void test() {
    for (int i = 0; i < 100; i++) {
        for (int j = 0; j < randomObjects.length; j++) {
            try {
                randomStrings[j] = randomObjects[j].toString();
            } catch (NullPointerException e) {
                randomStrings[j] = null;
                handleException(e);
            }
        }
    }
}
```

After a short while, I see output such as:

```
Duplicate Exception: class java.lang.NullPointerException  
count = 228  
java.lang.NullPointerException
```

This is true also for other exceptions that you could cause with the JVM. For example, the `ClassCastException`:

```
import java.util.*;

public class ClassCastTest extends DuplicateExceptionChecker {

    private final Object[] randomObjects =
        new Object[1000 * 1000];

    private final String[] randomStrings =

```

```

new String[1000 * 1000];

public static void main(String[] args) {
    ClassCastTest npt = new ClassCastTest();
    npt.fillArrays(0.01);
    npt.test();
}

public void notifyOfDuplicate(Exception e) {
    super.notifyOfDuplicate(e);
    System.exit(1);
}

private void fillArrays(double probabilityObjectIsNull) {
    Random random = new Random(0);
    for (int i = 0; i < randomObjects.length; i++) {
        if (random.nextDouble() < probabilityObjectIsNull) {
            randomObjects[i] = new Float(i);
        } else {
            randomObjects[i] = new Integer(i);
        }
    }
    Arrays.fill(randomStrings, null);
}

private void test() {
    for (int i = 0; i < 100; i++) {
        for (int j = 0; j < randomObjects.length; j++) {
            try {
                randomStrings[j] = ((Integer)randomObjects[j]).toString();
            } catch (ClassCastException e) {
                randomStrings[j] = null;
                handleException(e);
            }
        }
    }
}
}

```

We can see that this also causes an exception that eventually is replaced with a single instance exception.

Duplicate Exception: class java.lang.ClassCastException

```
count = 263
java.lang.ClassCastException
```

Lastly we can see that even an `ArrayIndexOutOfBoundsException` is eventually replaced with a single instance without a stack trace:

```
import java.util.*;

public class ArrayBoundsTest extends DuplicateExceptionChecker {
    private static final Object[] randomObjects =
        new Object[1000 * 1000];

    private static final int[] randomIndexes =
        new int[1000 * 1000];

    private static final String[] randomStrings =
        new String[1000 * 1000];

    public static void main(String[] args) {
        ArrayBoundsTest test = new ArrayBoundsTest();
        test.fillArrays(0.01);
        test.test();
    }

    public void notifyOfDuplicate(Exception e) {
        super.notifyOfDuplicate(e);
        System.exit(1);
    }

    private void fillArrays(double probabilityIndexIsOut) {
        Random random = new Random(0);
        for (int i = 0; i < randomObjects.length; i++) {
            randomObjects[i] = new Integer(i);
            randomIndexes[i] = (int) (Math.random() * i);
            if (random.nextDouble() < probabilityIndexIsOut) {
                randomIndexes[i] = -randomIndexes[i];
            }
        }
        Arrays.fill(randomStrings, null);
    }

    private void test() {
        for (int i = 0; i < 100; i++) {
```

```
        for (int j = 0; j < randomObjects.length; j++) {
            try {
                int index = randomIndexes[j];
                randomStrings[index] = randomObjects[index].toString();
            } catch (ArrayIndexOutOfBoundsException e) {
                randomStrings[j] = null;
                handleException(e);
            }
        }
    }
}
```

The question still remains - what is faster? I will not answer that question in this newsletter. Since we do not have the cost of object creation during the exception, the number of instructions would be roughly the same.

There is thus a difference between *causing* and *creating* exceptions. When we create them ourselves, they are very expensive to initialize due to the `fillInStackTrace()` method. But when we cause them inside the Java Virtual Machine, they may end up eventually not costing anything, depending on the exception caused. This is an optimization that we could of course add to our own code, but just remember that the stack trace is one of the most important parts of the exception. Leave that out and you might as well not throw it.

History Lesson

When I discovered this new feature, I was amazed as to how clever the Sun engineers had been to sneak this into the very latest Java 6 server HotSpot. Wanting to find out at exactly which point this was added, I went back in time and tried earlier versions of Java 6, then Java 5, 1.4.2, 1.4.1 and 1.4.0. Imagine my surprise when every single version that I tried had this feature? It was actually a bit depressing that the JVM has been doing this since at least February 2002 and that I did not know about it. I have also never read about this anywhere.

Turning Off Fast Exceptions

Ervin Varga sent me a JVM option that you can use to control this behaviour in the JVM, if you ever need to. Use the `-XX:-OmitStackTraceInFastThrow` to turn it off.

Thank you very much for reading this newsletter. I hope you enjoyed it.

Heinz

Issue 184 - Deadlocks through Cyclic Dependencies

Author: Dr. Heinz M. Kabutz

Date: 2010-06-04

Category: Concurrency

Java Versions: 1.2, 1.3, 1.4, 1.5, 6+

Abstract:

A common approach to ensuring serialization consistency in thread safe classes such as Vector, Hashtable or Throwable is to include a synchronized writeObject() method. This can result in a deadlock when the object graph contain a cyclic dependency and we serialize from two threads. Whilst unlikely, it has happened in production.

Welcome to the 184th edition of **The Java(tm) Specialists' Newsletter**, sent to you from Chorafakia, where the birds are singing outside and the blue sea smiles at me in the distance (not too much of a distance though :-)). On Wednesday my wife and I visited the police station, where I was asked to hand over my driver's license for two months. The **Law of Cretan Driving** caught up with me, as I knew it would. I will tell you the whole story later this evening when we cover that law in our Master Course day 1.

In about 6.5 hours from now, we are running the first day of the Java Specialist Master Course as a welcome gift for members of our **Java Specialist Club**. If you have not signed up yet, please do. I would not want you to miss out on this opportunity. And yes, it is completely free for club members. The course runs from 13:00-21:00 GMT today on June 4th. Once you have **signed up to the club**, please find the registration link by **visiting our club forum**. Please note that the webinar requires either Mac OS X or Windows. Linux is currently not supported.

Deadlocks through Cyclic Dependencies

In my **last newsletter**, I asked readers why Vector had a writeObject() method that just did the default, without a readObject() method. The answer is to simply make the writeObject() method synchronized. The quiz was easy, I admit.

One of my readers pointed out that in another JVM implementation, Vector had been coded specifically to avoid a deadlock situation that was found in production. Instead of making the entire method synchronized, they first cloned the Vector with the underlying Object[], but not the elements inside, and then wrote those out in a synchronized block.

This led me to the question - could I generate a deadlock by writing out two Vectors with a cyclic dependency from two threads? Turns out it was easy as pie:

```

import java.io.*;
import java.util.*;
import java.util.concurrent.*;

public class VectorSerializationDeadlock {
    public static void main(String[] args) {
        ExecutorService pool = Executors.newFixedThreadPool(2);

        final Vector[] vecs = {
            new Vector(),
            new Vector(),
        };
        vecs[0].add(vecs[1]);
        vecs[1].add(vecs[0]);

        for (int i = 0; i < 2; i++) {
            final int threadNumber = i;
            pool.submit(new Callable() {
                public Object call() throws Exception {
                    for (int i = 0; i < 1000 * 1000; i++) {
                        ObjectOutputStream out = new ObjectOutputStream(
                            new NullOutputStream()
                        );
                        out.writeObject(vecs[threadNumber]);
                        out.close();
                    }
                    System.out.println("done");
                    return null;
                }
            });
        }
    }

    public class NullOutputStream extends OutputStream {
        public void write(int b) throws IOException {
        }
    }
}

```

After a very short time, this causes a deadlock on the Sun's Java Virtual Machine (JVM). On other JVMs, this might not cause a deadlock, as they specifically coded around it.

Update: Java 7 was modified to avoid this deadlock. However, it can still happen if you wrap a `LinkedList` with a `SynchronizedList`, such as you can see in my

SynchronizedListSerializationDeadlock example.

When I mentioned this in the **Java Specialist Club**, Olivier Croisier pointed out that there are lots of cases in the JDK with a synchronized writeObject() method:

- java.beans.beancontext.BeanContextServicesSupport
- java.beans.beancontext.BeanContextSupport
- java.io.File
- java.lang.StringBuffer
- java.lang.Throwable
- java.net.Inet6Address
- java.net.SocketPermission
- java.net.URL
- java.util.Hashtable
- java.util.PropertyPermission
- java.util.Vector
- javax.security.auth.kerberos.DelegationPermission

He even managed to cause a deadlocks with Throwable, although I would argue that a cyclic dependency in Throwables would be a bug. If you try to print the stack space you will get a StackOverflowError. All he did was replace my Vectors with Throwables:

```
Throwable t1 = new Throwable("t1");
Throwable t2 = new Throwable("t2", t1);
t1.initCause(t2);

final Throwable[] vecs = {t1,t2};
```

My Vector example is contrived, meaning that it is extremely unlikely that with a simple object graph you would have two vectors that contain one another. However, with a complicated data structure, it is entirely possible that this could happen. In fact, it has happened to someone in production, which is why the "other" JVM had to code around it specifically.

Synchronized List

Another interesting point is that the List returned by the Collections.synchronizedList() method does not protect itself against concurrent updates in the writeObject() method. Collections.synchronizedCollection returns a class that does also use the synchronized

writeObject() approach. In the case of the Synchronized Collection, we might quite easily also cause deadlocks on the write, I imagine, though I have not tried that out.

Here is an example of how we can cause a ConcurrentModificationException with a Synchronized List. This does not happen with the Synchronized Collection.

```

import java.io.*;
import java.util.*;

public class MangledSynchronizedList {
    public static void main(String[] args) {
        final List<String> synchList = Collections.synchronizedList(
            new ArrayList<String>());
        Collections.addAll(synchList, "hello", "world");
        Thread tester = new Thread() {
            { setDaemon(true); }
            public void run() {
                while (true) {
                    synchList.add("hey there");
                    synchList.remove(2);
                }
            }
        };
        tester.start();

        while (true) {
            try {
                ObjectOutputStream out = new ObjectOutputStream(
                    new NullOutputStream()
                );
                for (int i = 0; i < 100 * 1000; i++) {
                    out.writeObject(synchList);
                }
                out.close();
            } catch (IOException e) {
                e.printStackTrace();
                break;
            }
        }
    }
}

```

After a short while, we see ConcurrentModificationException.

Kind regards

Heinz

Issue 179 - Escape Analysis

Author: Dr. Heinz M. Kabutz

Date: 2009-12-30

Category: Performance

Java Versions: 5+

Abstract:

Escape analysis can make your code run 110 times faster - if you are a really really bad programmer to begin with :-) In this newsletter we look at some of the places where escape analysis can potentially help us.

A hearty welcome to the 179th edition of **The Java(tm) Specialists' Newsletter**, sent to you from the beautiful warm and sunny island of Crete. Whilst my wife and oldest daughter are spending a week shivering in England, I'm left playing mom and dad back in Crete with our other two. Rather challenging finding time to research the topic for this month amongst cooking and various domestic tasks. If I did this job permanently I'd lose 20 pounds in the first month.

So, before we end 2009, here is one more little newsletter for all the diehard Java programmers who are still sitting behind their desks despite the many holidays.

Escape Analysis

Escape analysis has been hailed as a solution to GC problems for the last few years. Here are some articles written that explain what it is and why it can help us.

- **Sun Microsystems, 2009: Java SE 6u14 Update Release Notes** From the release notes: *The -XX:+DoEscapeAnalysis option directs HotSpot to look for objects that are created and referenced by a single thread within the scope of a method compilation. Allocation is omitted for such non-escaping objects, and their fields are treated as local variables, often residing in machine registers. Synchronization on non-escaping objects is also elided.*
- **Dan Dyer, 2009: Escape Analysis in Java 6 Update 14 - Some Informal Benchmarks**
- **Wolfgang Laun, 2009:** Pointed me to the latest flags in update 14 in a private email.
- **Tinou Bao, 2009: Lock Coarsening, Biased Locking, Escape Analysis for Dummies** (my favourite article on the subject)
- **Jeroen Borgers, 2007: Did escape analysis escape from Java 6?**
- **Brian Goetz, 2005: Java theory and practice: Urban performance legends, revisited**

It is easy to be confused by the results of the escape analysis flag since it does two things. It can omit constructing objects that do not escape, even keeping them in CPU registers. However, it also avoids synchronization on non-escaping objects. This can skew the results of microbenchmarks to make it seem that escape analysis is better than it really is.

One way to tell the difference is to log the GC output. If a benchmark runs faster with escape analysis turned on and the GC output (-Xloggc:file.gc) is the same, it is most likely the result of lock eliding.

Writing a benchmark to specifically test escape analysis is rather difficult. After more than a dozen dead ends, I came up with this one whilst taking my 3 year old for a walk around the countryside this morning. Escape analysis seems to give us the best performance gains with poorly written code, such as when we create lots of unnecessary objects. For example, here is a Calculator, that adds two ints together. Even though the method add() could be static, we wrote it as non-static to demonstrate the power of escape analysis.

```
public class Calculator {
    public int add(int i0, int i1) {
        return i0 + i1;
    }
}
```

In our poorly written CalculatorTest, we construct a new Calculator object every time we call the add method. You would hopefully agree that this is a harebrained way of coding Java. We use the return value of the calculation to ensure that the entire methods are not optimized away.

```
public class CalculatorTest {
    public static void main(String[] args) {
        long time = System.currentTimeMillis();
        long grandTotal = 0;
        for (int i = 0; i < 100000; i++) {
            grandTotal += test();
        }
        time = System.currentTimeMillis() - time;
        System.out.println("time = " + time + "ms");
        System.out.println("grandTotal = " + grandTotal);
    }
}
```

```

private static long test() {
    long total = 0;
    for (int i = 0; i < 10000; i++) {
        Calculator calc = new Calculator();
        total += calc.add(i, i/2);
    }
    return total;
}
}

```

I ran this with an old Java 1.6.0_03 32-bit server JVM on my Mac (Soylatte) and the latest 1.6.0_17 64-bit server JVM. Escape analysis has only been officially available since 1.6.0_14, so I could not use it for Soylatte.

| EA on | EA off | Old Java |
|-------|--------|----------|
| 2.2s | 8.6s | 7.0s |

With escape analysis turned off, we constructed 15 GB of heap objects. Even though GC was only 2% of CPU, we did create 1.3 GB per second. We call this *object churn*. The 32-bit Soylatte JVM constructed 7.6 GB of objects. Since the objects are half the size of the 64-bit objects, 8 bytes as opposed to 16 bytes, these values make sense. I tried to compress the OOPS using the new `-XX:+UseCompressedOops` flag, but that did not seem to make much difference. The objects are still each 16 bytes, according to the GC logs.

As I said before, escape analysis helps us to improve performance of poorly written code. Instead of taking 8.6 seconds, we only take 2.2 seconds. However, if we change our code to reuse the Calculator object, or even make the add method static, then the test executes in under a second.

An interesting application of escape analysis is with varargs. In one of my many experiments, I found that array objects only benefit from escape analysis when the size is 64 or less. So if you write really really bad code with more than 64 arguments for a varargs call, then your program will slow down to a crawl. Here is some sample code:

```

public class VarArgsTest {
    public static void main(String[] args) {

```

```

long time = System.currentTimeMillis();
long grandTotal = 0;
for (int i = 0; i < 100000; i++) {
    grandTotal += test();
}
time = System.currentTimeMillis() - time;
System.out.println("time = " + time + "ms");
System.out.println("grandTotal = " + grandTotal);
}

private static long test() {
    long total = 0;
    for (int i = 0; i < 10000; i++) {
        total += test(
            i, 1, 2, 3, 4, 5, 6, 7, 8, 9,
            i, 1, 2, 3, 4, 5, 6, 7, 8, 9,
            i, 1, 2, 3, 4, 5, 6, 7, 8, 9,
            i, 1, 2, 3, 4, 5, 6, 7, 8, 9,
            i, 1, 2, 3, 4, 5, 6, 7, 8, 9,
            i, 1, 2, 3, 4, 5, 6, 7, 8, 9,
            i, 1, 2, 3, 4, 5, 6, 7, 8, 9,
            i, 1, 2, 3
        );
    }
    return total;
}

public static int test(int... args) {
    return args[0] + args.length;
}
}

```

When we run this with 64 parameters, we get the following results, where escape analysis just made our code 110x faster:

| EA on | EA off | Old Java |
|-------|--------|----------|
| 1.2s | 140s | 120s |

Since escape analysis only seems to work with arrays of length 64 or less, if we add a single parameter to the method call above, it slows down to a crawl:

| EA on | EA off | Old Java |
|-------|--------|----------|
| 150s | 150s | 120s |

We will probably never write such bad code, taking 65 parameters. However, knowing that there is special treatment for arrays of length 64 or less means that we need to take that into account when writing our benchmarks. For example, adding three Strings together using a StringBuilder or StringBuffer is only sensible if the total length can sometimes exceed 64 characters.

Let's apply the knowledge of vararg improvements into a better Calculator, now called CalculatorVarArgs:

```
public class CalculatorVarArgs {
    public int add(int... is) {
        if (is.length == 0) throw new IllegalArgumentException();
        if (is.length == 1) return is[0];
        if (is.length == 2) return is[0] + is[1];
        if (is.length == 3) return is[0] + is[1] + is[2];
        if (is.length == 4) return is[0] + is[1] + is[2] + is[3];
        int total = 0;
        for (int i : is) {
            total += i;
        }
        return total;
    }
}
```

Note the rather convoluted syntax for dealing with cases where the length of the array is less than 5. I would've thought that the loops would be unrolled automatically. It does seem to make a rather large performance difference, so look at this if a vararg method is your bottleneck.

With a small test, we see that the varargs also does not create objects on the Java Heap as long as the array is small enough:

```

public class CalculatorVarargTest {
    public static void main(String[] args) {
        long time = System.currentTimeMillis();
        long grandTotal = 0;
        for (int i = 0; i < 100000; i++) {
            grandTotal += test();
        }
        time = System.currentTimeMillis() - time;
        System.out.println("time = " + time + "ms");
        System.out.println("grandTotal = " + grandTotal);
    }

    private static long test() {
        long total = 0;
        for (int i = 0; i < 10000; i++) {
            CalculatorVarArgs calc = new CalculatorVarArgs();
            total += calc.add(i, i/2);
        }
        return total;
    }
}

```

As you can see in our results, the varargs makes a difference when escape analysis is turned off, but otherwise not.

| | | |
|-------|--------|----------|
| EA on | EA off | Old Java |
| 2.2s | 23s | 18s |

Java SciMark 2.0

It's quite interesting running the **scimark benchmark** using the various Escape Analysis settings. The biggest difference is with the Monte Carlo calculation. On my laptop, it runs in 410 with EA on and 300 with EA off. However, there are hardly any objects collected, so my suspicion is that performance improvement is coming from synchronization eliding.

Real-Life Escape Analysis Improvements

Just because some microbenchmarks run 110x faster, does not mean that our real application code is going to perform better, unless we are really bad programmers.

On the other hand, perhaps now with escape analysis and lock eliding in place, we have lowered the bar of what a good programmer should be able to figure out? For example, as Jeroen Borges correctly points out in his article, `StringBuilder` is now obsolete, after just one Java version. The benefit we had with `StringBuilder` was as an unsynchronized version of `StringBuffer`. However, with lock eliding we should not need to concern ourselves with this anymore. Even `Vector` might become fashionable again.

Non-Escaping Object Storage

The Java release notes indicate that non-escaping objects are treated as local variables, maybe even being stored in machine registers. However, I have not managed to write a benchmark that demonstrates this. We would expect that if we use escape analysis that we would run out of stack space more quickly. However, that does not seem to be the case, even if we have larger objects, such as `int[64]`. It would be interesting to see a benchmark that shows how or where we can run out of resources differently with escape analysis enabled. A job for another day ...

My son asked me what type of weather we can expect tomorrow in Crete, considering that winter has officially started. Looks like 26 Celsius and sunny. Just thought you'd like to know that information :-)))

Kind regards

Heinz

Issue 178b - WalkingCollection Generics

Author: Dr. Heinz M. Kabutz

Date: 2009-11-17

Category: Language

Java Versions: 5+

Abstract:

Generics can be used to further improve the WalkingCollection, shown in our previous newsletter.

When I wrote the WalkingCollection, I was not very happy that PrintProcessor was defined with a generic type, since it could always be called on Object. I was also not happy with the WalkingCollection.iterate() method signature as it was too restrictive. At the very least, it should have been defined with <? super E> type parameter. One of my subscribers, Per Claesson, a Java specialist from Gothenburg, Sweden, picked up on this issue, so together we improved the code a bit.

We are planning an informal get-together next week Thursday evening (26th Nov 09) at 18:30 in Montreal (Quebec), where I will be presenting my "Secrets of Concurrency" talk. Seating is limited. There is no cover charge to attend. **Please let me know if you are interested.**

WalkingCollection Generics

In the [WalkingCollection newsletter](#), we looked at how we could control the iteration from within the Collection, thus making concurrency easier. However, the interface was more restrictive than necessary. Instead, we could have written the following iterate() method:

```
public void iterate(Processor<? super E> processor) {
    rwlock.readLock().lock();
    try {
        iterating.set(true);
        for (E e : wrappedCollection) {
            if (!processor.process(e)) break;
        }
    } finally {
        iterating.set(false);
        rwlock.readLock().unlock();
    }
}
```

```
}
```

This would allow us to write the PrintProcessor to be defined on Object, rather than on the same generic type parameter as our WalkingCollection. Infact, any Processor can be defined on a super type of the WalkinCollection's type parameter.

```
public class PrintProcessor implements Processor<Object> {
    public boolean process(Object o) {
        System.out.println("">>>> " + o);
        return true;
    }
}
```

The CompositeProcessor is now changed to contain a collection of super classes of E, as follows:

```
import java.util.*;

public class CompositeProcessor<E>
    implements Processor<E> {
    private final List<Processor<? super E>> processors =
        new ArrayList<Processor<? super E>>();

    public void add(Processor<? super E> processor) {
        processors.add(processor);
    }

    public boolean process(E e) {
        for (Processor<? super E> processor : processors) {
            if (!processor.process(e)) return false;
        }
        return true;
    }
}
```

Instead of constructing the PrintProcessor with a type parameter, we simply construct it with Object. The rest of the WalkingCollectionTest class remains the same:

```

public class WalkingCollectionTest {
    public static void main(String[] args) {
        WalkingCollection<Long> ages = new WalkingCollection<Long>(
            new java.util.ArrayList<Long>()
        );

        ages.add(10L);
        ages.add(35L);
        ages.add(12L);
        ages.add(33L);

        PrintProcessor pp = new PrintProcessor();
        ages.iterate(pp);

        AddProcessor<Long> ap = new AddProcessor<Long>();
        ages.iterate(ap);
        System.out.println("ap.getTotal() = " + ap.getTotal());

        // composite
        System.out.println("Testing Composite");
        ap.reset();

        CompositeProcessor<Long> composite =
            new CompositeProcessor<Long>();
        composite.add(new Processor<Long>() {
            private long previous = Long.MIN_VALUE;
            public boolean process(Long current) {
                boolean result = current >= previous;
                previous = current;
                return result;
            }
        });
        composite.add(ap);
        composite.add(pp);
        ages.iterate(composite);
        System.out.println("ap.getTotal() = " + ap.getTotal());
    }
}

```

Paul Cowan sent me another interesting piece of code that is related to what we have done here. It compiles in Java 6, but not in Java 5:

```

import java.util.*;
import java.util.concurrent.*;

public class CompilesOnJava6Not5 {
    public static void main(String args[]) throws Exception {
        List<WorkerTask> tasks = new ArrayList<WorkerTask>();
        ExecutorService executor = Executors.newCachedThreadPool();
        executor.invokeAll(tasks); // fails in Java 5, but not in 6
    }

    public static class WorkerTask implements Callable<String> {
        public String call() {
            return "some work";
        }
    }
}

```

The reason is a change in the signature of the invokeAll() method. Previously, it would only accept `Collection<Callable<T>>`. However, in Java 6 they changed that to also accept `Collection<Callable<? extends T>>`:

```

<T> List<Future<T>> invokeAll(Collection<? extends Callable<T>> tasks)
    throws InterruptedException;

```

As a result, we can now upcast the `List<WorkerTask>` to `Collection<? extends Callable<T>>`.

Kind regards

Heinz

Issue 178 - WalkingCollection

Author: Dr. Heinz M. Kabutz

Date: 2009-11-14

Category: Tips and Tricks

Java Versions: 5+

Abstract:

We look at how we could internalize the iteration into a collection by introducing a Processor interface that is applied to each element. This allows us to manage concurrency from within the collection.

Welcome to the 178th issue of **The Java(tm) Specialists' Newsletter**, sent to you from Chania on the beautiful island of Crete. A few days ago, we spontaneously decided to make a spit roast with some friends from Denver. When we bought our house, it came with a fire pit and hooks for a spit, so recently I invested in a motor for turning the meat around. So here we were, sitting outside in our shorts at ten in the night in Europe in November, watching the fat drip onto the coals. The spit roast must be one of the easiest way of cooking the meat, you simply put it on and then sit there watching it get ready, whilst chatting about politics and philosophy. Now why did I not buy this equipment when I still lived in South Africa? Do let me know if you come to Chania in Crete and we'll have a feast together, assuming that I am home ... :-)

I will be in Montreal from the 22-26 Nov 2009 and San Jose from the 6-11 Dec 2009. By the time I get home from those trips, I will have done 93 flights in this year! **Let me know please if you are interested in attending our Java Specialist Master Course in San Jose in December 6-11.** The course is gaining momentum and we expect it to sell out. We do not know when we will be able to run it again in San Jose.

WalkingCollection

The Iterator design pattern, described in the Gang-of-Four book, lists more implementation options than are available in the `java.util.*` classes. One of the questions asked is: who controls the iteration? In the standard Java Iterator idiom, the client controls it, for example like this:

```
Iterator<String> it = names.iterator();
while(it.hasNext()) {
    String name = it.next();
    System.out.println(name);
}
```

When we use the Java 5 for-each loop, the iteration is still controlled by the client, even though we do not explicitly call the `next()` method.

```
for(String name : names) {
    System.out.println(name);
}
```

Concurrency and iteration is tricky. The approach in Java 1 was to synchronize everything. However, we could still iterate through the vector whilst it was being changed, resulting in unpredictable behaviour. In the Java 2 idiom, collections were by default unsynchronized. We would thus need to lock the entire collection whilst we were iterating to avoid a `ConcurrentModificationException`. In the Java 5 idiom, we see classes like `CopyOnWriteArrayList`, where the underlying array is copied every time it is modified. This is expensive, but guarantees lock-free iteration.

In this alternative approach, we control the iteration from within the collection, making concurrency easier.

We define an interface `Processor` with a single method `process()` that returns a boolean. The return value is used to determine whether we should stop iterating. Our collection would then apply this processor to every element in the collection. For example, here is my `Processor` interface with my collection's `iterate()` method.

```
public interface Processor<E> {
    boolean process(E e);
}

// in our collection class
public void iterate(Processor<E> processor) {
    for (E e : wrappedCollection) {
        if (!processor.process(e)) break;
    }
}
```

We can then use the `ReadWriteLock` to differentiate between methods that modify the state of the collection and those that do not. For example, the `add(E)` method would lock on a write lock, but the `size()` method only on a read lock.

The easiest way to implement a collection is to extend `AbstractCollection`. All we have to do is implement the `size()` and `iterate()` methods, like so:

```
import java.util.*;

public class SimpleCollection<E> extends AbstractCollection<E> {
    private final E[] values;

    public SimpleCollection(E[] values) {
        this.values = values.clone();
    }

    public Iterator<E> iterator() {
        return new Iterator<E>() {
            private int pos = 0;
            public boolean hasNext() {
                return pos < values.length;
            }

            public E next() {
                // Thanks to Volker Glave for pointing out that we need
                // to throw a NoSuchElementException if we try to go past
                // the end of the array.
                if (!hasNext()) throw new NoSuchElementException();
                return values[pos++];
            }

            public void remove() {
                throw new UnsupportedOperationException();
            }
        };
    }

    public int size() {
        return values.length;
    }
}
```

This is an immutable collection, but by implementing the `add()` and `Iterator.remove()`

methods, we could make it into a normal mutable collection. Methods like `isEmpty()` and `toString()` simply use our internal `iterator()` and `size()` methods.

```
public class SimpleCollectionTest {
    public static void main(String[] args) {
        String[] arr = {"John", "Andre", "Neil", "Heinz", "Anton"};
        SimpleCollection<String> names =
            new SimpleCollection<String>(arr);

        for (String name : names) { // works
            System.out.println(name);
        }

        System.out.println(names); // works

        System.out.println(names.isEmpty()); // works

        names.add("Dirk"); // throws UnsupportedOperationException

        names.remove("Neil"); // throws UnsupportedOperationException
    }
}
```

In our WalkingCollection, we also use the concept of ReadWriteLocks to differentiate between read-only and read-write methods. We should be able to iterate with multiple threads at the same time, but only one thread may modify it at once. I am a bit wary of ReadWriteLocks due to the possibility of starvation, as described in [Newsletter #165](#).

An initial implementation of the WalkingCollection just overrides the minimum number of methods. If you want to use bulk update methods like `addAll()`, it would be more efficient to only acquire the lock a single time. However, I have not called the bulk update methods very often in my last 12.5 years of Java programming, so will leave that as an "exercise to the reader".

One of the issues with the ReentrantReadWriteLock implementation is that you can downgrade a write lock to a read lock, but you cannot upgrade the read to a write. You do not even get an exception, the thread just hangs up. In our implementation, we thus keep a ThreadLocal to indicate that we are busy iterating and check this before acquiring the write lock. An exception is better to see than for the thread to simply go into the BLOCKED state.

```
import java.util.*;
import java.util.concurrent.locks.*;

public class WalkingCollection<E>
    extends AbstractCollection<E> {
    private final static ThreadLocal<Boolean> iterating =
        new ThreadLocal<Boolean>() {
            protected Boolean initialValue() {
                return false;
            }
        };
    private final Collection<E> wrappedCollection;
    private final ReentrantReadWriteLock rwlock =
        new ReentrantReadWriteLock();

    public WalkingCollection(Collection<E> wrappedCollection) {
        this.wrappedCollection = wrappedCollection;
    }

    public void iterate(Processor<E> processor) {
        rwlock.readLock().lock();
        try {
            iterating.set(true);
            for (E e : wrappedCollection) {
                if (!processor.process(e)) break;
            }
        } finally {
            iterating.set(false);
            rwlock.readLock().unlock();
        }
    }

    public Iterator<E> iterator() {
        rwlock.readLock().lock();
        try {
            final Iterator<E> wrappedIterator =
                wrappedCollection.iterator();
            return new Iterator<E>() {
                public boolean hasNext() {
                    rwlock.readLock().lock();
                    try {
                        return wrappedIterator.hasNext();
                    } finally {
                        rwlock.readLock().unlock();
                    }
                }

                public E next() {
```

```
        rwlock.readLock().lock();
    try {
        return wrappedIterator.next();
    } finally {
        rwlock.readLock().unlock();
    }
}

public void remove() {
    checkForIteration();
    rwlock.writeLock().lock();
    try {
        wrappedIterator.remove();
    } finally {
        rwlock.writeLock().unlock();
    }
}
};

} finally {
    rwlock.readLock().unlock();
}
}

public int size() {
    rwlock.readLock().lock();
    try {
        return wrappedCollection.size();
    } finally {
        rwlock.readLock().unlock();
    }
}

public boolean add(E e) {
    checkForIteration();
    rwlock.writeLock().lock();
    try {
        return wrappedCollection.add(e);
    } finally {
        rwlock.writeLock().unlock();
    }
}

private void checkForIteration() {
    if (iterating.get())
        throw new IllegalMonitorStateException(
            "Cannot modify whilst iterating");
}
}
```

To use this, we define processors that can do something with the contents of the collection. For example, the `PrintProcessor` prints each element to the console:

```
public class PrintProcessor<E> implements Processor<E> {
    public boolean process(E o) {
        System.out.println("">>>> " + o);
        return true;
    }
}
```

The `AddProcessor` adds up the numbers in the collection and returns the total as a double:

```
public class AddProcessor<N extends Number>
    implements Processor<N> {
    private double total = 0;

    public boolean process(N n) {
        total += n.doubleValue();
        return true;
    }

    public double getTotal() {
        return total;
    }

    public void reset() {
        total = 0;
    }
}
```

We can even define a `CompositeProcessor` that aggregates several processors together:

```

import java.util.*;

public class CompositeProcessor<E>
    implements Processor<E> {
    private final List<Processor<E>> processors =
        new ArrayList<Processor<E>>();

    public void add(Processor<E> processor) {
        processors.add(processor);
    }

    public boolean process(E e) {
        for (Processor<E> processor : processors) {
            if (!processor.process(e)) return false;
        }
        return true;
    }
}

```

We can combine these in a test class that processes a bunch of numbers, as such:

```

public class WalkingCollectionTest {
    public static void main(String[] args) {
        WalkingCollection<Long> ages = new WalkingCollection<Long>(
            new java.util.ArrayList<Long>()
        );

        ages.add(10L);
        ages.add(35L);
        ages.add(12L);
        ages.add(33L);

        PrintProcessor<Long> pp = new PrintProcessor<Long>();
        ages.iterate(pp);

        AddProcessor<Long> ap = new AddProcessor<Long>();
        ages.iterate(ap);
        System.out.println("ap.getTotal() = " + ap.getTotal());

        // composite
        System.out.println("Testing Composite");
    }
}

```

```

ap.reset();

CompositeProcessor<Long> composite =
    new CompositeProcessor<Long>();
composite.add(new Processor<Long>() {
    private long previous = Long.MIN_VALUE;
    public boolean process(Long current) {
        boolean result = current >= previous;
        previous = current;
        return result;
    }
});
composite.add(ap);
composite.add(pp);
ages.iterate(composite);
System.out.println("ap.getTotal() = " + ap.getTotal());
}
}

```

Here is the output from our test code:

```

>>> 10
>>> 35
>>> 12
>>> 33
ap.getTotal() = 90.0
Testing Composite
>>> 10
>>> 35
ap.getTotal() = 45.0

```

One of the restrictions of the iterate() method is that we cannot modify the collection from within the process() methods. We can "downgrade" a write lock to a read lock, but not the other way round. During the iteration, we are holding the read lock. In our WalkingCollection, we throw an IllegalMonitorStateException when this happens:

```

public class WalkingCollectionBrokenTest {

```

```

public static void main(String[] args) {
    final WalkingCollection<String> names =
        new WalkingCollection<String>(
            new java.util.ArrayList<String>()
        );

    names.add("Maximilian");
    names.add("Constance");
    names.add("Priscilla");
    names.add("Evangeline");

    Processor<String> pp = new Processor<String>() {
        public boolean process(String s) {
            if ("Priscilla".equals(s)) names.remove(s);
            return true;
        }
    };
    names.iterate(pp);
}
}

```

We see the following output:

```

Exception in thread "main" java.lang.IllegalMonitorStateException:
    Cannot modify whilst iterating
    at WalkingCollection.checkForIteration(WalkingCollection.java:93)
    ... etc.

```

I created a course on Design Patterns before my daughter Connie was born. She is now 8 years old. Over the years, I added patterns and removed obsolete ones, but a surprising amount of material has survived 8 years of programming advances. Once a company has bought one of my Design Patterns courses, they usually end up wanting to train dozens (or even hundreds) of their developers, so it remains one of my best selling courses of all time.

[Click here for more information about the Design Patterns Course.](#)

Time to go out for a walk with my kids, so I hope you enjoyed this newsletter.

Kind regards

Heinz

Issue 176 - The Law of the Xerox Copier

Author: Dr. Heinz M. Kabutz

Date: 2009-09-22

Category: Tips and Tricks

Java Versions: 5+

Abstract:

Concurrency is easier when we work with immutable objects. In this newsletter, we define another concurrency law, The Law of the Xerox Copier, which explains how we can work with immutable objects by returning copies from methods that would ordinarily modify the state.

Welcome to the 176th issue of The Java(tm) Specialists' Newsletter, sent to you from Oslo in Norway. The **JavaZone** conference is wonderful, with at least one interesting topic per session. There are actually two other talks I'd like to attend when I am speaking tomorrow ...

Thanks for the feedback after my last newsletter, where I showed how to **construct objects without calling any constructor**. When I wrote the article, I felt it unnecessary to explicitly state that making a linkage to sun.* classes would be a problem for portability of your code, since that is rather obvious. So there, I've said it ... :-)

The Law of the Xerox Copier

In previous newsletters, we looked at a series of laws to make sense of Java concurrency. Just to remind you, here they are again. In this newsletter, I would like to add a new one to our list, the Law of the Xerox Copier.

1. [The Law of the Sabotaged Doorbell](#)
2. [The Law of the Distracted Spearfisherman](#)
3. [The Law of the Overstocked Haberdashery](#)
4. [The Law of the Blind Spot](#)
5. [The Law of the Leaked Memo](#)
6. [The Law of the Corrupt Politician](#)
7. [The Law of the Micromanager](#)
8. [The Law of Cretan Driving](#)
9. [The Law of Sudden Riches](#)
10. [The Law of the Uneaten Lutefisk](#)
11. [The Law of the Xerox Copier](#)

The Law of the Xerox Copier

Protect yourself by making copies of objects.

At the end of 2008, we found a beautiful house in Crete that we wanted to purchase. We only had one tiny little problem. Houses in general are kind of expensive and I didn't have all that much in my wallet. So we started the long process of begging banks for a loan, which involves giving them reams of paperwork.

You would imagine that it is safe to give a bank your documents, expecting them to look after them until they returned them. Well, the night before the bank was supposed to give back our paperwork, loan denied of course, they were attacked by a mob of angry protesters who completely trashed it, together with our documents. How often does an event like that happen? Fortunately, we had protected our original objects by only giving copies to the loan consultant.

Coming back to concurrency, it is usually a good idea to make your objects as immutable as possible. Instead of then changing the object, we would return a new copy containing the changes.

Immutable objects are always thread safe. We cannot have stale values, race conditions or early writes. According to [Brian Goetz](#), an immutable class is defined as the following:

- State cannot be modified after construction
- All the fields are final
- 'this' reference does not escape during construction

An example of an immutable class is String. Whenever we "modify" it, such as with the + operator or the toUpperCase() method. This causes additional work for the garbage collector, but concurrency is much easier.

In [Scala](#), we even have immutable lists. How could that work in Java? Here is an example:

```
import java.util.Iterator;

public class ImmutableList<E> implements Iterable<E> {
    private final Object[] elements;

    public ImmutableList() {
        this.elements = new Object[0];
    }

    private ImmutableList(Object[] elements) {
        this.elements = elements;
    }

    @Override
    public Iterator<E> iterator() {
        return new Iterator<E>() {
            int index = 0;

            @Override
            public boolean hasNext() {
                return index < elements.length;
            }

            @Override
            public E next() {
                if (!hasNext()) {
                    throw new NoSuchElementException();
                }
                return (E) elements[index];
            }
        };
    }
}
```

```

public int size() {
    return elements.length;
}

public ImmutableList<E> add(E e) {
    Object[] new_elements = new Object[elements.length + 1];
    System.arraycopy(elements, 0,
        new_elements, 0, elements.length);
    new_elements[new_elements.length - 1] = e;
    return new ImmutableList<E>(new_elements);
}

public Iterator<E> iterator() {
    return new Iterator<E>() {
        private int pos = 0;

        public boolean hasNext() {
            return pos < elements.length;
        }

        public E next() {
            return (E) elements[pos++];
        }

        public void remove() {
            throw new UnsupportedOperationException();
        }
    };
}
}

```

Making ArrayList immutable is probably going overboard, a better approach might be to just use the CopyOnWriteArrayList, which also gives us great concurrent performance.

We can use this ImmutableList in a more functional coding approach:

```

public class ImmutableListTest {
    public static void main(String[] args) {
        ImmutableList<String> ial =
            new ImmutableList<String>();
        ial = ial.add("Heinz").add("Max").add("Kabutz");
    }
}

```

```
for (Object obj : ial) {  
    System.out.println(obj);  
}  
}  
}
```

Making copies of objects has helped me out of a fix many times in the past, where rather complicated concurrency bugs would have been tricky to resolve otherwise.

Kind regards from Oslo

Heinz

Issue 174 - Java Memory Puzzle Explained

Author: Dr. Heinz M. Kabutz

Date: 2009-06-26

Category: Performance

Java Versions: Java 1.0 - 6.0

Abstract:

In this newsletter, we reveal the answer to the puzzle from last month and explain the reasons why the first class sometimes fails and why the second always succeeds.
Remember this for your next job interview ...

Welcome to the 174th issue of **The Java(tm) Specialists' Newsletter**. I've thoroughly enjoyed the last two weeks with my family in Crete, going to beaches, playing Age of Mythology with my two older kids and getting up early every morning to do exercise with my brother-in-law Costa, a professional gym instructor. As a result, I've done very little Java in the last two weeks, aside from answering the 400+ emails that I received as a result of last month's puzzle. Thank you all for participating in this puzzle - you can stop sending emails now ;-)

Java Memory Puzzle Explained

The [Initial JavaMemoryPuzzle](#) was based on a discussion that one of my readers, Dmitry Vyazelenko, had with some friends. They were arguing that you should always set local variables to `null` after you finish using them and were using code such as seen in our puzzle to prove their point. To understand why setting local variables to `null` is never necessary in the real world, we first need to understand why the first class fails.

Here is the JavaMemoryPuzzle code again:

```
public class JavaMemoryPuzzle {
    private final int dataSize =
        (int) (Runtime.getRuntime().maxMemory() * 0.6);

    public void f() {
    }
    byte[] data = new byte[dataSize];
}

byte[] data2 = new byte[dataSize];
```

```

    }

    public static void main(String[] args) {
        JavaMemoryPuzzle jmp = new JavaMemoryPuzzle();
        jmp.f();
    }
}

```

The easiest way to understand why it fails is to look at the disassembled class file, which we can generate using the `javap -c JavaMemoryPuzzle` command. I will show just the relevant parts:

```

public void f();
Code:
 0: aload_0
 1: getfield #6; //Field dataSize:I
 4: newarray byte
 6: astore_1
 7: aload_0
 8: getfield #6; //Field dataSize:I
11: newarray byte
13: astore_1
14: return

```

Instruction 0 loads the pointer **this** onto the stack. Instruction 1 reads the value of dataSize and leaves that on the stack. Instruction 4 creates the new **byte[]** and leaves a pointer to it on the stack. Instruction 6 now writes the pointer to the **byte[]** into stack frame position 1. Instruction 7 and 8 again load the dataSize value onto the stack. Instruction 11 is now supposed to create a new **byte[]**, but at this point, there is still a pointer to the old **byte[]** in the stack frame. The old **byte[]** can thus not be garbage collected, so the newarray byte instruction fails with an OutOfMemoryError.

The **data** variable is never used again after the object scope, but it is not necessarily cleared when we leave the block. Inside the byte code, there is no concept of code blocks as such. The generated code would be different if we did not have the code block, here is what would have been generated:

```
public void f() {
    byte[] data = new byte[dataSize];
    byte[] data2 = new byte[dataSize];
}
```

would result in byte code that does not reuse the stack frame 1, as we can see with instruction 13:

```
public void f();
Code:
 0: aload_0
 1: getfield #6; //Field dataSize:I
 4: newarray byte
 6: astore_1
 7: aload_0
 8: getfield #6; //Field dataSize:I
11: newarray byte
13: astore_2
14: return
```

The code block allows the compiler to generate code that reuses the stack frame for the next local variable and would be more akin to the following method f() :

```
public void f() {
    byte[] data = new byte[dataSize];
    data = new byte[dataSize];
}
```

When does JavaMemoryPuzzle not fail?

Several newsletter fans sent me an email pointing out that certain versions of the JVM, such as BEA JRockit and IBM's JVM, do not fail with the first version.

Others pointed out that it also does not fail with the new G1 collector, which you can use with

the following VM options: `-XX:+UnlockExperimentalVMOptions -XX:+UseG1GC`

Furthermore, Christian Glencross pointed out that when you compile the `JavaMemoryPuzzle` class with Java 1.0 or 1.1, it optimizes away the body of method `f()`, thus becoming:

```
public void f();
Code:
0: return
```

The `javac` compiler in older versions of Java was a lot more aggressive in its optimizations. It saw that we never used the fields and simply removed them from the code.

Joakim Sandström discovered that when you start the class with `-Xcomp`, it also never fails. Since Java 2, the JIT compiler takes the place of the static compiler for optimizing our code.

Christian Glencross also sent me code demonstrating that if you call method `f()` with a small `dataSize` often enough, then the JIT compiler will kick in and optimize the code. The large array construction then passes also. In this example, we call the `f()` method 100.000 times with a small `dataSize`. The HotSpot compiler will thus in all likelihood pick up that this method is being called a lot and then optimize it for us. Since the optimizing occurs in a separate thread, we need to call the method often enough so that we can be sure that the code has been optimized by the time we call it with the large `dataSize`. If you still get the `OutOfMemoryError`, just increase the number. Incidentally, you can also call the method fewer times, for example 10.000 times, and then sleep for a second, giving the JIT compiler time to optimize the code.

Make sure that you use the server HotSpot compiler, since the client compiler does not seem to do this optimization:

```
public class JavaMemoryPuzzleWithHotspotWarmup {
    private int dataSize = 0;

    public void f() {
        byte[] data = new byte[dataSize];
    }
}
```

```
byte[] data2 = new byte[dataSize];
}

public static void main(String[] args) {
    JavaMemoryPuzzleWithHotspotWarmup jmp =
        new JavaMemoryPuzzleWithHotspotWarmup();
    jmp.dataSize = 10;
    for (int i = 0; i < 1000 * 1000; i++) {
        jmp.f();
    }
    jmp.dataSize = (int) (Runtime.getRuntime().maxMemory() * 0.6);
}

jmp.f(); // probably no OutOfMemoryError
}
}
```

As we can see, the HotSpot compiler does a great job of optimizing our code even if we do not explicitly set local variables to `null` after we finish using them. In 12 years of coding Java, I have not once needed to null out local variables. The `JavaMemoryPuzzle` class is an edge case, from which we must not draw the wrong conclusions.

Why does the Polite version pass?

In our [previous newsletter](#), we then showed a "polite" version of the Java Memory Puzzle, which passes on all JVMs that I know of:

```
public class JavaMemoryPuzzlePolite {
    private final int dataSize =
        (int) (Runtime.getRuntime().maxMemory() * 0.6);

    public void f() {
        byte[] data = new byte[dataSize];
    }

    for(int i=0; i<10; i++) {
        System.out.println("Please be so kind and release memory");
    }
    byte[] data2 = new byte[dataSize];
}
```

```

public static void main(String[] args) {
    JavaMemoryPuzzlePolite jmp = new JavaMemoryPuzzlePolite();
    jmp.f();
    System.out.println("No OutOfMemoryError");
}
}

```

The majority of responses were incorrect and suggested that the for() loop either gave the GC time to do its work during the System.out.println() or that there was some obscure synchronization / JVM / voodoo happening at that point.

Some of my readers realised that it had nothing to do with the System.out.println and that a simple `int i = 0;` would suffice. If you declare *any* local variable immediately after the code block, you break the strong reference to the byte[] held in the stack frame 1 *before* you invoke the `new byte[]` the second time. Again, looking at the generated byte code of the polite puzzle explains this nicely:

```

public void f();
Code:
 0: aload_0
 1: getfield #6; //Field dataSize:I
 4: newarray byte
 6: astore_1
 7: iconst_0
 8: istore_1
 9: iload_1
10: bipush   10
12: if_icmpge 29
15: getstatic #7; //Field System.out
18: ldc      #8; //String Please be so kind and release memory
20: invokevirtual #9; //Method PrintStream.println
23: iinc    1, 1
26: goto    9
29: aload_0
30: getfield #6; //Field dataSize:I
33: newarray byte
35: astore_1
36: return

```

After Instruction 8 has completed, we do not have a strong reference to the first byte[] left,

so the GC can collect it at any time. The polite request "Please be so kind and release memory" was just to catch you out, and boy, was it successful :-)))

Heinz

Issue 173 - Java Memory Puzzle

Author: Dr. Heinz M. Kabutz

Date: 2009-05-28

Category: Performance

Java Versions: Java 1.0 - 6.0

Abstract:

In this newsletter we show you a puzzle, where a simple request causes memory to be released, that otherwise could not. Solution will be shown in the next newsletter.

Welcome to the 173rd issue of **The Java(tm) Specialists' Newsletter**, sent to you from Zürich in Switzerland. This is the 12th week I'm away from Crete this year alone, with another 2 weeks away before the middle of June. So out of 26 weeks in the first half of 2009, I will have been away for 14, which is more than 50%. Due to the economic crisis, I fully expected at least half of my engagements to get cancelled, but that did not happen. Our **Java Specialist Master Course** and **Design Patterns Course** are just way too popular. **HOWEVER**, as from the middle of June, I'll be on "holiday" in Crete for a few months, coinciding with my kids 14 week school holiday. This will give me more time to think and produce some nice newsletters. Also, I am talking to a publisher to perhaps put together a book of my most popular newsletters, so you will have something to take to the beach.

Java Memory Puzzle

Instead of telling you some mystery of Java memory, it is time for you to put on your thinking caps. I had a discussion a few weeks ago with one of my subscribers of whether you should **null** your local variables, to make things easier for the garbage collector. His understanding was that the local variables will be stored on the stack and thus popped off at the end of the method call anyway, so nulling them was a waste of time. In almost all situations, he is right. However, he had a class that did something most peculiar, something like this:

```
public class JavaMemoryPuzzle {
    private final int dataSize =
        (int) (Runtime.getRuntime().maxMemory() * 0.6);

    public void f() {
    {
        byte[] data = new byte[dataSize];
    }

    byte[] data2 = new byte[dataSize];
```

```

    }

    public static void main(String[] args) {
        JavaMemoryPuzzle jmp = new JavaMemoryPuzzle();
        jmp.f();
    }
}

```

When you run this you will always get an OutOfMemoryError, even though the local variable `data` is no longer visible outside of the code block.

So here comes the puzzle, that I'd like you to ponder about a bit. If you very politely ask the VM to release memory, then you don't get an OutOfMemoryError:

```

public class JavaMemoryPuzzlePolite {
    private final int dataSize =
        (int) (Runtime.getRuntime().maxMemory() * 0.6);

    public void f() {
        {
            byte[] data = new byte[dataSize];
        }

        for(int i=0; i<10; i++) {
            System.out.println("Please be so kind and release memory");
        }
        byte[] data2 = new byte[dataSize];
    }

    public static void main(String[] args) {
        JavaMemoryPuzzlePolite jmp = new JavaMemoryPuzzlePolite();
        jmp.f();
        System.out.println("No OutOfMemoryError");
    }
}

```

Why does this work? In my original newsletter, I asked my readers to send an answer. 400 emails later, I'd now prefer you to look at the [next newsletter](#) for the answer. Please [contact me via my website](#) if you have something new to add to the puzzle.

Heinz

Issue 171 - Throwing ConcurrentModificationException Early

Author: Dr. Heinz M. Kabutz

Date: 2009-03-09

Category: Tips and Tricks

Java Versions: Java 1.2 - 6.0

Abstract:

One of the hardest exceptions to get rid of in a system is the `ConcurrentModificationException`, which typically occurs when a thread modifies a collection whilst another is busy iterating. In this newsletter we show how we can fail on the modifying, rather than the iterating thread.

Welcome to the 171st issue of **The Java(tm) Specialists' Newsletter**, written at 30.000 feet en route to another [Java Specialist Master Course](#). The frequent traveler knows that an emergency exit seat in economy (coach) is often superior to what is on offer in business class on the shorter European flights. My dad was 1.94m (6'4") and when we traveled together, he always requested emergency exit seats. Fortunately my height stayed a manageable 1.88m, so I'm tall enough to beg for an emergency exit seat, but won't be crippled if I don't get one. I keep on telling my son Maxi that 1.88m is a perfect height, but he is adamant that he will overtake me in the next few years. Time will tell, but I fear he is right in his predictions.

My father took me along on some of his business trips. When I was 16, we almost pushed someone off the road by accident. My dad was rather large and also rather impatient. Driving the speed limit was just not an option, so he maneuvered his Mazda 323, with trailer, into oncoming traffic until he found a little gap, which, it turned out, was perfect for a Mazda 323 **without** trailer. A traffic cop pulled us over about thirty minutes later and the poor man who had almost been pushed into a ravine stopped behind us. However, they became most apologetic when my father got out of the car: A huge guy with a black beard and carrying a 9mm Browning on his hip. Looked a bit like Bud Spencer, just a bit taller. We were let off with a warning. My dad felt quite bad about the incident and rather sorry for the little terrified traffic cop. These short business trips taught me more about dealing with customers and how to sell, than all the years at school. If you have kids, do them a favour and teach them what you know.

Throwing ConcurrentModificationException Early

After my last newsletter, one of my readers asked whether I knew of a way to figure out who was modifying a collection concurrently, causing a `ConcurrentModificationException` on iteration. This could happen if you shared a collection between lots of objects and you had lost track of who was modifying it when. This is of course an example of bad design, but sometimes we inherit such code from others. I had often thought about this particular

exception and had used it as an example in my courses of an exception that causes problems because it is **not** thrown early. Also have a look at my [Newsletter on Exceptions](#).

One possible solution to this problem is to write a wrapper collection, much like the synchronized collections in the `java.util.Collections`, which would maintain a set of active iterators. If that set was non-empty, we would throw a `ConcurrentModificationException` on the collection's modification methods. Here is my attempt to define the term *active iterator*:

Active Iterator: *an iterator is active when `hasNext()` has not yet returned `false` AND when there exists a strong reference to it.*

We could thus easily mark an iterator as inactive when it has walked through the entire collection. The tricky case is where someone walks through a section of the collection and then stops. If they keep a strong reference to the iterator, it would stay active until they de-reference it.

Here is the method that checks how many *active iterators* are associated with the collection. The first step is to check whether an active iterator might exist, by checking whether the `iterators` set returns elements. If it does, we call `System.gc()` explicitly, which will hopefully clear the `WeakReferences` to the inactive iterators.

```

private final Map<WeakReference<Iterator<E>>, Object> iterators =
    new ConcurrentHashMap<WeakReference<Iterator<E>>, Object>();

private void checkForCurrentIteration() {
    boolean perhapsConcurrentIteration = false;
    for (WeakReference<Iterator<E>> ref : iterators.keySet()) {
        Iterator<E> iterator = ref.get();
        if (iterator != null) {
            perhapsConcurrentIteration = true;
            break;
        }
    }
    if (perhapsConcurrentIteration) {
        System.gc();
        for (WeakReference<Iterator<E>> ref : iterators.keySet()) {
            if (ref.get() == null) {
                iterators.remove(ref);
            }
        }
        int currentIterators = iterators.size();
        if (currentIterators != 0) {
    }
}

```

```

        throw new ConcurrentModificationException(
            "Iteration might currently be happening with " +
            currentIterators + " iterators");
    }
}
}

```

Under normal circumstances, calling `System.gc()` directly is strongly discouraged. In our case we are trying to discover a specific problem and so `System.gc()` is appropriate. However, it is not guaranteed that it will clear all the `WeakReferences`, so we could get false alarms. Our class is only intended for debugging and not for production code.

The rest of the collection class delegates to another collection and is easy to understand:

```

import java.lang.ref.WeakReference;
import java.util.*;
import java.util.concurrent.ConcurrentHashMap;

public class FailFastCollection<E> implements Collection<E> {
    private final Collection<E> delegate;

    private final Map<WeakReference<Iterator<E>>, Object> iterators =
        new ConcurrentHashMap<WeakReference<Iterator<E>>, Object>();

    public FailFastCollection(Collection<E> delegate) {
        this.delegate = delegate;
    }

    private void checkForCurrentIteration() {
        boolean perhapsConcurrentIteration = false;
        for (WeakReference<Iterator<E>> ref : iterators.keySet()) {
            Iterator<E> iterator = ref.get();
            if (iterator != null) {
                perhapsConcurrentIteration = true;
                break;
            }
        }
        if (perhapsConcurrentIteration) {
            System.gc();
            for (WeakReference<Iterator<E>> ref : iterators.keySet()) {
                if (ref.get() == null) {

```

```
        iterators.remove(ref);
    }
}

int currentIterators = iterators.size();
if (currentIterators != 0) {
    throw new ConcurrentModificationException(
        "Iteration might currently be happening with " +
        currentIterators + " iterators");
}
}

// Non-modifying safe operations
public int size() {
    return delegate.size();
}

public boolean isEmpty() {
    return delegate.isEmpty();
}

public boolean contains(Object o) {
    return delegate.contains(o);
}

public Object[] toArray() {
    return delegate.toArray();
}

public <T> T[] toArray(T[] a) {
    return delegate.toArray(a);
}

public boolean containsAll(Collection<?> c) {
    return delegate.containsAll(c);
}

public boolean equals(Object o) {
    return delegate.equals(o);
}

public int hashCode() {
    return delegate.hashCode();
}

public String toString() {
    return delegate.toString();
}
```

```
// Operations that might modify the underlying collection - unsafe
public boolean add(E e) {
    checkForCurrentIteration();
    return delegate.add(e);
}

public boolean remove(Object o) {
    checkForCurrentIteration();
    return delegate.remove(o);
}

public boolean addAll(Collection<? extends E> c) {
    checkForCurrentIteration();
    return delegate.addAll(c);
}

public boolean removeAll(Collection<?> c) {
    checkForCurrentIteration();
    return delegate.removeAll(c);
}

public boolean retainAll(Collection<?> c) {
    checkForCurrentIteration();
    return delegate.retainAll(c);
}

public void clear() {
    checkForCurrentIteration();
    delegate.clear();
}

public Iterator<E> iterator() {
    return new NoFailIterator(delegate.iterator());
}

private class NoFailIterator implements Iterator<E> {
    private final Iterator<E> delegate;
    private final WeakReference<Iterator<E>> reference;

    private NoFailIterator(Iterator<E> delegate) {
        this.delegate = delegate;
        reference = new WeakReference<Iterator<E>>(this);
        iterators.put(reference, "dummy");
    }

    public boolean hasNext() {
        if (delegate.hasNext()) {

```

```

        return true;
    } else {
        iterators.remove(reference);
        return false;
    }
}

public E next() {
    E e = delegate.next();
    if (!delegate.hasNext()) {
        iterators.remove(reference);
    }
    return e;
}

public void remove() {
    delegate.remove();
}
}
}

```

Test Code

We have several test cases that should all pass and which will show you the intended use of this FailFastCollection class:

```

import junit.framework.TestCase;

import java.util.*;
import java.util.concurrent.*;

public class FailFastCollectionTest extends TestCase {
    private final Collection<String> safe_names =
        new FailFastCollection<String>(new ArrayList<String>());

    protected void setUp() throws Exception {
        safe_names.clear();
        Collections.addAll(safe_names, "Anton", "John", "Heinz");
    }

    public void testBasicIteration() {
        for (String name : safe_names) {

```

```

    }

}

public void testBasicModification() {
    // we first iterate
    for (String name : safe_names) {
    }
    // then we add Dirk - should work as no client code has a
    // handle to the iterator and we ran the iterator to
    // completion
    safe_names.add("Dirk");
    assertEquals(4, safe_names.size());
    Iterator<String> it = safe_names.iterator();
    assertEquals("Anton", it.next());
    assertEquals("John", it.next());
    assertEquals("Heinz", it.next());
    assertEquals("Dirk", it.next());
}

public void testModificationFromAnotherThread() throws InterruptedException {
    // this test is more tricky. Once we have iterated through
    // two elements, we want to add "Neil" to the list. Since
    // that is done in another thread, we get the exception using
    // futures and check it is a ConcurrentModificationException

    final CountDownLatch latch = new CountDownLatch(2);

    ExecutorService executor = Executors.newSingleThreadExecutor();
    Future<?> future = executor.submit(new Callable<Object>() {
        public Object call() throws Exception {
            latch.await();
            safe_names.add("Neil");
            return null;
        }
    });

    for (String name : safe_names) {
        System.out.println(name);
        latch.countDown();
        Thread.sleep(50);
    }

    try {
        future.get();
        fail("expected a ConcurrentModificationException!");
    } catch (ExecutionException e) {
        if (e.getCause() instanceof ConcurrentModificationException) {
            // success
        }
    }
}

```

```

    } else {
        fail("expected a ConcurrentModificationException!");
    }
}

public void testModificationFromSameThread() {
    // iteration does not run to completion, but iterator is out
    // of scope
    try {
        for (String name : safe_names) {
            if (name.equals("John")) {
                safe_names.add("Andre");
            }
        }
    } catch (ConcurrentModificationException ex) {
        System.out.println("Expected the error " + ex);
    }

    // now we should be able to use the collection still.
    assertEquals(3, safe_names.size());
    Iterator<String> it = safe_names.iterator();
    assertEquals("Anton", it.next());
    assertEquals("John", it.next());
    assertEquals("Heinz", it.next());

    // we have run to completion, thus the iterator is inactive
    safe_names.add("Andre");

    assertEquals(4, safe_names.size());
    Iterator<String> it2 = safe_names.iterator();
    assertEquals("Anton", it2.next());
    assertEquals("John", it2.next());
    assertEquals("Heinz", it2.next());
    assertEquals("Andre", it2.next());
}
}

```

Set, List, Map Implementations

We can use the same approach for producing Fail-Fast wrappers for the Set, List and Map interfaces, but that is beyond the scope of this newsletter. List might be a bit tricky, since the ListIterator can also go back in the collection. Thus we can only define an inactive list iterator as one that is not being referenced from anywhere.

WeakReference vs. Finalizers?

We could also have overridden the `finalize()` method in our `NoFailIterator`, but that would not work well at all. Finalization occurs in a separate thread, which would introduce a racing condition into our code. The calling thread should know immediately if one of the references has been cleared, but with the Finalizer thread involved, it would have to wait an indeterminate amount of time to be sure that the `finalize()` method had in fact been called. In our solution we remove the blank reference with the calling thread.

WeakHashMap vs. ConcurrentHashMap?

Instead of the `ConcurrentHashMap`, we could also have used a `WeakHashMap`, which would have reduced our code a bit. However, I wanted to be able to access the map concurrently and `WeakHashMap` is not threadsafe. We could have made it threadsafe by wrapping it with a synchronized map, an idiom quite common in the JDK, but that would have added unnecessary contention.

Next time you have a hard-to-find `ConcurrentModificationException`, try this approach to help find the code that is causing the collection to be modified whilst you are iterating. Please let me know if this was of help to you.

Kind regards

Heinz

Issue 165 - Starvation with ReadWriteLocks

Author: Dr. Heinz M. Kabutz

Date: 2008-10-14

Category: Performance

Java Versions: Java 5 and 6

Abstract:

In this newsletter we examine what happens when a `ReadWriteLock` is swamped with too many reader or writer threads. If we are not careful, we can cause starvation of the minority in some versions of Java.

Welcome to the 165th issue of **The Java(tm) Specialists' Newsletter**, sent to you from Wall Street, New York City. New York has to date been the most expensive city that I have visited. Not that prices are particularly high in comparison to Crete, but you can purchase almost anything, twenty four hours a day! Usually when I go to New York, my bank sends me urgent emails confirming that I really am spending all that money on my credit card. In contrast, back home in Crete, items are often double the New York price, but there is very little temptation to spend.

Starvation with ReadWriteLocks

On my flight from Greece to New York, I decided to try a different approach to cope with jet lag. Instead of taking a nap, I forced myself to stay awake, thus adapting immediately to the new time zone. No jet lag at all this time round! To keep myself occupied, I began investigating what happens with `ReadWriteLocks` in contention situations, which resulted in this newsletter.

I had always thought that with the `ReadWriteLock`, the write lock would get preference over the read lock. Since the read locks can get allocated concurrently, they could essentially tag-team each other and thus never give a chance to a write lock to enter the critical section. However, something in the back of my mind always told me to try it out. The documentation certainly did not promise that write locks would be given preference, on the contrary, it gave no guarantee.

Writing tests like this can be tricky, so I showed my findings to some of the leading Java concurrency experts, who confirmed that they had similar experiences. My test is probably not perfect, but it does demonstrate what happens when we have many threads acquiring read locks and only few threads acquiring write locks.

To make it more interesting, I gave each thread a bunch of instructions to do in the critical section. Doug Lea recommended in an email that the `ReadWriteLock` should only be used in cases where the critical section is at least 2000 code statements. Doug also mentioned that

the Locks had been modified slightly in Java 6, which may have made the problem of starvation more pronounced. In my experiments, I found that Java 5 was much worse than Java 6 in this regard. Doug Lea also suggested that for most applications, we should rather use the concurrency classes, such as ConcurrentHashMap and ConcurrentLinkedQueue, rather than rely on the ReadWriteLock, which I wholeheartedly agree with.

To be able to test different types of locks, we define a LockFactory interface, with two implementations, ReadWriteLockFactory and StandardLockFactory:

```

import java.util.concurrent.locks.Lock;
public interface LockFactory {
    Lock createLock(boolean readOnly);
}

import java.util.concurrent.locks.*;
public class ReadWriteLockFactory implements LockFactory {
    private final ReadWriteLock rwl;

    public ReadWriteLockFactory(boolean fair) {
        rwl = new ReentrantReadWriteLock(fair);
    }

    public Lock createLock(boolean readOnly) {
        return readOnly ? rwl.readLock() : rwl.writeLock();
    }
}

import java.util.concurrent.locks.*;
public class StandardLockFactory implements LockFactory {
    private final Lock lock;

    public StandardLockFactory(boolean fair) {
        lock = new ReentrantLock(fair);
    }

    public Lock createLock(boolean readOnly) {
        return lock;
    }
}

```

We then define a class that implements Runnable and will acquire locks repeatedly, called

the LockAcquirer. The LockAcquirer will run until the shared AtomicBoolean running is set to **false**. We also give it the lock that must be acquired, which could either be a read or a write lock. The CountDownLatches ensure that the threads start and end at the same time. Lastly, the readOnly field helps us to distinguish what the number of calls for both read and write locks was.

In order to see real starvation of threads, I added a work() method, that adds some random numbers and then subtracts them again, thus fooling the hotspot compiler into not removing that particular code. As we've seen in other newsletters, a simple for() loop could be optimized away in a flash. On my dual-core MacBook Pro, we should see both cores at work when we use read locks, but only one at a time when we use write locks.

Here is the code for our LockAcquirer class:

```
import java.util.concurrent.CountDownLatch;
import java.util.concurrent.atomic.AtomicBoolean;
import java.util.concurrent.locks.Lock;
import java.util.Random;

public class LockAcquirer implements Runnable {
    private final AtomicBoolean running;
    private final Lock lock;
    private final CountDownLatch start;
    private final CountDownLatch finish;
    private final boolean readOnly;
    private long numberOfLockAcquisitions = 0;
    private int work = 50;
    private long offset = calculateOffset();

    private long calculateOffset() {
        Random r = new Random(0);
        long result = 0;
        for(int i = 0; i<work; i++) {
            result += r.nextInt();
        }
        return result;
    }

    public LockAcquirer(AtomicBoolean running,
                        Lock lock, CountDownLatch start,
                        CountDownLatch finish, boolean readOnly) {
        this.running = running;
        this.lock = lock;
        this.start = start;
        this.finish = finish;
    }
}
```

```
this.readOnly = readOnly;
}

public void run() {
    try {
        start.countDown();
        start.await();
        while (running.get()) {
            lock.lock();
            try {
                work();
            } finally {
                lock.unlock();
            }
        }
        finish.countDown();
    } catch (InterruptedException e) {
        return;
    }
}

private void work() {
    Random r = new Random(0);
    for(int i=0; i<work; i++) {
        numberOfLockAcquisitions += r.nextInt();
    }
    numberOfLockAcquisitions -= offset;
    numberOfLockAcquisitions++;
}

public String toString() {
    return String.format("%s%,15d",
        (readOnly ? "RO" : "RW"),
        numberOfLockAcquisitions);
}

public long getNumberOfLockAcquisitions() {
    return numberOfLockAcquisitions;
}

public boolean isReadOnly() {
    return readOnly;
}
}
```

The last class in the puzzle is called TestLocks. It creates one thread for each LockAcquirer. We use different numbers of reader and writer threads to test whether starvation does indeed occur. If you run this, you can change the field DETAILED_OUTPUT to specify how much information is displayed.

```
import java.util.*;
import java.util.concurrent.*;
import java.util.concurrent.atomic.AtomicBoolean;
import java.util.concurrent.locks.Lock;

public class TestLocks {
    private static final int DURATION_SECONDS = 5;
    private static final int TOTAL_LOCKS = 16;
    private static final boolean DETAILED_OUTPUT = false;

    public static void main(String[] args)
        throws InterruptedException {
        test(false);
        test(true);
    }

    private static void test(boolean fair)
        throws InterruptedException {
        readWriteLockTest(fair);
        lockTest(fair);
    }

    private static void readWriteLockTest(boolean fair)
        throws InterruptedException {
        System.out.println("ReentrantReadWriteLock test");
        for (int locks = 1; locks <= TOTAL_LOCKS; locks *= 2) {
            for (int reads = 0; reads <= locks; reads++) {
                int writes = locks - reads;
                test(new ReadWriteLockFactory(fair), reads, writes, fair);
            }
        }
    }

    private static void lockTest(boolean fair)
        throws InterruptedException {
        System.out.println("ReentrantLock test");
        for (int locks = 1; locks < TOTAL_LOCKS; locks++) {
            test(new StandardLockFactory(fair), 0, locks, fair);
        }
    }
}
```

```

public static void test(LockFactory factory, int reads,
                      int writes, boolean fair)
    throws InterruptedException {
    System.out.printf("RO=%2d, RW=%2d, fair=%b      ",
                      reads, writes, fair);
    CountDownLatch start = new CountDownLatch(reads + writes);
    CountDownLatch finish = new CountDownLatch(reads + writes);
    AtomicBoolean running = new AtomicBoolean(true);

    ExecutorService pool = Executors.newFixedThreadPool(
        reads + writes);

    Collection<LockAcquirer> acquirers =
        new ArrayList<LockAcquirer>();
    for (int i = 0; i < reads + writes; i++) {
        boolean isReadOnly = i >= writes;
        Lock lock = factory.createLock(isReadOnly);
        LockAcquirer acquirer = new LockAcquirer(running,
            lock, start, finish, isReadOnly);
        acquirers.add(acquirer);
        pool.submit(acquirer);
    }

    start.await();
    TimeUnit.SECONDS.sleep(DURATION_SECONDS);
    running.set(false);
    finish.await();
    long totalReads = 0;
    long totalWrites = 0;
    if (DETAILED_OUTPUT) System.out.println();
    for (LockAcquirer acquirer : acquirers) {
        long numberOfLockAcquisitions =
            acquirer.getNumberOfLockAcquisitions();
        if (DETAILED_OUTPUT)
            System.out.printf("##      %,17d%n",
                numberOfLockAcquisitions);

        if (acquirer.isReadOnly()) {
            totalReads += numberOfLockAcquisitions;
        } else {
            totalWrites += numberOfLockAcquisitions;
        }
    }
    System.out.printf(",17d%,17d%n", totalReads, totalWrites);
    pool.shutdown();
}
}

```

What are we looking for?

With microbenchmarks, we usually need to have some idea of what output to expect, based on our understanding of the code. If the benchmark is wildly wrong then either our understanding of the code is wrong or our benchmark is flawed. My microbenchmark is possibly a bit simplistic, but Cliff Click confirmed that his more sophisticated benchmark showed similar weaknesses with the `ReadWriteLock`.

I always assumed that `ReadWriteLock` would give preference to the `WriteLock`. If not, the writer thread could be starved of resources if new reader threads kept on entering the critical section in a tag-team fashion. That would result in the `WriteLock` literally **never** getting a chance. We're talking of total starvation here! Surely `ReadWriteLock` would not do that!?

My `TestLocks` class produced different output for Java 5 and 6. I used JDK 1.6.0_07 and JDK 1.5.0_13 on my MacBook Pro 2.6 dual core. My suspicion is that the more cores you add, the more pronounced the benchmark will become. Unfortunately I don't own enough cool hardware to verify that.

Java 5 Results

First the output for Java 5. Note that as expected, when we have two threads using `ReadOnly` locks, we get better throughput than with one (first highlighted line). It gets interesting when we use more than two threads. I don't think the number of cores is as important here. When we have three threads accessing `ReadLocks` and one accessing a `WriteLock`, the readers access the critical section 2,757,365 times, whereas the writer only 2 times (second highlighted line). We see the same effect with 3 readers / 5 writers and 3 readers / 13 writers (third and fourth highlighted lines). Thus it confirms my worry that although we get added throughput due to the readers being able to access the critical section concurrently, if these threads keep on repeatedly accessing the critical section, they will lock out all writer threads.

| ReentrantReadWriteLock test | | | |
|------------------------------------|------------------|-----------|--|
| RO= 0, RW= 1, fair=false | 0 | 2,117,034 | |
| RO= 1, RW= 0, fair=false | 2,090,567 | 0 | |
| RO= 0, RW= 2, fair=false | 0 | 1,238,352 | |
| RO= 1, RW= 1, fair=false | 339,871 | 913,259 | |
| 1. RO= 2, RW= 0, fair=false | 2,728,331 | 0 | |
| RO= 0, RW= 4, fair=false | 0 | 1,256,089 | |
| RO= 1, RW= 3, fair=false | 229,381 | 1,019,947 | |
| RO= 2, RW= 2, fair=false | 2,676,337 | 11,274 | |
| 2. RO= 3, RW= 1, fair=false | 2,757,365 | 2 | |
| RO= 4, RW= 0, fair=false | 2,877,727 | 0 | |
| RO= 0, RW= 8, fair=false | 0 | 1,246,185 | |
| RO= 1, RW= 7, fair=false | 72,935 | 1,170,020 | |

| | | | |
|----|---------------------------------|------------------|-----------|
| | RO= 2, RW= 6, fair=false | 2,649,444 | 19,693 |
| 3. | RO= 3, RW= 5, fair=false | 2,856,555 | 5 |
| | RO= 4, RW= 4, fair=false | 2,854,140 | 4 |
| | RO= 5, RW= 3, fair=false | 2,881,890 | 3 |
| | RO= 6, RW= 2, fair=false | 2,958,654 | 2 |
| | RO= 7, RW= 1, fair=false | 3,108,282 | 1 |
| | RO= 8, RW= 0, fair=false | 3,233,558 | 0 |
| | RO= 0, RW=16, fair=false | 0 | 1,234,477 |
| | RO= 1, RW=15, fair=false | 61,381 | 1,181,331 |
| | RO= 2, RW=14, fair=false | 976,371 | 749,589 |
| 4. | RO= 3, RW=13, fair=false | 2,855,692 | 13 |
| | RO= 4, RW=12, fair=false | 2,941,074 | 12 |
| | RO= 5, RW=11, fair=false | 2,918,760 | 11 |
| | RO= 6, RW=10, fair=false | 2,921,576 | 10 |
| | RO= 7, RW= 9, fair=false | 3,185,453 | 9 |
| | RO= 8, RW= 8, fair=false | 3,206,795 | 8 |
| | RO= 9, RW= 7, fair=false | 3,343,547 | 7 |
| | RO=10, RW= 6, fair=false | 3,359,844 | 1,018 |
| | RO=11, RW= 5, fair=false | 3,180,840 | 677 |
| | RO=12, RW= 4, fair=false | 3,542,919 | 4 |
| | RO=13, RW= 3, fair=false | 3,638,226 | 3 |
| | RO=14, RW= 2, fair=false | 3,694,536 | 2 |
| | RO=15, RW= 1, fair=false | 3,688,920 | 1 |
| | RO=16, RW= 0, fair=false | 3,677,340 | 0 |

The normal Locks show typical behaviour, so it is not that interesting to look at. Here are the values for completeness:

| ReentrantLock test | | | |
|--------------------|--------------------------|---|-----------|
| | RO= 0, RW= 1, fair=false | 0 | 2,105,577 |
| | RO= 0, RW= 2, fair=false | 0 | 1,233,695 |
| | RO= 0, RW= 3, fair=false | 0 | 1,199,802 |
| | RO= 0, RW= 4, fair=false | 0 | 1,231,030 |
| | RO= 0, RW= 5, fair=false | 0 | 1,229,788 |
| | RO= 0, RW= 6, fair=false | 0 | 1,216,533 |
| | RO= 0, RW= 7, fair=false | 0 | 1,229,848 |
| | RO= 0, RW= 8, fair=false | 0 | 1,285,035 |
| | RO= 0, RW= 9, fair=false | 0 | 1,251,308 |
| | RO= 0, RW=10, fair=false | 0 | 1,226,727 |
| | RO= 0, RW=11, fair=false | 0 | 1,239,005 |
| | RO= 0, RW=12, fair=false | 0 | 1,231,186 |
| | RO= 0, RW=13, fair=false | 0 | 1,237,608 |
| | RO= 0, RW=14, fair=false | 0 | 1,501,639 |
| | RO= 0, RW=15, fair=false | 0 | 1,303,815 |

It gets interesting again when we add fairness to the picture. With single threads or when there are *only* reads happening, we see similar throughput as with unfair locks (highlighted line 1). We also see that there is no more starvation happening with our readers. However, the throughput is also *much* lower than before. In the past, the total throughput (reads plus writes) was between 2.5 and 3.5 million, now it hardly reaches 0.5 million (highlighted lines 2 and 3). We see that fairness is a throughput killer, also confirmed in [Java Concurrency in Practice](#).

| ReentrantReadWriteLock test | | | |
|-----------------------------------|------------------|----------------|--|
| RO= 0, RW= 1, fair=true | 0 | 2,072,787 | |
| RO= 1, RW= 0, fair=true | 2,060,310 | 0 | |
| RO= 0, RW= 2, fair=true | 0 | 565,702 | |
| RO= 1, RW= 1, fair=true | 275,904 | 275,624 | |
| RO= 2, RW= 0, fair=true | 2,682,398 | 0 | |
| RO= 0, RW= 4, fair=true | 0 | 443,952 | |
| RO= 1, RW= 3, fair=true | 110,523 | 331,542 | |
| RO= 2, RW= 2, fair=true | 234,798 | 234,772 | |
| RO= 3, RW= 1, fair=true | 402,339 | 134,095 | |
| 1. RO= 4, RW= 0, fair=true | 2,918,238 | 0 | |
| RO= 0, RW= 8, fair=true | 0 | 439,975 | |
| RO= 1, RW= 7, fair=true | 54,965 | 384,712 | |
| RO= 2, RW= 6, fair=true | 111,827 | 335,442 | |
| RO= 3, RW= 5, fair=true | 171,148 | 285,232 | |
| RO= 4, RW= 4, fair=true | 238,927 | 238,915 | |
| RO= 5, RW= 3, fair=true | 318,833 | 191,296 | |
| RO= 6, RW= 2, fair=true | 415,278 | 138,424 | |
| RO= 7, RW= 1, fair=true | 556,224 | 79,317 | |
| RO= 8, RW= 0, fair=true | 3,145,546 | 0 | |
| 2. RO= 0, RW=16, fair=true | 0 | 438,235 | |
| RO= 1, RW=15, fair=true | 27,571 | 413,475 | |
| RO= 2, RW=14, fair=true | 54,852 | 383,873 | |
| RO= 3, RW=13, fair=true | 83,335 | 361,065 | |
| RO= 4, RW=12, fair=true | 111,132 | 333,361 | |
| RO= 5, RW=11, fair=true | 142,440 | 313,366 | |
| RO= 6, RW=10, fair=true | 171,931 | 286,555 | |
| 3. RO= 7, RW= 9, fair=true | 211,226 | 271,570 | |
| RO= 8, RW= 8, fair=true | 240,015 | 240,012 | |
| RO= 9, RW= 7, fair=true | 278,216 | 216,392 | |
| RO=10, RW= 6, fair=true | 322,074 | 193,246 | |
| RO=11, RW= 5, fair=true | 374,487 | 170,221 | |
| RO=12, RW= 4, fair=true | 422,841 | 140,949 | |
| RO=13, RW= 3, fair=true | 488,080 | 112,634 | |
| RO=14, RW= 2, fair=true | 565,221 | 80,748 | |
| RO=15, RW= 1, fair=true | 678,649 | 44,844 | |

| | |
|-------------------------|-----------|
| RO=16, RW= 0, fair=true | 3,721,636 |
|-------------------------|-----------|

| |
|---|
| 0 |
|---|

Just how fair the ReentrantLock is (and how poor the throughput can be), is seen in the figures below:

```
ReentrantLock test
RO= 0, RW= 1, fair=true      0      2,078,251
RO= 0, RW= 2, fair=true      0      555,530
RO= 0, RW= 3, fair=true      0      440,922
RO= 0, RW= 4, fair=true      0      443,428
RO= 0, RW= 5, fair=true      0      441,595
RO= 0, RW= 6, fair=true      0      441,867
RO= 0, RW= 7, fair=true      0      442,270
RO= 0, RW= 8, fair=true      0      441,479
RO= 0, RW= 9, fair=true      0      443,839
RO= 0, RW=10, fair=true      0      442,190
RO= 0, RW=11, fair=true      0      444,120
RO= 0, RW=12, fair=true      0      440,291
RO= 0, RW=13, fair=true      0      443,763
RO= 0, RW=14, fair=true      0      440,695
RO= 0, RW=15, fair=true      0      444,110
```

We saw that for Java 5, we can see serious starvation of our writer threads to the point where they never get a chance.

Java 6 Results

Let's now take a look at the figures for Java 6. The figures for fair locks and the normal ReentrantLock are very similar in their nature to Java 5, so I will just show unfair ReadWriteLocks. They do not degrade the writers as badly as in Java 5. An interesting phenomenon is that the readers seem to be starved by the writers (highlighted lines 1 and 2). Perhaps the locks were modified to counteract the starvation of Java 5 and this might have been overcompensated? We also see that as the readers increase, the writers get disadvantaged, but not as seriously as in Java 5 (highlighted line 3 and 4). Even though there are 6 readers and 10 writers, the readers manage to read more than 10 times more frequently (in total) than the writers. The detailed output for this case is shown below.

```
ReentrantReadWriteLock test
```

| | | | |
|----|---------------------------------|------------------|------------------|
| | RO= 0, RW= 1, fair=false | 0 | 4,386,864 |
| | RO= 1, RW= 0, fair=false | 3,929,583 | 0 |
| | RO= 0, RW= 2, fair=false | 0 | 2,724,154 |
| 1. | RO= 1, RW= 1, fair=false | 8,515 | 2,630,659 |
| | RO= 2, RW= 0, fair=false | 5,687,455 | 0 |
| | RO= 0, RW= 4, fair=false | 0 | 2,515,201 |
| | RO= 1, RW= 3, fair=false | 2,848 | 2,602,498 |
| | RO= 2, RW= 2, fair=false | 28,315 | 2,476,875 |
| | RO= 3, RW= 1, fair=false | 5,193,481 | 394,732 |
| | RO= 4, RW= 0, fair=false | 5,902,569 | 0 |
| | RO= 0, RW= 8, fair=false | 0 | 2,662,787 |
| 2. | RO= 1, RW= 7, fair=false | 647 | 2,662,818 |
| | RO= 2, RW= 6, fair=false | 3,460 | 2,693,680 |
| | RO= 3, RW= 5, fair=false | 3,474,177 | 1,234,051 |
| 3. | RO= 4, RW= 4, fair=false | 4,699,560 | 584,991 |
| | RO= 5, RW= 3, fair=false | 5,296,241 | 293,361 |
| | RO= 6, RW= 2, fair=false | 5,699,336 | 87,412 |
| | RO= 7, RW= 1, fair=false | 5,781,451 | 54,931 |
| | RO= 8, RW= 0, fair=false | 6,063,859 | 0 |
| | RO= 0, RW=16, fair=false | 0 | 2,654,262 |
| | RO= 1, RW=15, fair=false | 479 | 2,688,341 |
| | RO= 2, RW=14, fair=false | 2,052 | 2,693,743 |
| | RO= 3, RW=13, fair=false | 2,672,766 | 1,514,596 |
| | RO= 4, RW=12, fair=false | 3,402,001 | 1,206,118 |
| | RO= 5, RW=11, fair=false | 2,523,643 | 1,585,372 |
| 4. | RO= 6, RW=10, fair=false | 4,873,365 | 555,011 |
| | RO= 7, RW= 9, fair=false | 5,094,895 | 362,255 |
| | RO= 8, RW= 8, fair=false | 5,365,968 | 310,056 |
| | RO= 9, RW= 7, fair=false | 5,394,899 | 283,084 |
| | RO=10, RW= 6, fair=false | 5,650,911 | 220,294 |
| | RO=11, RW= 5, fair=false | 5,971,020 | 171,484 |
| | RO=12, RW= 4, fair=false | 5,916,624 | 159,303 |
| | RO=13, RW= 3, fair=false | 6,193,914 | 117,573 |
| | RO=14, RW= 2, fair=false | 5,969,253 | 74,979 |
| | RO=15, RW= 1, fair=false | 6,187,903 | 32,918 |
| | RO=16, RW= 0, fair=false | 6,415,678 | 0 |

Here is the detailed output for the case where we have 6 reader and 10 writer threads:

```
RO= 6, RW=10, fair=false
##          48,849
##          40,850
##          74,406
```

| | |
|----|------------------------|
| ## | 83,103 |
| ## | 45,319 |
| ## | 54,664 |
| ## | 62,691 |
| ## | 56,161 |
| ## | 35,511 |
| ## | 53,457 |
| ## | 958,901 |
| ## | 857,711 |
| ## | 867,904 |
| ## | 772,812 |
| ## | 804,360 |
| ## | 611,677 |
| | 4,873,365 555,011 |

We see that starvation has become better managed in Java 6, but it still appears to be a potential problem. We also see that when we have several readers, the throughput will be related to the number of actual cores available, which proves that the readers are happening in parallel.

Conclusion

In a way, I am surprised that `ReadWriteLocks` are as bad as they are. I really thought that write locks would get priority over read locks, since not doing that would result in obvious starvation scenarios. Instead, with Java 5, as soon as you have 3 or more read threads, you can expect complete starvation of the writer threads. With Java 6, the situation has improved somewhat.

In my understanding of these values, we should probably never use `ReadWriteLocks` in Java 5. It is just too risky. In Java 6, we can use the `ReadWriteLock` when we are willing to wait for our writers to get an opportunity to acquire the lock. It seems that they won't get locked out completely, but it might take some time before they are serviced. Fairness does not really help us consistently, since it also reduces the throughput.

Instead of using `ReadWriteLock`, we rather recommend that you use higher level concurrency classes such as the atomic classes, the `ConcurrentHashMap` and `ConcurrentLinkedQueue`.

Kind regards

Heinz

Issue 164 - Why 0x61c88647?

Author: Dr. Heinz M. Kabutz

Date: 2008-09-29

Category: Performance

Java Versions: Java 6

Abstract:

Prior to Java 1.4, ThreadLocals caused thread contention, rendering them useless for performant code. In the new design, each thread contains its own ThreadLocalMap, thus improving throughput. However, we still face the possibility of memory leaks due to values not being cleared out of the ThreadLocalMap with long running threads.

Welcome to the 164th issue of **The Java(tm) Specialists' Newsletter**, read in at least **116 countries**, with the recent addition of Tunisia. Our kids are back at school, after 14 weeks of summer holidays. When I was young, we used to get 6 weeks, of which a substantial amount of time was spent getting ready for Christmas and surviving New Year. In South Africa, Christmas is the height of summer :-) Back at school, Connie promptly caught a nasty flu virus, which I hope to be able to ward off with heavy doses of vitamins and zinc.

We were approached a few months ago to present our **Design Patterns Course** to a rather large IT company. Since it would be a larger contract, we agreed to completely overhaul our Design Patterns material. J2EE patterns were tossed out (at long last, I never liked them). In their place, we have some additional Gang-of-Four patterns. We also changed the structure of the exercises so that we can now accommodate up to 50 developers in one class! It was quite an experience teaching such a large class in one room.

Why 0x61c88647?

This newsletter is a product of widespread collaboration. It started with an email from Ant Kutschera, pointing out an **issue he had noticed with using ThreadLocals**.

I had noticed during one of my **Java Specialist Master Course** demonstrations that it was not always that obvious when values were released from memory with ThreadLocals, which could lead to memory leaks in applications that reuse threads. (This newsletter could have been entitled "When Do ThreadLocal Values Get Cleared?", but I wanted to pique your interest with that number.)

However, the code for ThreadLocal looked a tad complicated, with some rather intimidating numbers (0x61c88647), so I enlisted the help of **Joachim Ansorg**, a clever young German software engineer fresh out of university and my long-time buddy John Green.

In earlier versions of Java, ThreadLocals used to have contention when accessed from multiple threads, rendering them practically useless for multi-core applications. In Java 1.4, a new design was introduced, where the ThreadLocals were stored in the Thread directly. When we now call get() on the ThreadLocal, a call back is made to Thread, which returns an instance of ThreadLocalMap, an inner class of ThreadLocal.

I discovered by experimentation that when a thread exits, it also removes all of its ThreadLocal values. This happens before it gets garbage collected, in its exit() method. If we thus forget to call remove() on a ThreadLocal when we are done with it, the value will still be made eligible for garbage collection when the thread exits.

The ThreadLocalMap contains WeakReferences to the ThreadLocals and normal strong references to the values. However, it does not consult the ReferenceQueue to discover which weak references have been cleared, thus an entry might not be cleared out of the ThreadLocalMap immediately (or at all, as we shall see.)

Before we delve into the code and try to figure out how ThreadLocalMap works, I would like to demonstrate a simple example of how ThreadLocals could be used. Imagine we had a StupidInhouseFramework, where the author called an abstract method from the constructor. Something like this:

```
public abstract class StupidInhouseFramework {
    private final String title;

    protected StupidInhouseFramework(String title) {
        this.title = title;
        draw();
    }

    public abstract void draw();

    public String toString() {
        return "StupidInhouseFramework " + title;
    }
}
```

You might think that no one would ever call an abstract method from a constructor, but you are wrong. I even found places in the JDK where this is done, though I don't remember where they were. Here is the class that the poor user has constructed:

```

public class PoorUser extends StupidInhouseFramework {
    private final Long density;

    public PoorUser(String title, long density) {
        super(title);
        this.density = density;
    }

    public void draw() {
        long density_fudge_value = density + 30 * 113;
        System.out.println("draw ... " + density_fudge_value);
    }

    public static void main(String[] args) {
        StupidInhouseFramework sif = new PoorUser("Poor Me", 33244L);
        sif.draw();
    }
}

```

When we run this, we get a `NullPointerException`. The field is of type `Long`, the wrapper class. The method `draw()` is called from the superclass, at which point the `PoorUser`'s constructor has not been called yet. It is thus still set to null, which causes a `NullPointerException` when it is unboxed. We can solve this problem using `ThreadLocal` and even though this is not the typical use case, it is fun to look at.

```

public class HappyUser extends StupidInhouseFramework {
    private final Long density;

    private static final ThreadLocal<Long> density_param =
        new ThreadLocal<Long>();

    private static String setParams(String title, long density) {
        density_param.set(density);
        return title;
    }

    private long getDensity() {
        Long param = density_param.get();
        if (param != null) {
            return param;
        }
    }
}

```

```

    return density;
}

public HappyUser(String title, long density) {
    super(setParams(title, density));
    this.density = density;
    density_param.remove();
}

public void draw() {
    long density_fudge_value = getDensity() + 30 * 113;
    System.out.println("draw ... " + density_fudge_value);
}

public static void main(String[] args) {
    StupidInhouseFramework sif = new HappyUser("Poor Me", 33244L);
    sif.draw();
}
}

```

Just a warning, the example inside the JavaDocs for ThreadLocal in JDK 1.6 had an obvious typo in it, but they fortunately fixed it in JDK 1.7. See if you can spot the difference.

When Can ThreadLocal Values Be Garbage Collected?

We said already that they can be garbage collected when the owning thread exits. However, if the thread belongs to a thread pool, such as we would find in some application servers, the values may or may not be garbage collected, ever.

To demonstrate this, I created several classes with finalize() methods, which demonstrate when the object has reached the end of its life.

The first is a simple value that also shows when it is collected and notifies our test framework. This will allow us to write actual unit tests demonstrating our findings.

```

public class MyValue {
    private final int value;

    public MyValue(int value) {
        this.value = value;
    }
}
```

```

    }

    protected void finalize() throws Throwable {
        System.out.println("MyValue.finalize " + value);
        ThreadLocalTest.setMyValueFinalized();
        super.finalize();
    }
}

```

MyThreadLocal overrides ThreadLocal and prints out a message when it is being finalized:

```

public class MyThreadLocal<T> extends ThreadLocal<T> {
    protected void finalize() throws Throwable {
        System.out.println("MyThreadLocal.finalize");
        ThreadLocalTest.setMyThreadLocalFinalized();
        super.finalize();
    }
}

```

ThreadLocalUser is a class that would encapsulate a ThreadLocal. When this is no longer reachable, we would expect its ThreadLocal to also be collected. Note that in the JavaDocs we read: *ThreadLocal instances are typically private static fields in classes that wish to associate state with a thread (e.g., a user ID or Transaction ID).* By constructing lots of instance of ThreadLocals, we are exhibiting the problem in a more dramatic fashion.

```

public class ThreadLocalUser {
    private final int num;
    private MyThreadLocal<MyValue> value =
        new MyThreadLocal<MyValue>();

    public ThreadLocalUser() {
        this(0);
    }

    public ThreadLocalUser(int num) {
        this.num = num;
    }
}

```

```

protected void finalize() throws Throwable {
    System.out.println("ThreadLocalUser.finalize " + num);
    ThreadLocalTest.setThreadLocalUserFinalized();
    super.finalize();
}

public void setThreadLocal(MyValue myValue) {
    value.set(myValue);
}

public void clear() {
    value.remove();
}
}

```

The last class is MyThread, which shows when the thread is collected:

```

public class MyThread extends Thread {
    public MyThread(Runnable target) {
        super(target);
    }
    protected void finalize() throws Throwable {
        System.out.println("MyThread.finalize");
        ThreadLocalTest.setMyThreadFinalized();
        super.finalize();
    }
}

```

The first two test cases illustrate what happens when the thread local is cleared with the remove() method and when it is left for the garbage collector to take care of. The booleans are used to help us write the unit tests.

```

import junit.framework.TestCase;

import java.util.concurrent.*;

```

```
public class ThreadLocalTest extends TestCase {  
    private static boolean myValueFinalized;  
    private static boolean threadLocalUserFinalized;  
    private static boolean myThreadLocalFinalized;  
    private static boolean myThreadFinalized;  
  
    public void setUp() {  
        myValueFinalized = false;  
        threadLocalUserFinalized = false;  
        myThreadLocalFinalized = false;  
        myThreadFinalized = false;  
    }  
  
    public static void setMyValueFinalized() {  
        myValueFinalized = true;  
    }  
  
    public static void setThreadLocalUserFinalized() {  
        threadLocalUserFinalized = true;  
    }  
  
    public static void setMyThreadLocalFinalized() {  
        myThreadLocalFinalized = true;  
    }  
  
    public static void setMyThreadFinalized() {  
        myThreadFinalized = true;  
    }  
  
    private void collectGarbage() {  
        for (int i = 0; i < 10; i++) {  
            System.gc();  
            try {  
                Thread.sleep(50);  
            } catch (InterruptedException e) {  
                Thread.currentThread().interrupt();  
                break;  
            }  
        }  
    }  
  
    public void test1() {  
        ThreadLocalUser user = new ThreadLocalUser();  
        MyValue value = new MyValue(1);  
        user.setThreadLocal(value);  
        user.clear();  
        value = null;  
        collectGarbage();  
    }  
}
```

```

        assertTrue(myValueFinalized);
        assertFalse(threadLocalUserFinalized);
        assertFalse(myThreadLocalFinalized);
    }

    // weird case
    public void test2() {
        ThreadLocalUser user = new ThreadLocalUser();
        MyValue value = new MyValue(1);
        user.setThreadLocal(value);
        value = null;
        user = null;
        collectGarbage();
        assertFalse(myValueFinalized);
        assertTrue(threadLocalUserFinalized);
        assertTrue(myThreadLocalFinalized);
    }
}

```

In test3(), we demonstrate how a thread closing releases its ThreadLocal values:

```

public void test3() throws InterruptedException {
    Thread t = new MyThread(new Runnable() {
        public void run() {
            ThreadLocalUser user = new ThreadLocalUser();
            MyValue value = new MyValue(1);
            user.setThreadLocal(value);
        }
    });
    t.start();
    t.join();
    collectGarbage();
    assertTrue(myValueFinalized);
    assertTrue(threadLocalUserFinalized);
    assertTrue(myThreadLocalFinalized);
    assertFalse(myThreadFinalized);
}

```

In our next test, we see how a thread pool can cause the value to be held captive:

```

public void test4() throws InterruptedException {
    Executor singlePool = Executors.newSingleThreadExecutor();
    singlePool.execute(new Runnable() {
        public void run() {
            ThreadLocalUser user = new ThreadLocalUser();
            MyValue value = new MyValue(1);
            user.setThreadLocal(value);
        }
    });
    Thread.sleep(100);
    collectGarbage();
    assertFalse(myValueFinalized);
    assertTrue(threadLocalUserFinalized);
    assertTrue(myThreadLocalFinalized);
}

```

So far, we have not seen any great surprises. Now we get to the interesting test cases. In the next one, we construct a hundred ThreadLocals and then garbage collect them at the end. Note that not a single one of the MyValue objects is garbage collected:

```

public void test5() throws Exception {
    for (int i = 0; i < 100; i++) {
        ThreadLocalUser user = new ThreadLocalUser(i);
        MyValue value = new MyValue(i);
        user.setThreadLocal(value);
        value = null;
        user = null;
    }
    collectGarbage();

    assertFalse(myValueFinalized);
    assertTrue(threadLocalUserFinalized);
    assertTrue(myThreadLocalFinalized);
}

```

In test6(), we see that due to the forced garbage collection, some of the values are now being collected, but they lag behind the ThreadLocalUser collections.

```

public void test6() throws Exception {
    for (int i = 0; i < 100; i++) {
        ThreadLocalUser user = new ThreadLocalUser(i);
        MyValue value = new MyValue(i);
        user.setThreadLocal(value);
        value = null;
        user = null;
        collectGarbage();
    }

    assertTrue(myValueFinalized);
    assertTrue(threadLocalUserFinalized);
    assertTrue(myThreadLocalFinalized);
}

```

You can see how the collection of MyValues lags behind in the output. By the time the program finished, MyValues 98 and 99 had not been collected.

```

ThreadLocalUser.finalize 96
MyValue.finalize 94
ThreadLocalUser.finalize 97
MyThreadLocal.finalize
MyValue.finalize 96
MyValue.finalize 95
MyThreadLocal.finalize
ThreadLocalUser.finalize 98
ThreadLocalUser.finalize 99
MyThreadLocal.finalize
MyValue.finalize 97

```

Gaze Inside ThreadLocal

One of the first things that I noticed when I looked inside the ThreadLocal class was a big fat number 0x61c88647 staring back at me. This was the HASH_INCREMENT. Every time a new ThreadLocal is created, it gets a unique hash number by adding 0x61c88647 to the previous value. I spent most of yesterday trying to figure out **why** the engineers had chosen this particular number. If you google for 61c88647, you will find some articles in Chinese and a few to do with encryption. Besides that, not much else.

My friend John Green had the idea of turning this number into decimal and repeating the search. The number 1640531527 had more useful hits. However, in the contexts that we saw, it was used to in hashing to multiply hash values, not add them. Also, in all the contexts we found, the actual number was -1640531527. Some more digging revealed that this number is the 32-bit signed version of the unsigned number 2654435769.

This number represents the golden ratio ($\sqrt{5}-1$) times two to the power of 31. The result is then a golden number, either 2654435769 or -1640531527. You can see the calculation here:

```
public class ThreadHashTest {
    public static void main(String[] args) {
        long l1 = (long) ((1L << 31) * (Math.sqrt(5) - 1));
        System.out.println("as 32 bit unsigned: " + l1);
        int i1 = (int) l1;
        System.out.println("as 32 bit signed:    " + i1);
        System.out.println("MAGIC = " + 0x61c88647);
    }
}
```

For more information about the Golden Ratio, have a look at [the Wikipedia link](#) as well as a [book on data structures in C++](#). For completeness, I also looked up references to the golden ratio in Donald Knuth's [The Art of Computer Programming](#). Donald Knuth belongs on the desk of every serious computer programmer, in the same way that the [Merck Manual](#) adorns your health practitioner's bookshelf ([Available online](#)). Just don't expect to understand the contents ...

We established thus that the HASH_INCREMENT has something to do with fibonacci hashing, using the golden ratio. If we look carefully at the way that hashing is done in the ThreadLocalMap, we see why this is necessary. The standard java.util.HashMap uses linked lists to resolve clashes. The ThreadLocalMap simply looks for the next available space and inserts the element there. It finds the first space by bit masking, thus only the lower few bits are significant. If the first space is full, it simply puts the element in the next available space. The HASH_INCREMENT spaces the keys out in the sparse hash table, so that the possibility of finding a value next to ours is reduced.

When a ThreadLocal is garbage collected, the WeakReference key in the ThreadLocalMap is cleared. The question we then need to resolve is when this will be deleted from the ThreadLocalMap. It does *not* get cleared out when we call `get()` on another entry in the map. `java.util.WeakHashMap` expunges all stale entries from its map even on the `get()`. The `get()` would thus be a bit faster in ThreadLocalMap, but might leave the table with stale entries, hence memory leaks.

When a ThreadLocal is set(), it could fall into one of three categories:

1. Firstly, we could find the entry and simply set it. In this case, stale entries are not expunged at all.
2. Secondly, we might find that one of the entries before ours has become stale, in which case we expunge all stale entries within our run (that is, between two `null` values). If we find our key, it will be swapped with the stale entry.
3. Thirdly, our run might not have enough space to expand into in which case the entry is placed in the last null of the run and some of the stale entries are cleaned out. This stage uses a $O(\log_2 n)$ algorithm initially, but if it cannot get below the fill factor, then a full rehash is performed in $O(n)$.

Lastly, if a key is removed, then the entry is expunged, together with any other entries in that run.

If you want more information of how this works, you can get some ideas from [Knuth 6.4 Algorithm R](#).

Best Practices

If you must use ThreadLocal, make sure that you remove the value as soon as you are done with it and preferably before you return your thread to a thread pool. Best practice is to use `remove()` rather than `set(null)`, since that would cause the WeakReference to be removed immediately, together with the value.

Kind regards

Heinz

Issue 160 - The Law of the Uneaten Lutefisk

Author: Dr. Heinz M. Kabutz

Date: 2008-05-12

Category: Concurrency

Java Versions: 5+

Abstract:

Imagine a very stubborn viking father insisting that his equally stubborn child eat its lutefisk before going to sleep. In real life one of the "threads" eventually will give up, but in Java, the threads become deadlocked, with neither giving an inch. In this newsletter we discover how we can sometimes escape from such deadlocked situations in Java and learn why the stop() function should never ever ever be called.

Welcome to the 160th issue of **The Java(tm) Specialists' Newsletter**, sent to you from Belmont in San Francisco. I am here in the San Francisco Bay Area until the 21st of May, so give me a shout if you are in the vicinity and would like to get together one of the evenings or the coming weekend for a chat about Java or life in general.

I must apologize for the length of this newsletter. I try to keep them in nice bit-sized chunks. However, there is just too much material to cover on deadlocks. It should be worth reading though, even if it takes you a few days. In the end, we make some discoveries about how we can sometimes escape from deadlock situations in Java, something that was not possible in earlier versions of Java. Infact, I have not seen any of this information published elsewhere. This also brings us to the end of our series on concurrency. If this series interested you and you would like more information about concurrency and other advanced Java topics, please consider attending our [Java Master Course](#).

The Law of the Uneaten Lutefisk

We are looking at a series of laws to make sense of Java concurrency. Just to remind you, here they are again. In this newsletter, we will look at the Law of the Uneaten Lutefisk.

1. [The Law of the Sabotaged Doorbell](#)
2. [The Law of the Distracted Spearfisherman](#)
3. [The Law of the Overstocked Haberdashery](#)
4. [The Law of the Blind Spot](#)
5. [The Law of the Leaked Memo](#)
6. [The Law of the Corrupt Politician](#)
7. [The Law of the Micromanager](#)
8. [The Law of Cretan Driving](#)
9. [The Law of Sudden Riches](#)
10. [The Law of the Uneaten Lutefisk](#)
11. [The Law of the Xerox Copier](#)

The Law of the Uneaten Lutefisk

A deadlock in Java can usually only be resolved by restarting the Java Virtual Machine.

Definition: Lutefisk - A traditional Scandinavian dish prepared by soaking air-dried cod in a lye solution for several weeks before skinning, boning, and boiling it, a process that gives the dish its characteristic gelatinous consistency.

Imagine a very stubborn viking father insisting that his equally stubborn child eat its lutefisk (eggplant, tomatoes, etc.) before going to bed. Each child seems to have at least one food that they will not eat, irrespective of the amount of coaxing by their parents. When I was a kid, I could not stomach hotdog sausages. This was particularly challenging, as they were the food of choice at birthday parties. Food can create deadlocks between parents and children. Fortunately in real life, one of the parties usually gives in after a while, which means the chances of getting to sleep improve. In Java, when we get into a deadlock situation, we do not always have an easy way out!

Java has several ways of locking critical sections. When we lock using synchronized, we call this monitor locking. The Java 5 locks would just be called locking.

Deadlocks can occur when two threads are locking on two different locks, in opposite order. Usually this happens inadvertently. The deadlock can occur between two monitors or between two locks or a hybrid of the two.

Calling Thread.stop()

You have probably heard that Thread.stop() should never be called. There are several reasons why. One of them was sent to me this morning by Brian Goetz. What connects Brian and me is not just the last two letters of our surnames, but also a keen interest in Java concurrency. His book titled **Java Concurrency in Practice** belongs on every Java Specialist's book shelf.

Brian sent me this example of code that would break a class invariant if stop() is called at an inappropriate time on thread Foo.

```
public static final Object lock = new Object();
@GuardedBy("lock") public static int x;
@GuardedBy("lock") public static int y;
// invariant: x + y = 10

class Foo extends Thread {
```

```

public void run() {
    for (int i=0; i<ONE_BILLION; i++) {
        synchronized(lock) {
            x++;
            y--;
        }
    }
}

```

I put Brian's idea into a class that you can run and which will predictably produce a failed class invariant. In addition, you can change it to terminate the thread using the interrupt() mechanism, by just setting a constant USE_INTERRUPT to true:

```

public class StopIsBad {
    private static final boolean USE_INTERRUPT = false;
    public static final Object lock = new Object();
    public static int x = 5;
    public static int y = 5;
    // invariant: x + y = 10

    static class Foo extends Thread {
        public void run() {
            while (!isInterrupted()) {
                synchronized (lock) {
                    x++;
                    sleepQuietly(); // make sure stop() happens here
                    y--;
                }
            }
        }
    }

    private void sleepQuietly() {
        try {
            sleep(10);
        } catch (InterruptedException ex) {
            Thread.currentThread().interrupt();
        }
    }
}

public static void main(String[] args)
    throws InterruptedException {

```

```

while (true) {
    System.out.println("Creating new foo");
    Foo foo = new Foo();
    foo.start();
    Thread.sleep(50);
    if (USE_INTERRUPT) {
        foo.interrupt();
    } else {
        foo.stop();
    }
    synchronized (lock) {
        if (x + y != 10) {
            throw new IllegalStateException(
                "Invariant is broken: " + (x + y));
        }
    }
    foo.join();
}
}

```

When we run this with USE_INTERRUPT==false, we almost immediately break the class invariant:

```

Creating new foo
Exception in thread "main" java.lang.IllegalStateException:
    Invariant is broken: 11
    at StopIsBad.main(StopIsBad.java:42)

```

When we use the interrupt() mechanism to shut down the code, assuming that we listened to **The Law of the Sabotaged Doorbell**, the class invariant stays intact:

```

Creating new foo

```

```
Creating new foo
```

```
...
```

The `Thread.stop()` method is by its nature dangerous. During the course of this newsletter, I will show you two other occasions where `stop()` produces unexpected results.

Dining Philosophers

Imagine a group of philosophers sitting around a table. Each philosopher has a bowl of rice and a chopstick on either side of the bowl, each of which he shares with his neighbour. Philosophers alternate between the states `THINKING` and `EATING`. To go from `THINKING` to eating, he first picks up the left chopstick then the right chopstick. If they all pick up the left chopstick at the same time, they will have a deadlock. None of them can go from state `THINKING` to `EATING`, because they each need the right chopstick, which is held by their neighbour.

Depending on how long he waits between taking each chopstick and the number of philosophers, the chances of him deadlocking can increase.

For example, here is our Philosopher, with a left and right chopstick (both synchronized monitors). In order to change from `THINKING` to `EATING` state, he needs to first acquire the left chopstick, followed by the right chopstick. In order to make the deadlock occur quicker, I make him ponder for a while inbetween moves.

```
public class Philosopher implements Runnable {
    private final Object left;
    private final Object right;

    public Philosopher(Object left, Object right) {
        this.left = left;
        this.right = right;
    }

    private void ponder() throws InterruptedException {
        Thread.sleep((int) Math.random() * 10) + 10);
    }

    public void run() {
        try {
            while (true) {
                ponder(); // thinking
                synchronized (left) {
                    synchronized (right) {
                        System.out.println("Philosopher " + hashCode() +
                            " is eating");
                    }
                }
            }
        } catch (InterruptedException e) {
            System.out.println("Philosopher " + hashCode() +
                " was interrupted");
        }
    }
}
```

```
    synchronized (left) {
        ponder();
        synchronized (right) {
            ponder(); // eating
        }
    }
}

} catch (InterruptedException e) {
    Thread.currentThread().interrupt();
    return;
}
}
```

Next we need a way to discover deadlocks. Fortunately the ThreadMXBean can help us find these. Before I show you my DeadlockChecker, here is the Dinner, with 5 philosophers:

```
import java.lang.management.ThreadInfo;
import java.util.Collection;

public class Dinner {
    public static void main(String[] args) throws Exception {
        final Philosopher[] philosophers = new Philosopher[5];
        Object[] chopsticks = new Object[philosophers.length];
        for (int i = 0; i < chopsticks.length; i++) {
            chopsticks[i] = new Object();
        }
        for (int i = 0; i < philosophers.length; i++) {
            Object left = chopsticks[i];
            Object right = chopsticks[(i + 1) % chopsticks.length];
            philosophers[i] = new Philosopher(left, right);
            Thread t = new Thread(philosophers[i], "Phil " + (i + 1));
            t.start();
        }
    }

    DeadlockChecker checker = new DeadlockChecker();
    while (true) {
        Collection<ThreadInfo> threads = checker.check();
        if (!threads.isEmpty()) {
            System.out.println("Found deadlock:");
            for (ThreadInfo thread : threads) {
                System.out.println("\t" + thread.getThreadName());
            }
        }
    }
}
```

```
        System.exit(1);  
    }  
    Thread.sleep(1000);  
}  
}  
}  
}
```

Let's examine the output with our 5 philosophers:

```
Found deadlock:  
    Phil 1  
    Phil 2  
    Phil 3  
    Phil 4  
    Phil 5
```

Setting up the dining philosophers is fairly straightforward. What is more complicated is finding out whether their eating habits have caused a thread deadlock in our system. To do that, we use the ThreadMXBean, as already described in [Newsletter 130](#). I modified it here for our purposes of testing several scenarios in which deadlocks can occur:

```
import java.lang.management.*;
import java.util.*;

/**
 * Used to check whether there are any known deadlocks by
 * querying the ThreadMXBean. It looks for threads deadlocked
 * on monitors (synchronized) and on Java 5 locks. Call check()
 * to get a set of deadlocked threads (might be empty).
 *
 * We can also add threads to the ignore set by calling the
 * ignoreThreads(Thread[]) method. For example, once we have
 * established that a certain deadlock exists, we might want to
 * ignore that until we have shut down our program and instead
 * concentrate on new deadlocks.
 */
public class DeadlockChecker {
```

```

private final static ThreadMXBean tmb =
    ManagementFactory.getThreadMXBean();
private final Set<Long> threadIdsToIgnore =
    new HashSet<Long>();

/**
 * Returns set of ThreadInfos that are part of the deadlock;
 * could be empty if there is no deadlock.
*/

public Collection<ThreadInfo> check() {
    Map<Long, ThreadInfo> map = new HashMap<Long, ThreadInfo>();
    findDeadlockInfo(map, tmb.findMonitorDeadlockedThreads());
    findDeadlockInfo(map, tmb.findDeadlockedThreads());
    return map.values();
}

public void ignoreThreads(Thread[] threads) {
    for (Thread thread : threads) {
        threadIdsToIgnore.add(thread.getId());
    }
}

private void findDeadlockInfo(Map<Long, ThreadInfo> result,
    long[] ids) {
    if (ids != null && ids.length > 0) {
        for (long id : ids) {
            if (!threadIdsToIgnore.contains(id)) {
                result.put(id, tmb.getThreadInfo(id));
            }
        }
    }
}
}

```

Now that we have the basic building blocks for figuring out when a deadlock occurs, let's examine what we can do once we find a deadlock.

Resolving Deadlocks

Databases also have deadlock situations and they typically resolve these by choosing a deadlock victim. Killing one of the queries frees up the other one to finish its work. The deadlock victim then tries again, hopefully with more success this time.

When I tried this the last time in Java, the monitor locked threads could not break out of the deadlock, neither with `stop()` nor with `interrupt()`. When the thread is in state BLOCKED, it

does not respond well to either of these signals.

However, with the new Java 5 locks, it might actually be possible to resolve a deadlock in Java code. In order to try out these ideas, I wrote an interface with two methods, method1() and method2(). These two methods should be called by two separate threads within a short time interval, which should then cause a deadlock. For example, here is the DeadlockingClass interface with the DeadlockedOnMonitors implementation. You will notice that the interface throws InterruptedException from the methods; we will use this in other examples to break out of deadlocks cleanly.

```
/**
 * Method1 and method2 should reliably deadlock every time they
 * get called at roughly the same time.
 */
public interface DeadlockingClass {
    void method1() throws InterruptedException;
    void method2() throws InterruptedException;
}

public class DeadlockedOnMonitors implements DeadlockingClass {
    private final Object lock1 = new Object();
    private final Object lock2 = new Object();

    public void method1() throws InterruptedException {
        synchronized (lock1) {
            Thread.sleep(1000);
            synchronized (lock2) {
                }
            }
        }
    }

    public void method2() throws InterruptedException {
        synchronized (lock2) {
            Thread.sleep(1000);
            synchronized (lock1) {
                }
            }
        }
    }
}
```

Next we show two implementations of DeadlockingClass. The first uses basic Java 5 locks, whereas the second locks interruptibly on the Java 5 locks. This allows us to interrupt a deadlock and thus exit cleanly. As you will see by the results later on, the best approach for

locking in Java 5 is to **always** use at least `Lock.lockInterruptibly()` and maybe even `Lock.tryLock()`.

```
import java.util.concurrent.locks.*;

public class DeadlockedOnLocks implements DeadlockingClass {
    private final Lock lock1 = new ReentrantLock();
    private final Lock lock2 = new ReentrantLock();

    public void method1() throws InterruptedException {
        lock1.lock();
        try {
            Thread.sleep(1000);
            lock2.lock();
            try {
            } finally {
                lock2.unlock();
            }
        } finally {
            lock1.unlock();
        }
    }

    public void method2() throws InterruptedException {
        lock2.lock();
        try {
            Thread.sleep(1000);
            lock1.lock();
            try {
            } finally {
                lock1.unlock();
            }
        } finally {
            lock2.unlock();
        }
    }
}

import java.util.concurrent.locks.*;

public class DeadlockedOnLocksInterruptibly
```

```

implements DeadlockingClass {
private final Lock lock1 = new ReentrantLock();
private final Lock lock2 = new ReentrantLock();

public void method1() throws InterruptedException {
    lock1.lockInterruptibly();
    try {
        Thread.sleep(1000);
        lock2.lockInterruptibly();
        try {
        }
        finally {
            lock2.unlock();
        }
    }
    finally {
        lock1.unlock();
    }
}

public void method2() throws InterruptedException {
    lock2.lockInterruptibly();
    try {
        Thread.sleep(1000);
        lock1.lockInterruptibly();
        try {
        }
        finally {
            lock1.unlock();
        }
    }
    finally {
        lock2.unlock();
    }
}
}

```

The last two types of deadlocking classes involve hybrid solutions (in keeping with the modern craze everything ecologically sustainable). In the first, thread one locks on a monitor, whereas thread two locks on a Java 5 lock. In the second, it's the other way round. The case where the Java 5 lock is locked interruptibly is left as an exercise to the reader.

```
import java.util.concurrent.locks.*;
```

```
public class DeadlockedOnMonitorThenLock implements DeadlockingClass {
    private final Object lock1 = new Object();
    private final Lock lock2 = new ReentrantLock();

    public void method1() throws InterruptedException {
        synchronized (lock1) {
            Thread.sleep(1000);
            lock2.lock();
            try {
            } finally {
                lock2.unlock();
            }
        }
    }

    public void method2() throws InterruptedException {
        lock2.lock();
        try {
            Thread.sleep(1000);
            synchronized (lock1) {
            }
        } finally {
            lock2.unlock();
        }
    }
}

import java.util.concurrent.locks.*;

public class DeadlockedOnLockThenMonitor implements DeadlockingClass {
    private final Lock lock1 = new ReentrantLock();
    private final Object lock2 = new Object();

    public void method1() throws InterruptedException {
        lock1.lock();
        try {
            Thread.sleep(1000);
            synchronized (lock2) {
            }
        } finally {
            lock1.unlock();
        }
    }
}
```

```

public void method2() throws InterruptedException {
    synchronized (lock2) {
        Thread.sleep(1000);
        lock1.lock();
        try {
        } finally {
            lock1.unlock();
        }
    }
}

```

Now that you have seen the five deadlocking classes, I need to explain how the test will be set up. For each deadlocking class, I create two threads (T1 and T2) that invoke methods `method1()` and `method2()` respectively. They should all be deadlocked within two seconds of starting the threads. In the first test, we call `T1.stop()`, wait for a second, then call `T2.stop()`. We should never call `stop()` in real code, but this is just to see what would happen if we did.

For the second test, we create another two threads (T3 and T4) and try out what happens if we `interrupt()` the threads. Here, the only class that would work is `DeadlockedOnLocksInterruptibly`.

In our tests, we can see that Java 5 locks are released when `Thread.stop()` is called. Threads locked on monitors with `synchronized` are not released. The hybrid deadlocks are released only because the thread locked on the Java 5 lock is stopped correctly. However, the `stop()` method does strange things to the threads deadlocked purely on monitors. You can see that it actually breaks the deadlock detection mechanism of Java. After we have called `stop()` on the `DeadlockedOnMonitors` class, we end up with a single thread being shown as deadlocked, which is clearly not correct. *If you ever needed another reason not to call the `Thread.stop()` method, this has got to be it!*

Since `Thread.stop()` is completely unsafe (with a third example of why to follow), the only way that we should attack a deadlock is with `Thread.interrupt()`. At the very least, we could try `interrupt()` as a first level attack, after choosing our deadlock victim. If that does not resolve the deadlock, we can interrupt the other thread. After that we could call `stop()` on both of the threads, to try to dislodge the deadlock. Irrespective of what we do, we also have to log a severe error that we have seen a deadlock in our system.

Here is my test code:

```

import java.lang.management.ThreadInfo;
import java.util.Collection;
import java.util.concurrent.TimeUnit;

public class DeadlockTest {
    private static final DeadlockChecker checker =
        new DeadlockChecker();

    public static void main(String[] args) throws Exception {
        DeadlockingClass[] problemClasses = {
            new DeadlockedOnLocks(),
            new DeadlockedOnLocksInterruptibly(),
            new DeadlockedOnMonitorThenLock(),
            new DeadlockedOnLockThenMonitor(),
            new DeadlockedOnMonitors(),
        };
        DeadlockTest test = new DeadlockTest();
        for (DeadlockingClass problemClass : problemClasses) {
            String name = problemClass.getClass().getSimpleName();
            System.out.println(name);
            System.out.println(name.replaceAll(".", "="));
            test.resolveByStop(problemClass);
            System.out.println();
            test.resolveByInterrupt(problemClass);
            System.out.println();
            System.out.println();
            System.out.println();
        }
    }

    private void resolveByStop(DeadlockingClass pc) {
        System.out.println("Testing deadlock resolve by stop()");
        System.out.println("-----");
        Thread[] threads = setupThreads(pc, "t1", "t2");
        sleepQuietly(2);
        check();
        System.out.println("Trying to resolve through t1.stop()");
        threads[0].stop();
        sleepQuietly(1);
        check();
        System.out.println("Trying to resolve through t2.stop()");
        threads[1].stop();
        sleepQuietly(1);
        check();
        checker.ignoreThreads(threads);
    }
}

```

```

private void resolveByInterrupt(DeadlockingClass pc) throws InterruptedException {
    System.out.println("Testing deadlock resolve by interrupt()");
    System.out.println("-----");
    Thread[] threads = setupThreads(pc, "t3", "t4");
    sleepQuietly(2);
    check();
    System.out.println("Trying to resolve by t3.interrupt()");
    threads[0].interrupt();
    sleepQuietly(1);
    check();
    System.out.println("Trying to resolve by t4.interrupt()");
    threads[1].interrupt();
    sleepQuietly(1);
    check();
    checker.ignoreThreads(threads);
}

private static void sleepQuietly(int seconds) {
    try {
        TimeUnit.SECONDS.sleep(seconds);
    } catch (InterruptedException e) {
        Thread.currentThread().interrupt();
    }
}

private static Thread[] setupThreads(
    final DeadlockingClass problemClass,
    final String name1, final String name2) {
    Thread[] threads = {
        new Thread(name1) {
            public void run() {
                try {
                    problemClass.method1();
                } catch (InterruptedException e) {
                    Thread.currentThread().interrupt();
                    System.out.println(name1 + " interrupted");
                } finally {
                    System.out.println(
                        "***** " + name1 + " Deadlock resolved");
                }
            }
        },
        new Thread(name2) {
            public void run() {
                try {
                    problemClass.method2();
                } catch (InterruptedException e) {

```

```

        System.out.println(name2 + " interrupted");
        Thread.currentThread().interrupt();
    } finally {
        System.out.println(
            "***** " + name2 + " Deadlock resolved");
    }
}
};

for (Thread thread : threads) {
    thread.setDaemon(true);
    thread.start();
    System.out.println("Started thread " + thread.getName() +
        " (" + thread.getId() + ")");
}
return threads;
}

private final static void check() {
    printInfo(checker.check());
}

private final static void printInfo(
    Collection<ThreadInfo> threads) {
    if (threads.isEmpty()) {
        System.out.println("++ No deadlock (detected)");
    } else {
        System.out.print("--- We have detected a deadlock: ");
        for (ThreadInfo info : threads) {
            System.out.print(info.getThreadName() +
                " (id=" + info.getThreadId() + ") ");
        }
        System.out.println();
    }
}
}
}

```

When we run this, we see the following output:

```

DeadlockedOnLocks
=====
Testing deadlock resolve by stop()
-----
Started thread t1 (8)

```

```
Started thread t2 (9)
--- We have detected a deadlock: t1 (id=8)  t2 (id=9)
Trying to resolve through t1.stop()
***** t1 Deadlock resolved
***** t2 Deadlock resolved
+++ No deadlock (detected)
Trying to resolve through t2.stop()
+++ No deadlock (detected)

Testing deadlock resolve by interrupt()
-----
Started thread t3 (10)
Started thread t4 (11)
--- We have detected a deadlock: t3 (id=10)  t4 (id=11)
Trying to resolve by t3.interrupt()
--- We have detected a deadlock: t3 (id=10)  t4 (id=11)
Trying to resolve by t4.interrupt()
--- We have detected a deadlock: t3 (id=10)  t4 (id=11)

DeadlockedOnLocksInterruptibly
=====
Testing deadlock resolve by stop()
-----
Started thread t1 (12)
Started thread t2 (13)
--- We have detected a deadlock: t1 (id=12)  t2 (id=13)
Trying to resolve through t1.stop()
***** t1 Deadlock resolved
***** t2 Deadlock resolved
+++ No deadlock (detected)
Trying to resolve through t2.stop()
+++ No deadlock (detected)

Testing deadlock resolve by interrupt()
-----
Started thread t3 (14)
Started thread t4 (15)
--- We have detected a deadlock: t3 (id=14)  t4 (id=15)
Trying to resolve by t3.interrupt()
***** t4 Deadlock resolved
t3 interrupted
***** t3 Deadlock resolved
+++ No deadlock (detected)
Trying to resolve by t4.interrupt()
+++ No deadlock (detected)
```

```
DeadlockedOnMonitorThenLock
=====
Testing deadlock resolve by stop()
-----
Started thread t1 (16)
Started thread t2 (17)
--- We have detected a deadlock: t2 (id=17)  t1 (id=16)
Trying to resolve through t1.stop()
***** t1 Deadlock resolved
***** t2 Deadlock resolved
+++ No deadlock (detected)
Trying to resolve through t2.stop()
+++ No deadlock (detected)
```

```
Testing deadlock resolve by interrupt()
```

```
-----
Started thread t3 (18)
Started thread t4 (19)
--- We have detected a deadlock: t4 (id=19)  t3 (id=18)
Trying to resolve by t3.interrupt()
--- We have detected a deadlock: t4 (id=19)  t3 (id=18)
Trying to resolve by t4.interrupt()
--- We have detected a deadlock: t4 (id=19)  t3 (id=18)
```

```
DeadlockedOnLockThenMonitor
```

```
=====
Testing deadlock resolve by stop()
-----
Started thread t1 (20)
Started thread t2 (21)
--- We have detected a deadlock: t2 (id=21)  t1 (id=20)
Trying to resolve through t1.stop()
--- We have detected a deadlock: t2 (id=21)  t1 (id=20)
Trying to resolve through t2.stop()
***** t2 Deadlock resolved
***** t1 Deadlock resolved
+++ No deadlock (detected)
```

```
Testing deadlock resolve by interrupt()
```

```
-----
Started thread t3 (22)
Started thread t4 (23)
+++ No deadlock (detected)
Trying to resolve by t3.interrupt()
```

```

+++ No deadlock (detected)
Trying to resolve by t4.interrupt()
+++ No deadlock (detected)

DeadlockedOnMonitors
=====
Testing deadlock resolve by stop()
-----
Started thread t1 (24)
Started thread t2 (25)
--- We have detected a deadlock: t2 (id=25)  t1 (id=24)
Trying to resolve through t1.stop()
--- We have detected a deadlock: t2 (id=25)  t1 (id=24)
Trying to resolve through t2.stop()
--- We have detected a deadlock: t2 (id=25)  t1 (id=24)

Testing deadlock resolve by interrupt()
-----
Started thread t3 (26)
Started thread t4 (27)
--- We have detected a deadlock: t4 (id=27)
Trying to resolve by t3.interrupt()
--- We have detected a deadlock: t4 (id=27)
Trying to resolve by t4.interrupt()
--- We have detected a deadlock: t4 (id=27)

```

As you can see from the output, the only locking mechanism that manages to recover from a deadlock situation cleanly is the Java 5 Lock.lockInterruptibly. Instead of putting the calling thread in a BLOCKED state, it enters the WAITING state, in which it can then be interrupted.

Stop() Breaks Condition.await()

As if you needed another reason not to call stop(), but in the following code:

```

lock.lock();
try {
    while(list.isEmpty()) {
        condition.await();
    }
    return list.removeFirst();
}

```

```
    } finally {
        lock.unlock();
    }
}
```

The `lock.unlock()` might throw an exception. In the synchronized construct, when you call `stop()` on a thread that is in the WAITING state, it first picks up the lock again before continuing. In Java 5 locks, they don't do that. Therefore, when we try to unlock(), we actually don't own the lock anymore. There is a workaround where we use the `ReentrantLock` interface to figure out whether our own thread holds the lock before unlocking, but that is also not really satisfactory. The best solution is to never ever ever. That is never with a big N. Never call `thread.stop()`.

In addition, if you do get deadlocks, proceed with extreme caution. You should use my deadlock detector to find them more quickly, but then you need to get rid of them from your code. Interrupting the threads is a bit like putting a band aid onto a gaping wound. It might buy you time and keep your system limping along for a bit more, but should not be seen as a long-term solution.

Kind regards

Heinz

P.S. I will be in the San Francisco Bay Area for the next few days for those of you who would like to get together for a chat. Just send me a note please so we can arrange something.

Issue 159 - The Law of Sudden Riches

Author: Dr. Heinz M. Kabutz

Date: 2008-05-05

Category: Concurrency

Java Versions: 5+

Abstract:

We all expect faster hardware to make our code execute faster and better. In this newsletter we examine why this is not always true. Sometimes the code breaks on faster servers or executes slower than on worse hardware.

Welcome to the 159th issue of **The Java(tm) Specialists' Newsletter**, sent from Athens. My two older kids and I went camping at Menies this week, a fairly deserted beach where people used to come from all over the world to worship Diktinna. We clambered up a hill with an amazing view of the Cretan sea and found a whole bunch of parts of ancient columns. An archeologists dream!

The Law of Sudden Riches

We are looking at a series of laws to make sense of Java concurrency. Just to remind you, here they are again. In this newsletter, we will look at the Law of Sudden Riches.

1. [The Law of the Sabotaged Doorbell](#)
2. [The Law of the Distracted Spearfisherman](#)
3. [The Law of the Overstocked Haberdashery](#)
4. [The Law of the Blind Spot](#)
5. [The Law of the Leaked Memo](#)
6. [The Law of the Corrupt Politician](#)
7. [The Law of the Micromanager](#)
8. [The Law of Cretan Driving](#)
9. [The Law of Sudden Riches](#)
10. [The Law of the Uneaten Lutefisk](#)
11. [The Law of the Xerox Copier](#)

The Law of Sudden Riches

Additional resources (faster CPU, disk or network, more memory) for seemingly stable system can make it unstable.

Imagine receiving an email from a stranger, telling you that you have inherited part of an estate from someone you never met. In the days before mass email scams, we received an email like that. My wife had inherited a motor home in the USA from someone who only

vaguely knew her father. Nowadays I would just delete an email like that, but in those days, we were trusting enough to believe them. As a result, we received a bequest of several thousand dollars. She put the bequest to good use and we are still receiving the blessings from it today.

Now imagine for a moment that instead of it having been a few thousand dollars, if it had been a few hundred million dollars. Our lives would most definitely have changed and I am not sure for the better! Very often, these sudden wins or massive inheritances, cause seemingly stable systems to start shaking.

We saw this happen with a company that had bought a faster server. The older server was coping fairly well with the Java application. The new server was 4x faster and had more cores. However, the new server would occasionally stop users from logging in to the system, typically when there was a high load. So on Friday afternoons, when the client reports needed to be generated, the system would not allow them to log in anymore.

This baffled us initially, since we would expect the opposite to happen. We thought that with better hardware, the system should be faster. However, we have seen this in several systems already, where better hardware makes the overall program perform worse (or not at all).

We traced the problem to a DOM tree that got corrupted due to a data race that happened less frequently in the older system. See also [The Law of the Corrupt Politician](#).

Seemingly Stable System

I need to make the point here that the bug was in the system all along, even in the old system. However, it occurred seldomly in the old system. Once a year, perhaps, the users could not log in. The administrator would restart the server and we would be none the wiser. With the faster server, which also corresponded with an increase in users, we got to the data race more quickly, thus causing the error once a week.

Let's look at some code that demonstrates this. To run this effectively, you need two CPUs, one modern and one old. We start with a BankAccount with a thread racing condition:

```
public class BankAccount {
    private int balance;

    public BankAccount(int balance) {
        this.balance = balance;
    }
}
```

```

public void deposit(int amount) {
    balance += amount;
}

public void withdraw(int amount) {
    deposit(-amount);
}

public int getBalance() {
    return balance;
}
}

```

To demonstrate that we have a problem in the code, we write a test where two threads continually deposit and withdraw 100. A third thread prints out the value of balance once a second.

```

import java.util.concurrent.*;

public class BankAccountTest {
    public static void main(String[] args) {
        final BankAccount account = new BankAccount(1000);
        for (int i = 0; i < 2; i++) {
            new Thread() {
                {
                    start();
                }

                public void run() {
                    while (true) {
                        account.deposit(100);
                        account.withdraw(100);
                    }
                }
            };
        }
        ScheduledExecutorService timer =
            Executors.newSingleThreadScheduledExecutor();
        timer.scheduleAtFixedRate(new Runnable() {
            public void run() {
                System.out.println(account.getBalance());
            }
        }, 1, 1, TimeUnit.SECONDS);
    }
}

```

```
}
```

If I run this with JDK 1.6 using the -client hotspot compiler, I see the following effect on my older machine. We see that the racing condition becomes evident, but only after running the code for some time.

```
Old Machine, JDK 1.6.0_05 -client
1000
1000
1100
1000
1000
1200
1100
1200
1100
1100
1200
1200
1200
1300 // incorrect
1300 // incorrect
1400 // incorrect
1500 // incorrect
1500 // incorrect
```

In the next test, I run the exact same code on my new MacBook Pro, with a 2.6GHz dual core 64-bit processor. I don't know which the exact processor model is, so if any of you clever Mac freaks have any idea how to figure that out, please drop me a line :-) The default Apple Java VM only supports the -server HotSpot compiler, so I run this example with the SoyLatte JVM distribution:

```
New MacBook Pro
java version "1.6.0_03-p3" Client VM
-118100
-116400
```

```
-112500  
-107500  
-102800  
-98100  
-94500  
-90600  
-86800
```

Right off the bat, this version was completely wrong. Thus if we have code that works with my old computer, it could go horrendously wrong in my new computer.

We now add the `-server` HotSpot compiler to the mix. In Java 6, the `-server` compiler optimizes really quite aggressively. However, this can make the system appear correct, even though it isn't. Let's run the code again using the `-server` hotspot compiler:

```
New MacBook Pro  
java version "1.6.0_03-p3" Server VM  
1000  
1000  
1000  
1000  
1000  
1000  
1000  
1000  
1000  
1000  
1000  
1000  
1000  
1000  
1000  
1000  
1000
```

It *appears* as if the system works correctly, without the need for synchronization, since we always get correct results. However, let's run the code again:

```
New MacBook Pro  
java version "1.6.0_03-p3" Server VM  
3241600  
3241600
```

```
3241600
3241600
3241600
3241600
3241600
3241600
3241600
```

You can probably see what is happening here - the value of the balance is "hoisted" into the thread stack, after which it is not updated, even if the two threads do actually update it. We can test this premise by making the balance variable volatile. This prevents the thread from caching the value. See [The Law of the Blind Spot](#).

```
public class BankAccount {
    private volatile int balance;

    public BankAccount(int balance) {
        this.balance = balance;
    }

    public void deposit(int amount) {
        balance += amount;
    }

    public void withdraw(int amount) {
        deposit(-amount);
    }

    public int getBalance() {
        return balance;
    }
}
```

When we run the code again with the `-server` option, we get more realistic results:

```
New MacBook Pro
balance set to "volatile"
```

```
java version "1.6.0_03-p3" Server VM  
-415480100  
-826545300  
-1243123400  
-1648635600  
-2019877800  
1875291096  
1460629096  
1056496196  
645105696  
230742396  
-183617704  
-597808704  
-1007827804
```

The lesson here is to be really careful when moving to better and faster hardware. Your system might be incorrect, without you realizing it and the new hardware might show you up.

Kind regards

Heinz

Issue 158 - Polymorphism Performance Mysteries Explained

Author: Dr. Heinz M. Kabutz

Date: 2008-04-01

Category: Performance

Java Versions: 6

Abstract:

In this newsletter, we reveal some of the polymorphism mysteries in the JDK. The HotSpot Server Compiler can distinguish between mono-morphism, bi-morphism and poly-morphism.

The bi-morphism is a special case which executes faster than poly-morphism.

Mono-morphism can be inlined by the compiler in certain circumstances, thus not costing anything at all.

A warm welcome to **The Java(tm) Specialists' Newsletter**, sent to you from the Island of Crete in the Mediterranean. Last week I was in Estonia, a delightful little country just south of Finland. The total population is about 1.2 million people. For those Dilbert lovers out there, Estonia is not Elbonia ... ;-) Therefore, do not expect to find waist-high mud and pigs on phone lines. Instead, you will find a booming little country with super fast internet, plus if you are lucky, a bit of snow and ice. It is a country that is worth visiting, especially now that they are a member of the Schengen states, which simplifies travel for people on odd passports (such as my South African). We had several programmers from other cities and countries on our Java Specialist Master Course.

Polymorphism Performance Mysteries Explained

After our [previous newsletter](#), Cliff Click, the author of the Server HotSpot VM, confirmed what we had observed by experimentation. Before we go into the details of mono-morphism, bi-morphism and poly-morphism, I would like to show you some code that you can use today to demonstrate to your friends and colleagues that Java is "infinitely" faster than C++ / C# / Pascal / [their favourite language here]. Infinitely faster, that is, at doing nothing :-)

Let's take the following class FastJavaFast. It will call a method 10000000000000000000000000000000 times in under a second. That is 10 to the power of 27. The method will not just be any method, but an overridden method (like virtual methods in C++). You need to start the program with the -server flag in Java 6.

```
public class FastJavaFast {
    public void test() {
```

```

for (int i = 0; i < 1000 * 1000 * 1000; i++) {
    test2();
}
}

public void test2() {
    for (int i = 0; i < 1000 * 1000 * 1000; i++) {
        test3();
    }
}

public void test3() {
    for (int i = 0; i < 1000 * 1000 * 1000; i++) {
        foo();
    }
}

// this is a "virtual" method, in the C++ sense
public void foo() {
}

public static void main(String[] args) {
    FastJavaFast fast = new FastJavaFast() {
        // I am overriding the foo() method
        public void foo() {
            // do nothing
        }
    };
    long time = System.currentTimeMillis();
    fast.foo();
    fast.test3();
    fast.test2();
    fast.test();
    time = System.currentTimeMillis() - time;
    System.out.println("time = " + time);
}
}

```

On my old Dell D800 notebook, I can execute the test in just a couple of milliseconds! Calling an overridden method 1,000,000,000,000,000,000,000,000 in under a second is simply blazingly fast! If we keep on calling test(), it eventually executes in 0 milliseconds. Swallow that one, C#! No other language can do *nothing* as fast as Java :-)

Now for the explanation. The first time that a method is run, the HotSpot Compiler tries to optimize the code. It uses something called On-Stack-Replacement (OSR), where it replaces some of the byte codes with optimized code. This is not the best optimization, but a start. If

you call the method a second time, the newly optimized method is called. Since we call the `foo()` method first, the HotSpot compiler determines that there is only one implementation of the method being called. When we call the `test3()` method, it has already determined that `foo()` can be inlined, and it eliminates the loop altogether. We then call `test2()`, where it again has already determined that `test3()` does nothing, thus eliminating the loop in `test2()`. Lastly, we call `test()` and similarly, the loop is optimized out.

You can test this by inlining the calls to `test2()` and `test3()`, thus having a method `test()` such as:

```
public void test() {
    for (int i = 0; i < 1000 * 1000 * 1000; i++) {
        for (int j = 0; j < 1000 * 1000 * 1000; j++) {
            for (int k = 0; k < 1000 * 1000 * 1000; k++) {
                foo();
            }
        }
    }
}
```

I estimate this should take approximately 79,274,479,959 years to complete on my Dell ...

The amazing performance of inlining code should affect how we code Java. We should stay clear of long methods and rather break them up more. Lots of small methods usually result in less duplicate copy and paste. Since the inlining is done for you by the Server HotSpot Compiler, there is no cost in calling the method.

This was one of the reasons for the strange results in my [previous newsletter](#), where the initial results were faster than subsequent runs. Instead of calling a test method many times in our `main()` method, we should rather extract that code into a separate method and call that from `main()`.

Thanks to Cliff Click's input, I changed my test code to show these effects. First off, instead of hard-coding the instances, I generate a bunch of random B subclasses and put them into an array. In my test, I then iterate over the A instances and call the `run()` method. Instead of showing you all the classes again, I will just show you one example and refer you to my [website for a zip file containing all the sources](#).

We start with a common interface for all the tests, called `Test.java`. The `run()` method calls B's `foo()` function, which might be a subclass instance. The `description()` returns a String showing what the test is doing.

```
public interface Test {
    public void run();
    public String description();
}
```

For example, here we have A3, B3, C3:

```
public class A3 implements Test {
    private final B3 b;
    public A3(B3 b) {
        this.b = b;
    }
    public void run() {
        b.f();
    }
    public String description() {
        return "Bi-Morphic: Two subclasses, via interface";
    }
}

public interface B3 {
    public void f();
}

public class C3 implements B3 {
    public void f() {
    }
}
```

In this new test class, we choose the test at start-up with a command-line argument. Depending on the test chosen, we generate instances of different test cases. The command line arguments are:

0 - No-morphic: Single subclass, pointed to directly

- 1 - Mono-Morphic: Single subclass, via interface
- 2 - Bi-Morphic: Two subclasses, via interface
- 3 - Poly-Morphic: Three subclasses, via interface
- 4 - Poly-Morphic: Four subclasses, via interface
- 5 - Poly-Morphic: Eight subclasses, via interface

We generate the test data by making instances of the relevant classes (A1, A2, etc.) with the corresponding B* classes and subclasses. For A4, the interface would be B4 and the implementations of B4 would be C4, D4, E4. When we run the test, we pick a random set of 10 instance, and then call the run() method on those. In order to "warm-up" the HotSpot Server Compiler, we run through all of the tests first, to let the compiler settle. Once that is completed, we do the actual runs. You can see from our results below that our standard deviation is rather low.

```

import java.lang.reflect.Constructor;
import java.util.Random;

public class PolymorphismCliffRandomTest {
    private static final int UPTO = 100 * 1000 * 1000;

    public static void main(String[] args) throws Exception {
        int offset = Integer.parseInt(args[0]);

        Test[] tests = generateTestData(offset);

        printDescriptions(tests);

        System.out.println("Warmup");
        testAll(tests);

        System.out.println("Actual run");
        printHeader(tests);

        testAll(tests);
    }

    private static void testAll(Test[] tests) {
        for (int j = 0; j < 10; j++) {
            runTests(tests);
        }
    }

    private static void printDescriptions(Test[] tests) {

```

```
System.out.println(tests[0].getClass().getSimpleName() +  
    ": " + tests[0].description());  
System.out.println();  
}  
  
public static void runTests(Test[] tests) {  
    long time = System.currentTimeMillis();  
    test(tests);  
    time = System.currentTimeMillis() - time;  
    System.out.print(time + "\t");  
    System.out.flush();  
    System.out.println();  
}  
  
public static void test(Test[] sources) {  
    Test t0 = makeRandomTest(sources);  
    Test t1 = makeRandomTest(sources);  
    Test t2 = makeRandomTest(sources);  
    Test t3 = makeRandomTest(sources);  
    Test t4 = makeRandomTest(sources);  
    Test t5 = makeRandomTest(sources);  
    Test t6 = makeRandomTest(sources);  
    Test t7 = makeRandomTest(sources);  
    Test t8 = makeRandomTest(sources);  
    Test t9 = makeRandomTest(sources);  
    for (int i = 0; i < UPTO / 10; i++) {  
        t0.run();  
        t1.run();  
        t2.run();  
        t3.run();  
        t4.run();  
        t5.run();  
        t6.run();  
        t7.run();  
        t8.run();  
        t9.run();  
    }  
}  
  
private static Test makeRandomTest(Test[] sources) {  
    return sources[((int) (Math.random() * sources.length))];  
}  
  
private static void printHeader(Test[] tests) {  
    System.out.print(tests[0].getClass().getSimpleName());  
    System.out.print('\t');  
    System.out.println();  
}
```

```

private static Test[] generateTestData(int offset)
    throws Exception {
    switch (offset) {
        default:
            throw new IllegalArgumentException("offset:" + offset);
        case 0:
            return fillSources(A1.class, B1.class, B1.class);
        case 1:
            return fillSources(A2.class, B2.class, C2.class);
        case 2:
            return fillSources(
                A3.class, B3.class, C3.class, D3.class);
        case 3:
            return fillSources(
                A4.class, B4.class, C4.class, D4.class, E4.class);
        case 4:
            return fillSources(
                A5.class, B5.class, C5.class, D5.class, E5.class,
                F5.class);
        case 5:
            return fillSources(
                A6.class, B6.class, C6.class, D6.class, E6.class,
                F6.class, G6.class, I6.class, J6.class);
    }
}

private static Test[] fillSources(
    Class<? extends Test> aClass,
    Class<?> bClass, Class<?>... bClasses)
    throws Exception {
    Test[] sources = new Test[1000];
    Random rand = new Random(0);
    Constructor<? extends Test> constr =
        aClass.getConstructor(bClass);
    for (int i = 0; i < sources.length; i++) {
        int offset = Math.abs(rand.nextInt() % bClasses.length);
        Object b = bClasses[offset].newInstance();
        sources[i] = constr.newInstance(b);
    }
    return sources;
}
}

```

We run this six times, one for each test case. Here are the test cases:

- A1: No-morphic: Single subclass, pointed to directly
- A2: Mono-Morphic: Single subclass, via interface
- A3: Bi-Morphic: Two subclasses, via interface
- A4: Poly-Morphic: Three subclasses, via interface
- A5: Poly-Morphic: Four subclasses, via interface
- A6: Poly-Morphic: Eight subclasses, via interface

On my server, the average results are:

| A1 | A2 | A3 | A4 | A5 | A6 |
|-------|-------|---------|-----------|-----------|-----------|
| 0 (0) | 0 (0) | 495 (9) | 2855 (31) | 2838 (18) | 2837 (10) |

When you run it with Java 6 -server, A1 and A2 come back with an average of 0 (0). The standard deviation is in brackets. In the case of a simple pointer to B or where there is a single subclass of B, the method call is inlined.

A3 is the bi-morphic case, where we have two subclasses of B. In this case, the results come back as 495 (9). The Server HotSpot compiler, according to Cliff Click, deals with bi-morphism as a special case of poly-morphism: "Where the server compiler can prove only **two** classes reach a call site, it will insert a type-check and then statically call both targets (which may then further inline, etc)."

The cases A4, A5 and A6 are all where we have more than two subclasses of B. In these cases, the results come back as 2855 (31), 2838 (18) and 2837 (10). They are pretty much equal and much slower than the bi-morphic case. According to Cliff Click: "*HotSpot uses an inline-cache for calls where the compiler cannot prove only a single target can be called. An inline-cache turns a virtual (or interface) call into a static call plus a few cycles of work. It's a 1-entry cache inlined in the code; the Key is the expected class of the 'this' pointer, the Value is the static target method matching the Key, directly encoded as a call instruction. As soon as you need 2+ targets for the same call site, you revert to the much more expensive dynamic lookup (load/load/load/jump-register). Both compilers use the same runtime infrastructure, but the server compiler is more aggressive about proving a single target.*"

It was nice having my discoveries confirmed by the guy who wrote the JVM Server Hotspot Compiler :-) Thanks to Cliff Click for taking the time to explain how it works. We also saw that Java is really fast at doing nothing, so be extra careful with empty loops that you inserted into your code instead of Thread.sleep(). Since Java 6, they are in danger of being removed.

We see again how good design wins: short methods, lots of polymorphism. If we do not need the polymorphism, it is optimized out of our code. Thus the penalty for using it is extremely small, usually non-existent where it counts.

I am in the process of expanding my Java Design Patterns Course to include more patterns. This has been one of my most popular courses to date. If this would interest you, [please let me know :-\)](#).

Kind regards

Heinz

P.S. In my last newsletter, I mentioned that we were trying to buy an olive grove on which to build our family house. We managed to finally buy it yesterday. If you can believe this, the lawyers are on strike in Greece at the moment. Even the kids at school strike! They say that you have not experienced true Greece if you have not been inconvenienced by at least one strike whilst on holiday here ;-)

Issue 157 - Polymorphism Performance Mysteries

Author: Dr. Heinz M. Kabutz

Date: 2008-03-06

Category: Performance

Java Versions: 5+

Abstract:

Late binding is supposed to be a bottleneck in applications - this was one of the criticisms of early Java. The HotSpot Compiler is however rather good at inlining methods that are being called through polymorphism, provided that we do not have very many implementation subclasses.

A warm welcome to the 157th issue of **The Java(tm) Specialists' Newsletter** sent to you from the beautiful island of Crete. We signed a purchase agreement on an olive grove on which we intend building our family home. Most of the 100 olive trees should survive our building project, which leaves me in a tricky situation. To harvest the olives will take a fair amount of manpower. The trees should produce a few hundred litres of fine pure cold pressed virgin olive oil, but this would require quite a bit of manual labour. Perhaps we should do a Java work camp where we harvest olives during the day and Java at night? Any volunteers? ;-)

Polymorphism Performance Mysteries

A few years ago, when I was still teaching **Bruce Eckel's** excellent Handson Java course in South Africa, I wrote some code to test the effects of polymorphism on performance. We know that late binding is supposed to be more expensive than static binding.

Polymorphism is essential to good design. In my **Design Patterns Course** we look at how polymorphism features even in the Singleton pattern. So is there a cost involved and if so, how much?

For microbenchmarks, I find it quite pointless to execute them on the client HotSpot compiler. Almost all of our serious Java programs will be executing on servers, so we might as well concentrate on the server HotSpot compiler.

Let's start with classes A1 and B1. A1 contains a field pointing to B1. In the run() method of A1, we call the f() method in B1. We would expect the HotSpot compiler to inline the method call, thus making the test exceedingly fast. The method f() is not **final**, but it should still be able to determine that it is not being overwritten in any subclasses, thus allowing it to be inlined.

```

public class A1 {
    private final B1 b;
    public A1(B1 b) {
        this.b = b;
    }
    public void run() {
        b.f();
    }
}

public class B1 {
    public void f() { }
}

```

The second set of classes is A2, B2 and C2. B2 is an **interface** with C2 the actual implementation. We would expect that the call to b.f() would take longer than in our first experiment, since we would expect to see late binding thanks to polymorphism.

```

public class A2 {
    private final B2 b;
    public A2(B2 b) {
        this.b = b;
    }
    public void run() {
        b.f();
    }
}

public interface B2 {
    public void f();
}

public class C2 implements B2 {
    public void f() {
    }
}

```

The third set of classes are A3, B3, C3 and D3. B3 is again an **interface** but this time with two implementations, C3 and D3.

```

public class A3 {
    private final B3 b;
    public A3(B3 b) {
        this.b = b;
    }
    public void run() {
        b.f();
    }
}

public interface B3 {
    public void f();
}

public class C3 implements B3 {
    public void f() {
    }
}

public class D3 implements B3 {
    public void f() {
    }
}

```

The last set of classes are similar to A3, but with two additional implementation classes. Say hello to: A4, B4, C4, D4, E4 and F4.

```

public class A4 {
    private final B4 b;

    public A4(B4 b) {
        this.b = b;
    }

    public void run() {
        b.f();
    }
}

public interface B4 {

```

```

public void f();
}
public class C4 implements B4 {
    public void f() {
    }
}
public class D4 implements B4 {
    public void f() {
    }
}
public class E4 implements B4 {
    public void f() {
    }
}
public class F4 implements B4 {
    public void f() {
    }
}

```

You might be wondering where I am going with this. Once we have these classes with the polymorphic call to f(), we can see if there is any difference at all in the performance of run().

```

public class PolymorphismTest {
    private static final int UPTO = 1000 * 1000 * 1000;

    public static void main(String[] args) {

        System.out.println("Empty\tA1\tA2\tA3\tA3/C+D\tA4");
        for (int j = 0; j < 10; j++) {

            long time = System.currentTimeMillis();
            for (int i = 0; i < UPTO; i++) {
                //empty loop
            }
            time = System.currentTimeMillis() - time;
            System.out.print(time + "\t");
            System.out.flush();

            A1 a = new A1(new B1());
            time = System.currentTimeMillis();
            for (int i = 0; i < UPTO; i++) {
                a.run();
            }
        }
    }
}
```

```

time = System.currentTimeMillis() - time;
System.out.print(time + "\t");
System.out.flush();

A2 a2 = new A2(new C2());
time = System.currentTimeMillis();
for (int i = 0; i < UPTO; i++) {
    a2.run();
}
time = System.currentTimeMillis() - time;
System.out.print(time + "\t");
System.out.flush();

A3 a3 = new A3(new C3());
time = System.currentTimeMillis();
for (int i = 0; i < UPTO; i++) {
    a3.run();
}
time = System.currentTimeMillis() - time;
System.out.print(time + "\t");
System.out.flush();

A3 a3_c3 = new A3(new C3());
A3 a3_d3 = new A3(new D3());
time = System.currentTimeMillis();
for (int i = 0, upto=UPTO/2; i < upto; i++) {
    a3_c3.run();
    a3_d3.run();
}
time = System.currentTimeMillis() - time;
System.out.print(time + "\t");
System.out.flush();

A4 a4_c4 = new A4(new C4());
A4 a4_d4 = new A4(new D4());
A4 a4_e4 = new A4(new E4());
A4 a4_f4 = new A4(new F4());
time = System.currentTimeMillis();
for (int i = 0, upto=UPTO/4; i < upto; i++) {
    a4_c4.run();
    a4_d4.run();
    a4_e4.run();
    a4_f4.run();
}
time = System.currentTimeMillis() - time;
System.out.println(time);
}

}

```

```
}
```

Before I show you the results, I need to warn you that the actual time to call the method is really small, irrespective of whether we use polymorphism or not. **Good design will produce consistently faster code.** So please do not go changing your code to remove polymorphism, but rather look at the results and try to figure out what is going on.

For the JDK 1.6.0_05 Server HotSpot, I got the following results. The numbers in brackets is the standard deviation.

| | A1 | A2 | A3 | A3/C+D | A4 |
|-------|---------|---------|---------|---------|-----------|
| Empty | 1126(1) | 1691(6) | 2255(9) | 1691(7) | 19469(28) |
| 0(0) | | | | | |

The empty loop has probably been eliminated by the HotSpot compiler. The call to f() in A1 has probably been inlined. Calls A2 and A3/C+D are virtually the same, whilst A3 takes longer. A4 looks like it has been optimized very little. It seems like the more active subclass objects we have, the worse the polymorphism performance.

For completeness, we should also look at the Client HotSpot compiler, plus the latest JDK 1.5.0_15.

| | A1 | A2 | A3 | A3/C+D | A4 |
|----------|------------|---------|-----------|-----------|-----------|
| Empty | 11299(482) | 9031(6) | 31583(34) | 30457(28) | 29896(24) |
| 1694(10) | | | | | |

| | A1 | A2 | A3 | A3/C+D | A4 |
|-------|---------|---------|----------|----------|------------|
| Empty | 1126(1) | 2258(9) | 3680(34) | 3384(10) | 13279(914) |
| 0(0) | | | | | |

| | A1 | A2 | A3 | A3/C+D | A4 |
|----------|------------|------------|-----------|------------|------------|
| Empty | 10311(429) | 10885(275) | 19659(99) | 19107(124) | 18511(287) |
| 2259(10) | | | | | |

I do not have a conclusive explanation for some of these values (hence the title of this newsletter).

There is another mystery that I have not mentioned yet. Usually when you run a benchmark, you discard the first set of values, since they usually are much slower than the rest. In this case, the opposite happens. Look at the first three rows of raw data in running the test with the 1.6.0_05 Server HotSpot compiler:

| Empty | A1 | A2 | A3 | A3/C+D | A4 |
|-------|------|------|------|--------|-------|
| 9 | 9 | 97 | 13 | 1151 | 19575 |
| 0 | 1125 | 1692 | 2268 | 1688 | 19524 |
| 0 | 1125 | 1687 | 2251 | 1690 | 19489 |

If you run just A1, A2 and A3 repeatedly as tests, they very quickly deteriorate in performance, so the method call is then not completely eliminated anymore. This is the first time that I have found an example where the HotSpot compiler slows down after warming up. Perhaps it recognizes that there may be other side effects with the additional classes being added? I do not know, but I do find the results curious.

I need to emphasize this once more, just to ensure that I minimize misunderstandings: *Good design beats minute performance gains*. Please do not go changing production code! The one place where this knowledge can be useful is when you are writing benchmarking code, just so you are aware of possible causes of interference in your test.

Final Methods

Making the f() methods **final** in the implementation classes did not change my results at all. In JDK 1.0 and 1.1, the final methods were inlined by the static compiler, whereas in JDK 1.2 onwards, they are inlined by the dynamic HotSpot compiler. I only ever make methods **final** for design reasons.

"Cost of Casting" Scapegoat for Polymorphism Cost?

The first time that I presented my **Java Specialist Master Course** last year, we had a discussion as to whether casting of objects is expensive. One of my students vaguely remembered an example where by reducing the hierarchy depth, they managed to significantly improve the performance. Unfortunately he could not remember any details, so we were not able to reproduce this scenario. However, I have for a while now had the suspicion that some of the alleged cost of casting might have been the cost of late binding?

Kind regards

Heinz

Issue 156 - The Law of Cretan Driving

Author: Dr. Heinz M. Kabutz

Date: 2008-02-26

Category: Concurrency

Java Versions: 5+

Abstract:

The Law of Cretan Driving looks at what happens when we keep on breaking the rules. Eventually, we might experience a lot of pain when we migrate to a new architecture or Java Virtual Machine. Even if we decide not to obey them, we need to know what they are. In this newsletter, we point you to some essential reading for every Java Specialist.

Welcome to the 156th issue of **The Java(tm) Specialists' Newsletter**. Whilst my brother was enjoying beautiful spring weather in Germany last week, we were building a snow man at our house. Due to a freak of nature, we had about 2 millimeters of snow at the beginning of last week. This shut down all the schools in Crete for the day and all the civil servants took the day off. Having grown up in South Africa, I had never built a snowman before since the vital ingredient (snow) was missing. However, I went to a German school as a kid, and they taught us the theory of snowman building ;-)

The Law of Cretan Driving

We are looking at a series of laws to make sense of Java concurrency. Just to remind you, here they are again. In this newsletter, we will look at the Law of Cretan Driving.

1. [The Law of the Sabotaged Doorbell](#)
2. [The Law of the Distracted Spearfisherman](#)
3. [The Law of the Overstocked Haberdashery](#)
4. [The Law of the Blind Spot](#)
5. [The Law of the Leaked Memo](#)
6. [The Law of the Corrupt Politician](#)
7. [The Law of the Micromanager](#)
8. [The Law of Cretan Driving](#)
9. [The Law of Sudden Riches](#)
10. [The Law of the Uneaten Lutefisk](#)
11. [The Law of the Xerox Copier](#)

The Law of Cretan Driving

The JVM does not enforce all the rules. Your code is probably wrong, even if it works.

After living here on the Island of Crete for sixteen months, we have come to appreciate the

local residents. The Cretans are extremely hospitable and welcoming, once you have broken through the initial barrier. It is a charming and amazing place to raise your children.

Traffic in Crete takes some getting used to. It still amazes me when someone parks his car in the middle of traffic, puts the emergency lights on and then goes to buy a packet of cigarettes from the periptero (little store on the side of the road). No one hoots or complains. We just wait until he is finished and then carry on driving. The driver obviously needed cigarettes, so "what can you do?"

Most of the speed limit signs have bullet wounds or are covered with stickers advertising the latest concerts. From what I can tell, very few drivers even know the rules. Occasionally we have a visitor from a stricter country, who can then cause chaos on the roads by crawling along at the official speed limit. Not that we drive overly fast; the roads are rough on the car.

The majority of drivers fail to wear a seat belt, even though the law requires it. My American friend soon fitted in and drove without wearing it. One day he was stopped by a policeman and slapped with a \$500 fine! Of course we were all rather surprised. When rules are hardly ever enforced, we begin ignoring them. This can have painful consequences.

The Java Virtual Machine Specification was written with a large range of hardware architectures in mind. It is thus not surprising that it contains several rules that would usually not be enforced. However, if you break the rules, especially in concurrency, you might be surprised by a rather large fine one day. What makes this tricky is that even though your code might execute correctly on all the JVM implementations to date, you might still contain concurrency bugs that only show up on other architectures.

As Java Specialists, we should have read the [Java Virtual Machine Specification](#) and the [Java Memory Model JSR 133](#) at least once. The [Java Language Specification](#) is also recommended reading for Java experts. (Now you have enough reading material to cure your insomnia...)

For example, here is an extract from the Java Memory Model:

"VM implementers are encouraged to avoid splitting their 64-bit values where possible. Programmers are encouraged to declare shared 64-bit values as volatile or synchronize their programs correctly to avoid this."

That is like saying: "Traffic police are encouraged to allow people to not wear seat belts, where possible. Drivers are encouraged to wear seat belts to avoid fines." The only way for the driver to be safe from fines would thus be to wear a seat belt all the time. However, the probability of a fine would still be so low that most of the time you would get away with breaking the law.

The Memory Model extract implies the following. Let's take the following code:

```
public class LongFields {
    private long value;
    public void set(long v) { value = v; }
    public long get()      { return value; }
}
```

If we call this method `set()` with values `0x12345678ABCD0000L` and `0x1111111111111111L` from two different threads, the value field could end up as `0x11111111ABCD0000L` or `0x1234567811111111L`.

If this happens in your code, it is a bug in *your* code, *not* in the JVM. I have not seen it happen in real life, since none of the JVMs that I used split the 64 bit values, to my knowledge.

Synchronize at the Right Places

Synchronization cannot be sprinkled around randomly. We need to synchronize in only the correct places, but nowhere else.

Too much synchronization causes contention. We can spot this when an increase in CPUs does not improve performance. See [The Law of the Micromanager](#).

Not having enough synchronization leads to racing conditions and corrupt data. See [The Law of the Corrupt Politician](#).

Fields might be written early if we try to avoid synchronization. See [The Law of the Leaked Memo](#).

Changes to shared fields might not be visible. See [The Law of the Blind Spot](#) (formally named The Law of South African Crime).

Kind regards from Crete. I am planning an excursion tomorrow with a friend visiting me from South Africa. Hopefully we will be going to [Balos tomorrow](#). Please let me know if you are planning to visit Crete so I can give you some good pointers on places worth visiting.

Heinz

Issue 155 - The Law of the Micromanager

Author: Dr. Heinz M. Kabutz

Date: 2008-01-16

Category: Concurrency

Java Versions: 5+

Abstract:

In good Dilbert style, we want to avoid having Pointy-Haired-Bosses (PHBs) in our code. Commonly called micromanagers, they can make a system work extremely inefficiently. My prediction is that in the next few years, as the number of cores increases per CPU, lock contention is going to be the biggest performance problem facing companies.

Welcome to the 155th issue of **The Java(tm) Specialists' Newsletter**. Last Tuesday, we had a bit of a break from the rain here on Crete, so I packed my laptop in my Suzuki Jimny and headed for a more **interesting location**.

As you are a Java Specialist reader this will almost certainly be of interest. We have recently launched a course specifically designed for top Java professionals like yourself. The course is the result of all the knowledge and experience gained from publishing 150 advanced Java Newsletters, teaching hundreds of seminars and writing thousands of lines of Java code and offers Java programmers the chance to truly master the Java Programming Language. For more information check out the **Master's Course**.

The Law of the Micromanager

We are looking at a series of laws to make sense of Java concurrency. Just to remind you, here they are again. In this newsletter, we will look at the Law of the Micromanager.

1. **The Law of the Sabotaged Doorbell**
2. **The Law of the Distracted Spearfisherman**
3. **The Law of the Overstocked Haberdashery**
4. **The Law of the Blind Spot**
5. **The Law of the Leaked Memo**
6. **The Law of the Corrupt Politician**
7. **The Law of the Micromanager**
8. **The Law of Cretan Driving**
9. **The Law of Sudden Riches**
10. **The Law of the Uneaten Lutefisk**
11. **The Law of the Xerox Copier**

The Law of the Micromanager

Even in life, it wastes effort and frustrates the other threads.

micro-management: to manage or control with excessive attention to minor details.

When tuning the performance of a system, we typically start by measuring the utilisation of hardware (CPU, disk, memory and IO). If we do not find the bottleneck, we go one step up and look at the number of threads plus the garbage collector activity. If we still cannot find the bottleneck, in all likelihood we have a thread contention problem, unless our measurements or test harness were incorrect.

In the past few laws, I have emphasised the importance of protecting data against corruption. However, if we add too much protection and at the wrong places, we may end up serializing the entire system. The CPUs can then not be utilized fully, since they are waiting for one core to exit a critical section.

A friend of mine sent me some code that he found during a code review. The programmers had declared `String WRITE_LOCK_OBJECT`, pointing to a constant String, like so:

```
String WRITE_LOCK_OBJECT = "WRITE_LOCK_OBJECT";
// Later on in the class
synchronized(WRITE_LOCK_OBJECT) { ... }
```

But wait a minute? String constants are stored in the intern table, thus if the String "`WRITE_LOCK_OBJECT`" occurs in two classes, they will point to the same object in the constant pool. We can demonstrate this with classes A and B, which each contain fields with a constant String.

```
public class A {
    private String WRITE_LOCK_OBJECT = "WRITE_LOCK_OBJECT";
    public void compareLocks(Object other) {
        if (other == WRITE_LOCK_OBJECT) {
            System.out.println("they are identical!");
            System.out.println(
                System.identityHashCode(WRITE_LOCK_OBJECT));
            System.out.println(
                System.identityHashCode(other));
        } else {
            System.out.println("they do not match");
        }
    }
}
```

```

    }

public static void main(String[ ] args) {
    A a1 = new A();
    A a2 = new A();
    a1.compareLocks(a2.WRITE_LOCK_OBJECT);
}
}

```

When you run A.main(), you see that with two instances of A, the field WRITE_LOCK_OBJECT is pointing to the same String instance.

```

they are identical!
11394033
11394033

```

Similarly, in B we compare the internal String to the String inside A, and again they are identical:

```

public class B {
    private String WRITE_LOCK_OBJECT = "WRITE_LOCK_OBJECT";

    public static void main(String[ ] args) {
        B b = new B();
        A a = new A();
        a.compareLocks(b.WRITE_LOCK_OBJECT);
    }
}

they are identical!
11394033
11394033

```

If the String had been created with **new**, it would have been a different object, but I still think this is a bad idiom to use for locking:

```

public class C {
    private String WRITE_LOCK_OBJECT =
        new String("WRITE_LOCK_OBJECT");

    public static void main(String[] args) {
        C c = new C();
        A a = new A();
        a.compareLocks(c.WRITE_LOCK_OBJECT);
        a.compareLocks(c.WRITE_LOCK_OBJECT.intern());
    }
}

```

As we would expect, the Strings are now different objects, but if we `intern()` it, it would point to the same object again:

```

they do not match
they are identical!
11394033
11394033

```

Since he had repeated this pattern throughout his code, his entire system was synchronizing on one lock object. This would not only cause terrible contention, but also raise the possibilities of deadlocks and livelocks. You could also lose signals by having several threads waiting on a lock by mistake. Basically, this is a really bad idea, so *please* do not use it in your code.

Amdahl's Law

Before we continue, we should consider Amdahl's Law in relation to parallelization. According to [Wikipedia](#), Amdahl's law states that if F is the fraction of a calculation that is sequential (i.e. cannot benefit from parallelization), and $(1 - F)$ is the fraction that can be parallelized, then the maximum speedup that can be achieved by using N processors is

$$\frac{1}{F + (1 - F)/N}$$

For example, if F is even just 10%, the problem can be sped up by a *maximum* of a factor of 10, no matter what the value of N is. For all practical reasons, the benefit of adding more cores decreases as we get closer to the theoretical maximum speed-up of 10. Here is an example of how we can calculate it:

```
public class Amdahl {
    private static double amdahl(double f, int n) {
        return 1.0 / (f + (1 - f) / n);
    }
    public static void main(String[] args) {
        for (int i = 1; i < 10000; i *= 3) {
            System.out.println("amdahl(0.1, " + i + ") = " + amdahl(0.1, i));
        }
    }
}
```

We can see from the output that the benefit of adding more cores decreases as we get closer to the theoretical maximum of 10:

```
amdahl(0.1, 1) = 1.0
amdahl(0.1, 3) = 2.5
amdahl(0.1, 9) = 5.0
amdahl(0.1, 27) = 7.5
amdahl(0.1, 81) = 9.0
amdahl(0.1, 243) = 9.642857142857142
amdahl(0.1, 729) = 9.878048780487804
amdahl(0.1, 2187) = 9.959016393442623
amdahl(0.1, 6561) = 9.986301369863012
```

(Thanks to Jason Oikonomidis and Scott Walker for pointing this out)

Atomics and Compare-And-Swap (CAS)

Since Java 5, the language includes support for atomics. Instead of synchronizing access to our fields, we can use atomic references or atomic primitives. Atomics use the **Compare-And-Swap approach**, supported by hardware (the CMPXCHG instruction on Intel). For example, if you want to do a `++i` operation with `AtomicInteger`, you would call the `AtomicInteger.addAndGet(int delta)` method. This would use an optimistic algorithm that assumes we will not have a conflict. If we do have a conflict, we simply try again. The `addAndGet()` method does something like this:

1. get the current value and store it in a local variable **current**
2. store the new value (`current + delta`) in a local variable **next**
3. call `compareAndSet` with the **current** value and the **next** value
 1. inside the `compareAndSet`, if the **current** value matches the value in the `AtomicInteger`, then return true; otherwise false
4. if `compareAndSet` returns true, we return **next**; otherwise start from 1.

We can thus have thread safe code without explicit locking. There is still a memory barrier though, since the field inside the `AtomicInteger` is marked as **volatile** to prevent the visibility problem seen (or perhaps *not seen* would be more appropriate ;-) in **The Law of the Blind Spot**.

If we look back at the example in our previous law, **The Law of the Corrupt Politician**, we can simplify it greatly by using an `AtomicInteger`, instead of explicitly locking. In addition, the throughput should be better as well:

```
import java.util.concurrent.atomic.AtomicInteger;

public class BankAccount {
    private final AtomicInteger balance =
        new AtomicInteger();

    public BankAccount(int balance) {
        this.balance.set(balance);
    }
    public void deposit(int amount) {
        balance.addAndGet(amount);
    }
    public void withdraw(int amount) {
        deposit(-amount);
    }
    public int getBalance() {
```

```
    return balance.intValue();
}
}
```

There are other approaches for reducing contention. For example, instead of using `HashMap` or `Hashtable` for shared data, we could use the `ConcurrentHashMap`. This map partitions the buckets into several sections, which can then be modified independently. When we construct the `ConcurrentHashMap`, we can specify how many partitions we would like to have, called the `concurrencyLevel` (default is 16). The `ConcurrentHashMap` is an excellent class for reducing contention.

Another useful class in the JDK is the `ConcurrentLinkedQueue`, which uses an efficient wait-free algorithm based on one described in [Simple, Fast, and Practical Non-Blocking and Blocking Concurrent Queue Algorithms](#). It also uses the compare-and-swap approach that we saw with the atomic classes.

Since Java 6, we also have the `ConcurrentSkipListMap` and the `ConcurrentSkipListSet` as part of the JDK.

There is a lot more that I could write on the subject of contention, but I encourage you to do your own research on this very important topic, which will become even more essential as the number of cores increases.

Kind regards from Crete

Heinz

Issue 154 - ResubmittingScheduledPoolExecutor

Author: Dr. Heinz M. Kabutz

Date: 2007-12-04

Category: Tips and Tricks

Java Versions: 5+

Abstract:

Timers in Java have suffered from the typical Command Pattern characteristics. Exceptions could stop the timer altogether and even with the new ScheduledPoolExecutor, a task that fails is cancelled. In this newsletter we explore how we could reschedule periodic tasks automatically.

Welcome to the 154th issue of **The Java(tm) Specialists' Newsletter**, which I started writing at 36000 feet flying in a brand new Airbus 320-200 en route to Frankfurt Sun Tech Days. Lovely plane, nice and smooth ride, probably partly due to the altitude it was flying at. The idea for this newsletter came from one of the participants of my recent talk in London on the Secrets of Concurrency; unfortunately this particular talk was not accessible to the general public.

Attention New Yorkers: On the 11th of December 2007, I have been invited to present a talk at the **NYC Java SIG** on the Secrets of Concurrency. If you are in the area and available that evening, it would be great to meet you! On the website they say that you need to RSVP. Our meeting place is unfortunately not that large, so I would strongly encourage you to let them know if you would like to come.

ResubmittingScheduledPoolExecutor

In the early days of Java, when you needed a timer to periodically do some task, you would create a thread, do the task, sleep for some set time, and repeat ad infinitum. If an Error or RuntimeException occurred during the execution of the task, the thread would typically die, also stopping the scheduled task. It suffered from another problem, in that we needed a new thread for every scheduled task in the system. Most of these threads would sit around idle most of the time.

Enter JDK 1.3. With this version, we had a `java.util.Timer` class that shared one thread between lots of different periodic tasks. We therefore have less unnecessary threads, but this brings with it other problems: First off, if one of the tasks causes an Error or a RuntimeException, the TimerTask thread dies, thus none of the other tasks progress either. Here is a demonstration program:

```

import java.util.*;

public class TimerTest {
    public static void main(String[] args) {
        Timer timer = new Timer();
        for(int i=0; i<5; i++) {
            final int i1 = i;
            timer.schedule(new TimerTask() {
                public void run() {
                    System.out.println("i = " + i1);
                    if (Math.random() < 0.1) {
                        throw new RuntimeException();
                    }
                }
            }, 1000, 1000);
        }
    }
}

```

Secondly, if a task takes a long time to complete, the other tasks might be held-up longer than desired, since the TimerTask only uses one thread to execute the tasks.

These problems were solved in Java 5 with the ScheduledThreadPoolExecutor class. First off, when one task misbehaves by causing unchecked exceptions, only it gets cancelled and the thread pool continues execution. Secondly, since you can create this ExecutorService with several threads, the likelihood that one task is blocking others is decreased.

According to the [definition in Goetz](#), livelock is a form of liveness failure in which a thread, while not blocked, still cannot make progress because it keeps retrying an operation that will always fail. This form of livelock often comes from overeager errorrecovery code that mistakenly treats an unrecoverable error as a recoverable one.

Therefore, cancelling the task that causes an exception is the only safe and sensible thing to do in general; otherwise we could potentially create a livelock. However, there are cases where we might want to reschedule a task that has failed.

By extending the ScheduledThreadPoolExecutor class and overriding the afterExecute(Runnable, Throwable) method, we are able to immediately discover when a task has failed. Unfortunately the Runnable that we see in the afterExecute() method parameter is not the same as the submitted Runnable. It is a wrapper object, that does not allow us to obtain the original submitted task. However, it is the same object as is returned by the schedule() methods. We also need to know what additional parameters were submitted to the ScheduledThreadPoolExecutor so that we can resubmit it.

As a start, we define a `ScheduledExceptionHandler` interface that is notified when an exception occurs. The `exceptionOccurred()` method then needs to return `true` if the task must be rescheduled; `false` otherwise.

```
public interface ScheduledExceptionHandler {
    /**
     * @return true if the task should be rescheduled;
     *         false otherwise
     */
    boolean exceptionOccurred(Throwable e);
}
```

The overridden class now uses that to bind the cancelled task to the submitted task. We use an `IdentityHashMap` to make sure that we are comparing actual objects, not their calculated hash code. Something which can help us is that the `Future.isDone()` method returns true if a task has been cancelled.

```
import java.util.*;
import java.util.concurrent.*;

public class ResubmittingScheduledThreadPoolExecutor
    extends ScheduledThreadPoolExecutor {

    /** Default exception handler, always reschedules */
    private static final ScheduledExceptionHandler NULL_HANDLER =
        new ScheduledExceptionHandler() {
            public boolean exceptionOccurred(Throwable e) {
                return true;
            }
        };
    private final Map<Object, FixedRateParameters> runnables =
        new IdentityHashMap<Object, FixedRateParameters>();
    private final ScheduledExceptionHandler handler;

    /**
     * @param reschedule when an exception causes a task to be
     *                   aborted, reschedule it and notify the
     *                   exception listener
     */
    public ResubmittingScheduledThreadPoolExecutor(int poolSize) {
```

```

    this(poolSize, NULL_HANDLER);
}

public ResubmittingScheduledThreadPoolExecutor(
    int poolSize, ScheduledExceptionHandler handler) {
    super(poolSize);
    this.handler = handler;
}

private class FixedRateParameters {
    private Runnable command;
    private long period;
    private TimeUnit unit;

    /**
     * We do not need initialDelay, since we can set it to period
     */
    public FixedRateParameters(Runnable command, long period,
                               TimeUnit unit) {
        this.command = command;
        this.period = period;
        this.unit = unit;
    }
}

public ScheduledFuture<?> scheduleAtFixedRate(
    Runnable command, long initialDelay, long period,
    TimeUnit unit) {
    ScheduledFuture<?> future = super.scheduleAtFixedRate(
        command, initialDelay, period, unit);
    runnables.put(future,
        new FixedRateParameters(command, period, unit));
    return future;
}

protected void afterExecute(Runnable r, Throwable t) {
    ScheduledFuture future = (ScheduledFuture) r;
    // future.isDone() is always false for scheduled tasks,
    // unless there was an exception
    if (future.isDone()) {
        try {
            future.get();
        } catch (ExecutionException e) {
            Throwable problem = e.getCause();
            FixedRateParameters parms = runnables.remove(r);
            if (problem != null && parms != null) {
                boolean resubmitThisTask =
                    handler.exceptionOccurred(problem);
            }
        }
    }
}

```

```
        if (resubmitThisTask) {
            scheduleAtFixedRate(parms.command, parms.period,
                parms.period, parms.unit);
        }
    }
} catch (InterruptedException e) {
    Thread.currentThread().interrupt();
}
}
```

We can see how this works by looking at an example that decides to retry five times and then gives up. Such a system would be more robust than just giving up after the first failure, and since we put a limit on the retries, we also avoid livelock issues.

```
import java.util.concurrent.*;
import java.util.concurrent.atomic.AtomicInteger;

public class ResubmittingTest {
    public static void main(String[] args)
        throws InterruptedException {
    ScheduledExecutorService service2 =
        new ResubmittingScheduledThreadPoolExecutor(
            5, new MyScheduledExceptionHandler());
    service2.scheduleAtFixedRate(
        new MyRunnable(), 2, 1, TimeUnit.SECONDS);
}

private static class MyRunnable implements Runnable {
    public void run() {
        if (Math.random() < 0.3) {
            System.out.println("I have a problem");
            throw new IllegalArgumentException("I have a problem");
        }
        System.out.println("I'm happy");
    }
}

/** As an example, we give up after 5 failures. */
private static class MyScheduledExceptionHandler
    implements ScheduledExceptionHandler {
    private AtomicInteger problems = new AtomicInteger();
```

```
public boolean exceptionOccurred(Throwable e) {  
    e.printStackTrace();  
    if (problems.incrementAndGet() >= 5) {  
        System.err.println("We give up!");  
        return false;  
    }  
    System.err.println("Resubmitting task to scheduler");  
    return true;  
}  
}  
}
```

That's all for now - hope you enjoyed this newsletter and will find it useful in your work!

Heinz

P.S. Another solution to managing exceptions is to catch them in the tasks themselves, thus not letting them bubble up. In our case we were looking for a general solution to the cancelled task problems.

Issue 153 - Timeout on Console Input

Author: Dr. Heinz M. Kabutz

Date: 2007-11-25

Category: Tips and Tricks

Java Versions: 5+

Abstract:

In this newsletter, we look at how we can read from the console input stream, timing out if we do not get a response by some timeout.

Welcome to the 153rd issue of **The Java(tm) Specialists' Newsletter**. Last week I presented my first Java Specialist **Master** Course (<http://www.javaspecialists.eu/courses/master.jsp>), so I would like to give some feedback on what happened. It is hard to describe the feelings that I had when I was teaching it. At first I was quite nervous about the pace, since I have never taught so much advanced information in such a short period of time. But then, after each day, instead of being drained, I felt refreshed and excited how it was panning out. I am now convinced that this is the *most comprehensive advanced Java course* you will find *anywhere*. Just the section on serialization is 70 slides, without being repetitive and boring. If you are a *good* Java programmer, and you want to go further, **have a look at the outline**.

Last week I received a rather interesting email from Maximilian Eberl, who was trying to write a simple Java console application to receive input from nurses who look after elderly and handicapped people. I must admit that my first suspicion was that this was a school project, so I quizzed him as to its application. Due to the proliferation of plagiarism on the internet, I refuse to help anybody who I suspect of either asking for my assistance with their school work (that is what tutors and teachers are supposed to be for) or when I feel that they are trying to embellish their skill, without giving credit for my help to their bosses. However, after working out a solution, and reading Maximilian's detailed explanation of why he was trying to do this, I was convinced that this is not school project material. (Besides that, he has the same name as my son, so that worked in his favour as well :-) It was thus an absolute pleasure helping him solve this interesting problem.

Timeout on Console Input

The problem that Maximilian was trying to solve was to have a simple console based application that would give users a certain time to answer the question. It would then timeout and retry a few times.

The reason this was a difficult problem to solve was that `System.in` blocks on the `readLine()` method. The thread state is RUNNABLE when you are blocking on IO, not WAITING or TIMED_WAITING. It therefore does not respond to interruptions. Here is some

sample code that shows the thread state of the Thread:

```
import java.io.*;

public class ReadLineTest {
    public static void main(String[] args) throws IOException {
        BufferedReader in = new BufferedReader(
            new InputStreamReader(System.in)
        );
        in.readLine();
    }
}
```

The thread dump clearly shows the state (I have stripped out unnecessary lines from the stack trace):

```
"main" prio=10 tid=0x08059000 nid=0x2e8a runnable
  java.lang.Thread.State: RUNNABLE
    at java.io.BufferedReader.readLine(BufferedReader.java:362)
    at ReadLineTest.main(ReadLineTest.java:8)
```

Since blocking reads cannot be interrupted with `Thread.interrupt()`, we traditionally stop them by closing the underlying stream. In our **Java Specialist Master Course**, one of the working examples is how to write a non-blocking server using Java NIO. (Told you it was a comprehensive course :-)) However, since `System.in` is a traditional stream, we cannot use non-blocking techniques. Also, we cannot close it, since that would close it for all readers.

One little method in the `BufferedStream` will be able to help us. We can call `BufferedStream.ready()`, which will only return true if the `readLine()` method can be called without blocking. This implies that the stream not only contains data, but also a newline character.

The first problem is therefore solved. However, if we read the input in a thread, we still need to find a way to get the `String` input back to the calling thread. The `ExecutorService` in Java 5 will work well here. We can implement `Callable` and return the `String` that was read. Unfortunately we need to poll until something has been entered. Currently we sleep for 200 milliseconds between checks, but we could probably make that much shorter if we want

instant response. Since we are sleeping, thus putting the thread in the TIMED_WAITING state, we can interrupt this task at any time. One last catch was that we do not want to accept an empty line as a valid input.

```

import java.io.*;
import java.util.concurrent.Callable;

public class ConsoleInputReadTask implements Callable<String> {
    public String call() throws IOException {
        BufferedReader br = new BufferedReader(
            new InputStreamReader(System.in));
        System.out.println("ConsoleInputReadTask run() called.");
        String input;
        do {
            System.out.println("Please type something: ");
            try {
                // wait until we have data to complete a readLine()
                while (!br.ready()) {
                    Thread.sleep(200);
                }
                input = br.readLine();
            } catch (InterruptedException e) {
                System.out.println("ConsoleInputReadTask() cancelled");
                return null;
            }
        } while ("".equals(input));
        System.out.println("Thank You for providing input!");
        return input;
    }
}

```

The next task is to call the `ConsoleInputReadTask` and timeout after some time. We do that by calling `get()` on the `Future` that is returned by the `submit()` method on `ExecutorService`.

```

import java.util.concurrent.*;

public class ConsoleInput {
    private final int tries;
    private final int timeout;

```

```

private final TimeUnit unit;

public ConsoleInput(int tries, int timeout, TimeUnit unit) {
    this.tries = tries;
    this.timeout = timeout;
    this.unit = unit;
}

public String readLine() throws InterruptedException {
    ExecutorService ex = Executors.newSingleThreadExecutor();
    String input = null;
    try {
        // start working
        for (int i = 0; i < tries; i++) {
            System.out.println(String.valueOf(i + 1) + ". loop");
            Future<String> result = ex.submit(
                new ConsoleInputReadTask());
            try {
                input = result.get(timeout, unit);
                break;
            } catch (ExecutionException e) {
                e.getCause().printStackTrace();
            } catch (TimeoutException e) {
                System.out.println("Cancelling reading task");
                result.cancel(true);
                System.out.println("\nThread cancelled. input is null");
            }
        }
    } finally {
        ex.shutdownNow();
    }
    return input;
}
}

```

We can put all this to the test with a little test class. It takes the number of tries and the timeout in seconds from the command line and instantiates that `ConsoleInput` class, reading from it and displaying the String:

```

import java.util.concurrent.TimeUnit;

public class ConsoleInputTest {
    public static void main(String[] args)

```

```

throws InterruptedException {
if (args.length != 2) {
    System.out.println(
        "Usage: java ConsoleInputTest <number of tries> " +
        "<timeout in seconds>");
    System.exit(0);
}

ConsoleInput con = new ConsoleInput(
    Integer.parseInt(args[0]),
    Integer.parseInt(args[1]),
    TimeUnit.SECONDS
);

String input = con.readLine();
System.out.println("Done. Your input was: " + input);
}
}

```

This seems to satisfy all the requirements that we were trying to fulfill. To be honest, when I first saw the problem, I did not think it could be done.

There is at least one way you could potentially get this program to fail. If you call the `ConsoleInput.readLine()` method from more than one thread, you run the very real risk of a data race between the `ready()` and `readLine()` methods. You would then block on the `BufferedReader.readLine()` method, thus potentially never completing.

Hopefully the rest of the program will be straightforward and soon the nurses for the disabled and elderly people will have some computer assistance, thanks to Maximilian Eberl. Good luck!

Kind regards from an airport somewhere in Europe :-)

Heinz

Issue 152 - The Law of the Corrupt Politician

Author: Dr. Heinz M. Kabutz

Date: 2007-11-12

Category: Concurrency

Java Versions: 5+

Abstract:

Corruption has a habit of creeping into system that do not have adequate controls over their threads. In this law, we look at how we can detect data races and some ideas to avoid and fix them.

Welcome to the 152nd issue of **The Java(tm) Specialists' Newsletter**, sent from a cold and windy island in the Mediterranean. We drove down to our favourite beach today, to let the children climb the trees and go wild on the sand. A lone Greek, Apostolis, who of his own accord keeps the beach spotlessly clean, was the only one brave enough to go for a swim. When the rain started to come down, we decided to rather head for our dry house, so here I am putting the finishing touches to this newsletter :-)

The Law of the Corrupt Politician

We are looking at a series of laws to make sense of Java concurrency. Just to remind you, here they are again. In this newsletter, we will look at the Law of the Corrupt Politician.

1. [The Law of the Sabotaged Doorbell](#)
2. [The Law of the Distracted Spearfisherman](#)
3. [The Law of the Overstocked Haberdashery](#)
4. [The Law of the Blind Spot](#)
5. [The Law of the Leaked Memo](#)
6. [**The Law of the Corrupt Politician**](#)
7. [The Law of the Micromanager](#)
8. [The Law of Cretan Driving](#)
9. [The Law of Sudden Riches](#)
10. [The Law of the Uneaten Lutefisk](#)
11. [The Law of the Xerox Copier](#)

The Law of the Corrupt Politician

In the absence of proper controls, corruption is unavoidable.

In a letter written in 1887, Lord Acton, famous historian, wrote: "Power tends to corrupt, and absolute power corrupts absolutely."

When we start programming threads, we wield a lot of power, perhaps more than is good for us. We need to make sure that we put controls in the right places to stop data races from corrupting our object state.

Here is one of my favourite examples for showing the effects of data races causing corrupted object state:

```
public class BankAccount {
    private int balance;
    public BankAccount(int balance) {
        this.balance = balance;
    }
    public void deposit(int amount) {
        balance += amount;
    }
    public void withdraw(int amount) {
        deposit(-amount);
    }
    public int getBalance() { return balance; }
}
```

One of the problems with this code is that the `+=` operation is not atomic.

We start with a `BankAccount` containing \$1000. Let us now imagine that two threads are calling `deposit` and `withdraw` at the same time. Thread 1 is scheduled to run first and adds \$100 to the account, thus updating the balance to \$1100. Thread 2 is then scheduled to run, it withdraws \$100, and writes the final balance of \$1000 back to balance.

However, there are also other effects that might become visible. Since we have no concurrency *controls* around our code, Thread 1 could start, get the balance field value, but before writing the new value, get swapped out. Thread 2 could now come in, read the old value of \$1000, withdraw \$100 and write the final value of \$900 into the field. Thread 1 would now start up again, not realising that the value has changed, overwrite the value \$900 with its own \$1100.

You can see the effects of data races very quickly with this test class. Be careful however, since some of the effects seen could also be due to **The Law of the Blind Spot** or **The Law of the Leaked Memo**

```

import java.util.concurrent.CountDownLatch;
import java.util.*;

public class TestCorruption {
    private static final int THREADS = 2;
    private static final CountDownLatch latch =
        new CountDownLatch(THREADS);
    private static final BankAccount heinz =
        new BankAccount(1000);

    public static void main(String[] args) {
        for (int i = 0; i < THREADS; i++) {
            addThread();
        }
        Timer timer = new Timer(true);
        timer.schedule(new TimerTask() {
            public void run() {
                System.out.println(heinz.getBalance());
            }
        }, 100, 1000);
    }

    private static void addThread() {
        new Thread() {
            { start(); }
            public void run() {
                latch.countDown();
                try {
                    latch.await();
                } catch (InterruptedException e) {
                    return;
                }
                while (true) {
                    heinz.deposit(100);
                    heinz.withdraw(100);
                }
            }
        };
    }
}

```

Before I propose a fix, I would like to discuss how you can detect data races. Unfortunately, I do not know of any easy solution. You could possibly instrument the JVM to detect when an object is modified from two threads, but this type of problem usually occurs in production under heavy load. It is not easy to diagnose. There are some tell-tale signs though:

- NullPointerException: when you see this in code that you would definitely not expect, have a look whether you might have encountered a race condition.
- Broken Assertions: when our postconditions fail, it could be a race condition.
- Corrupt Data Structures: a XML DOM tree that suddenly becomes corrupt, could quite possibly be a race condition on the nodes in the tree.

As a first step, we should perhaps add assertions to make sure that if it goes wrong, we know about it early on. I do not like the idea that you can switch off an assertion, but prefer the **Pragmatic Programmer** approach, where we leave them on, even in production. You need to then handle the assertion correctly if it fails:

```
public class BankAccount {
    private int balance;
    public BankAccount(int balance) {
        this.balance = balance;
    }
    public void deposit(int amount) {
        int check = balance + amount;
        balance = check;
        if (balance != check) {
            throw new AssertionError("Data Race Detected");
        }
    }
    public void withdraw(int amount) {
        deposit(-amount);
    }
    public int getBalance() { return balance; }
}
```

Interestingly, the assertion never fires on my machine, though it might well happen on your machine. My suspicion is that this is due to the local caching of the balance field (**The Law of the Blind Spot**). Let's eliminate that possibility by making balance **volatile**.

```
public class BankAccount {
    private volatile int balance;
    public BankAccount(int balance) {
        this.balance = balance;
    }
    public void deposit(int amount) {
```

```

int check = balance + amount;
balance = check;
if (balance != check) {
    throw new AssertionError("Data Race Detected");
}
}

public void withdraw(int amount) {
    deposit(-amount);
}
public int getBalance() { return balance; }
}

```

Now I can see the assertion failing very quickly:

```

1000
1100
Exception in thread "Thread-1" AssertionError: Data Race Detected
    at BankAccount.deposit(BankAccount.java:10)
    at BankAccount.withdraw(BankAccount.java:14)
    at TestCorruption$2.run(TestCorruption.java:38)
1000
1000
1000
1100

```

We are now at the point where we can look at some solutions to solve this problem. If we are running in an application server, we could set appropriate transaction isolation levels to avoid multiple threads calling the method at the same time.

Pre-Java 5, we used the **synchronized** mechanism to mark the critical section. If we use any sort of locking, we do not need to make the field volatile. Unfortunately, programmers have fallen into the habit of synchronizing on **this**, which is usually a bad idea. There is at least one other place in your code with a reference to your object, otherwise it would get garbage collected. If that other place also synchronizes on your object, you could see all sorts of strange liveness issues. A much better approach is to synchronize on a private final object:

```

public class BankAccount {
    private int balance;
    private final Object lock = new Object();
    public BankAccount(int balance) {
        this.balance = balance;
    }
    public void deposit(int amount) {
        synchronized(lock) {
            int check = balance + amount;
            balance = check;
            if (balance != check) {
                throw new AssertionError("Data Race Detected");
            }
        }
    }
    public void withdraw(int amount) {
        deposit(-amount);
    }
    public int getBalance() {
        synchronized(lock) {
            return balance;
        }
    }
}

```

To avoid visibility problems with balance, we synchronize the getBalance() method. Visibility problems could also be solved by keeping balance **volatile**.

The test class now prints either 1000, 1100 or 1200.

There are some issues with using standard Java locks. First off, you can never interrupt a thread that is blocked on a lock. This makes it impossible to shut down a deadlocked application cleanly.

To illustrate, we can use the Java 5 ReentrantLock to lock the critical section.

```

import java.util.concurrent.locks.*;

public class BankAccount {
    private int balance;
    private final Lock lock = new ReentrantLock();

```

```

public BankAccount(int balance) {
    this.balance = balance;
}

public void deposit(int amount) throws InterruptedException {
    lock.lockInterruptibly();
    try {
        int check = balance + amount;
        balance = check;
        if (balance != check) {
            throw new AssertionError("Data Race Detected");
        }
    } finally {
        lock.unlock();
    }
}

public void withdraw(int amount) throws InterruptedException {
    deposit(-amount);
}

public int getBalance() throws InterruptedException {
    lock.lockInterruptibly();
    try {
        return balance;
    } finally {
        lock.unlock();
    }
}
}

```

Another nice feature of the Java 5 locks is that you can differentiate between reads and writes. This is particularly useful when you have lots of threads reading concurrently, but only a few writing. The reads do not block each other, but the writes have exclusive access to the critical section. Here is how you could use it:

```

import java.util.concurrent.locks.*;

public class BankAccount {
    private int balance;
    private final ReadWriteLock lock =
        new ReentrantReadWriteLock();
}

```

```
public BankAccount(int balance) {
    this.balance = balance;
}

public void deposit(int amount) throws InterruptedException {
    lock.writeLock().lockInterruptibly();
    try {
        int check = balance + amount;
        balance = check;
        if (balance != check) {
            throw new AssertionError("Data Race Detected");
        }
    } finally {
        lock.writeLock().unlock();
    }
}

public void withdraw(int amount) throws InterruptedException {
    deposit(-amount);
}

public int getBalance() throws InterruptedException {
    lock.readLock().lockInterruptibly();
    try {
        return balance;
    } finally {
        lock.readLock().unlock();
    }
}
```

Note that it is easy to write a microbenchmark that shows that using the `ReadWriteLock` is slower than simply using the `synchronized` keyword. It is important to remember the effects that contention can have on system throughput.

Kind regards from Greece

Heinz

Issue 151 - The Law of the Leaked Memo

Author: Dr. Heinz M. Kabutz

Date: 2007-10-04

Category: Concurrency

Java Versions: 5+

Abstract:

In this fifth law of concurrency, we look at a deadly law where a field value is written early.

Welcome to the 151st issue of **The Java(tm) Specialists' Newsletter**, sent to you from the Island of Crete somewhere in the Mediterranean. I am sitting inside writing this newsletter, so no beach stories today. Outside is a chilly 24 Celsius / 75 Fahrenheit.

This coming Saturday (6th October 2007) I am presenting a talk on the Secrets of Concurrency at the **Java Hellenic User Group in Athens**. Next week on Thursday (11th October 2007), I am presenting the same talk at the **JFall 07** conference in Bussum, Netherlands. Come say "hi" if you are at either of these events :-)

The Law of the Leaked Memo

We are looking at a series of laws to make sense of Java concurrency. Just to remind you, here they are again. In this newsletter, we will look at the Law of the Leaked Memo.

1. [The Law of the Sabotaged Doorbell](#)
2. [The Law of the Distracted Spearfisherman](#)
3. [The Law of the Overstocked Haberdashery](#)
4. [The Law of the Blind Spot](#)
5. [The Law of the Leaked Memo](#)
6. [The Law of the Corrupt Politician](#)
7. [The Law of the Micromanager](#)
8. [The Law of Cretan Driving](#)
9. [The Law of Sudden Riches](#)
10. [The Law of the Uneaten Lutefisk](#)
11. [The Law of the Xerox Copier](#)

The Law of the Leaked Memo

This may never happen, but when it does, check your synchronization.

Imagine for a moment that you are the director of a listed company. You are going to announce tomorrow that 50% of the staff is being retrenched. You have secretly typed the

memo on your own laptop and personally made 500 photocopies. The reason why this information must not be leaked is because it will undoubtedly affect the share price, allowing someone with insider knowledge to make a quick buck and landing you in jail. However, after you make the 500 copies, you forget to remove the original memo from the photocopy machine. The secretary thread picks up the memo accidentally and soon there is widespread panic in the company.

It can happen in Java that without proper synchronization, a field value might be leaked before it should. The Java Memory Model gives the implementors the ability to reorder statements. To the Java thread, everything will appear to be in the same order, but other threads might see your early writes.

Let's look at an example:

```
public class EarlyWrites {
    private int x;
    private int y;

    public void f() {
        int a = x;
        y = 3;
    }

    public void g() {
        int b = y;
        x = 4;
    }
}
```

If two threads call f() and g(), what are the possible values for a and b at the end of the methods? The obvious answers are a=4, b=0 and a=0, b=3. Another fairly obvious answer is a=0, b=0 if the scheduler swaps between the threads after the first line of either method was executed.

However, there is a fourth possibility. Both threads might reorder the statements, resulting in:

```
public void f() {
    y = 3;
```

```

    int a = x;
}

public void g() {
    x = 4;
    int b = y;
}

```

This can result in a=4, b=3.

This reordering can happen at most places of the program. There are some restrictions though. For example, if you have a synchronized block, the hotspot compiler can reorder statements inside the block and it can move statements that lie outside the synchronized block into the block. However, it may not move code that is inside a synchronized block to the outside. We can use this restriction to prevent early writes from happening in critical sections of our code.

Another way of preventing early writes is to mark the field as **volatile**. We read in the Java Memory Model:

There is a happens-before edge from a write to a volatile variable v to all subsequent reads of v by any thread (where subsequent is defined according to the synchronization order)

This early writing is one of the reasons why the double-checked locking is broken in Java. The value of the field might get assigned before the constructor is called. Thus you might receive an uninitialised instance. A fix is to make the instance field volatile.

This law takes a bit of time to get your mind around. It is able to refute a lot of "clever" code that people have written to avoid synchronization.

Kind regards from Greece

Heinz

Issue 150 - The Law of the Blind Spot

Author: Dr. Heinz M. Kabutz

Date: 2007-09-05

Category: Concurrency

Java Versions: 5+

Abstract:

In this fourth law of concurrency, we look at the problem with visibility of shared variable updates. Quite often, "clever" code that tries to avoid locking in order to remove contention, makes assumptions that may result in serious errors.

Welcome to the 150th issue of **The Java(tm) Specialists' Newsletter**, sent to you from Marathi Beach, Crete. We have now lived here on Crete for almost a whole year. Did you know that in Greece, the children get a 14 week school holiday during summer? I think the long holiday is rooted in agriculture. During harvest time, the whole family used to work. For example, my wife spent some time living in Greece as a teenager and she had to assist in the tobacco harvest. For many Greek families, the summer is the main work time, with tourists coming in droves.

We have had the most incredible time as a family. We went to the beach on most mornings, snorkeling and swimming. I shifted my working hours to the afternoons and nights, giving us an amazing family time. It is hard to describe summer on the Greek Isles to someone who has not experienced it. I am already looking forward to the 14 weeks of school holidays in June 2008 :-)

The Law of the Blind Spot

We are looking at a series of laws to make sense of Java concurrency. Just to remind you, here they are again. In this newsletter, we will look at the Law of the Blind Spot.

1. [The Law of the Sabotaged Doorbell](#)
2. [The Law of the Distracted Spearfisherman](#)
3. [The Law of the Overstocked Haberdashery](#)
4. [**The Law of the Blind Spot**](#)
5. [The Law of the Leaked Memo](#)
6. [The Law of the Corrupt Politician](#)
7. [The Law of the Micromanager](#)
8. [The Law of Cretan Driving](#)
9. [The Law of Sudden Riches](#)
10. [The Law of the Uneaten Lutefisk](#)
11. [The Law of the Xerox Copier](#)

The Law of the Blind Spot

It is not always possible to see what other threads (cars) are doing with shared data (road).

Since I moved to Crete, the problem with the blind spot has seldom been a possibility. Almost all our roads on Crete are single-threaded (single-laned). However, in a multi-lane environment, this can cause serious accidents. The problem is with visibility: One thread (your car) cannot see what another thread (someone else's car) is doing on the shared data (the road that you are both travelling on). This typically causes problems with the check-then-act idiom. We **check** in our rear-view and side mirrors to see if there are any cars next to us. We then **act** by moving over into the slow lane. If someone was creeping up behind us, trying to overtake us on the inside, we will hear a lot of tire screeching, hooting and possibly cause a serious accident.

The Java Memory Model allows each thread to keep a local copy of fields. This flexibility allows JVM developers to add optimizations, based on the underlying architecture.

As a developer, you need to be aware that if you do not have adequate access control for your shared data, another thread might never see your change to the data.

This usually happens when we try to reason with logic in order to avoid synchronization. Unless you are synchronizing your data in some way, you can get visibility problems. Crime has become just a perception.

For example, if we look at the following code, MyThread might never end, even if we call `shutdown()`:

```
public class MyThread extends Thread {
    private boolean running = true;
    public void run() {
        while(running) {
            // do something
        }
    }
    public void shutdown() {
        running = false;
    }
}
```

The difficulty with writing correct code is that you do not necessarily know if your threaded code is correct on all implementations of the JVM through experimentation. You cannot prove the correctness of your code by running it successfully.

There are three ways of preventing the problem of field visibility:

1. Use locks to read and write access. Either the **synchronized** or the new Java 5 locks would work. It is important here to lock even if you are reading, otherwise you might see an outdated version of the field from your local cache.
2. Make field **volatile**. This will ensure that threads always read the value from the field and do not cache it locally. It is a cheaper option than synchronizing but is more difficult to get right. Typically, you would use **volatile** fields where the new value is not dependent on the previous value. Operations such as ++ and -- are not atomic so volatile will not be the correct solution.
3. The cheapest way to guarantee correct visibility is to make the field **final**. This puts a "freeze" on the value when the constructor has completed. Threads may then keep a local copy of the field and are guaranteed to see the correct value. This solution of course does not work if you need to change the field at some time, as in our example above.

We use **volatile** to solve the visibility problem:

```
public class MyThread extends Thread {
    private boolean volatile running = true;
    public void run() {
        while(running) {
            // do something
        }
    }
    public void shutdown() {
        running = false;
    }
}
```

The most up-to-date value of variable **running** is now visible to all threads reading it. We do not have a problem with each thread having its own *visibility of the value*, the way we have with the blind spot.

To summarise, the lesson of this newsletter is to be careful when reducing synchronization. Your code might run correctly on your JVM, but could fail when running on a production system, during high load.

Kind regards from Greece

Heinz

P.S. This law used to be called the Law of South African Crime, but was renamed to make it more understandable to non-South Africans :-)

Issue 149 - The Law of the Overstocked Haberdashery

Author: Dr. Heinz M. Kabutz

Date: 2007-08-20

Category: Concurrency

Java Versions: 5+

Abstract:

Learn how to write correct concurrent code by understanding the Secrets of Concurrency. This is the third part of a series of laws that help explain how we should be writing concurrent code in Java. In this section, we look at why we should avoid creating unnecessary threads, even if they are not doing anything.

Welcome to the 149th issue of **The Java(tm) Specialists' Newsletter**. We are continuing with the Secrets of Concurrency. Thank you for all your feedback. Enjoy!

The Law of the Overstocked Haberdashery

In this newsletter, I am continuing in looking at the laws that will help us write correct concurrent code in Java. Just to remind you, here they are again. In this newsletter, we will look at The Law of the Overstocked Haberdashery.

1. [The Law of the Sabotaged Doorbell](#)
2. [The Law of the Distracted Spearfisherman](#)
3. [**The Law of the Overstocked Haberdashery**](#)
4. [The Law of the Blind Spot](#)
5. [The Law of the Leaked Memo](#)
6. [The Law of the Corrupt Politician](#)
7. [The Law of the Micromanager](#)
8. [The Law of Cretan Driving](#)
9. [The Law of Sudden Riches](#)
10. [The Law of the Uneaten Lutefisk](#)
11. [The Law of the Xerox Copier](#)

The Law of the Overstocked Haberdashery

Having too many threads is bad for your application. Performance will degrade and debugging will become difficult.

Before we start with the law, I need to define the word haberdashery: A shop selling sewing wares, e.g. threads and needles.

One of the services I offer companies is to help them debug complex Java problems. This can take the form of code reviews or performance tuning. When I still lived in South Africa, I was contacted by a company that wanted an experienced Java programmer to fix some minor problems in their application. During our telephone conversation, I asked them to describe their system and immediately realised that they were creating a huge number of inactive threads.

I visited the director to discuss the scope of the work. Unfortunately for everyone involved, I made a fatal mistake right at the beginning of the engagement. I completely underquoted the hourly rate. The hourly rate should have been approximately four times what I quoted. I was immediately classed as a POJP (plain old Java programmer), rather than as a world-class Java *consultant*. Since I was now a POJP, the rate I had quoted sounded too high to the director and he tried to squeeze me even on that.

During the first discussions, I again raised the problem of them creating far too many threads. The director, who had now classed me as technically way below him, insisted that there would be no problem creating hundreds of thousands of threads, as long as they were not all active at the same time.

The technical manager then showed me the system and told me that they had a memory problem, or so he thought. He had personally refactored the code, but now it did not work at all anymore. I would need to go back to the original code and reduce the memory footprint. For example, I should change Vector to ArrayList and other such fantastic optimizations.

With no unit tests, I spent a few very frustrating weeks trying to follow the instructions of my employer. Of course, none of the things I did made any difference, because the problem lay with too many inactive threads being created. The haberdashery was overstocked with too many threads.

Every few days, the director would walk in and ask if I had fixed the problem yet and my answer would be "No, but I think you are creating too many threads." He would then tell me patiently that the number of threads was theoretically unlimited, you could have hundreds of thousands of threads without it causing any problems for your system.

Eventually, after being thoroughly fed up, I wrote a simple Java program that kept on creating threads, but kept them inactive by just putting them into the WAITING state. I ran it with him watching and it showed clearly that on their machine, it was limited to a few thousand.

The solution to the problem was simple and was something that I had mentioned in our very first meeting. All they had to do was to change from standard blocking IO to non-blocking New IO. Had I charged a high consulting rate, chances are that they would have listened to me in the first meeting and fixed the architecture. Instead of hiring me for three weeks, they would have had to pay for two hours of consulting. It would have saved time, money and frustration. It reminded me of the secret of consulting: **The more they pay you, the more they love you.**

Here is a small piece of code that you can run to find out how many *inactive* threads you can start on your JVM:

```

import java.util.concurrent.atomic.AtomicInteger;
import java.util.concurrent.CountDownLatch;

public class ThreadCreationTest {
    public static void main(String[] args)
        throws InterruptedException {
        final AtomicInteger threads_created = new AtomicInteger(0);
        while (true) {
            final CountDownLatch latch = new CountDownLatch(1);
            new Thread() {
                { start(); }
                public void run() {
                    latch.countDown();
                    synchronized (this) {
                        System.out.println("threads created: " +
                            threads_created.incrementAndGet());
                        try {
                            wait();
                        } catch (InterruptedException e) {
                            Thread.currentThread().interrupt();
                        }
                    }
                }
            };
            latch.await();
        }
    }
}

```

Depending on various factors (operating system, JVM version, Hotspot compiler version, JVM parameters, etc.), we will get different results. In my case, I managed to cause the actual JVM to crash:

```

Exception in "main" OutOfMemoryError: unable to create native thread
at java.lang.Thread.start0(Native Method)
at java.lang.Thread.start(Thread.java:597)

```

```

at ThreadCreationTest$1.<init>(ThreadCreationTest.java:8)
at ThreadCreationTest.main(ThreadCreationTest.java:7)
#
# An unexpected error has been detected by Java Runtime Environment:
#
# Internal Error (455843455054494F4E530E4350500134) #
# Java VM: Java HotSpot(TM) Client VM (1.6.0_01-b06 mixed mode)
# An error report file with more information is saved as hs_err.log
#
Aborted (core dumped)

```

We can increase our maximum number of inactive threads by decreasing the stack size. One of the limiting factors is the thread stack size, which stores local variables, parameters and the call stack. It typically does not need to be too large, unless you have very deep recursion. However, this does not necessarily solve the problem, because usually there is a problem in the architecture to start with.

You can reduce the stack size with the `-Xss<size_of_stack>` JVM parameter. For example, with `java -Xss48k ThreadCreationTest`, I was able to create 32284 threads. However, when I tried to create the 32285th thread, the JVM hung up so badly that I could not stop it, except with `kill -9`.

How many Threads is Healthy?

Creating a new thread should improve the performance of your system. In the same way that you should not use exceptions to control program flow, you should not use threads in that way either.

The number of *active* threads is related to the number of physical cores that you have in your system and on the pipeline length of your CPU. Typically running 4 active threads per CPU core is acceptable. Some architectures allow you to run up to 20 active threads per core, but they are the exception.

The number of *inactive* threads that you should have is architecture specific. However, having 9000 inactive threads on one core is far too many. What would happen if they all became active at once? Also, they consume a lot of memory. This you could manage by decreasing the stack size. However, if you do reach the limit, the JVM can crash without warning.

Traffic Calming

Instead of creating new threads directly, it is advisable to use thread pools. Thread pools are now part of the JDK and can be created either as fixed or expanding pools. One of the

benefits of thread pools is that you can limit the maximum number of active threads that will be used. We then submit our jobs to the pool and one of the available threads will execute it. By limiting the number of threads, we prevent our system being swamped under strain.

I have heard from various quarters that creating a thread is not as expensive as it used to be and that we should not use thread pools just to avoid the cost of creating a thread. My measurements tell a different tale. I would like to see evidence to prove the assertion that threads are cheap to create, before making a definite decision. Perhaps this is something that can be tuned with JVM parameters.

Here is a short test that creates a few thousand threads. We create a semaphore of size 10, which we will use to stop the system creating too many inactive threads:

```
import java.util.concurrent.*;

public class ThreadConstructTest {
    private static final int NUMBER_OF_THREADS = 100 * 1000;
    private Semaphore semaphore = new Semaphore(10);
    private final Runnable job = new Runnable() {
        public void run() {
            semaphore.release();
        }
    };

    public void noThreadPool() throws InterruptedException {
        for (int i = 0; i < NUMBER_OF_THREADS; i++) {
            semaphore.acquire();
            new Thread(job).start();
        }
        // wait for all jobs to finish
        semaphore.acquire(10);
        semaphore.release(10);
    }

    public void fixedThreadPool() throws InterruptedException {
        ExecutorService pool = Executors.newFixedThreadPool(12);
        for (int i = 0; i < NUMBER_OF_THREADS; i++) {
            semaphore.acquire();
            pool.submit(job);
        }
        semaphore.acquire(10);
        semaphore.release(10);
        pool.shutdown();
    }
}
```

```
public static void main(String[] args) throws Exception {
    ThreadConstructTest test = new ThreadConstructTest();

    long time = System.currentTimeMillis();
    test.noThreadPool();
    System.out.println(System.currentTimeMillis() - time);

    time = System.currentTimeMillis();
    test.fixedThreadPool();
    System.out.println(System.currentTimeMillis() - time);
}
```

I will not embarrass my laptop by publishing figures :-) Besides, I am writing this newsletter on battery power whilst sitting on Tersanas Beach, so it is even slower than usual. However, please run the code on your machine and let me know if you get figures that show that creating a new thread is insignificant on your architecture.

To summarise, the lesson of this newsletter is to not create too many threads, even if they are inactive. If you do use thread pools for traffic calming, make sure that you make the number of threads configurable. You might want to one day run your program on a 768 core **Azul Systems** machine.

Kind regards from Greece

Heinz

Issue 147 - The Law of the Distracted Spearfisherman

Author: Dr. Heinz M. Kabutz

Date: 2007-07-20

Category: Concurrency

Java Versions: 5+

Abstract:

Learn how to write correct concurrent code by understanding the Secrets of Concurrency.

This is the second part of a series of laws that help explain how we should be writing concurrent code in Java. We look at how to debug a concurrent program by knowing what every thread in the system is doing.

Welcome to the 147th issue of **The Java(tm) Specialists' Newsletter**. A few weeks ago, my laptop monitor gave up its ghost, after approximately 30000 hours of operation. I phoned Dell, full of fear that they might not have a service department on the Island of Crete. To my delight, the next morning, the parts arrived via courier and in the early evening, their engineer came to my house and swapped out the monitor.

The new laptop monitor is brighter than the original one, allowing me to sit in the shade at the beach and write this newsletter, while our children are splashing in the sea at Marathi Beach, with the fishing boats and the Lefka Ori Mountains in the background. Picture perfect. So thanks Dell, you made my day again :-)

The Law of the Distracted Spearfisherman

In my [previous newsletter](#), I introduced the ten laws of concurrency that can help you make sense of some rather tricky coding. Just to remind you, here they are again. In this newsletter, we will look at the Law of the Distracted Spearfisherman.

1. [The Law of the Sabotaged Doorbell](#)
2. [The Law of the Distracted Spearfisherman](#)
3. [The Law of the Overstocked Haberdashery](#)
4. [The Law of the Blind Spot](#)
5. [The Law of the Leaked Memo](#)
6. [The Law of the Corrupt Politician](#)
7. [The Law of the Micromanager](#)
8. [The Law of Cretan Driving](#)
9. [The Law of Sudden Riches](#)
10. [The Law of the Uneaten Lutefisk](#)
11. [The Law of the Xerox Copier](#)

The Law of the Distracted Spearfisherman

Focus on one thread at a time. The school of threads will blind you.

One of my hobbies as a teenager was catching fish with a speargun. Rather than sit on the rocks with a line and a baited hook, we would jump into the waves, where we could be more selective as to what exactly we wanted to catch. As I got older, the thought of being eaten by a great white shark became less appealing than the thrill of catching my own food.

The best defence for a fish is to swim right next to a bigger, better, fish. Something I learned from trying to shoot fish was that you need to stay focused, otherwise you end up with nothing. Once you zone in on a particular specimen, you need to commit to catching that one. If you let your eye wander to the next bigger fish, by the time you pull the trigger, you are too late and the fish are gone.

Let's apply this to Java Concurrency. You need to understand what **every** thread is doing in your system. Maybe you should try that now - create a thread dump of all of your threads and try to figure out what they are all doing.

Most of us know how to generate thread dumps by now, but here it is again for those who do not. The simplest way is to go to the console window that is running your Java process and then press CTRL+Break on Windows and CTRL+\ on Unix. In Unix you can also invoke kill -3 on the Java process, which dumps the thread dump to the console that is running the Java program.

If you do not have access to the console window or if you are running the process without a console, you have several other options for finding out what the threads are doing.

Since Java 5, we have a tool called jstack that ships as part of the JDK and which we can use to generate a thread dump. We can attach this to an existing running Java process to see what is going on. At the talk in Barcelona, someone mentioned that jstack does not show the deadlocks, but only the states of the threads, so I decided to create a deadlock in order to try this out.

Here is a class that should cause a deadlock after some time:

```
public class BadClass extends Thread {
    private final Object lock1;
    private final Object lock2;

    public BadClass(Object lock1, Object lock2) {
        this.lock1 = lock1;
```

```

    this.lock2 = lock2;
}

public void run() {
    while(true) {
        synchronized(lock1) {
            synchronized(lock2) {
                System.out.print('.');
                System.out.flush();
            }
        }
    }
}

public static void main(String[] args) {
    Object lock1 = new Object();
    Object lock2 = new Object();
    BadClass bc1 = new BadClass(lock1, lock2);
    BadClass bc2 = new BadClass(lock2, lock1);
    bc1.start();
    bc2.start();
}
}
}

```

When we run this in Java 5, we can see that the output of jstack is not that useful. The state of the threads is shown as BLOCKED, whereas some of them are actually in the WAITING state. It does not show the name of the threads.

```

JVM version is 1.5.0_11-b03
Thread 21068: (state = BLOCKED)

Thread 21077: (state = BLOCKED)
- BadClass.run() @bci=13, line=13 (Compiled frame)

Thread 21076: (state = BLOCKED)
- BadClass.run() @bci=13, line=13 (Compiled frame)

Thread 21072: (state = BLOCKED)

Thread 21071: (state = BLOCKED)
- java.lang.Object.wait(long) @bci=0 (Interpreted frame)
- java.lang.ref.ReferenceQueue.remove(long) @bci=44, line=116
- java.lang.ref.ReferenceQueue.remove() @bci=2, line=132

```

```

- Finalizer$FinalizerThread.run() @bci=3, line=159

Thread 21070: (state = BLOCKED)
- java.lang.Object.wait(long) @bci=0 (Interpreted frame)
- java.lang.Object.wait() @bci=2, line=474 (Interpreted frame)
- Reference$ReferenceHandler.run() @bci=46, line=116

```

In Java 6, they have improved jstack to also show detected deadlocks. You can now see exactly what is going on.

```

Full thread dump Java HotSpot(TM) Client VM (1.6.0_01-b06):

"Attach Listener" daemon prio=10 tid=0x08078400 nid=0x60ee
    runnable [0x00000000..0x00000000]
    java.lang.Thread.State: RUNNABLE

"DestroyJavaVM" prio=10 tid=0x08058400 nid=0x6084
    waiting on condition [0x00000000..0xb7d90080]
    java.lang.Thread.State: RUNNABLE

"Thread-1" prio=10 tid=0x080a2800 nid=0x608d
    waiting for monitor entry [0xb5842000..0xb5842fb0]
    java.lang.Thread.State: BLOCKED (on object monitor)
        at BadClass.run(BadClass.java:13)
        - waiting to lock <0x8c509ee0> (a java.lang.Object)
        - locked <0x8c509ee8> (a java.lang.Object)

"Thread-0" prio=10 tid=0x080a1400 nid=0x608c
    waiting for monitor entry [0xb5893000..0xb5893f30]
    java.lang.Thread.State: BLOCKED (on object monitor)
        at BadClass.run(BadClass.java:13)
        - waiting to lock <0x8c509ee8> (a java.lang.Object)
        - locked <0x8c509ee0> (a java.lang.Object)

"Low Memory Detector" daemon prio=10 tid=0x0808d400 nid=0x608a
    runnable [0x00000000..0x00000000]
    java.lang.Thread.State: RUNNABLE

"CompilerThread0" daemon prio=10 tid=0x0808bc00 nid=0x6089
    waiting on condition [0x00000000..0xb59e6838]
    java.lang.Thread.State: RUNNABLE

"Signal Dispatcher" daemon prio=10 tid=0x0808a800 nid=0x6088

```

```

    runnable [0x00000000..0xb5a37e10]
java.lang.Thread.State: RUNNABLE

"Finalizer" daemon prio=10 tid=0x08081800 nid=0x6087
    in Object.wait() [0xb5aca000..0xb5acb0b0]
java.lang.Thread.State: WAITING (on object monitor)
    at java.lang.Object.wait(Native Method)
    - waiting on <0x8c50a128> (a ReferenceQueue$Lock)
    at java.lang.ref.ReferenceQueue.remove
    - locked <0x8c50a128> (a ReferenceQueue$Lock)
    at java.lang.ref.ReferenceQueue.remove
    at java.lang.ref.Finalizer$FinalizerThread.run

"Reference Handler" daemon prio=10 tid=0x08080400 nid=0x6086
    in Object.wait() [0xb5b1b000..0xb5b1c030]
java.lang.Thread.State: WAITING (on object monitor)
    at java.lang.Object.wait(Native Method)
    - waiting on <0x8c509eb8> (a Reference$Lock)
    at java.lang.Object.wait(Object.java:485)
    at java.lang.ref.Reference$ReferenceHandler.run
    - locked <0x8c509eb8> (a Reference$Lock)

"VM Thread" prio=10 tid=0x08077000 nid=0x6085 runnable

"VM Periodic Task Thread" prio=10 tid=0x0808f000 nid=0x608b
    waiting on condition

JNI global references: 623

```

```

Found one Java-level deadlock:
=====
"Thread-1":
    waiting to lock monitor 0x080829d8
        (object 0x8c509ee0, a java.lang.Object),
    which is held by "Thread-0"
"Thread-0":
    waiting to lock monitor 0x08082974
        (object 0x8c509ee8, a java.lang.Object),
    which is held by "Thread-1"

```

```
Java stack information for the threads listed above:
=====
```

```

"Thread-1":
    at BadClass.run(BadClass.java:13)
    - waiting to lock <0x8c509ee0> (a java.lang.Object)
    - locked <0x8c509ee8> (a java.lang.Object)
"Thread-0":
```

```

at BadClass.run(BadClass.java:13)
- waiting to lock <0x8c509ee8> (a java.lang.Object)
- locked <0x8c509ee0> (a java.lang.Object)

```

Found 1 deadlock.

Now take some time to go over that thread dump again and make sure that you know exactly what every thread is doing. Don't go to the next one until you have figured out the one you are looking at. Remember the Law of the Distracted Spearfisherman! If you jump around between threads, without knowing exactly what each one is doing, you will end up with nothing at all!

Javva the Hutt tried out the advice in this newsletter and found lots of threads that did not need to be there. In his case, the quantity of threads was not an issue, but he concedes that it was a good exercise to know what is going on in the application. [Read the article here.](#)

One of the challenges of using tools like jstack and CTRL+Break is that the threads come back unsorted. This makes it difficult to compare two thread dumps of a live system, a technique that can help to see the progress of thread snapshots.

In [newsletter 132](#), I show how to generate nicely sorted thread dumps using JSP. You can include this on your server in an administrator page so that you can compare thread dumps taken at different times. Since the threads are sorted by name, you also have a better grouping of the threads, making it much easier to understand.

The JSP thread dumper does not show deadlocks either, but for that you can use my [Deadlock Detector](#) newsletter.

You might think that you have too many threads in your system and that you do not need to know what they are all up to. That is asking for trouble if you are writing multi-threaded code. A deadlock cannot be resolved in Java since you cannot force another thread to release its locks.

So remember the Law of the Distracted Spearfisherman. Zone in one thread at a time when you are trying to debug the system and make sure that you understand what it is doing before going any further. *The best defence for a fish is to swim next to a bigger, better fish!*

Kind regards from Marathi Beach :-)

Heinz

Issue 146 - The Secrets of Concurrency (Part 1)

Author: Dr. Heinz M. Kabutz

Date: 2007-06-26

Category: Concurrency

Java Versions: 5+

Abstract:

Learn how to write correct concurrent code by understanding the Secrets of Concurrency. This is the first part of a series of laws that help explain how we should be writing concurrent code in Java.

Welcome to the 146th issue of **The Java(tm) Specialists' Newsletter**. I am on my way to TheServerSide Java Symposium in Barcelona, where on Friday I will be presenting a new talk entitled "The Secrets of Concurrency". After many days of feverishly trying to come up with an outline for my talk, we ended up **praying for inspiration**. The writer's block disappeared immediately!

As mentioned in the previous newsletter, from July 2007, I will be offering **Java code reviews** for your team to get an expert's viewpoint of your Java system.

Secrets of Concurrency (Part 1)

Have you ever used the **synchronized** keyword? Are you absolutely sure that your code is correct? Here are ten laws that will help you obey the rules and write correct code, which I will explain over the next few newsletters:

1. **The Law of the Sabotaged Doorbell**
2. **The Law of the Distracted Spearfisherman**
3. **The Law of the Overstocked Haberdashery**
4. **The Law of the Blind Spot**
5. **The Law of the Leaked Memo**
6. **The Law of the Corrupt Politician**
7. **The Law of the Micromanager**
8. **The Law of Cretan Driving**
9. **The Law of Sudden Riches**
10. **The Law of the Uneaten Lutefisk**
11. **The Law of the Xerox Copier**

Introduction

After watching myself and those around me for 35 years, I have come to the conclusion that

we are exceedingly dim-witted. We walk through life only half-awake. I have gotten in my car to drive somewhere, then arrived at a completely different place to where I wanted to go to. Lost in thought, then lost on the road.

There are some people amongst us who are wide awake. They will hear a concept once and will immediately understand it and forever remember it. These are the exceptions. If like me, you struggle to understand concepts quickly or to remember things for a long time, *you are not alone*.

This is what makes conferences challenging. I know that I risk exposing my own lack of intelligence by saying this, but most of the talks I have attended have ended up going completely over my head. The first quarter of the talk would normally be alright, but suddenly I would be lost, with no way of recovery. I have great admiration for those that sit through an entire talk without opening up their laptops.

Just to illustrate the point, [here is a picture](#) taken of me during the "Enterprise Java Tech Day" in Athens. The caption on the Java Champion read "Heinz Kabutz busy preparing for his talk ..." Had the picture been taken from behind, the caption would have had to be "Heinz Kabutz busy playing backgammon on his laptop ..." In fairness, the speaker was talking in Greek, which I could not follow anyway.

To make my talk as simple as possible to understand, I have summarised The Secrets of Concurrency in ten easily remembered laws. Here is a challenge. Try to forget the name "The Law of the Sabotaged Doorbell". Let me know if you manage.

Law 1: The Law of the Sabotaged Doorbell

Instead of suppressing interruptions, manage them properly.

Many years ago we used to live in a house without a fence. We would constantly be interrupted by people coming to ring our door bell. Neighbours who had dropped a ball behind our house, kids selling brownies, men looking for a gardening job (after seeing how I looked after my lawn). When my son was born, we had to contend with a newborn baby and a constant stream of people waking up said baby with the chime. After some frustrating pacing, holding our new bundle of joy, I got fed up and sabotaged the doorbell by removing the battery.

Now all was calm, except that I was also hiding those chimes that were actually quite important. These InterruptedExceptions were being caused by other threads and sometimes I was supposed to listen to it.

How often have you seen code like this? I have even seen Sun Java Evangelists doing this in their talks!

```
try {
    Thread.sleep(1000); // 1 second
} catch(InterruptedException ex) {
    // ignore - won't happen
}
```

There are two questions we will try to answer:

1. What does InterruptedException mean?
2. How *should* we handle it?

The first question has a simple and a difficult answer. The simple answer is that the thread was interrupted by another thread.

A more difficult answer is to examine what happens and then see how we can use this in our coding. The thread is interrupted when another thread calls its `interrupt()` method. The first thing that happens is that the interrupted status of the thread is set to `true`. This interrupted status is important when we look at methods that put a thread into a WAITING or a TIMED_WAITING state. Examples of these methods are: `wait()`, `Thread.sleep()`, `BlockingQueue.get()`, `Semaphore.acquire()` and `Thread.join()`.

If the thread is currently in either the WAITING or TIMED_WAITING states, it immediately causes an `InterruptedException` and returns from the method that caused the waiting state. If the thread is not in a waiting state, then only the interrupted status is set, nothing else. However, if later on the thread calls a method that would change the state into WAITING or TIMED_WAITING, the `InterruptedException` is immediately caused and the method returns.

Note that attempting to lock on a monitor with `synchronized` puts the thread in BLOCKED state, not in WAITING nor TIMED_WAITING. Interrupting a thread that is blocked will do nothing except set the interrupted status to true. You cannot stop a thread from being blocked by interrupting it. Calling the `stop()` method similarly has no effect when a thread is blocked. We will deal with this in a later law.

The interrupted status is nowadays commonly used to indicate when a thread should be shut down. The problem with the `Thread.stop()` method was that it would cause an asynchronous exception at any point of your thread's execution code. The `InterruptedException` is thrown at well defined places. Compare it to firing employees when they are idle, rather than when they are in the middle of important work.

Note that the interrupted status is set to false when an `InterruptedException` is caused or when the `Thread.interrupted()` method is explicitly called. Thus, when we catch an `InterruptedException`, we need to remember that the thread is now not interrupted anymore! In order to have orderly shutdown of the thread, we should keep the thread set to "interrupted".

What should we do when we call code that may cause an `InterruptedException`? Don't immediately yank out the batteries! Typically there are two answers to that question:

1. **Rethrow the `InterruptedException` from your method.** This is usually the easiest and best approach. It is used by the new `java.util.concurrent.*` package, which explains why we are now constantly coming into contact with this exception.
2. **Catch it, set interrupted status, return.** If you are running in a loop that calls code which may cause the exception, you should set the status back to being interrupted. For example:

```
while (!Thread.currentThread().isInterrupted()) {
    // do something
    try {
        TimeUnit.SECONDS.sleep(1000);
    } catch (InterruptedException e) {
        Thread.currentThread().interrupt();
        break;
    }
}
```

Remember the Law of the Sabotaged Doorbell - don't just ignore interruptions, manage them properly!

In the next newsletter, I will explain the Law of the Distracted Spearfisherman and the Law of the Overstocked Haberdashery.

Kind regards from Athens Airport

Heinz

Issue 135 - Are you really Multi-Core?

Author: Dr. Heinz M. Kabutz

Date: 2006-11-06

Category: Performance

Java Versions: JDK 5

Abstract:

With Java 5, we can measure CPU cycles per thread. Here is a small program that runs several CPU intensive tasks in separate threads and then compares the elapsed time to the total CPU time of the threads. The factor should give you some indication of the CPU based acceleration that the multi cores are giving you.

Welcome to the 135th edition of **The Java(tm) Specialists' Newsletter**, now sent from a beautiful little island in Greece. We arrived safely two weeks ago and have been running around organising the basics, such as purchasing a vehicle, opening a bank account, getting cell phone contracts. Things happen really quickly in Greece. We can get my wife's Greek birth certificate in one week. In South Africa, this took me about 4 months to do. In about a week's time, I should be ready to apply for permanent residence here in Greece, so now I am the "First Java Champion in Greece" :))

The Java Performance Tuning course almost didn't happen, due to the hotel being washed into the sea by the storms. Fortunately my friend George Niavradakis (who sells **real estate in Crete**) jumped in and organised a new venue for us. And the dinner at Irene's was unforgettable, as always!

Are you really Multi-Core?

A few weeks ago, I presented a Java 5 and a Design Patterns Course in Cape Town to a bunch of developers. They were mostly developing in Linux, and one of the chaps was impressing us all with his multi-core machine. A Dell Latitude notebook, with tons of RAM, a great graphics card, etc. It looked really fast, especially the 3D effects of his desktop.

One of the exercises that we do in the Java 5 course is to measure the CPU cycles that a thread has used, as opposed to elapsed time. If you have one CPU in your machine, then these should be roughly the same. However, when you have several CPUs in your machine, the CPU cycles should be a factor more than the elapsed time. The factor should never be more than the number of actual CPUs, and may be less when you either have other processes running, or too many threads per CPU. Also, as all good computer scientists know, you can never scale completely linearly on one machine, so as you approach a large number of CPUs, the factor will grow more slowly.

Here is a short piece of code that starts 5 threads. Each thread runs through a loop from 0 to

999999999. For each thread we measure the thread CPU time with the new ThreadMXBean. These are added up and then we divide the total by the elapsed time (also called "wall clock time"). In order to not introduce contention, I'm using the AtomicLong and the CountDownLatch.

```

import java.lang.management.*;
import java.util.concurrent.CountDownLatch;
import java.util.concurrent.atomic.AtomicLong;

public class MultiCoreTester {
    private static final int THREADS = 5;
    private static CountDownLatch ct = new CountDownLatch(THREADS);
    private static AtomicLong total = new AtomicLong();

    public static void main(String[] args)
        throws InterruptedException {
        long elapsedTime = System.nanoTime();
        for (int i = 0; i < THREADS; i++) {
            Thread thread = new Thread() {
                public void run() {
                    total.addAndGet(measureThreadCpuTime());
                    ct.countDown();
                }
            };
            thread.start();
        }
        ct.await();
        elapsedTime = System.nanoTime() - elapsedTime;
        System.out.println("Total elapsed time " + elapsedTime);
        System.out.println("Total thread CPU time " + total.get());
        double factor = total.get();
        factor /= elapsedTime;
        System.out.printf("Factor: %.2f%n", factor);
    }

    private static long measureThreadCpuTime() {
        ThreadMXBean tm = ManagementFactory.getThreadMXBean();
        long cpuTime = tm.getCurrentThreadCpuTime();
        long total=0;
        for (int i = 0; i < 1000 * 1000 * 1000; i++) {
            // keep ourselves busy for a while ...
            // note: we had to add some "work" into the loop or Java 6
            // optimizes it away. Thanks to Daniel Einspanjer for
            // pointing that out.
            total += i;
            total *= 10;
        }
    }
}

```

```

    }
    cpuTime = tm.getCurrentThreadCpuTime() - cpuTime;
    System.out.println(total + " ... " + Thread.currentThread() +
        ": cpuTime = " + cpuTime);
    return cpuTime;
}
}
}

```

When I run this on my little D800 Latitude, I get:

```

Thread[Thread-3,5,main]: cpuTime = 1920000000
Thread[Thread-2,5,main]: cpuTime = 1920000000
Thread[Thread-1,5,main]: cpuTime = 1930000000
Thread[Thread-4,5,main]: cpuTime = 1920000000
Thread[Thread-0,5,main]: cpuTime = 1940000000
Total elapsed time 9759677000
Total thread CPU time 9630000000
Factor: 0.99

```

As always with performance testing, we have to be careful to run it on a quiet machine. If I copy a large file at the same time while running the test, I get:

```

Thread[Thread-0,5,main]: cpuTime = 1920000000
Thread[Thread-4,5,main]: cpuTime = 1990000000
Thread[Thread-2,5,main]: cpuTime = 1960000000
Thread[Thread-1,5,main]: cpuTime = 1980000000
Thread[Thread-3,5,main]: cpuTime = 1960000000
Total elapsed time 10979895000
Total thread CPU time 9810000000
Factor: 0.89

```

When I run the program twice in parallel on a quiet system, the Factor should be close to 0.5, hopefully:

```
Thread[Thread-3,5,main]: cpuTime = 4090000000
Thread[Thread-4,5,main]: cpuTime = 4070000000
Thread[Thread-0,5,main]: cpuTime = 2660000000
Thread[Thread-2,5,main]: cpuTime = 4020000000
Thread[Thread-1,5,main]: cpuTime = 2970000000
Total elapsed time 33988220000
Total thread CPU time 17810000000
Factor: 0.52
```

and the second run, started slightly later

```
Thread[Thread-1,5,main]: cpuTime = 3320000000
Thread[Thread-3,5,main]: cpuTime = 3120000000
Thread[Thread-4,5,main]: cpuTime = 3190000000
Thread[Thread-0,5,main]: cpuTime = 2590000000
Thread[Thread-2,5,main]: cpuTime = 3070000000
Total elapsed time 32353817000
Total thread CPU time 15290000000
Factor: 0.47
```

When we ran this program on the student's supa-dupa multi-core system, we were puzzled in that the factor was just below 1. We rebooted the machine into Windows, and the factor went up to just below 2. Fortunately we had a system administrator in the group, and he pointed out that the kernel on that Linux machine was incorrect. By simply putting the correct kernel on, the dream machine laptop was able to run at double the CPU cycles.

Your exercise for today is to find a multi-core or multi-cpu machine and see what factor you get. You need at least a JDK 5. Let me know how you fare ... :)

Just a hint: the number of threads should probably be a multiple of the number of CPUs or cores that you have available.

Kind regards from Greece

Heinz

Issue 134 - DRY Performance

Author: Kirk Pepperdine

Date: 2006-10-08

Category: Performance

Java Versions: JDK 1.5

Abstract:

As developers we often hear that performance often comes at the price of good design. However when we have our performance tuning hats on, we often find that good design is essential to help achieve good performance. In this article we will explore one example of where a good design choice has been essential in an effort to improve performance.

Welcome to the 134th edition of **The Java(tm) Specialists' Newsletter**. This week, I have the honour of welcoming Kirk Pepperdine as a contributing author to our newsletter. It is the first newsletter that Kirk is writing for us, and hopefully not the last. Kirk is a well known authority on the topic of Java Performance. I have listened to his talks at TSSJS and JavaZone, and have even shared the stage with Kirk in Johannesburg earlier this year.

Kirk Pepperdine is presenting his incredible **Java Performance Tuning course** in Greece in the week of the 23rd. On one of the evenings, I will join you for a wonderful meal at "Irene's", a little restaurant in the village of Xorafakia. Run by Irene herself, this tiny taverna cooks typical Cretan dishes and is frequented by the locals (always a good sign) and of course tourists. Come join us for an evening of good food and good company :)

And with that, I hand you over to Kirk Pepperdine.

DRY Performance

Just recently I had a conversation with a colleague who had determined that String was at the root of his performance woes. Normally I am sceptical when someone tells me that String utilization is the problem. It isn't that String utilization couldn't be a problem; it is just that String is used so ubiquitously that fixing it not only time consuming, there is no guarantee that "fixing" it will improve performance. However the application in question was performing a lot of String concatenations. The resulting String was then used to perform a lookup in a HashMap. What should have been a quick performance win was going to take a lot of effort because the original developer failed to adhere to the Don't Repeat Yourself (DRY) design principle.

One of the goals in this exercise was to dispel the myth that you need to sacrifice good design or write overly complex code in order to achieve good performance. That there is wide spread belief in this myth is unfortunate because I have found that violating design principles or writing overly complex code is often the stumbling block to achieving good

performance. In this case, not following DRY left the String concatenation operation and subsequent HashMap lookup scattered all throughout the application. This has two knock-on effects. The first being that any operation that is a bottleneck will be difficult to find as the scattering dilutes or hides the effect. The second is that if you are lucky enough to find that an operation is a bottleneck, you will be obliged to trawl through the code looking for every instance of that operation. Lastly what happens is that teams blindly change String operations without any evidence to say that a change would make a difference.

Fortunately my colleague had identified a single use case and that allowed me to write a focused benchmark. A benchmark in which we could explore the benefits of one strategy over another. In addition, I wanted to demonstrate how DRY would help and without hurting performance. Thus I structured the benchmark to follow that design principle.

```

public class Person {
    private int id;
    private final String firstName;
    private final String lastName;
    public Person(String firstName, String lastName) {
        this.firstName = firstName;
        this.lastName = lastName;
    }
    public int getId() { return id; }
    public void setId(int id) { this.id = id; }
}

import java.util.HashMap;
public class AllPersons {
    private HashMap personsById = new HashMap();
    private HashMap personsByName = new HashMap();
    public Person addPerson(String firstName, String lastName) {
        Person person = new Person(firstName, lastName);
        personsById.put(new Integer(person.getId()), person);
        personsByName.put(firstName + lastName, person);
        return person;
    }
    public Person findPersonsById(int id) {
        return (Person) personsById.get(new Integer(id));
    }
    public Person findPersonsByName(String firstName,
                                   String lastName) {
        return (Person) personsByName.get(firstName + lastName);
    }
}

```

As you can see, the AllPersons class provides a nice neat home for all of the needed CRUD operations. Note that the class contains two HashMaps. The second is not necessary for this benchmark but demonstrates one of the benefits of following DRY. As the queries build up you may find that you need alternative indexing schemes. Just as adding an extra index in a database, the personsByIndex Map provides that functionality. Other benefits will become apparent when we move to the second part of the benchmark. The next step was to setup a benchmark harness. The code for this is just below.

```

import java.util.*;
public class TestHarness implements Runnable {
    private static String[] FIRST_NAMES = {"Ted", "Fred", "Andy",
        "Gromit", "Wallace"};
    private static String[] LAST_NAMES = {"Mouse", "Duck", "Pascal",
        "Kabutz", "Monster", "Dread", "Crocket"};
    private StringArray firstNames =
        new StringArray(FIRST_NAMES);
    private StringArray lastNames =
        new StringArray(LAST_NAMES);
    private AllPersons allPersons;
    private int numberofIterations;

    public TestHarness(int numberofIterations) {
        this.numberofIterations = numberofIterations;
    }
    public void init() {
        allPersons = new AllPersons();
        for (int i = 0; i < 250000; i++) {
            allPersons.addPerson(
                firstNames.nextAsDeepCopy(),
                lastNames.nextAsDeepCopy());
        }
    }

    public void run() {
        for (int i = 0; i < numberofIterations; i++) {
            allPersons.findPersonsByName(
                firstNames.next(),
                lastNames.next());
        }
    }

    public static void main(String[] args) throws InterruptedException {
        ArrayList processes = new ArrayList();
        ArrayList threads = new ArrayList();
        long setup = System.currentTimeMillis();
        for (int i = 0; i < Integer.parseInt(args[0]); i++) {
    
```

```

TestHarness harness = new TestHarness(
    Integer.parseInt(args[1]));
harness.init();
processes.add(harness);
}
setup = System.currentTimeMillis() - setup;

System.gc();
Thread.sleep(1000);
System.gc();
Thread.sleep(1000);

long processing = System.currentTimeMillis();
for (Iterator it = processes.iterator(); it.hasNext();) {
    TestHarness harness = (TestHarness) it.next();
    Thread thread = new Thread(harness);
    thread.start();
    threads.add(thread);
}
System.out.println("waiting");
for (Iterator it = threads.iterator(); it.hasNext();) {
    Thread thread = (Thread) it.next();
    thread.join();
}
processing = System.currentTimeMillis() - processing;
System.out.println("Setup time : " + setup);
System.out.println("Processing time : " + processing);
}

public static class StringArray {
    private int nextString = 0;
    private String[] array;

    public StringArray(String[] strings) {
        this.array = strings.clone();
    }

    public String next() {
        String result = array[nextString++];
        nextString %= array.length;
        return result;
    }

    public String nextAsDeepCopy() {
        return new String(next());
    }
}
}

```

The harness is pretty straight forward. After doing some initialization I clear the deck to prepare for the run. The clearing consists of a pair of calls to System.gc() followed by a sleep. The reason why gc is called twice is that the second call takes care of collecting any residual objects that may have required finalization. The call to sleep removes any interference from the concurrent portion of the call to gc. With the cleanup complete, we have more assurance that we are only measuring what we believe we are measuring.

What I wanted a measure of was the effect of memory management on the time to complete a fixed unit of work. The results of the runs are listed in table below.

| Setup | Test |
|--------|--------|
| 25.187 | 324.32 |
| 25.328 | 327.81 |
| 25.328 | 316.73 |

There is some variation in the numbers but what is important to note is that the magnitude remains stable. During the run I made the observation that CPU utilization bounced about in the range of 77 to 89%. My guesstimate was that overall CPU utilization averaged 83%. This was a bit surprising given that the benchmark runs un-throttled but more on this later on. The next step was to implement the alternative solution. Since the key in the lookup was a composite of two String objects, what I proposed was to introduce a new class. In the source for CompositeKey we can see that the concatenation has been replaced two instance variables.

```

public class CompositeKey {
    private String key1, key2;
    public CompositeKey(String key1, String key2) {
        this.key1 = key1;
        this.key2 = key2;
    }

    public final boolean equals(Object o) {
        if (this == o) return true;
        if (!(o instanceof CompositeKey)) return false;

        CompositeKey that = (CompositeKey) o;
        if (key1 != null ? !key1.equals(that.key1) : that.key1 != null)
            return false;
        if (key2 != null ? !key2.equals(that.key2) : that.key2 != null)
            return false;
    }
}

```

```

    return true;
}

public final int hashCode() {
    int result;
    result = (key1 != null ? key1.hashCode() : 0);
    result = 31 * result + (key2 != null ? key2.hashCode() : 0);
    return result;
}

public String getKey1() {
    return key1;
}
public String getKey2() {
    return key2;
}
}

```

Since we followed DRY, altering the benchmark to use the new class is simple. All of our changes have been isolated to our domain specific collection class. Note that changes to the harness are missing from the code (listed below) that has been changed. This implies that we are presenting the benchmark the same unit of work offered under the same conditions. This is an important property that allows us to directly compare results.

```

public Person addPerson(String firstName, String lastName) {
    Person person = new Person(firstName, lastName);
    personsById.put(new Integer(person.getId()), person);
    personsByName.put(new CompositeKey(firstName, lastName), person);
    return person;
}
public Person findPersonsByName(String firstName, String lastName) {
    return (Person)personsByName.get(new CompositeKey(firstName, lastName));
}

```

With these changes in place I was quickly able to re-run the benchmark. The results of these runs are listed in the next table.

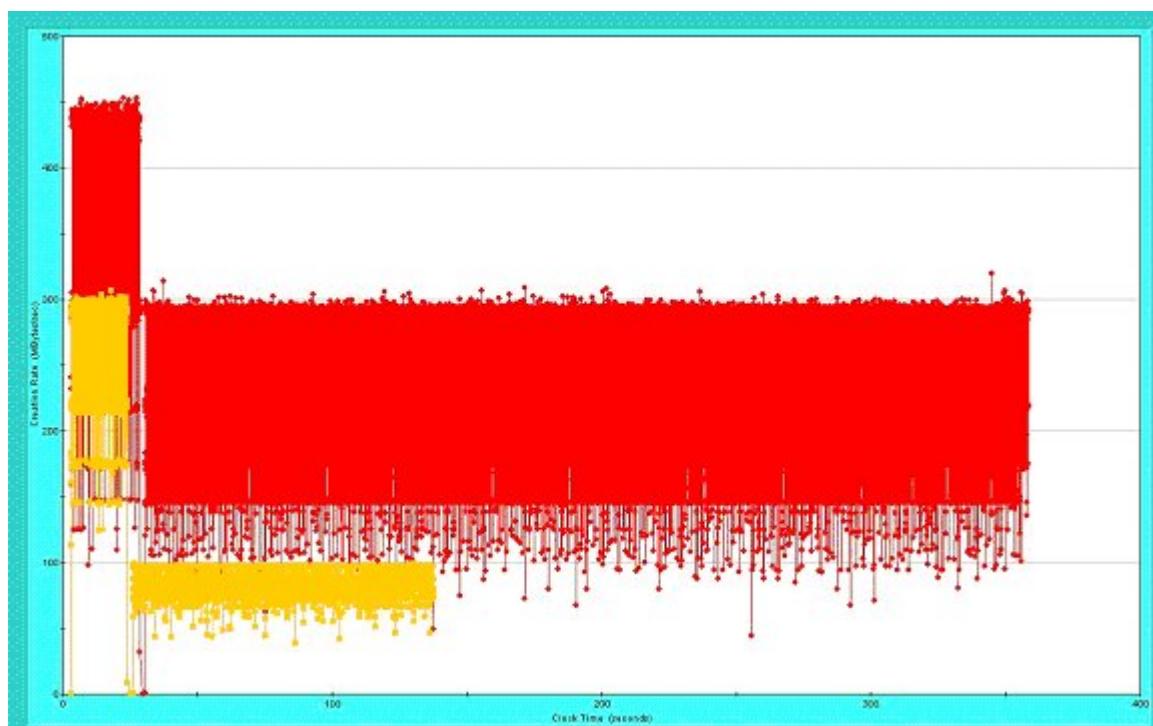
| Setup | Test |
|--------|---------|
| 20.640 | 112.000 |

| | |
|--------|---------|
| 20.750 | 111.797 |
| 20.704 | 111.719 |

As we can see there is a dramatic improvement in performance of the benchmark. As for my previous observation regarding CPU utilization, I guesstimated that the new improved version was averaging of about 95%.

From this evidence we can start to draw conclusions. The first conclusion was that abandoning String in favour of the (more suitable) CompositeKey class provided us with a stunning improvement in performance. The second conclusion was that following DRY made it extremely easy for us to alter the performance of the application. If this were all that could be said than it would be more than enough, however, since the focus of this benchmark was on the effects of memory management, garbage collection activity was monitored.

To monitor GC, the `-verbose:gc` JVM flag was set and the subsequent output was processed by HP JMeter. The resulting graphical display is shown in figure 1.



What we are looking at is a comparison of object creation rates of both the original benchmark in red and the "improved" version in yellow. From this view we can see that String version created 79,354.22 Mbytes of data where as the improved version created 14,661.595 Mbytes. I am not sure if there are any words that I can use that could add to the visual impact.

There remains yet one unexplained point in this benchmark, the differences in CPU utilization between the two. I ran the benchmark on a multi-core 3.2 GHz Pentium IV processor running Windows XP Pro. As is standard with all my machines, I have swap memory turned off. This effectively pins the JVM into memory, as there is no room for it to swap. I mention this because the inability to fully utilize a CPU is almost always attributed to

disk or network I/O or some form of contention such as locking. And with that note I will provide one other clue before I leave the mystery to you to solve. Since the second benchmark also was not able to reach 100% sustained utilization there is likely more than one source of contention.

As is the case with most benchmarking exercises I find that I am once again pleasantly surprised by the results. Also both Heinz and I often talk of the benefits of DRY with a lot of hand waving. We do this in a world where it is widely believed that you must violate best practices in order to achieve good performance. Thus it is satisfying to demonstrate how helpful DRY is with a concrete example.

Kind regards,

Kirk Pepperdine

Issue 132 - Thread Dump JSP in Java 5

Author: Dr. Heinz M. Kabutz

Date: 2006-09-11

Category: Tips and Tricks

Java Versions: JDK 1.5

Abstract:

Sometimes it is useful to have a look at what the threads are doing in a light weight fashion in order to discover tricky bugs and bottlenecks. Ideally this should not disturb the performance of the running system. In addition, it should be universally usable and cost nothing. Have a look at how we do it in this newsletter.

Welcome to the 132nd edition of **The Java(tm) Specialists' Newsletter**. I am sitting about 30000 feet above the Norwegian Sea on route to Oslo for what is said to be one of the finest Java conferences. **JavaZone** is put together by the javaBin JUG in Norway. Apparently all the Java companies in Norway shut down this week. Imagine 2 days, and almost 100 talks to attend! These Norwegians show no mercy, in line with their Viking ancestry, and have even scheduled talks during lunch time :-)

Thanks for all the kind wishes for the birth of our daughter, Evangeline Kineta Kabutz. She has only woken me up twice in the last six weeks. She is growing beautifully and will have a tough lean frame like her brother.

Thread Dump JSP in Java 5

Ten days ago, I received a desperate phone call from a large company in Cape Town. Their Java system tended to become unstable after some time, and especially during peak periods. Since the users were processing millions of dollars on this system, they should be able to log in at any time.

We managed to solve their problem. As you probably guessed, it was due to incorrectly handled concurrency. I cannot divulge how we find such problems or how to fix it, that is our competitive advantage. **Contact me offlist** if your company has bugs or performance issues that you cannot solve and where an extra pair of eyes can be useful.

One of the measurements we looked at was to inspect what the threads were doing. In this case, it did not reveal much, but it can be of great value in finding other issues. For example, at another customer, we stumbled upon an infinite loop by looking at what the threads were up to.

There are several ways of doing that. If you are using Unix, you can send a "kill -3" to the

process. With Windows, CTRL+Break on the console will give you that information.

This server was running on Windows (don't laugh). The application server did not allow us to start the JVM in a console window, which meant that we could not press CTRL+Break.

Another approach would have been to painstakingly step through the threads with JConsole. That was not an option to me.

One of the annoying parts of the typical thread dump is that the threads are not sorted, so it becomes a bit tricky to group them. It would also be nice to see a summary of the state in a table, to make it easier to find problems. In addition, we should be able to copy the text and diff it to see how things change between refreshes.

In good OO fashion, we separate model and view. Let's first define the model:

```
package com.cretesoft.tjsn.performance;

import java.io.Serializable;
import java.util.*;

public class ThreadDumpBean implements Serializable {
    private final Map<Thread, StackTraceElement[]> traces;

    public ThreadDumpBean() {
        traces = new TreeMap<Thread, StackTraceElement[]>(THREAD_COMP);
        traces.putAll(Thread.getAllStackTraces());
    }

    public Collection<Thread> getThreads() {
        return traces.keySet();
    }

    public Map<Thread, StackTraceElement[]> getTraces() {
        return traces;
    }

    /**
     * Compare the threads by name and id.
     */
    private static final Comparator<Thread> THREAD_COMP =
        new Comparator<Thread>() {
            public int compare(Thread o1, Thread o2) {
                int result = o1.getName().compareTo(o2.getName());
                if (result == 0) {
                    result = o1.getId() - o2.getId();
                }
                return result;
            }
        };
}
```

```

        if (result == 0) {
            Long id1 = o1.getId();
            Long id2 = o2.getId();
            return id1.compareTo(id2);
        }
        return result;
    }
};

}
}

```

We also write a bit of JSP, making use of the Expression Language \${}.

```

<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>

<jsp:useBean id="threadDump"
              class="com.cretesoft.tjsn.performance.ThreadDumpBean"
              scope="request"/>

<HTML>
<BODY>
<h2>Thread Summary:</h2>
<table cellpadding="5" cellspacing="5">
    <tr>
        <th>Thread</th>
        <th>State</th>
        <th>Priority</th>
        <th>Daemon</th>
    </tr>
    <c:forEach items="${threadDump.threads}" var="thr">
        <tr>
            <td><a href="#${thr.id}">${thr.name}</a></td>
            <td>${thr.state}</td>
            <td>${thr.priority}</td>
            <td>${thr.daemon}</td>
        </tr>
    </c:forEach>
</table>

<h2>Stack Trace of JVM:</h2>
<c:forEach items="${threadDump.traces}" var="trace">
    <h4><a name="${trace.key.id}">${trace.key}</a></h4>
    <pre>
        <c:forEach items="${trace.value}" var="traceline">

```

```
        at ${traceline}</c:forEach>
    </pre>
</c:forEach>

</BODY>
</HTML>
```

This will generate first a summary of the threads with hyperlinks to the individual stack traces. The summary shows the state of each thread.

A word of warning: you should not make this JSP page publicly accessible on your server, as it opens up the guts of the app server to the general population. Even if they cannot change the state of the threads, it might be bad enough for them to see what methods are being executed.

You might have to change the GET_STACK_TRACE_PERMISSION setting to allow yourself access into the application server.

This is a tool that I will keep handy whenever I do performance tuning or difficult bug fixing on a J2EE server running Java 5.

I look forward to hearing from you how you expanded this idea to make it even more useful :)

Kind regards

Heinz

Issue 130 - Deadlock Detection with new Locks

Author: Dr. Heinz M. Kabutz

Date: 2006-07-29

Category: Performance

Java Versions: JDK 1.5 - 1.6

Abstract:

Java level monitor deadlocks used to be hard to find. Then along came JDK 1.4 and showed them in CTRL+Break. In JDK 1.5, we saw the addition of the ThreadMXBean, which made it possible to continually monitor an application for deadlocks. However, the limitation was that the ThreadMXBean only worked for synchronized blocks, and not the new java.util.concurrent mechanisms. In this newsletter, we take a fresh look at the deadlock detector and show what needs to change to make it work under JDK 1.6. Also, we have a look at what asynchronous exceptions are, and how you can post them to another thread.

Welcome to the 130th edition of **The Java(tm) Specialists' Newsletter**, now sent to **112 countries** with the recent addition of Gibraltar. We had a fantastic meeting in Cologne earlier this month, organised by the JUG Cologne. Just a pity that Germany lost their soccer match. That was a bit of a let-down to the evening.

In October 2006 we will be moving to Europe to be closer to our customer base. This will open up possibilities where I can help companies in Europe with short 1-2 day consulting jobs, code reviews, performance reviews, short seminars, etc. Moving a whole family between continents is always challenging (2 years of planning) and I know that I will find it hardest to adjust to the new culture in Greece. Fortunately Helene is half-Greek so I have had some practice :)

Deadlock Detection with new Locks

One of the challenges of doing multi-threaded applications is that you need to keep an eye on both correctness and liveness. If you do not use locks, then it might happen that your objects suddenly get an invalid state (i.e. correctness is affected). It is not always easy to see this, since the problems are rarely reproducible.

The next step is to sprinkle locks all over the place, which can then lead to liveness problems. In a recent project, we needed to find some deadlock situations in Swing code. We ended up posting a Swing event every few seconds and then raising an alarm if it was not processed within a set amount of time. This helped us root out the problems quickly.

In my [newsletter # 93](#), I presented a class that automatically detected thread deadlocks. Bruce Eckel pointed out that this mechanism only worked for *monitor* locks (i.e. synchronized) and not for the java.util.concurrent owned locks.

In Java 1.6 (beta 2), the deadlocks are found for both the *monitor* and *owned* locks. However, the deadlock detector needs to be modified in order to pick up the *owned* locks that have deadlocked.

In Java 1.5, the ThreadMXBean had only the findMonitorDeadlockedThreads() method, which found deadlocks caused by the **synchronized** keyword. We now also have the findDeadlockedThreads() method that is enabled when the Java virtual machine supports monitoring of ownable synchronizer usage. (i.e. the new locks or related constructs)

New Deadlock Detector

This ThreadDeadlockDetector periodically checks whether there are threads that appear to be deadlocked. On some architectures, this could be quite resource-intensive, so we make the check interval configurable.

In Java 5, it only picked up monitor locks, not owned locks. Since Java 6, we can now see owned locks deadlocking, and a combination of owned and monitor locks.

The deadlock detection algorithm is not that accurate, in that it has trouble figuring out whether we have a deadlock when we use the tryLock(time) method. This method tries to lock for a set delay, and if it cannot manage, then it returns false. We can use this method to avoid getting deadlocks. However, the ThreadMXBean does still appear to be detecting these as real deadlocks.

The fallout is that whereas before, once you had reported on a thread being deadlocked, that thread would remain in that state forever. With the new Java 6 locks, this is not necessarily the case anymore.

Without further ado, here is the code:

```
import java.lang.management.*;
import java.util.*;
import java.util.concurrent.CopyOnWriteArraySet;

public class ThreadDeadlockDetector {
    private final Timer threadCheck =
        new Timer("ThreadDeadlockDetector", true);
    private final ThreadMXBean mbean =
        ManagementFactory.getThreadMXBean();
    private final Collection<Listener> listeners =
```

```

    new CopyOnWriteArraySet<Listener>();

/**
 * The number of milliseconds between checking for deadlocks.
 * It may be expensive to check for deadlocks, and it is not
 * critical to know so quickly.
 */
private static final int DEFAULT_DEADLOCK_CHECK_PERIOD = 10000;

public ThreadDeadlockDetector() {
    this(DEFAULT_DEADLOCK_CHECK_PERIOD);
}

public ThreadDeadlockDetector(int deadlockCheckPeriod) {
    threadCheck.schedule(new TimerTask() {
        public void run() {
            checkForDeadlocks();
        }
    }, 10, deadlockCheckPeriod);
}

private void checkForDeadlocks() {
    long[] ids = findDeadlockedThreads();
    if (ids != null && ids.length > 0) {
        Thread[] threads = new Thread[ids.length];
        for (int i = 0; i < threads.length; i++) {
            threads[i] = findMatchingThread(
                mbean.getThreadInfo(ids[i]));
        }
        fireDeadlockDetected(threads);
    }
}

private long[] findDeadlockedThreads() {
    // JDK 1.5 only supports the findMonitorDeadlockedThreads()
    // method, so you need to comment out the following three lines
    if (mbean.isSynchronizerUsageSupported())
        return mbean.findDeadlockedThreads();
    else
        return mbean.findMonitorDeadlockedThreads();
}

private void fireDeadlockDetected(Thread[] threads) {
    for (Listener l : listeners) {
        l.deadlockDetected(threads);
    }
}

```

```

private Thread findMatchingThread(ThreadInfo inf) {
    for (Thread thread : Thread.getAllStackTraces().keySet()) {
        if (thread.getId() == inf.getThreadId()) {
            return thread;
        }
    }
    throw new IllegalStateException("Deadlocked Thread not found");
}

public boolean addListener(Listener l) {
    return listeners.add(l);
}

public boolean removeListener(Listener l) {
    return listeners.remove(l);
}

/**
 * This is called whenever a problem with threads is detected.
 */
public interface Listener {
    void deadlockDetected(Thread[] deadlockedThreads);
}
}

```

I also wrote a DefaultDeadlockListener that prints out an error message and shows the stack traces. If you get a deadlock, you probably want to rather sent a high alert to the support team.

```

public class DefaultDeadlockListener implements
    ThreadDeadlockDetector.Listener {
    public void deadlockDetected(Thread[] threads) {
        System.err.println("Deadlocked Threads:");
        System.err.println("-----");
        for (Thread thread : threads) {
            System.err.println(thread);
            for (StackTraceElement ste : thread.getStackTrace()) {
                System.err.println("\t" + ste);
            }
        }
    }
}

```

To test this, I defined an interface that I could implement with several different deadlock conditions.

```
public interface DeadlockingCode {
    void f();
    void g();
}
```

The most basic error is to acquire two locks in a different sequence, such as in DeadlockingCodeSynchronized below. Our JDK 1.5 deadlock detector would have picked this one up as well:

```
public class DeadlockingCodeSynchronized implements DeadlockingCode {
    private final Object lock = new Object();
    public synchronized void f() {
        synchronized(lock) {
            // do something
        }
    }
    public void g() {
        synchronized(lock) {
            f();
        }
    }
}
```

The output from my default deadlock listener is:

```
Deadlocked Threads:
-----
Thread[DeadlockingCodeSynchronized g(),5,main]
    DeadlockingCodeSynchronized.f(DeadlockingCodeSynchronized.java:4)
    DeadlockingCodeSynchronized.g(DeadlockingCodeSynchronized.java:9)
```

```
DeadlockedThreadsTest$3.run(DeadlockedThreadsTest.java:42)
Thread[DeadlockingCodeSynchronized f(),5,main]
    DeadlockingCodeSynchronized.f(DeadlockingCodeSynchronized.java:4)
    DeadlockedThreadsTest$2.run(DeadlockedThreadsTest.java:34)
```

The more difficult deadlock is when it is on an owned lock, such as:

```
import java.util.concurrent.locks.*;

public class DeadlockingCodeNewLocksBasic implements DeadlockingCode {
    private final Lock lock1 = new ReentrantLock();
    private final Lock lock2 = new ReentrantLock();

    public void f() {
        lock1.lock();
        try {
            lock2.lock();
            try {
                // do something
            } finally {
                lock2.unlock();
            }
        } finally {
            lock1.unlock();
        }
    }

    public void g() {
        lock2.lock();
        try {
            f();
        } finally {
            lock2.unlock();
        }
    }
}
```

The new mechanism also picks up mixed deadlocks, where the one side has acquired a monitor and the other an owned lock, and now a deadlock has occurred.

Recovering From Deadlocks?

As before, there is little that can be done when you encounter a deadlock situation. If you even remotely suspect that you might cause a deadlock, rather use the `tryLock()` method together with a rollback if unsuccessful.

It is possible to `stop()` a thread that is deadlocked on an *owned* lock, but I found situations in which it retained the locks that it had already obtained. Also, you should never use `stop()`, since it will stop your thread at any place, even in critical code. Basically, the only thing that you can do is to notify the administrator that a deadlock has occurred, let him know the threads which are deadlocked, and exit the program.

The precursor of Java was called Oak. About 4 years ago, I wrote a newsletter entitled **Once upon an Oak**, where I showed how Java was influenced by Oak. There were several surprises. For example, there was no **private** keyword.

One of the scary features of Oak was Asynchronous Exceptions. Here is an excerpt from the **Oak Manual 0.2**:

| | Asynchronous Exceptions |
|---|--|
| Margin comment: The default will probably be changed to <i>not</i> allow asynchronous exceptions except in explicitly <i>unprotected</i> sections of code. | <p>Generally, exceptions are synchronous - they are thrown by code executed sequentially by an Oak program. However, in programs that have multiple threads of execution, one thread can throw an exception (using Thread's <code>postException()</code> instance method) to another thread. The second thread can't predict exactly when it will be thrown an exception, so the exception is <i>asynchronous</i>.</p> <p>By default, asynchronous exceptions can happen at any time. To prevent asynchronous exceptions from occurring in a critical section of code, you can mark the code with the protect keyword, as shown below:</p> <pre><code>protect { /* critical section goes here */ }</code></pre> |
| <i>Heinz: Note that instead of making everything protected by default, they made everything unprotected!</i> | To allow asynchronous exceptions to occur in an otherwise protected section of code, use the unprotect keyword, as follows: |

```
unprotect {
    /* code that can afford asynchronous exceptions */
}
```

In Java, the `Thread.postException()` method is called `Thread.stop(Throwable)`. You can send any exception to another thread, provided that the `SecurityManager` allows you to. Here is some rather confusing code:

```
public class ConfusingCode {
    public static long fibonacci(int n) {
        if (n < 2) return n;
        return fibonacci(n - 1) + fibonacci(n - 2);
    }

    public static void main(String[] args)
        throws InterruptedException {
        Thread fibThread = new Thread() {
            public void run() {
                try {
                    for (int i = 37; i < 90; i++) {
                        System.out.println("fib(" + i + ") = " + fibonacci(i));
                    }
                } catch (NullPointerException ex) {
                    ex.printStackTrace();
                }
            }
        };
        fibThread.start();

        Thread.sleep(10000);
        fibThread.stop(new NullPointerException("whoops"));
    }
}
```

When I run this code, I get the following output on my machine:

```
fib(37) = 24157817
fib(38) = 39088169
fib(39) = 63245986
fib(40) = 102334155
java.lang.NullPointerException: whoops
    at ConfusingCode.main(ConfusingCode.java:23)
```

As you can see, the line number does point to where `Thread.stop()` is being called. However, you could have some rather sinister fun if you were to start throwing `NullPointerExceptions` asynchronously at working code, especially if you subclasses it and changed the stack trace.

The problem with asynchronous exceptions in Java is that we do not have the **protect** keyword anymore. So therefore, critical sections can be stopped half-way through, causing serious errors in your application. It even appeared that the killed threads sometimes held onto locks.

JConsole

I have to mention JConsole, otherwise I will be flooded with emails about it :) You can also detect deadlocks with JConsole. Furthermore, you can attach JConsole onto remote processes. Since Java 1.6, you also do not need to start your program with any special system properties in order to see it locally.

However, what I like about my approach is that it automatically checks the deadlocks for you.

That's all for now - I hope that you can start using this mechanism in your code. For JDK 1.5, comment out the two lines that I marked in the `ThreadDeadlockDetector` class.

Kind regards from a chilly South Africa,

Heinz

Issue 125 - Book Review: Java Concurrency in Practice

Author: Dr. Heinz M. Kabutz

Date: 2006-04-17

Category: Book Review

Java Versions: JDK 6.0

Abstract:

We review Java Concurrency in Practice by Brian Goetz. Brian's book is the most readable on the topic of concurrency in Java, and deals with this difficult subject with a wonderful hands-on approach. It is interesting, useful, and relevant to the problems facing Java developers today.

Welcome to the 125th edition of **The Java(tm) Specialists' Newsletter**, sent to you from the "Dark Continent" (a term for Africa, coined about 130 years ago). A few weeks ago we suffered recurrent power failures, which made the name "Dark Continent" rather apt.

Book Review: Java Concurrency in Practice.

In my course on the new features in Java 5, we examine the "new" concurrency constructs of Java. Most of these are based on classes that have been freely available on [Doug Lea's website](#) for **at least** six years, and were well described in his excellent book **Concurrent Programming in Java**. However, I am yet to meet someone, either on a course or during my contracting / consulting, that has read Doug Lea's book.

Java Concurrency in Practice is split into four distinct sections:

1. Fundamentals
2. Structuring Concurrent Applications
3. Liveness, Performance, and Testing
4. Advanced Topics

Brian does an excellent job of explaining, and his examples are more similar to the real world than you would find with other books.

Something else I like about the book is that it mentions all sorts of new features that are available in Java 6. Whilst I cannot use that yet in production, it is good to know what is coming.

At the end of the first section on Fundamentals, Brian goes through the steps of building an

efficient, scalable result cache that you could use in a web server. The idea is to remember the result from a previous calculation in order to reduce latency and increase throughput, at the cost of a bit more memory usage.

The problem with building this cache is that if we are not careful, we could easily turn it into a scalability bottleneck. Brian starts with a basic `HashMap`, then looks at ways that we can make it more scalable.

We first define interface `Computable`, which performs some calculation:

```
public interface Computable<A, V> {
    V compute(A arg) throws InterruptedException;
}
```

The next class is `ExpensiveFunction`, which takes a long time to compute the result:

```
import java.math.BigInteger;

public class ExpensiveFunction
    implements Computable<String, BigInteger> {
    public BigInteger compute(String arg) {
        // after deep thought...
        try {
            Thread.sleep(1000);
        } catch (InterruptedException e) {
            Thread.currentThread().interrupt();
        }
        return new BigInteger(arg);
    }
}
```

We now build a `Computable` wrapper that remembers the result from the previous calculations and returns these transparently. This process is called memoization.

```

import java.util.*;

public class Memoizer1<A, V> implements Computable<A, V> {
    @GuardedBy("this")
    private final Map<A, V> cache = new HashMap<A, V>();
    private final Computable<A, V> c;
    public Memoizer1(Computable<A, V> c) {
        this.c = c;
    }
    public synchronized V compute(A arg) throws InterruptedException {
        V result = cache.get(arg);
        if (result == null) {
            result = c.compute(arg);
            cache.put(arg, result);
        }
        return result;
    }
}

```

The **@GuardedBy** annotation is one of several described in the book that help us to document our assumptions regarding thread safety:

```

import java.lang.annotation.*;

/**
 * The field or method to which this annotation is applied can only
 * be accessed when holding a particular lock, which may be a
 * built-in (synchronization) lock, or may be an explicit
 * java.util.concurrent.Lock.
 */
@Target({ElementType.FIELD, ElementType.METHOD})
@Retention(RetentionPolicy.RUNTIME)
public @interface GuardedBy {
    String value();
}

```

Since `HashMap` is not threadsafe, we need to take the conservative approach and lock on all access. If you have several threads queued up to execute `compute`, then it might actually take longer than without caching.

A slight improvement occurs when you change the HashMap to be a ConcurrentHashMap instance:

```
import java.util.Map;
import java.util.concurrent.ConcurrentHashMap;

public class Memoizer2<A, V> implements Computable<A, V> {
    private final Map<A, V> cache = new ConcurrentHashMap<A, V>();
    private final Computable<A, V> c;
    public Memoizer2(Computable<A, V> c) { this.c = c; }
    public V compute(A arg) throws InterruptedException {
        V result = cache.get(arg);
        if (result == null) {
            result = c.compute(arg);
            cache.put(arg, result);
        }
        return result;
    }
}
```

The problem with Memoizer2 is that even if one thread starts an expensive computation, other threads may still start the same computation. Instead, we should have some way of representing the notion that "thread X is currently computing f (27)", so that if another thread arrives looking for f (27), it knows that it should wait until Thread X has finished its work.

Yet another option is to use the FutureTask class

```
import java.util.Map;
import java.util.concurrent.*;

public class Memoizer3<A, V> implements Computable<A, V> {
    private final Map<A, Future<V>> cache
        = new ConcurrentHashMap<A, Future<V>>();
    private final Computable<A, V> c;
    public Memoizer3(Computable<A, V> c) { this.c = c; }
    public V compute(final A arg) throws InterruptedException {
        Future<V> f = cache.get(arg);
        if (f == null) {
            Callable<V> eval = new Callable<V>() {
                public V call() throws InterruptedException {

```

```
        return c.compute(arg);
    }
};

FutureTask<V> ft = new FutureTask<V>(eval);
f = ft;
cache.put(arg, ft);
ft.run(); // call to c.compute happens here
}

try {
    return f.get();
} catch (ExecutionException e) {
    // Kabutz: this is my addition to the code...
    try {
        throw e.getCause();
    } catch (RuntimeException ex) {
        throw ex;
    } catch (Error ex) {
        throw ex;
    } catch (Throwable t) {
        throw new IllegalStateException("Not unchecked", t);
    }
}
}
```

The Memoizer3 is almost perfect, but there still is a window of vulnerability when two threads might compute the same value. The window is smaller than in Memoizer2, but it still is there. Memoizer3 is vulnerable to this problem because a compound action (putif-absent) is performed on the backing map that cannot be made atomic using locking.

The next approach in the [book](#) is to use the atomic `putIfAbsent()` method from `ConcurrentMap`.

```
import java.util.concurrent.*;

public class Memoizer<A, V> implements Computable<A, V> {
    private final ConcurrentHashMap<A, Future<V>> cache
        = new ConcurrentHashMap<A, Future<V>>();
    private final Computable<A, V> c;
    public Memoizer(Computable<A, V> c) { this.c = c; }
    public V compute(final A arg) throws InterruptedException {
        while (true) {
```

```
Future<V> f = cache.get(arg);
if (f == null) {
    Callable<V> eval = new Callable<V>() {
        public V call() throws InterruptedException {
            return c.compute(arg);
        }
    };
    FutureTask<V> ft = new FutureTask<V>(eval);
    f = cache.putIfAbsent(arg, ft);
    if (f == null) {
        f = ft;
        ft.run();
    }
}
try {
    return f.get();
} catch (CancellationException e) {
    cache.remove(arg, f);
} catch (ExecutionException e) {
    // Kabutz: this is my addition to the code...
    try {
        throw e.getCause();
    } catch (RuntimeException ex) {
        throw ex;
    } catch (Error ex) {
        throw ex;
    } catch (Throwable t) {
        throw new IllegalStateException("Not unchecked", t);
    }
}
}
```

When I read the Java samples, the problem and the solution both appeared quite straightforward to me. Brian has picked plausible real-world example that clearly demonstrate the techniques that he presents.

This book should be available in your bookshops in the next few weeks, so keep your eye open for this one! Alternatively, you can also [pre-order it on Amazon.com](#).

Kind regards

Heinz

Issue 124 - Copying Arrays Fast

Author: Dr. Heinz M. Kabutz

Date: 2006-03-28

Category: Performance

Java Versions: JDK 1.5.0_06

Abstract:

In this newsletter we look at the difference in performance between cloning and copying an array of bytes. Beware of the Microbenchmark! We also show how misleading such a test can be, but explain why the cloning is so much slower for small arrays.

Welcome to the 124th edition of **The Java(tm) Specialists' Newsletter**, which I started writing for you in Las Vegas, where I had the fortune of attending TheServerSide Java Symposium. One of the highlights of these conferences is the networking during coffee breaks. Shaking hands with the likes of Ted Neward, Kirk Pepperdine, Bruce Snyder, Bruce Tate, Stephan Janssen, Rod Johnson, Adrian Coyler, Gregor Hohpe and Eugene Ciurana. At the TSSJS conferences you can get easy access to the speakers. It was also great linking up with my old friends from Estonia and with subscribers from around the world, such as David Jones (who wrote **J2EE Singleton** for us many years ago) and Robert Lentz, a Java Architect from Germany.

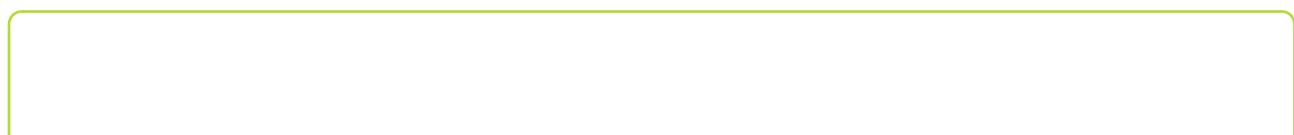
On Friday night Kirk Pepperdine and I led a Birds-of-a-Feather (BoF) on whether performance testing was still a relevant activity. Some people think that instead of tuning the performance, one could simply buy more hardware. This is not always the case. For example, if you are hitting the limits of memory, the garbage collector may degrade the performance of the entire system to such an extent that the only option is to fix the problem.

The difficulty with looking at performance, as we will see in this newsletter, is eliminating the noise from the problem. *Beware of the microbenchmark!*

Copying Arrays Fast

A month ago, my friend Paul van Spronsen sent me this puzzle (Paul wrote an excellent newsletter on **multicasting in Java**):

Look at the attached class. Guess which test will be faster and then run X.main. Startling result.



```
import java.util.Random;

public class X {
    private final byte[] byteValue = new byte[16];

    X() {
        new Random(0).nextBytes(byteValue);
    }

    public byte[] testClone() {
        return byteValue.clone();
    }

    public byte[] testNewAndCopy() {
        byte[] b = new byte[byteValue.length];
        System.arraycopy(byteValue, 0, b, 0, byteValue.length);
        return b;
    }

    public static void main(String[] args) {
        doTest();
        doTest();
    }

    private static void doTest() {
        X x = new X();
        int m = 50000000;

        long t0 = System.currentTimeMillis();
        for (int i = 0; i < m; i++) {
            x.testClone();
        }
        long t1 = System.currentTimeMillis();

        System.out.println("clone(): " + (t1 - t0));

        t0 = System.currentTimeMillis();
        for (int i = 0; i < m; i++) {
            x.testNewAndCopy();
        }
        t1 = System.currentTimeMillis();

        System.out.println("arraycopy(): " + (t1 - t0));
    }
}
```

I guessed, based on previous experience, that the `testNewAndCopy()` would be faster. The `System.arraycopy()` method uses JNI to copy the memory and I knew that was fast. (I also knew my friend Paul would only send me a puzzle if the result was surprising) Here we see that `clone` takes about 5 times longer than copying the array:

```
clone(): 26598
arraycopy(): 5618
clone(): 26639
arraycopy(): 5648
```

We could run off now and change all our code to use the more verbose approach instead of `clone()`, and expect to see an improvement in performance.

Beware of the Microbenchmark! I cannot emphasize this enough. When we encounter surprises, we need to find out why they are there. Changing code before knowing why it is slower can result in ugly, slow code. When I showed this to Kirk Peppardine at TheServerSide Java Symposium, he suggested that I would need to look at the source code of `clone()` to see why there was such a large difference.

But before we do that, let's have a look at some more robust testing classes. First off, a class that runs a method repeatedly within some time period. Here, higher numbers are better. This time I added some JavaDocs to explain how it works. Please let me know if you like seeing JavaDocs in the code, or if I can strip them out?

```
import java.util.*;

/**
 * The PerformanceChecker tries to run the task as often as possible
 * in the allotted time. It then returns the number of times that
 * the task was called. To make sure that the timer stops the test
 * timeously, we check that the difference between start and end
 * time is less than EPSILON. After it has tried unsuccessfully for
 * MAXIMUM_ATTEMPTS times, it throws an exception.
 *
 * @author Heinz Kabutz
 * @since 2006/03/27
 */
public class PerformanceChecker {
```

```

* Whether the test should continue running. Will expire after
* some time specified in testTime. Needs to be volatile for
* visibility.
*/
private volatile boolean expired = false;
/** The number of milliseconds that each test should run */
private final long testTime;
/** The task to execute for the duration of the test run. */
private final Runnable task;
/**
 * Accuracy of test. It must finish within 20ms of the testTime
 * otherwise we retry the test. This could be configurable.
*/
public static final int EPSILON = 20;
/**
 * Number of repeats before giving up with this test.
*/
private static final int MAXIMUM_ATTEMPTS = 3;

/**
 * Set up the number of milliseconds that the test should run, and
 * the task that should be executed during that time. The task
 * should ideally run for less than 10ms at most, otherwise you
 * will get too many retry attempts.
*
* @param testTime the number of milliseconds that the test should
*                 execute.
* @param task      the task that should be executed repeatedly
*                 until the time is used up.
*/
public PerformanceChecker(long testTime, Runnable task) {
    this.testTime = testTime;
    this.task = task;
}

/**
 * Start the test, and after the set time interrupt the test and
 * return the number of times that we were able to execute the
 * run() method of the task.
*/
public long start() {
    long numberofLoops;
    long start;
    int runs = 0;
    do {
        if (++runs > MAXIMUM_ATTEMPTS) {
            throw new IllegalStateException("Test not accurate");
        }
    }
}

```

```

expired = false;
start = System.currentTimeMillis();
numberOfLoops = 0;
Timer timer = new Timer();
timer.schedule(new TimerTask() {
    public void run() {
        expired = true;
    }
}, testTime);
while (!expired) {
    task.run();
    numberOfLoops++;
}
start = System.currentTimeMillis() - start;
timer.cancel();
} while (Math.abs(start - testTime) > EPSILON);
collectGarbage();
return numberOfLoops;
}

/**
 * After every test run, we collect garbage by calling System.gc()
 * and sleeping for a short while to make sure that the garbage
 * collector has had a chance to collect objects.
 */
private void collectGarbage() {
    for (int i = 0; i < 3; i++) {
        System.gc();
        try {
            Thread.sleep(10);
        } catch (InterruptedException e) {
            Thread.currentThread().interrupt();
            break;
        }
    }
}
}

```

I do not like running performance tests without calculating the standard deviation of the average, otherwise we cannot detect how noisy the environment is. Here is a class to calculate mean and standard deviation:

```
import java.util.*;
```

```

/**
 * Calculates the mean and average of a series of numbers. Not the
 * most efficient algorithm, but fast enough.
 *
 * @author Heinz Kabutz
 */
public class Average {
    /** The set of values stored as doubles. Autoboxed. */
    private Collection<Double> values = new ArrayList<Double>();

    /**
     * Add a new value to the series. Changes the values returned by
     * mean() and stddev().
     * @param value the new value to add to the series.
     */
    public void add(double value) {
        values.add(value);
    }

    /**
     * Calculate and return the mean of the series of numbers.
     * Throws an exception if this is called before the add() method.
     * @return the mean of all the numbers added to the series.
     * @throws IllegalStateException if no values have been added yet.
     * Otherwise we could cause a NullPointerException.
     */
    public double mean() {
        int elements = values.size();
        if (elements == 0) throw new IllegalStateException("No values");
        double sum = 0;
        for (double value : values) {
            sum += value;
        }
        return sum / elements;
    }

    /**
     * Calculate and return the standard deviation of the series of
     * numbers. See Stats 101 for more information...
     * Throws an exception if this is called before the add() method.
     * @return the standard deviation of numbers added to the series.
     * @throws IllegalStateException if no values have been added yet.
     * Otherwise we could cause a NullPointerException.
     */
    public double stddev() {
        double mean = mean();
        double stddevtotal = 0;

```

```

for (double value : values) {
    double dev = value - mean;
    stddevtotal += dev * dev;
}
return Math.sqrt(stddevtotal / values.size());
}
}

```

I know we will cause noise in the system, but what I definitely want to prevent is objects ending up in the Old Space. The point at which an object is put into Old Space is when it is larger than 512kb. Since `byte[]` takes up 8 bytes for the object pointer and 4 bytes for the array length, the largest byte array that will fit into Young Space is $512 * 1024 - 12$. Try it out! We experiment with this in our [Java Performance Tuning Course](#), an essential course if you are coding in Java.

Here we calculate the performance of a `PerformanceChecker` instance, based on the given number of runs. The result that comes back is an instance of `Average`. The standard deviation should be as small as possible. If it is large, then we know that there was background noise and that the values might or might not be invalid.

```

/**
 * This class calculates the performance of a PerformanceChecker
 * instance, based on the given number of runs.
 *
 * @author Heinz Kabutz
 */
public class PerformanceHarness {
    /**
     * We calculate the average number of times that the check
     * executed, together with the standard deviation.
     * @param check The test that we want to evaluate
     * @param runs How many times it should be executed
     * @return an average number of times that test could run
    */
    public Average calculatePerf(PerformanceChecker check, int runs) {
        Average avg = new Average();
        // first we warm up the hotspot compiler
        check.start(); check.start();
        for(int i=0; i < runs; i++) {
            long count = check.start();
            avg.add(count);
        }
    }
}

```

```
    return avg;
}
}
```

We now need to run this with the clone and array copy tests. First, I define some code for each of these test cases.

```
import java.util.Random;

public class ArrayCloneTest implements Runnable {
    private final byte[] byteValue;

    public ArrayCloneTest(int length) {
        byteValue = new byte[length];
        // always the same set of bytes...
        new Random(0).nextBytes(byteValue);
    }

    public void run() {
        byte[] result = byteValue.clone();
    }
}

import java.util.Random;

public class ArrayNewAndCopyTest implements Runnable {
    private final byte[] byteValue;

    public ArrayNewAndCopyTest(int length) {
        byteValue = new byte[length];
        // always the same set of bytes...
        new Random(0).nextBytes(byteValue);
    }

    public void run() {
        byte[] b = new byte[byteValue.length];
        System.arraycopy(byteValue, 0, b, 0, byteValue.length);
    }
}
```

We can now run the complete benchmark, by writing a CompleteTest class that tests everything thoroughly. In order to make this interesting, I test the difference between clone() and copy() for various sizes of byte[]. As mentioned before, we have to be careful not to exceed the maximum size of byte[] that can exist in the Young Space, otherwise the performance will degrade to such an extent that the whole test will fail.

```

public class CompleteTest {
    private static final int RUNS = 10;
    private static final long TEST_TIME = 100;

    public static void main(String[] args) throws Exception {
        test(1);
        test(10);
        test(100);
        test(1000);
        test(10000);
        test(100000);
    }

    private static void test(int length) {
        PerformanceHarness harness = new PerformanceHarness();
        Average arrayClone = harness.calculatePerf(
            new PerformanceChecker(TEST_TIME,
                new ArrayCloneTest(length)), RUNS);
        Average arrayNewAndCopy = harness.calculatePerf(
            new PerformanceChecker(TEST_TIME,
                new ArrayNewAndCopyTest(length)), RUNS);

        System.out.println("Length=" + length);
        System.out.printf("Clone %.0f\t%.0f%n",
            arrayClone.mean(), arrayClone.stddev());
        System.out.printf("Copy %.0f\t%.0f%n",
            arrayNewAndCopy.mean(), arrayNewAndCopy.stddev());
        System.out.printf("Diff %.2fx%n",
            arrayNewAndCopy.mean() / arrayClone.mean());
        System.out.println();
    }
}

```

When we now run this more comprehensive test, we see an interesting phenomenon. As the byte[] increases in size, the difference between the two techniques disappears. Why is that, you might wonder? Let's first look at the result, before we try to find out why...

```
Length=1
Clone 253606      19767
Copy   1282556    139950
Diff   5.06x
```

```
Length=10
Clone 240096      10105
Copy   1159128    59049
Diff   4.83x
```

```
Length=100
Clone 167628      13144
Copy   464809     43279
Diff   2.77x
```

```
Length=1000
Clone 53575       3535
Copy   68080       7455
Diff   1.27x
```

```
Length=10000
Clone 8842        162
Copy   7547        713
Diff   0.85x
```

```
Length=100000
Clone 807          19
Copy   763          90
Diff   0.95x
```

Oops, it seems that once the array becomes longer, the performance of the two is almost equal! Infact, the cloning seems marginally faster. Aren't you glad that you have not changed all your code yet?

I followed Kirk's advice and looked at the source code of the `clone()` method, which you will find in the `JVM_Clone` method in `jvm.c` (You will need to download the complete JVM source code from Sun's website). After getting my head around the old C code, I realised that it does two things in addition to plain copying of the memory:

1. Check whether instance is normal object or array.
2. Check whether array is of primitives or objects.

It has to be extra careful when copying an object array to not publish pointers without telling the garbage collector, or you would get nasty memory leaks.

These tests are in themselves not significant, but when the rest of our work is little (such as when the byte[] is small), then this causes our experiment to skew against cloning. Let's try out what would happen if we were to add those two checks to our test:

```
import java.util.Random;

public class ArrayNewAndCopyTest implements Runnable {
    private final byte[] byteValue;

    public ArrayNewAndCopyTest(int length) {
        byteValue = new byte[length];
        // always the same set of bytes...
        new Random(0).nextBytes(byteValue);
    }

    public void run() {
        Class cls = byteValue.getClass();
        if (cls.isArray()) {
            if (!cls.getComponentType().isAssignableFrom(Object.class)) {
                byte[] b = new byte[byteValue.length];
                System.arraycopy(byteValue, 0, b, 0, byteValue.length);
                return;
            }
        }
        throw new RuntimeException();
    }
}
```

The results are now almost identical:

| | |
|----------|--------|
| Length=1 | |
| Clone | 237416 |
| Copy | 235780 |
| Diff | 0.99x |

| | |
|-----------|--------|
| Length=10 | |
| Clone | 226804 |
| Copy | 9614 |

```

Copy 231363      12176
Diff 1.02x

Length=100
Clone 153981      6809
Copy 169900      9851
Diff 1.10x

Length=1000
Clone 50406 2835
Copy 52498 2579
Diff 1.04x

Length=10000
Clone 7769 281
Copy 6807 271
Diff 0.88x

Length=100000
Clone 724 24
Copy 680 49
Diff 0.94x

```

This leads me to conclude that the only reason why Paul's test seemed so much faster was because he picked a byte[] size that was small enough that the actual copying was dwarfed by the two **if** statements. Using clone() for copying arrays is less code and the performance difference is, as we saw, only significant for tiny arrays. I think that in future I will rather use clone() than System.arraycopy().

It would have been great to eliminate more noise from the experiment, but since we are testing the speed of copying of arrays, we need to create new objects all the time, which then need to be garbage collected.

An interesting method that I saw in Brian Goetz's new book on Java Concurrency in Practice (more details soon) is `java.util.Arrays.copyOf(byte[] original, int newLength)` which was added in JDK 1.6. The `copyOf` method uses `System.arraycopy()` to make a copy of the array, but is more flexible than `clone` since you can make copies of parts of an array.

Our new website is now running on a dedicated server, which has stayed up beautifully. I want to make it as easy as possible for you to browse through my past newsletters. Please let me know if you think of ways to make this part of the website more navigable.

Kind regards

Heinz

P.S. Sometimes it helps to cache byte[]'s especially if they are all of the same size and you need to create lots of them.

Issue 115 - Young vs. Old Generation GC

Author: Dr. Heinz M. Kabutz

Date: 2005-10-13

Category: Performance

Java Versions: Sun JDK 1.4, 1.5

Abstract:

A few weeks ago, I tried to demonstrate the effects of old vs. new generation GC. The results surprised me and reemphasized how important GC is to your overall program performance.

Welcome to the 115th edition of **The Java(tm) Specialists' Newsletter**. I am on the island of Crete again, this time with the whole family. We are enjoying the beauty and hospitality. Swimming was possible almost every day, even though it is already October! One of my dreams is to run Java courses here, far away from the distractions of the office. Just imagine: Java exercises combined with snorkeling and swimming in the sea. We are planning some amazing specials for our subscribers so that you can also experience this wonderful island.

My short trips to France and USA were very enjoyable. In France we did a design workshop followed by some Design Patterns. In the USA, I spoke at the **Java In Action** conference. Much to my delight, the room was packed to full capacity. Thanks all for attending! And thanks to the organisers for putting on a great conference! This was the very first time I got to go to the USA, and it certainly was quite an experience!

We've travelled enough (for now) and are looking forward to going home to South Africa soon.

Young vs. Old Generation GC

It is common knowledge that it is more efficient to GC young than old generations. You should therefore avoid middle-aged objects. This is also a reason why object pooling is discouraged. You might end up with more objects in old space, which will slow down your GC. How **much** it slows down is interesting to watch.

Let's have a look at some simple classes derived from a CreateTest class. ObjectCreationTest1 contains an infinite loop that simply creates objects. A question here: what percentage of CPU time do you estimate is spent collecting garbage? 40%, 50%, 60%, 70%? ObjectCreationTest2 keeps an array of the last million created objects. Does the JVM need more CPU to collect garbage?

```

public abstract class CreateTest {
    private long count;
    public long getCount() {
        return count;
    }
    protected final void incCount() {
        count++;
    }
    public abstract void run();
}

public class ObjectCreationTest1 extends CreateTest {
    public void run() {
        while (true) {
            new Object();
            incCount();
        }
    }
}

public class ObjectCreationTest2 extends CreateTest {
    public void run() {
        Object[] pool = new Object[1 * 1024 * 1024];
        int count = 0;
        while (true) {
            pool[(count++) % pool.length] = new Object();
            incCount();
        }
    }
}

```

In order to try this out, I wrote a Launcher class that after 120 seconds kills the JVM using the System.exit() command:

```

import java.util.*;

public class Launcher {
    public static void main(String[] args) throws Exception {
        String testName = args[0];
        final CreateTest job =
            (CreateTest) Class.forName(testName).newInstance();
        Timer timer = new Timer();
        timer.schedule(new TimerTask() {

```

```

public void run() {
    System.out.println(job.getCount());
    System.exit(0);
}
}, 120000);
job.run();
}
}

```

This is how we could run it:

```

java -Xloggc:nopool.log Launcher ObjectCreationTest1
java -Xloggc:pool.log Launcher ObjectCreationTest2

```

In my run, ObjectCreationTest1 created 8,762,510,385 objects and ObjectCreationTest2 only managed to create 389,105,915 objects.

A nice free tool that I learned about at [Java In Action](#) is JTune by HP which we can use to see what happened in the GC logs. I do not have a direct link to the tool, but you should be able to find one on the [Java Performance Tuning.com](#) website.

Using HP JTune, we should see that with ObjectCreationTest1, we get statistics such as:

- **Duration of the measurement:** 123s
- **Total bytes allocated:** 66,980 MB
- **Number of GC events:** 133,961
- **Time spent in GC:** 11s
- **Percentage of time in GC:** 9%

It must be said that the 134 thousand GCs all occurred in the young generation. Since the object references were immediately cleared, the objects were never moved into the old generation.

Again using HP JTune, we should see that with ObjectCreationTest2, we get statistics such as:

- **Duration of the measurement:** 123s
- **Total bytes allocated:** 2,975 MB
- **Number of GC events:** 1,455
- **Time spent in GC:** 96s
- **Percentage of time in GC:** 78%

Here the created objects live long enough to be pushed into the old generation or the young space could not accommodate new objects. The system only did 1165 young gen collections, and 290 old gen collections.

Object pools tend to let objects survive longer, thereby pushing them into the old generation space. This may be bad for performance due to the additional overhead of old gen GC, as seen here.

During Kirk Pepperdine's talk on performance, the question arose as to what percentage of GC activity was acceptable in a real system. In Kirk's experience, you should aim for 5%, but maximum 10%. The first example effectively does `while(true) { new Object(); }` but it only spends 9% of its CPU doing garbage collection. We should be careful to not be too generous in allocating CPU cycles to the GC.

Another fun display of GC activity is with [Sun's jvmstat visualgc](#). Great tool that gives insight (using sampling) of what is happening inside running JVMs. [Click here to see the graph for ObjectCreationTest1](#) and [here for ObjectCreationTest2](#).

Sanity Check

Here is another test that sits in the infinite loop and is counting how quickly the looping works, which is basically what would happen when *no objects* are created. The loop executed 24,362,038,313 times in two minutes.

```
public class ObjectCreationTest3 extends CreateTest {
    public void run() {
        while (true) {
            incCount();
        }
    }
}
```

To summarise, creating and destroying objects is quite fast when the objects live in the young space, but rather slow when they live in old space. Even faster is *creating no objects*

(obviously).

Kind regards

Heinz

P.S. **A picture of my kids enjoying the beach** near to where we want to run the Java courses on the island of Crete. **Watch this space!**

Issue 114 - Compile-time String Constant Quiz

Author: Dr. Heinz M. Kabutz

Date: 2005-09-16

Category: Language

Java Versions: All

Abstract:

When we change libraries, we need to do a full recompile of our code, in case any constants were inlined by the compiler. Find out which constants are inlined in this latest newsletter.

Welcome to the 114th edition of **The Java(tm) Specialists' Newsletter**. I learned last Tuesday that I had been nominated as a **Java Champion**. The nomination was approved this Monday. So now I am one of the world's first elected Java Champions. I do not fully understand yet what that means, but I am overwhelmed that I had been noticed, considering how small our subscription base is :)

Compile-time String Constant Quiz

This week's newsletter is based on a quiz sent to me by Clark Updike from the John Hopkins University Applied Physics Laboratory. Thanks Clark :)

Consider the following interface (remember that all fields in an interface are automatically **public static final**):

```
public interface StaticFinalTest {
    String LITERAL = "Literal";
    String LITERAL_PLUS = "Literal" + "Plus";
    String LITERAL_NEW = new String("LiteralNew");
    String LITERAL_CONCAT = "LiteralConcat".concat("");
}
```

And we can use this as follows:

```
public class StaticFinalTestClient {  
    public static void main(String[] args) {  
        System.out.println(StaticFinalTest.LITERAL);  
        System.out.println(StaticFinalTest.LITERAL_PLUS);  
        System.out.println(StaticFinalTest.LITERAL_NEW);  
        System.out.println(StaticFinalTest.LITERAL_CONCAT);  
    }  
}
```

When we run the program, we see:

```
Literal  
LiteralPlus  
LiteralNew  
LiteralConcat
```

Now Change StaticFinalTest, compile it, but do not compile StaticFinalTestClient. If you in an IDE, you will have to compile it from the command line.

```
public interface StaticFinalTest {  
    String LITERAL = "LiteralXXX";  
    String LITERAL_PLUS = "Literal" + "PlusXXX";  
    String LITERAL_NEW = new String("LiteralNewXXX");  
    String LITERAL_CONCAT = "LiteralConcat".concat("XXX");  
}
```

Here is the quiz: What is the output? (scroll down for the answer)


```
Literal
LiteralPlus
LiteralNewXXX
LiteralConcatXXX
```

At compile time, all **final** fields that have a constant value, and are either primitive or String, get inlined by the compiler. This includes of course the Strings in our StaticFinalTest class. Since the NEW and CONCAT values are not compile time literals they are not inlined. So, when you change libraries, you have to recompile all your code. This limits how you change between libraries. You cannot simply swap out libraries at runtime, because if you as soon as you use constants, they are inlined and require a full recompile of your code.

Whilst I would think this is really widely understood, I have met a number of Java programmers who did not realise this.

Unit Testing

Be careful when writing unit tests. If you look at the following (incorrect) code, we can write unit tests that make it appear correct:

```

public class Car {
    private final String registrationNumber;
    public Car(String registrationNumber) {
        this.registrationNumber = registrationNumber;
    }
    public boolean equals(Object o) {
        if (!(o instanceof Car)) return false;
        return registrationNumber == ((Car) o).registrationNumber;
    }
    public int hashCode() {
        return registrationNumber.hashCode();
    }
}

```

The code is obviously incorrect, because it compares Strings using the `==` operator, instead of `equals()`. But look at this unit test, which one would you write?

```

import junit.framework.TestCase;

public class CarTest extends TestCase {
    public void testIncorrect() {
        assertEquals(
            new Car("CET192233"),
            new Car("CET192233"));
    }
    public void testCorrect() {
        assertEquals(
            new Car(new String("CET192233")),
            new Car("CET192233"));
    }
}

```

The first test is incorrect, since it compares the Car objects with identical strings, so the `equals()` method appears correct.

In JDK 1.1 and 1.0, `final` methods were inlined as well if you compiled with the `-O` option. Since Java 2, `final` methods are only inlined at runtime by the hotspot compiler (if necessary).

Kind regards

Heinz

Issue 111 - What is faster - LinkedList of ArrayList?

Author: Dr. Heinz M. Kabutz

Date: 2005-07-12

Category: Performance

Java Versions: Sun JDK 1.2 - 5.0

Abstract:

Welcome to the 111th edition of **The Java(tm) Specialists' Newsletter**. After a quick visit home to Cape Town, I am yet again in Johannesburg, this time to present some courses on **Design Patterns**.

The old archive page was getting a bit difficult to navigate, so I have upgraded the structure of my website somewhat, and put all the newsletters into categories. You can view the new structure by going to the [archive page](#). I hope this will make it easier for you :)

I was deeply disturbed this morning by a [newspaper article](#) reporting that since 2003, there have been an additional estimated 700'000 cases of HIV/AIDS infection in South Africa alone, bringing the number of people living with HIV/AIDS, to over 6.2 million. Women between 25 - 29 years of age are hardest hit, with a 40% infection rate. I know this has nothing to do with Java, but it does concern us as human beings.

What is faster - LinkedList of ArrayList?

As programmers, we often try to eke the last ounce of performance out of our collections. An interesting statement is this: "ArrayList is faster than LinkedList, except when you remove an element from the middle of the list." I have heard this on more than one occasion, and a few months ago, decided to try out how true that statement really was. Here is some code that I wrote during last week's Java 5 Tiger course:

```
import java.util.*;

public class ListTest {
    private static final int NUM_ELEMENTS = 100 * 1000;
    public static void main(String[] args) {
        List ar = new ArrayList();
        for (int i = 0; i < NUM_ELEMENTS; i++) {
            ar.add(i);
        }
    }
}
```

```

testListBeginning(ar);
testListBeginning(new LinkedList(ar));
testListMiddle(ar);
testListMiddle(new LinkedList(ar));
testListEnd(ar);
testListEnd(new LinkedList(ar));
}
public static void testListBeginning(List list) {
    long time = System.currentTimeMillis();
    for (int i = 0; i < 10000; i++) {
        list.add(0, new Object());
        list.remove(0);
    }
    time = System.currentTimeMillis() - time;
    System.out.println("beginning " +
        list.getClass().getSimpleName() + " took " + time);
}
public static void testListMiddle(List list) {
    long time = System.currentTimeMillis();
    for (int i = 0; i < 10000; i++) {
        list.add(NUM_ELEMENTS / 2, new Object());
        list.remove(NUM_ELEMENTS / 2);
    }
    time = System.currentTimeMillis() - time;
    System.out.println("middle " +
        list.getClass().getSimpleName() + " took " + time);
}
public static void testListEnd(List list) {
    long time = System.currentTimeMillis();
    for (int i = 0; i < 10000000; i++) {
        list.add(new Object());
        list.remove(NUM_ELEMENTS);
    }
    time = System.currentTimeMillis() - time;
    System.out.println("end " +
        list.getClass().getSimpleName() + " took " + time);
}
}

```

One small little addition in Java 5 is the method `getSimpleName()` defined inside Class. I do not know how many times I have needed such a method and have had to write it.

The output is obvious, but surprising nevertheless in the extent of the differences:

```
beginning ArrayList took 4346
beginning LinkedList took 0
middle    ArrayList took 2104
middle    LinkedList took 26728
end      ArrayList took 731
end      LinkedList took 1242
```

Finding the element in the middle of the `LinkedList` takes so much longer that the benefits of just changing the pointer are lost. So, `LinkedList` is **worse** than `ArrayList` for removing elements in the middle, except perhaps if you are already there (although I have not tested that).

So, when should you use `LinkedList`? For a long list that works as a FIFO queue, the `LinkedList` should be faster than the `ArrayList`. However, even faster is the `ArrayBlockingQueue` or the [CircularArrayList that I wrote a few years ago](#). The answer is probably "never".

Kind regards

Heinz

Issue 105 - Performance Surprises in Tiger

Author: Dr. Heinz M. Kabutz

Date: 2005-03-28

Category: Performance

Java Versions: Sun JDKs 1.3.1_12, 1.4.0_04, 1.4.1_07, 1.4.2_05, 1.5.0_02

Abstract:

Welcome to the 105th edition of **The Java(tm) Specialists' Newsletter**, now also sent to Venezuela, our 109th country. It has been extremely busy this year, with company audits, and lots of travelling to far away countries. I am flying to Austria and Germany in April to present our **Design Patterns Course**, Java 5 (Tiger) Course, and last, but definitely not least, a section of the **Java Performance Tuning Course** authored by Kirk Pepperdine and Jack Shirazi. They are all presented in German, which is possible since that is my mother tongue, even though I grew up in South Africa. On my last visit, an official at the duty office remarked that he would never have guessed that I had been born outside of Germany :-)

Another trip is planned for May. One week training in Germany followed by a visit to Crete in Greece, where I plan to give a lecture on software development at the University of Crete in Iraklion, if all goes well. I went to Crete in December 2004, and it was easily the most hospitable place I have gone to.

Lots of travelling, and everywhere is far away when you live in the stunningly beautiful city of Cape Town.

Oh, on another note, I spoke at an event organised by ITWeb in South Africa. If you want to see what I look and sound like under pressure, [check this out](#). I tried to be reasonably coherent on the video clip, but that short video clip was completely unrehearsed, and if I had had time to prepare, I would have said something else.

Performance Surprises in Tiger

One thing that is certain about performance measurements in Java is that there is no certainty. Computers are by their nature deterministic, but you are still at the mercy of the compiler writers, the hotspot compilers, etc. With every version of Java, some parts are faster others slower. This becomes especially annoying when you have spent effort finetuning an application based on knowledge of what has always been true in Java, but suddenly changed without warning. Oh well, at least it will keep me writing newsletters ;-)

StringBuffer

For example, in the past, `StringBuffer.toString()` shared its `char[]` with the `String` object. If you changed the `StringBuffer` after calling `toString()`, it would make a copy of the `char[]` inside the `StringBuffer`, and work with that. Please have a look at my [newsletter #68](#) where I discussed this phenomenon. Incidentally, Sun changed the `setLength()` method of JDK 1.4.1 back to what it was in JDK 1.4.0. I was discussing this behaviour with some Java programmers, and wanted to demonstrate that instead of calling `setLength(0)`, you may as well just create a new `StringBuffer`, since the costly part of the creation is the `char[]`.

Let's examine some code snippets:

```
// SB1: Append, convert to String and release StringBuffer
StringBuffer buf = new StringBuffer();
buf.append(0);
buf.append(1);
buf.append(2);
// etc.
buf.toString();
```

```
// SB2: Create with correct length, throw away afterwards
StringBuffer buf = new StringBuffer(3231);
buf.append(0);
buf.append(1);
buf.append(2);
// etc.
buf.toString();
```

```
// SB3: buf defined somewhere else
buf.setLength(0);
buf.append(0);
buf.append(1);
buf.append(2);
// etc.
buf.toString(); // don't release buf
```

If we look at the table below, we see that for JDK 1.4.x, SB3 was approximately the same speed as SB1, and it was always slower than SB2. In JDK 1.5.0_02, suddenly SB2 and SB3 are approximately the same, with both being much faster than SB1.

| Java Version | Hotspot Type | SB1 | SB2 | SB3 |
|--------------|--------------|------|------|------|
| 1.4.0_04 | Client | 1653 | 1452 | 1632 |
| 1.4.0_04 | Server | 1082 | 951 | 1072 |
| 1.4.1_07 | Client | 1752 | 1582 | 1723 |
| 1.4.1_07 | Server | 1101 | 962 | 1061 |
| 1.4.2_05 | Client | 1072 | 871 | 1071 |
| 1.4.2_05 | Server | 630 | 551 | 681 |
| 1.5.0_02 | Client | 1032 | 501 | 490 |
| 1.5.0_02 | Server | 811 | 400 | 381 |

Since JDK 1.5, when we do not need StringBuffer to be synchronized, we can replace it by the StringBuilder. Running the tests using the new class yields better results:

| Java Version | Hotspot Type | SB1 | SB2 | SB3 |
|--------------|--------------|-----|-----|-----|
| 1.5.0_02 | Client | 921 | 441 | 431 |
| 1.5.0_02 | Server | 721 | 350 | 341 |

For completeness, here is the code used to test the performance:

```
public class StringBufferTest {
    private static final int UPTO = 10 * 1000;
    private final int repeats;

    public StringBufferTest(int repeats) {
        this.repeats = repeats;
    }

    private long testNewBufferDefault() {
        long time = System.currentTimeMillis();
        for (int i = 0; i < repeats; i++) {
            StringBuffer buf = new StringBuffer();
            for (int j = 0; j < UPTO; j++) {
                buf.append(j);
            }
            buf.toString();
        }
        time = System.currentTimeMillis() - time;
        return time;
    }
}
```

```

    return time;
}

private long testNewBufferCorrectSize() {
    long time = System.currentTimeMillis();
    for (int i = 0; i < repeats; i++) {
        StringBuffer buf = new StringBuffer(38890);
        for (int j = 0; j < UPTO; j++) {
            buf.append(j);
        }
        buf.toString();
    }
    time = System.currentTimeMillis() - time;
    return time;
}

private long testExistingBuffer() {
    StringBuffer buf = new StringBuffer();
    long time = System.currentTimeMillis();
    for (int i = 0; i < repeats; i++) {
        buf.setLength(0);
        for (int j = 0; j < UPTO; j++) {
            buf.append(j);
        }
        buf.toString();
    }
    time = System.currentTimeMillis() - time;
    return time;
}

public String testAll() {
    return testNewBufferDefault() + "," +
        testNewBufferCorrectSize() + "," + testExistingBuffer();
}

public static void main(String[] args) {
    System.out.print(System.getProperty("java.version") + ",");
    System.out.print(System.getProperty("java.vm.name") + ",");
    // warm up the hotspot compiler
    new StringBufferTest(10).testAll();
    System.out.println(new StringBufferTest(400).testAll());
}
}

```

Initialising Singletons

I discovered this one by pure chance, and I suspect that it is a bug in the server hotspot compiler. When I initialise Singletons, I usually do so in the static initialiser block. For example:

```
public class Singleton1 {
    private final static Singleton1 instance = new Singleton1();
    public static Singleton1 getInstance() {
        return instance;
    }
    private Singleton1() {}
}
```

Sometimes, I will initialise it in a synchronized block inside the getInstance() method. This is necessary when we combine the Singleton with polymorphism, and when we therefore want to choose which subclass to use. The code would then be:

```
public class Singleton2 {
    private static Singleton2 instance;
    // lazy initialization
    public static Singleton2 getInstance() {
        synchronized (Singleton2.class) {
            if (instance == null) {
                instance = new Singleton2();
            }
        }
        return instance;
    }
    private Singleton2() {}
}
```

However, thanks to byte code reordering by the hotspot compiler, I would avoid using the double-checked locking approach, otherwise I might return a half-initialised object to the caller of getInstance(). So I would **not** write it like this (since Java 5, the correct way would be to make the instance field volatile):

```

public class Singleton3 {
    private static Singleton3 instance;
    // double-checked locking - broken in Java - don't use it!
    public static Singleton3 getInstance() {
        if (instance == null) {
            synchronized (Singleton3.class) {
                if (instance == null) {
                    instance = new Singleton3();
                }
            }
        }
        return instance;
    }
    private Singleton3() {}
}

```

The table below contains the relative speed between calling getInstance() on each of the Singletons. Note the outliers marked in red. JDK 1.4.0_04 client hotspot for some reason performs really badly for the double-checked locking example. On the other hand, this particular test is terrible for the server hotspot in JDK 1.5.0_02. The Singleton1 is 1000 times slower in JDK 1.5.0 than in JDK 1.4.0, without changing a single line of code, and without even compiling anew. It is surprises like these that can make your project suddenly perform awefully.

| Java Version | Hotspot Type | Singleton1 | Singleton2 | Singleton3 |
|--------------|--------------|------------|------------|------------|
| 1.3.1_12 | Client | 220 | 1923 | 851 |
| 1.3.1_12 | Server | 220 | 1883 | 971 |
| 1.4.0_04 | Client | 360 | 1923 | 6339 |
| 1.4.0_04 | Server | 10 | 1633 | 530 |
| 1.4.1_07 | Client | 340 | 2013 | 1562 |
| 1.4.1_07 | Server | 20 | 1593 | 530 |
| 1.4.2_05 | Client | 330 | 1983 | 1632 |
| 1.4.2_05 | Server | 20 | 1783 | 631 |
| 1.5.0_02 | Client | 290 | 2144 | 1261 |
| 1.5.0_02 | Server | 10385 | 10455 | 8962 |

My explanation for these results is that either in the Server VM the access to the instance is not inlined in JDK 1.5, or it is simply a bug in the server hotspot compiler. If we compare the speed of using interpreted mode vs. mixed mode, we can see that the interpreted speed is similar to the server hotspot VM:

| Java Version | Hotspot Type | Singleton1 | Singleton2 | Singleton3 |
|--------------|--------------------|------------|------------|------------|
| 1.5.0_02 | Client Interpreted | 10415 | 14962 | 9423 |
| 1.5.0_02 | Server Interpreted | 7742 | 21611 | 9543 |

For completeness, here is the code for SingletonTest:

```
public class SingletonTest {
    private static final int UPTO = 100 * 1000 * 1000;
    public static void main(String[] args) {
        System.out.print(System.getProperty("java.version") + ",");
        System.out.print(System.getProperty("java.vm.name") + ",");

        long time;

        time = System.currentTimeMillis();
        for (int i = 0; i < UPTO; i++) {
            Singleton1.getInstance();
        }
        time = System.currentTimeMillis() - time;
        System.out.print(time + ",");

        time = System.currentTimeMillis();
        for (int i = 0; i < UPTO; i++) {
            Singleton2.getInstance();
        }
        time = System.currentTimeMillis() - time;
        System.out.print(time + ",");

        time = System.currentTimeMillis();
        for (int i = 0; i < UPTO; i++) {
            Singleton3.getInstance();
        }
        time = System.currentTimeMillis() - time;
        System.out.println(time);
    }
}
```

Update: In modern JVMs, we have something called biased locking, which means that tests of synchronized really need to be written with multiple threads calling the code. At the time of writing, this was ok, but in 2014 I would recommend using the [Java Microbenchmarking Harness](#). See also [Issue 217](#) and [Issue 217b](#).

Keep coding, and don't leave performance tests for too late in the game.

Kind regards

Heinz

Issue 101 - Causing Deadlocks in Swing Code

Author: Dr. Heinz M. Kabutz

Date: 2005-01-18

Category: GUI

Java Versions: Sun JDK 1.5.0_01

Abstract:

Welcome to the 101st edition of **The Java(tm) Specialists' Newsletter**, now sent to 107 countries. That is right, we had 5 new countries since the last edition: Four from Africa: Eritrea, Malawi, Zambia and Senegal, and one from Europe: The Former Yugoslav Republic of Macedonia. A special welcome to you :)

The worst time of year to be working in Cape Town is between Christmas and New Year. Our city grows by a few hundred thousand people, as the "Vaalies" (South Africans who live across the Vaal river, such as in Johannesburg) descend on Cape Town, together with lots of German and English tourists. All work stops, and if you are the odd-one-out who *is* trying to get something done, that week is the most depressing of the whole year. Since 1997, I got suckered EVERY year to work during that week, but this last December, I refused. The result? When I arrived in Germany last week, I was told that I looked 2 years younger :)

Causing Deadlocks in Swing Code

My first newsletter was about thread deadlocks that can occur in many places of Java, including the GUI. I then presented some code that automatically detected thread deadlocks in **Issue 93**.

It is easy to demonstrate deadlocks, but up till now, I did not have an example where incorrect coding in the Swing classes caused a deadlock. I had seen it "in the field", several times, but I did not have a one-page sample that demonstrated it. Many thanks to Dan Breen from USA who sent me this code sample :)

In this code, several factors come together. I am constructing the frame from within the static initializer block. Then, when I add the new MyJDesktopPane to the content pane, and set it to visible, it starts the Swing thread. However, I then construct a new JInternalFrame (with the main thread!) and at the same time, the Swing thread tries to call `paintComponent()` of MyJDesktopPane, which then hangs up when calling the `staticMethod()`.

```

import javax.swing.*;
import java.awt.*;

public class StrangeProblem extends JFrame {
    static {
        new StrangeProblem();
    }

    private static void staticMethod() {
        System.out.println("This is never reached");
    }

    private StrangeProblem() {
        getContentPane().add(new MyJDesktopPane());
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        setSize(300, 300);
        setVisible(true);
        // If commented out, program works fine, otherwise it hangs
        new JInternalFrame();
    }

    private class MyJDesktopPane extends JDesktopPane {
        protected void paintComponent(Graphics g) {
            super.paintComponent(g);
            System.out.println("We will now call the static method...");
            staticMethod();
            System.out.println("Static method was called.");
        }
    }

    public static void main(String[] args) {
    }
}

```

You run this code by compiling it and typing `java StrangeProblem`. This will load the class (before calling `main`) which will call the static initializer block. You will notice that the line "This is never reached" is never seen, and neither is the line "Static method was called." When you dump the threads, you see:

```

We will now call the static method...
Full thread dump Java HotSpot(TM) Client VM (1.5.0_01-b08 mixed mode, sharing):

"AWT-EventQueue-0" prio=7 tid=0x009f87c0 nid=0x524 in Object.wait() [0x02f4f000].

```

```

        at StrangeProblem$MyJDesktopPane.paintComponent(StrangeProblem.java:26)
*snip*
        - locked <0x22b37728> (a java.awt.Component$AWTTreeLock)

"main" prio=5 tid=0x00236040 nid=0xc2c waiting for monitor entry [0x0006f000..0x
        at java.awt.Component.setFont(Unknown Source)
        - waiting to lock <0x22b37728> (a java.awt.Component$AWTTreeLock)
*snip*
        at javax.swing.JInternalFrame.<init>(Unknown Source)
        at StrangeProblem.<init>(StrangeProblem.java:19)
        at StrangeProblem.<clinit>(StrangeProblem.java:6)

```

What I found amazing is that this code **always** causes a deadlock (or is that a livelock?) on Sun's VM's going right back to version 1.2.2_014.

Fixing the code is easy, and almost anything that we change will make the problem go away. For example, we could only call `setVisible(true)` after we finish adding components. Then, we could construct the object from within `main()` rather than the static initializer block.

However, the best change is to obey the Swing single-thread rule, which tells us to only ever call GUI code from within the Swing thread. In our case, we need to change the static initializer block to:

```

static {
    SwingUtilities.invokeLater(new Runnable() {
        public void run() {
            new StrangeProblemFixed();
        }
    });
}

```

The lesson learnt is, when writing any GUI code, make sure that only the Swing event thread is changing or reading the GUI. This applies even to code that constructs a frame from within the `main()` function.

I have a newsletter written by Aleksey Gureev, which I will probably send in my next issue, which offers another excellent approach to automatically detecting thread deadlocks in the GUI.

Kind regards

Heinz

Issue 098 - References

Author: Dr. Heinz M. Kabutz

Date: 2004-11-08

Category: Performance

Java Versions: Sun JDK 1.5.0-b64, Sun JDK 1.3.1_12

Abstract:

Welcome to the 98th edition of **The Java(tm) Specialists' Newsletter**. Did you know that the world is split into six floral kingdoms? The largest in size is the Holarctic Kingdom (42%), consisting of North America and Eurasia. The smallest is Capensis (0.04%) consisting of Cape Town and surrounding areas. Down here at the most south-west part of Africa, we have the greatest density of diverse plant life on the planet, which we generically call "fynbos". This is wonderful for plant lovers, but terrible for hayfever sufferers like me. If it was up to me, I would chop down all the fynbos and cover it all with concrete! [ok, I know I made enemies with that statement, but if we could swap bodies for one week, you would agree with me ;-]

I usually run my **Java courses** at the Faculty Training Institute in Cape Town. From our balcony, we overlook the Kenilworth Race Track. When the horse racing track was built, the inside of the track was kept in its original state. This is the last remaining habitat of some micro frogs living in the original fynbos. What happened was that the horse race track guarded the original plants, protecting them from the alien plant invasion from Australia.

References

How many articles have you read about soft, weak and phantom references? How many of those did you understand? I tried, and found most of the articles extremely hard to make sense of. This is an attempt to explain some of the behaviour of soft and weak references, and then to invent a new type of references that is, dare I say, more useful than a weak reference?

Soft vs Weak References

In **newsletter 15**, I described a SoftReference based HashMap, where the values were embedded in soft references. (The WeakHashMap has the keys embedded in weak references, so that would not be useful for building a cache.) At the time of writing, I was using JDK 1.3.x, and I did not notice any difference in the behaviour of soft vs. weak references. It turns out that there appeared to be some design flaw in JDK 1.3.x that rendered SoftReferences useless in the Sun JVM. Consider the following test code:

```
public class LargeObject {
    private final byte[] space = new byte[1024 * 1024];
    private final int id;
    public LargeObject(int id) {
        this.id = id;
    }
    public int getId() {
        return id;
    }
}
```

```
import java.lang.ref.*;
import java.util.*;

public class ReferenceTest {
    private static final int NUMBER_OF_REFERENCES = 30;
    public static void main(String[] args) {
        // the map contains reference objects as keys, and the id
        // as a value.
        final Map refs = new HashMap();
        final ReferenceQueue queue = new ReferenceQueue();
        // We need a thread that reads processed references from the
        // reference queue so that we can see in what order they are
        // reclaimed.
        new Thread() {
            {
                setDaemon(true);
                start();
            }

            public void run() {
                try {
                    while (true) {
                        Reference ref = queue.remove();
                        Integer id = (Integer) refs.remove(ref);
                        if (ref instanceof SoftReference) {
                            System.out.println("SOFT " + id);
                        } else if (ref instanceof WeakReference) {
                            System.out.println("WEAK " + id);
                        } else {
                            throw new IllegalArgumentException();
                        }
                    }
                }
            }
        }.start();
        for (int i = 0; i < NUMBER_OF_REFERENCES; i++) {
            refs.put(new SoftReference(i), i);
        }
        for (int i = 0; i < NUMBER_OF_REFERENCES; i++) {
            refs.put(new WeakReference(i), i);
        }
    }
}
```

```
        }

    } catch (InterruptedException e) {
        return;
    }
}

};

for (int i = 0; i < NUMBER_OF_REFERENCES; i++) {
    System.out.println("NEW " + i);
    Integer num = new Integer(i);
    // must keep a strong reference to the actual reference,
    // otherwise it will not be enqueued.
    refs.put(new SoftReference(new LargeObject(i), queue), num);
    refs.put(new WeakReference(new LargeObject(i), queue), num);
}

byte[][] buf = new byte[1024][];
System.out.println("Allocating until OOME...\"");
for (int i = 0; i < buf.length; i++) {
    buf[i] = new byte[1024 * 1024];
}
}
```

We need to run this with the various JVMs. Very few companies still use JDK 1.2.x, so I will not consider that case. In addition, I could not find a difference between JDK 1.4.x and JDK 1.5.x, so will only compare JDK 1.3.x with JDK 1.5.x.

Let us start by looking at Sun JDK 1.3.1_12. When we run the code, we can see that soft and weak references are released at about the same time. This is not dependent on how many references we have. It does not matter whether we have 10 or 1000 references. The JavaDocs (hard to understand) of the reference classes seem to hint that weak references should be released before soft references. However, in JDK 1.3.x, this was not the case:

```
java version "1.3.1_12"
Java(TM) 2 Runtime Environment, Standard Edition (build 1.3.1_12-b03)
Java HotSpot(TM) Client VM (build 1.3.1_12-b03, mixed mode)

NEW    0
SOFT   0           soft ref released first!
NEW    1
WEAK   0
NEW    2
SOFT   1
```

```

WEAK 1
NEW 3
NEW 4
SOFT 3
WEAK 2
SOFT 2
WEAK 3
NEW 5
SOFT 4
WEAK 4
NEW 6
SOFT 5
WEAK 5
NEW 7
SOFT 6
WEAK 6
SOFT 7
NEW 8
WEAK 7
NEW 9
SOFT 8
WEAK 8
Allocating until OOME...
SOFT 9      hardly any soft references still left on heap
WEAK 9
Exception in thread "main" java.lang.OutOfMemoryError
    <<no stack trace available>>

```

The JDK 1.5.0 behaves more closely to what we would expect. When we run the code, we can see that weak references are typically released first, and soft references are mainly released when the memory becomes low:

```

java version "1.5.0"
Java(TM) 2 Runtime Environment, Standard Edition (build 1.5.0-b64)
Java HotSpot(TM) Client VM (build 1.5.0-b64, mixed mode, sharing)

NEW 0
NEW 1
SOFT 0
WEAK 0
NEW 2
WEAK 1
NEW 3

```

```

WEAK 2
NEW 4
SOFT 2
SOFT 3
SOFT 1
WEAK 3
NEW 5
WEAK 4
NEW 6
WEAK 5
NEW 7
WEAK 6
NEW 8
WEAK 7
NEW 9
WEAK 8
Allocating until OOME...
WEAK 9
SOFT 4          lots of soft references still left on heap
SOFT 8
SOFT 5
SOFT 9
SOFT 6
SOFT 7
Exception in thread "main" java.lang.OutOfMemoryError: Java heap space

```

I put this observation to Mark Reinhold, who was kind enough to send me a quick response:

> [I know you are busy, so a simple: "Yes, JDK 1.3.x had a useless
> implementation." will suffice :-]

In fact that's exactly the case. HotSpot didn't have a useful
implementation of soft references until 1.4.

There we go. From the author of the `java.lang.ref` package himself. No wonder Sydney and I needed such a strange approach in [newsletter 15](#).

New SoftHashMap

I then wrote a new `SoftReference` based `HashMap`, but this time based on generics, and

using my discoveries of the behaviour of the new SoftReferences. (Thanks to Jason Walton for pointing out two possible memory leaks in the original code.)

```

import java.lang.ref.*;
import java.util.*;
import java.io.Serializable;

public class SoftHashMap <K, V> extends AbstractMap<K, V>
    implements Serializable {
    /** The internal HashMap that will hold the SoftReference. */
    private final Map<K, SoftReference<V>> hash =
        new HashMap<K, SoftReference<V>>();

    private final Map<SoftReference<V>, K> reverseLookup =
        new HashMap<SoftReference<V>, K>();

    /** Reference queue for cleared SoftReference objects. */
    private final ReferenceQueue<V> queue = new ReferenceQueue<V>();

    public V get(Object key) {
        expungeStaleEntries();
        V result = null;
        // We get the SoftReference represented by that key
        SoftReference<V> soft_ref = hash.get(key);
        if (soft_ref != null) {
            // From the SoftReference we get the value, which can be
            // null if it has been garbage collected
            result = soft_ref.get();
            if (result == null) {
                // If the value has been garbage collected, remove the
                // entry from the HashMap.
                hash.remove(key);
                reverseLookup.remove(soft_ref);
            }
        }
        return result;
    }

    private void expungeStaleEntries() {
        Reference<? extends V> sv;
        while ((sv = queue.poll()) != null) {
            hash.remove(reverseLookup.remove(sv));
        }
    }

    public V put(K key, V value) {

```

```

expungeStaleEntries();
SoftReference<V> soft_ref = new SoftReference<V>(value, queue);
reverseLookup.put(soft_ref, key);
SoftReference<V> result = hash.put(key, soft_ref);
if (result == null) return null;
reverseLookup.remove(result);
return result.get();
}

public V remove(Object key) {
    expungeStaleEntries();
    SoftReference<V> result = hash.remove(key);
    if (result == null) return null;
    return result.get();
}

public void clear() {
    hash.clear();
    reverseLookup.clear();
}

public int size() {
    expungeStaleEntries();
    return hash.size();
}

/**
* Returns a copy of the key/values in the map at the point of
* calling. However, setValue still sets the value in the
* actual SoftHashMap.
*/
public Set<Entry<K,V>> entrySet() {
    expungeStaleEntries();
    Set<Entry<K,V>> result = new LinkedHashSet<Entry<K, V>>();
    for (final Entry<K, SoftReference<V>> entry : hash.entrySet()) {
        final V value = entry.getValue().get();
        if (value != null) {
            result.add(new Entry<K, V>() {
                public K getKey() {
                    return entry.getKey();
                }
                public V getValue() {
                    return value;
                }
                public V setValue(V v) {
                    entry.setValue(new SoftReference<V>(v, queue));
                    return value;
                }
            });
        }
    }
}

```

```
        } );  
    }  
}  
return result;  
}  
}
```

We can now use the SoftHashMap just like an ordinary HashMap, except that the entries will disappear if we are running low on memory. Ideal if you want to build a cache. Here is some test code:

```
import java.util.*;  
  
public class SoftHashMapTest {  
    private static void print(Map<String, Integer> map) {  
        System.out.println("One=" + map.get("One"));  
        System.out.println("Two=" + map.get("Two"));  
        System.out.println("Three=" + map.get("Three"));  
        System.out.println("Four=" + map.get("Four"));  
        System.out.println("Five=" + map.get("Five"));  
    }  
  
    private static void testMap(Map<String, Integer> map) throws InterruptedException {  
        System.out.println("Testing " + map.getClass());  
        map.put("One", new Integer(1));  
        map.put("Two", new Integer(2));  
        map.put("Three", new Integer(3));  
        map.put("Four", new Integer(4));  
        map.put("Five", new Integer(5));  
        print(map);  
        Thread.sleep(2000);  
        print(map);  
        try {  
            byte[] block = new byte[200 * 1024 * 1024];  
        } catch (OutOfMemoryError ex) {  
            ex.printStackTrace();  
        }  
        print(map);  
    }  
  
    public static void main(String[] args) throws InterruptedException {  
        testMap(new HashMap<String, Integer>());  
        testMap(new SoftHashMap<String, Integer>());  
    }  
}
```

```
    }  
}
```

When you run this code, you get:

```
Testing class java.util.HashMap  
One=1  
Two=2  
Three=3  
Four=4  
Five=5  
One=1  
Two=2  
Three=3  
Four=4  
Five=5  
java.lang.OutOfMemoryError: Java heap space  
One=1  
Two=2  
Three=3  
Four=4  
Five=5  
Testing class SoftHashMap  
One=1  
Two=2  
Three=3  
Four=4  
Five=5  
One=1  
Two=2  
Three=3  
Four=4  
Five=5  
java.lang.OutOfMemoryError: Java heap space  
One=null  
Two=null  
Three=null  
Four=null  
Five=null
```

GhostReference

Did you know that you can write your own reference? A few months ago, I was having dinner at Mariner's Wharf in the village of Hout Bay near Cape Town, with three of Sun's Java evangelists. One of the points of discussion was that none of us had ever found a good use for the PhantomReference. The only suggestion that I could come up with is with object counting, and to use it to determine when objects are garbage collected ([newsletter 38](#)).

I spent several hours today thinking about PhantomReference. I did a search on google and could not find any good use case for the PhantomReference. On very careful inspection, I discovered some differences between the phantom and weak references. Both are released rather quickly, but the phantom reference is enqueued in the reference queue **before** its referent is cleared, whereas the weak reference is enqueued **after** the referent is cleared. One hitch - the PhantomReference.get() method always returns **null**. Hah - time for reflection! Another difference is that the PhantomReference is enqueued only **after** the finalize() method has been called.

Another hitch with references is this: If you want the reference to be added to the reference queue, you have to keep a **strong** reference to the reference. To solve this problem, I keep a Collection of currently active references inside my GhostReference. As is the case with all PhantomReference objects, you have to call clear() on the reference once you have completed working with it.

```
import java.lang.ref.*;
import java.lang.reflect.Field;
import java.util.*;

public class GhostReference extends PhantomReference {
    private static final Collection currentRefs = new HashSet();
    private static final Field referent;

    static {
        try {
            referent = Reference.class.getDeclaredField("referent");
            referent.setAccessible(true);
        } catch (NoSuchFieldException e) {
            throw new RuntimeException("Field \"referent\" not found");
        }
    }

    public GhostReference(Object referent, ReferenceQueue queue) {
        super(referent, queue);
        currentRefs.add(this);
    }
}
```

```

public void clear() {
    currentRefs.remove(this);
    super.clear();
}

public Object getReferent() {
    try {
        return referent.get(this);
    } catch (IllegalAccessException e) {
        throw new IllegalStateException("referent should be accessible!");
    }
}
}

```

We now have a more sensible Reference, with a cool name. It is not necessary to have a strong reference to the Reference object, in order for it to be enqueued, and it is possible to get hold of the referent (the object that we want a reference to) from the enqueued Reference:

```

import java.lang.ref.*;

public class GhostReferenceTest {
    private static final int NUMBER_OF_REFERENCES = 10;
    public static void main(String[] args) {
        final ReferenceQueue queue = new ReferenceQueue();
        new Thread() { { setDaemon(true); start(); } }
        public void run() {
            try {
                while (true) {
                    GhostReference ref = (GhostReference) queue.remove();
                    LargeObject obj = (LargeObject) ref.getReferent();
                    System.out.println("GHOST " + obj.getId());
                    ref.clear();
                }
            } catch (InterruptedException e) {
                return;
            }
        }
    };
    for (int i = 0; i < NUMBER_OF_REFERENCES; i++) {
        System.out.println("NEW " + i);
        // We do not need to keep strong reference to the actual
        // reference anymore, and we also do not need a reverse
    }
}

```

```
// lookup anymore
new GhostReference(new LargeObject(i), queue);
}
byte[][] buf = new byte[1024][];
System.out.println("Allocating until OOME... ");
for (int i = 0; i < buf.length; i++) {
    buf[i] = new byte[1024 * 1024];
}
}
```

Warning: the PhantomReference is only called after the finalize() has completed. This means that we are resurrecting an object that will then never be finalized again. However, this is only a theoretical limitation, as in practice you should never use the finalize() method anyway.

I have still to this day (2009-11-12) not used the PhantomReference in real programming work. What is interesting is that it is also not used once in the whole of the JDK, whereas both weak and soft references are. The WeakReference is subclassed in order to implement Finalizers.

Kind regards

Heinz

P.S. If you can come up with a miracle cure that will solve my hayfever, I will reconsider my stance on the fynbos eradication ;-)

P.P.S. The fynbos is safe - I moved to Crete in Greece in 2006, which has a lot less pollen in the air.

Issue 093 - Automatically Detecting Thread Deadlocks

Author: Dr. Heinz M. Kabutz

Date: 2004-07-28

Category: Concurrency

Java Versions: Sun JDK 1.5.0-beta2

Abstract:

Deadlocks between synchronized monitors are impossible to resolve in Java. In this newsletter we look at a new MXBean that can detect deadlocks between threads and use it to warn us.

Welcome to the 93rd edition of **The Java(tm) Specialists' Newsletter**. I posted a link to our [last newsletter](#) to [The Serverside](#). Didn't get too many comments (maybe not controversial enough ?!) but at least it made the front page for a few days :-)

Here is a really funny animated cartoon about [what Sun Microsystems could do with their \\$2,000,000,000](#) [Sorry, it is not available anymore]. I was teaching last week about Swing, and coincidentally a friend sent me [a hilarious animated cartoon about GridBagLayout](#). The nicest approach to GUI development that I have seen is in IntelliJ IDEA. They separate the layout from the rest of the GUI, and inject bytecode into your classes, instead of generating loads of source code. Once you get the hang of IDEA's GUI editor tool, you can smack together a GridBagLayout GUI in seconds, rather than minutes or days.

Automatically Detecting Thread Deadlocks

After our last newsletter, Kris Schneider mentioned that he had seen the Permanent Generation run out of space. He has kindly posted his improved version of my MemoryWarningSystem on [The Serverside](#).

Scott Sobel brought up the issue of OOME occurring when too many threads were created. The problem with too many threads is that they need stack space. Both Scott and I have experienced where increasing the maximum heap space actually *decreased* the number of threads that we could create before getting an OOME. Unfortunately this maximum number of threads seems to be kind-of magical, and depends on the operating system and on the initial stack size per thread.

This brought me to this new warning system, that notifies me if we have too many threads. In order to not get too many notifications, I take the approach that you get *one* warning when we pass the thread count threshold. If you slip below the threshold, and go above it again, you will get another warning notification. This is the same approach taken by the memory bean. Better would probably be to have a high- and low-water mark.

In addition, it can also tell if there are deadlocked threads. [My very first Java newsletter](#), sent in 2000 to some friends and colleagues gleaned from my contact list, demonstrated how you could find thread deadlocks by hand. This system finds them automatically for you. Seems we have progressed in the last 4 years! Looking for deadlocked threads is potentially slow, so this code could affect your performance. However, I would rather have a slower *correct* program than a lightning fast *incorrect* program.

There is absolutely nothing you can do with a deadlocked thread. You cannot stop it, you cannot interrupt it, you cannot tell it to stop trying to get a lock, and you also cannot tell it to let go of the locks that it owns. This is one of the criticism in [Doug Lea's book](#) about the primitive monitor-based locking mechanisms. Once you try to get a lock, you will forever try and never give up. The concurrency handling mechanisms of Doug's book are now in the java.util.concurrent package of JDK 1.5.

This brought me to the question, what is the definition of a deadlock? In Webopedia.com, they describe it nicely:

A condition that occurs when two processes are each waiting for the other to complete before proceeding. The result is that both processes hang. Deadlocks occur most commonly in multitasking and client/server environments. Ideally, the programs that are deadlocked, or the operating system, should resolve the deadlock, but this doesn't always happen. A deadlock is also called a deadly embrace. (Source [Webopedia.com](#))

Enough theory, here is the code to the ThreadWarningSystem, that detects when there are too many threads, and finds thread deadlocks:

```
import java.lang.management.*;
import java.util.*;

public class ThreadWarningSystem {
    private final Timer threadCheck = new Timer("Thread Monitor", true);
    private final ThreadMXBean mbean = ManagementFactory.getThreadMXBean();
    private final Collection<Listener> listeners = new ArrayList<Listener>();
    /**
     * The number of milliseconds between checking for deadlocks.
     * It may be expensive to check for deadlocks, and it is not
     * critical to know so quickly.
     */
    private static final int DEADLOCK_CHECK_PERIOD = 500;
    /**
     * The number of milliseconds between checking number of
     * threads. Since threads can be created very quickly, we need
     * to check this frequently.
     */
}
```

```

private static final int THREAD_NUMBER_CHECK_PERIOD = 20;
private static final int MAX_STACK_DEPTH = 30;
private boolean threadThresholdNotified = false;
private Set deadlockedThreads = new HashSet();

<**
 * Monitor only deadlocks.
 */

public ThreadWarningSystem() {
    threadCheck.schedule(new TimerTask() {
        public void run() {
            long[] ids = mbean.findMonitorDeadlockedThreads();
            if (ids != null && ids.length > 0) {
                for (Long l : ids) {
                    if (!deadlockedThreads.contains(l)) {
                        deadlockedThreads.add(l);
                        ThreadInfo ti = mbean.getThreadInfo(l, MAX_STACK_DEPTH);
                        fireDeadlockDetected(ti);
                    }
                }
            }
        }
    }, 10, DEADLOCK_CHECK_PERIOD);
}

<**
 * Monitor deadlocks and the number of threads.
 */

public ThreadWarningSystem(final int threadThreshold) {
    this();
    threadCheck.schedule(new TimerTask() {
        public void run() {
            if (mbean.getThreadCount() > threadThreshold) {
                if (!threadThresholdNotified) {
                    fireThresholdExceeded();
                    threadThresholdNotified = true;
                }
            } else {
                threadThresholdNotified = false;
            }
        }
    }, 10, THREAD_NUMBER_CHECK_PERIOD);
}

private void fireDeadlockDetected(ThreadInfo thread) {
    // In general I avoid using synchronized. The surrounding
    // code should usually be responsible for being threadsafe.
    // However, in this case, the timer could be notifying at
}

```

```

// the same time as someone is adding a listener, and there
// is nothing the calling code can do to prevent that from
// occurring. Another tip though is this: when I synchronize
// I use a private field to synchronize on, instead of
// "this".
synchronized (listeners) {
    for (Listener l : listeners) {
        l.deadlockDetected(thread);
    }
}
}

private void fireThresholdExceeded() {
    ThreadInfo[] allThreads = mbean.getThreadInfo(mbean.getAllThreadIds());
    synchronized (listeners) {
        for (Listener l : listeners) {
            l.thresholdExceeded(allThreads);
        }
    }
}

public boolean addListener(Listener l) {
    synchronized (listeners) {
        return listeners.add(l);
    }
}

public boolean removeListener(Listener l) {
    synchronized (listeners) {
        return listeners.remove(l);
    }
}

public interface Listener {
    /**
     * @param deadlockedThread The deadlocked thread, with stack
     * trace of limited depth.
     */
    void deadlockDetected(ThreadInfo deadlockedThread);
    /**
     * @param allThreads All the threads in the JVM, without
     * stack traces.
     */
    void thresholdExceeded(ThreadInfo[] allThreads);
}

```

```

    }
}

```

The following test code creates many threads, then waits a few seconds before creating a whole new batch. We will see the warning system go off when we exceed the threshold for the first time, then again after we have waited for the first batch of threads to die.

```

import java.lang.management.*;

public class TooManyThreadsTest {
    public static void main(String[] args) throws InterruptedException {
        ThreadWarningSystem tws = new ThreadWarningSystem(500);
        tws.addListener(new ThreadWarningSystem.Listener() {
            public void deadlockDetected(ThreadInfo thread) { }
            public void thresholdExceeded(ThreadInfo[] threads) {
                System.out.println("Threshold Exceeded");
                System.out.println("threads.length = " + threads.length);
            }
        });
        createBatchOfThreads();
        Thread.sleep(10000);
        System.out.println("We should've dipped below the threshold");
        createBatchOfThreads();
    }

    private static void createBatchOfThreads() {
        for (int i=0; i<1000; i++) {
            new Thread() { {start();} }
            public void run() {
                try {
                    Thread.sleep(5000);
                } catch (InterruptedException e) { }
            }
        };
    }
}

```

On my machine I get the following output:

```

Threshold Exceeded
threads.length = 571
We should've dipped below the threshold
Threshold Exceeded
threads.length = 507

```

Testing for deadlocks is even more interesting. I have the classic case where two threads want lock1 and lock2, but in opposite orders. Then I have a more difficult case, where we have three threads that lock each other out:

```

import java.lang.management.ThreadInfo;

public class DeadlockedThreadsTest {
    public static void main(String[] args) {
        ThreadWarningSystem tws = new ThreadWarningSystem();
        tws.addListener(new ThreadWarningSystem.Listener() {
            public void deadlockDetected(ThreadInfo inf) {
                System.out.println("Deadlocked Thread:");
                System.out.println("-----");
                System.out.println(inf);
                for (StackTraceElement ste : inf.getStackTrace()) {
                    System.out.println("\t" + ste);
                }
            }
            public void thresholdExceeded(ThreadInfo[] threads) { }
        });

        // deadlock with three locks
        Object lock1 = new String("lock1");
        Object lock2 = new String("lock2");
        Object lock3 = new String("lock3");

        new DeadlockingThread("t1", lock1, lock2);
        new DeadlockingThread("t2", lock2, lock3);
        new DeadlockingThread("t3", lock3, lock1);

        // deadlock with two locks
        Object lock4 = new String("lock4");
        Object lock5 = new String("lock5");

        new DeadlockingThread("t4", lock4, lock5);
        new DeadlockingThread("t5", lock5, lock4);
    }
}

```

```

// There is absolutely nothing you can do when you have
// deadlocked threads. You cannot stop them, you cannot
// interrupt them, you cannot tell them to stop trying to
// get a lock, and you also cannot tell them to let go of
// the locks that they own.
private static class DeadlockingThread extends Thread {
    private final Object lock1;
    private final Object lock2;

    public DeadlockingThread(String name, Object lock1, Object lock2) {
        super(name);
        this.lock1 = lock1;
        this.lock2 = lock2;
        start();
    }
    public void run() {
        while (true) {
            f();
        }
    }
    private void f() {
        synchronized (lock1) {
            g();
        }
    }
    private void g() {
        synchronized (lock2) {
            // do some work...
            for (int i = 0; i < 1000 * 1000; i++) ;
        }
    }
}
}

```

Not surprisingly, it takes longer for the deadlock to happen with the three threads than with the two threads. Here is the output from the program:

Deadlocked Thread:

```

-----
Thread t5 (Id = 13) BLOCKED java.lang.String@de6ced
    DeadlockedThreadsTest$DeadlockingThread.g(DeadlockedThreadsTest.java:65)
    DeadlockedThreadsTest$DeadlockingThread.f(DeadlockedThreadsTest.java:61)

```

```

        DeadlockedThreadsTest$DeadlockingThread.run(DeadlockedThreadsTest.java:5
Deadlocked Thread:
-----
Thread t4 (Id = 12) RUNNABLE null
    DeadlockedThreadsTest$DeadlockingThread.g(DeadlockedThreadsTest.java:65)
    DeadlockedThreadsTest$DeadlockingThread.f(DeadlockedThreadsTest.java:61)
    DeadlockedThreadsTest$DeadlockingThread.run(DeadlockedThreadsTest.java:5
Deadlocked Thread:
-----
Thread t3 (Id = 11) BLOCKED java.lang.String@c17164
    DeadlockedThreadsTest$DeadlockingThread.g(DeadlockedThreadsTest.java:65)
    DeadlockedThreadsTest$DeadlockingThread.f(DeadlockedThreadsTest.java:61)
    DeadlockedThreadsTest$DeadlockingThread.run(DeadlockedThreadsTest.java:5
Deadlocked Thread:
-----
Thread t1 (Id = 9) BLOCKED java.lang.String@1fb8ee3
    DeadlockedThreadsTest$DeadlockingThread.g(DeadlockedThreadsTest.java:65)
    DeadlockedThreadsTest$DeadlockingThread.f(DeadlockedThreadsTest.java:61)
    DeadlockedThreadsTest$DeadlockingThread.run(DeadlockedThreadsTest.java:5
Deadlocked Thread:
-----
Thread t2 (Id = 10) RUNNABLE null
    DeadlockedThreadsTest$DeadlockingThread.g(DeadlockedThreadsTest.java:65)
    DeadlockedThreadsTest$DeadlockingThread.f(DeadlockedThreadsTest.java:61)
    DeadlockedThreadsTest$DeadlockingThread.run(DeadlockedThreadsTest.java:5

```

You can get back to the original Thread from the ThreadInfo class if necessary by calling Thread.getAllStackTraces(). I would presume that this is an expensive operation, so use it with caution. The Thread's ID matches the ThreadInfo's ID, so we can always get back to the Thread from the ThreadInfo.

```

private static Thread findMatchingThread(ThreadInfo inf) {
    Map<Thread, StackTraceElement[]> all = Thread.getAllStackTraces();
    for (Map.Entry<Thread, StackTraceElement[]> entry : all.entrySet()) {
        if (entry.getKey().getId() == inf.getThreadId()) {
            return entry.getKey();
        }
    }
    throw new NoSuchElementException();
}

```

Apparently, JDK 1.5 beta 3 has been released. Exciting stuff and I can hardly wait for it to be properly released, so that I can start slowly bringing this into some products. In the past, I would say to customers: "If the application stops responding, please go to the console and press CTRL+Break and then email me the threads that are printed on the screen." Now we can get notified automatically. Ohhhh, what joy!

Even though the actual code of this newsletter was easy, it took a long time to write this newsletter, and I don't know why. Maybe I have been too distracted of late. In Germany, I went to visit my good friend Dr Jung, who wrote the [masterpiece on dynamic proxies](#). We were a bit late for the train, so Christoph drove at breakneck speed to the train station, then grabbing my luggage ran full steam and hopped onto the train. A few minutes later, Christoph appeared and informed me that I had gotten onto the wrong train! Perhaps I am getting older, or maybe I am just becoming more nutty :-)

Kind regards

Heinz

Issue 070 - Too many dimensions are bad for you

Author: Dr. Heinz M. Kabutz

Date: 2003-05-17

Category: Performance

Java Versions:

Abstract:

Welcome to the 70th edition of **The Java(tm) Specialists' Newsletter**, where we have a quick look at multi-dimensional arrays. Many articles are available about this problem, so in this newsletter, I will briefly point out the problem and give you one example. I would like to thank Roy Emmerich (Peralex in Cape Town) and Pieter Potgieter and Louis Pool from Grintek Ewation for reminding me of this problem and for sending me a test program that illustrated it.

This past week we had a Design Patterns Course at one of the finest hotels of Cape Town. OK, it is *renowned* to be one of the finest ;-) The lunches were fantastic, but the coffees, well! Let me say that they are probably not used to having programmers at their hotel? There is a direct correlation between quality of coffee and quality of code, so we are quite discerning when it comes to the magic brew. But the lunches more than made up for it! On the first day we had baked sirloin steak, on the second we had filled chicken breasts and on the third we were treated to excellently prepared line fish (Cape Salmon, otherwise known as Geelbek). The discussions about the Design Patterns were deep, and we all left thoroughly exhausted (especially me!).

Too many dimensions are bad for you

Something you learn in Java nursery school is to avoid multi-dimensional arrays like the plague. An array is an object in its own right, and when you have multi-dimensional arrays, you have arrays of objects. Navigating these will take considerable processing power. Instead of making a multi-dimensional array of size n by m, rather make a one-dimensional array of size n times m and then do the index calculation yourself.

```
public class MultiDimensions {  
    private final static int NUM_BINS = 1000;  
    private final static int ITERATIONS = 10000;  
  
    public static void main(String[] args) {  
        testMultiArray();  
    }  
}
```

```
testMultiArray2();
testSingleArray();
}

private static void testMultiArray() {
    long time = -System.currentTimeMillis();
    // just making sure that the number of operations is equal
    int ops = 0;

    for (int repeat = 0; repeat < ITERATIONS; repeat++) {
        int[][] aTwoDim = new int[NUM_BINS][4];
        for (int i = 0; i < aTwoDim.length; i++) {
            for (int j = 0; j < aTwoDim[i].length; j++) {
                ops++;
                aTwoDim[i][j] = j;
            }
        }
    }

    time += System.currentTimeMillis();
    System.out.println(ops);
    System.out.println("Time Elapsed for [][4] - " + time);
}

private static void testMultiArray2() {
    long time = -System.currentTimeMillis();
    int ops = 0;

    for (int repeat = 0; repeat < ITERATIONS; repeat++) {
        int[][] aTwoDim = new int[4][NUM_BINS];
        for (int i = 0; i < aTwoDim.length; i++) {
            for (int j = 0; j < aTwoDim[i].length; j++) {
                ops++;
                aTwoDim[i][j] = j;
            }
        }
    }

    time += System.currentTimeMillis();
    System.out.println(ops);
    System.out.println("Time Elapsed for [4][] - " + time);
}

private static void testSingleArray() {
    long time = -System.currentTimeMillis();
    int ops = 0;

    for (int repeat = 0; repeat < ITERATIONS; repeat++) {
```

```

int[] aOneDim = new int[NUM_BINS * 4];
for (int i = 0; i < aOneDim.length/4; i++) {
    for (int j = 0; j < 4; j++) {
        ops++;
        aOneDim[i*4 + j] = j;
    }
}

time += System.currentTimeMillis();
System.out.println(ops);
System.out.println("Time Elapsed for [] - " + time);
}
}

```

When I run this code with the client hotspot under JDK 1.4.1, I get the following results:

```

40000000
Time Elapsed for [][4] - 4226
40000000
Time Elapsed for [4][] - 631
40000000
Time Elapsed for [] - 671

```

The server hotspot fares slightly better:

```

40000000
Time Elapsed for [][4] - 3675
40000000
Time Elapsed for [4][] - 350
40000000
Time Elapsed for [] - 561

```

The results are enlightning. If you have an array with a big first index, you will be better off using a single-dimensional array. Under server hotspot, the multi-dimensional array with a small first index fares quite a bit better than the single-dimensional array.

That's all that I would like to say about this issue. I warned you that this newsletter would be short :-)

Kind regards

Heinz

Issue 068 - Appending Strings

Author: Dr. Heinz M. Kabutz

Date: 2003-04-21

Category: Performance

Java Versions:

Abstract:

Welcome to the 68th edition of **The Java(tm) Specialists' Newsletter**, sent to 6400 Java Specialists in 95 countries.

Since our last newsletter, we have had two famous Java authors join the ranks of subscribers. It gives me great pleasure to welcome **Mark Grand** and **Bill Venners** to our list of subscribers. Mark is famous for his three volumes of Java Design Patterns books. You will notice that I quote Mark in the [brochure of my Design Patterns course](#). Bill is famous for his book [Inside The Java Virtual Machine](#). Bill also does a lot of work training with [Bruce Eckel](#).

Our last newsletter on [BASIC Java](#) produced gasps of disbelief. Some readers told me that they now wanted to unsubscribe, which of course I supported 100%. Others enjoyed it with me. It was meant in humour, as the warnings at the beginning of the newsletter clearly indicated.

Appending Strings

The first code that I look for when I am asked to find out why some code is slow is concatenation of Strings. When we concatenate Strings with `+=` a whole lot of objects are constructed.

Before we can look at an example, we need to define a Timer class that we will use for measuring performance:

```
/**
 * Class used to measure the time that a task takes to execute.
 * The method "time" prints out how long it took and returns
 * the time.
 */
public class Timer {
```

```

/**
 * This method runs the Runnable and measures how long it takes
 * @param r is the Runnable for the task that we want to measure
 * @return the time it took to execute this task
 */
public static long time(Runnable r) {
    long time = -System.currentTimeMillis();
    r.run();
    time += System.currentTimeMillis();
    System.out.println("Took " + time + "ms");
    return time;
}
}

```

In the test case, we have three tasks that we want to measure. The first is a simple `+= String` append, which turns out to be extremely slow. The second creates a `StringBuffer` and calls the `append` method of `StringBuffer`. The third method creates the `StringBuffer` with the correct size and then appends to that. After I have presented the code, I will explain what happens and why.

```

public class StringAppendDiff {
    public static void main(String[] args) {
        System.out.println("String += 10000 additions");
        Timer.time(new Runnable() {
            public void run() {
                String s = "";
                for(int i = 0; i < 10000; i++) {
                    s += i;
                }
                // we have to use "s" in some way, otherwise a clever
                // compiler would optimise it away. Not that I have
                // any such compiler, but just in case ;-)
                System.out.println("Length = " + s.length());
            }
        });
        System.out.println(
            "StringBuffer 300 * 10000 additions initial size wrong");
        Timer.time(new Runnable() {
            public void run() {
                StringBuffer sb = new StringBuffer();
                for(int i = 0; i < (300 * 10000); i++) {
                    sb.append(i);
                }
            }
        });
    }
}

```

```
String s = sb.toString();
System.out.println("Length = " + s.length());
}

});

System.out.println(
    "StringBuffer 300 * 10000 additions initial size right");
Timer.time(new Runnable() {
    public void run() {
        StringBuffer sb = new StringBuffer(19888890);
        for(int i = 0; i < (300 * 10000); i++) {
            sb.append(i);
        }
        String s = sb.toString();
        System.out.println("Length = " + s.length());
    }
});
}
```

This program does use quite a bit of memory, so you should set the maximum old generation heapspace to be quite large, for example 256mb. You can do that with the -Xmx256m flag. When we run this program, we get the following output:

```
String += 10000 additions
Length = 38890
Took 2203ms

StringBuffer 300 * 10000 additions initial size wrong
Length = 19888890
Took 2254ms

StringBuffer 300 * 10000 additions initial size right
Length = 19888890
Took 1562ms
```

You can observe that using StringBuffer directly is about 300 times faster than using `+=`. Another observation that we can make is that if we set the initial size to be correct, it only takes 1562ms instead of 2254ms. This is because of the way that `java.lang.StringBuffer` works. When you create a new `StringBuffer`, it creates a `char[]` of size 16. When you append, and there is no space left in the `char[]` then it is doubled in size. This means that if you size it first, you will reduce the number of `char[]`s that are constructed.

The time that the `+=` String append takes is dependent on the compiler that you use to

compile the code. I discovered this accidentally during my Java course last week, and much to my embarrassment, I did not know why this was. If you compile it from within Eclipse, you get the result above, and if you compile it with Sun's javac, you get the output below. I think that Eclipse uses jikes to compile the code, but I am not sure. Perhaps it even has an internal compiler?

```
String += 10000 additions
Length = 38890
Took 7912ms
StringBuffer 300 * 10000 additions initial size wrong
Length = 19888890
Took 2634ms
StringBuffer 300 * 10000 additions initial size right
Length = 19888890
Took 1822ms
```

Why the difference between compilers?

This took some head-scratching, resulting in my fingers being full of wood splinters. I started by writing a class that did the basic String append with `+=`.

```
public class BasicStringAppend {
    public BasicStringAppend() {
        String s = "";
        for(int i = 0; i < 100; i++) {
            s += i;
        }
    }
}
```

When in doubt about what the compiler does, disassemble the classes. Even when I disassembled them, it took a while before I figured out what the difference was and why it was important. The part where they differ is in *italics*. You can disassemble a class with the tool `javap` that is in the bin directory of your java installation. Use the `-c` parameter:

```
javap -c BasicStringAppend
```

Compiled with Eclipse:

Compiled from BasicStringAppend.java

```

public class BasicStringAppend extends java.lang.Object {
    public BasicStringAppend();
}

Method BasicStringAppend()
  0  aload_0
  1  invokespecial #9 <Method java.lang.Object()>
  4  ldc #11 <String "">
  6  astore_1
  7  iconst_0
  8  istore_2
  9  goto 34
 12 new #13 <Class java.lang.StringBuffer>
 15 dup
 16 aload_1
 17 invokestatic #19 <Method java.lang.String valueOf(java.lang.Object)>
 20 invokespecial #22 <Method java.lang.StringBuffer(java.lang.String)>
 23 iload_2
 24 invokevirtual #26 <Method java.lang.StringBuffer append(int)>
 27 invokevirtual #30 <Method java.lang.String toString()>
 30 astore_1
 31 iinc 2 1
 34 iload_2
 35 bipush 100
 37 if_icmplt 12
 40 return

```

Compiled with Sun's javac:

Compiled from BasicStringAppend.java

```

public class BasicStringAppend extends java.lang.Object {
    public BasicStringAppend();
}

Method BasicStringAppend()
  0  aload_0
  1  invokespecial #1 <Method java.lang.Object()>
  4  ldc #2 <String "">
  6  astore_1
  7  iconst_0
  8  istore_2

```

```

9 goto 34
12 new #3 <Class java.lang.StringBuffer>
15 dup
16 invokespecial #4 <Method java.lang.StringBuffer()>
19 aload_1
20 invokevirtual #5 <Method java.lang.StringBuffer append(java.lang.String)>
23 iload_2
24 invokevirtual #6 <Method java.lang.StringBuffer append(int)>
27 invokevirtual #7 <Method java.lang.String toString()>
30 astore_1
31 iinc 2 1
34 iload_2
35 bipush 100
37 if_icmplt 12
40 return

```

Instead of explaining what every line does (which I hope should not be necessary on a Java **Specialists' Newsletter**) I present the equivalent Java code for both IBM's Eclipse and Sun. The differences, which equate to the disassembled difference, is again in *italics*:

```

public class IbmBasicStringAppend {
    public IbmBasicStringAppend() {
        String s = "";
        for(int i = 0; i < 100; i++) {
            s = new StringBuffer(String.valueOf(s)).append(i).toString();
        }
    }
}

```

```

public class SunBasicStringAppend {
    public SunBasicStringAppend() {
        String s = "";
        for(int i = 0; i < 100; i++) {
            s = new StringBuffer().append(s).append(i).toString();
        }
    }
}

```

It does not actually matter which compiler is better, either is terrible. The answer is to avoid

`+=` with Strings wherever possible.

Throw the used StringBuffers away!

You should never reuse a StringBuffer object. Construct it, fill it, convert it to a String, and then throw it away.

Why is this? StringBuffer contains a `char[]` which holds the characters to be used for the String. When you call `toString()` on the StringBuffer, does it make a copy of the `char[]`? No, it assumes that you will throw the StringBuffer away and constructs a String with a pointer to the *same char[] that is contained inside StringBuffer!* If you do change the StringBuffer after creating a String, it makes a copy of the `char[]` and uses that internally. Do yourself a favour and read the source code of StringBuffer - it is enlightning.

But it gets worse than this. In JDK 1.4.1, Sun changed the way that `setLength()` works. Before 1.4.1, it was safe to do the following:

```
... // StringBuffer sb defined somewhere else
sb.append(...);
sb.append(...);
sb.append(...);
String s = sb.toString();
sb.setLength(0);
```

The code of `setLength` pre-1.4.1 used to contain the following snippet of code:

```
if (count < newLength) {
    // *snip*
} else {
    count = newLength;
    if (shared) {
        if (newLength > 0) {
            copy();
        } else {
            // If newLength is zero, assume the StringBuffer is being
            // stripped for reuse; Make new buffer of default size
            value = new char[16];
            shared = false;
        }
    }
}
```

```
    }
}
```

It was replaced in the 1.4.1 version with:

```
if (count < newLength) {
    // *snip*
} else {
    count = newLength;
    if (shared) copy();
}
```

Therefore, if you reuse a StringBuffer in JDK 1.4.1, and any one of the Strings created with that StringBuffer is big, all future Strings will have the same size `char[]`. This is not very kind of Sun, since it causes bugs in many libraries. However, my argument is that you should not have reused StringBuffers *anyway*, since you will have less overhead simply creating a new one than setting the size to zero again.

This memory leak was pointed out to me by Andrew Shearman during one of my courses, thank you very much! For more information, you can visit [Sun's website](#).

When you read those posts, it becomes apparent that JDOM reuses StringBuffers extensively. It was probably a bit mean to change StringBuffer's `setLength()` method, although I think that it is not a bug. It is simply highlighting bugs in many libraries.

For those of you that use JDOM, I hope that JDOM will be fixed soon to cater for this change in the JDK. For the rest of us, let us remember to throw away used StringBuffers.

So long...

Heinz

Issue 066 - Book Review: Java Performance Tuning by Jack Shirazi

Author: Dr. Heinz M. Kabutz

Date: 2003-03-21

Category: Book Review

Java Versions:

Abstract:

In this book, Jack outlines the process used to make Java systems run faster. He gives lots of tips on how to find your bottlenecks and then also gives specific tricks to make your code just that bit faster. A must-have for Java programmers who care about the speed of their programs.

Welcome to the 66th edition of **The Java(tm) Specialists' Newsletter**, sent to 6138 Java Experts in 94 countries. The newslist server is going to start throwing out dead email addresses, so the side effect is that the number of subscribers will likely decrease. If you did not receive this email, please respond by replying to this email and telling me that you did not receive it.

Last week I had great fun with some programmers in Frankfurt. We did the **Design Patterns Course** and good laughs together and of course we all learned new things. Thanks especially to my friend **Carl Smotricz** (who used to host our newsletter archive) for organising this event. One of the highlights of my trip was meeting with some of our newsletter subscribers for a good German beer.

Meeting Jack Shirazi

During my recent travels, I had the opportunity to meet with Jack Shirazi, famous as the author of the landmark book **Java Performance Tuning**. Besides consulting on performance related issues, Jack also maintains an **excellent website**, which is the #1 resource for expert Java programmers who want to keep up to date with Java performance. I particularly like his website because true to the topic, it is fast to navigate. The website does not look particularly pretty, but you can easily get to the information you want. There are no frames, no pictures, no special fonts, but gosh, does it render quickly! When I chatted to Jack about this, he told me that according to the principles in his book, the fastest content that you could possibly serve is static content, i.e. content that is not dynamically generated. Jack therefore generates his entire website and uploads the differences. This works because the content does not change all the time, it would be a bit difficult to build an entire internet banking site like this. However, even the internet banking site should be serving static content whenever this is possible.

Jack and I spent a good few hours chatting about all sorts of topics, from the state of

consulting to business ideas, to how many kids we have (at this point I knew that Jack was hopelessly superior at a 3:1 ratio and we agreed to decide on greatness by means of an arm-wrestle). The waitress was quite amused since neither of us really had any idea of what we were doing. There are two types of people in this world: those that can program, and those that can arm-wrestle. Our inability to achieve neither victory nor defeat has placed us firmly into the camp of programmers :-)

Ah, here is a good job interview question: "Hi, so you want to work for the XYZ company where we build important software. What did you think of the {rugby|cricket|soccer|football|basketball|baseball} game on Saturday?" An answer like "Was there a match? Who played?" will get you the job as programmer without any further questions, but saying "Yeah, Ted performed well - he has recovered well since his knee operation that he had in high school - he went to Osborne High from 1984-1989 where he was Valedictorian, did you know?" will only get you a position in the sales department. Mind you, the sales department is not a bad place to be - at least you can watch sport on the weekend and tell your boss and spouse that you are doing important work.

Let us not get sidetracked...

Java Performance Tuning 2nd Edition

Naturally I did not just travel 10000km to speak to Jack about kids and lucrative consulting jobs. [To those of you who do not have kids, once you have them, you will realise that they are far more important than anything else in life, even Java.]

The real reason I met with Jack was to speak about the 2nd Edition of his book **Java Performance Tuning**. When Jack wrote the first edition of his book, there was not much else available. The one book that I have mentioned in this newsletter is **Java 2 Performance and Idiom Guide**, and that was available before Jack's. However, Jack's book focuses far more on performance than the other book. In addition, Jack adds value to the reader with his **informative website** that is worthwhile to explore.

The book contains chapters on the following performance topics:

- **Chapter 1** gives general guidelines on how to tune. It will help you to formulate a good tuning strategy following a methodical process.
- **Chapter 2** covers the tools you need to use while tuning.
- **Chapter 3** looks at the SDK, including Virtual Machines (VMs) and compilers. Jack does not cover JDK 1.4.1, only up to JDK 1.4.0. This is understandable, because at the time of writing, JDK 1.4.1 had not been released.
- **Chapters 4 - 12** cover various techniques you can apply to Java code.
- **Chapter 12** looks at tuning techniques specific to distributed applications.
- **Chapter 13** steps back from the low-level code-tuning techniques examined throughout most of the book and considers tuning at all other stages of the development process.
- **Chapter 14** is a quick look at some operating system-level tuning techniques.
- **Chapters 15 - 18** are new in this edition, providing the information you need to tune

J2EE applications. Chapter 15 describes tuning considerations that are common to all J2EE applications. Chapter 16 looks at specifics for tuning JDBC, including optimizing transactions and SQL, while Chapter 17 provides important information for speeding up servlets and JSPs. Chapter 18 describes performance considerations for Enterprise JavaBeans (EJBs).

- Finally, **Chapter 19** covers a wide range of additional resources where you can learn more about Java performance tuning.

Chapter 1: Introduction

The book starts with a chapter outlining a performance tuning strategy that you should follow when attempting to tune an application. Follow it, and you will be successful. Ignore it, and you will forever be tuning and never get any acknowledgements for your achievements. At the end of each chapter is a performance checklist of the major points of that chapter. For example:

- Specify the required performance.
- Make your benchmarks long enough: over five seconds is a good target.
- Break down distributed application measurements into components, transfer layers, and network transfer times.
- Tune systematically: understand what affects the performance; define targets; tune; monitor and redefine targets when necessary.
- Work with user expectations to provide the appearance of better performance.
- Quality-test the application after any optimizations have been made.
- Document optimizations fully in the code. Retain old code in comments.

Chapter 6: Exceptions, Assertions, Casts, and Variables

Many moons ago, when Java had just been invented, programmers invented strange ways to write optimal code. One of the weirdest abominations was the logic that a **try/catch** was for free when no exception was thrown and that it was cheaper than **if** and **instanceof** statements. For example:

```
public class TryCatch {
    public static boolean test1(Object o) {
        try {
            Integer i = (Integer)o;
            return false;
        } catch(Exception e) {
            return true;
        }
    }
}
```

```

public static boolean test2(Object o) {
    if (o instanceof Integer) {
        Integer i = (Integer)o;
        return false;
    } else {
        return true;
    }
}
}

```

The thinking was that a try block was free, so therefore if you almost always have Integer objects, it will be faster to just cast and hope for the best, than to first test if it really is an Integer. `test1()` should therefore be faster than `test2()` most of the time, unless we passed in an object that was not an Integer.

But is it really faster? Let us have a look:

```

public class TryCatchCostTest extends TryCatch {
    public static void main(String[] args) {
        Integer i = new Integer(3);
        Boolean b = new Boolean(true);
        long TEST_DURATION = 2 * 1000;
        boolean res;
        long stop;

        stop = TEST_DURATION + System.currentTimeMillis();
        int test1_i = 0;
        // we do not want to test with every increment, otherwise the
        // call to System.currentTimeMillis() will dwarf the rest of
        // our calls.
        while(((test1_i % 1000) != 0)
            || (System.currentTimeMillis() < stop)) {
            test1_i++;
            res = test1(i);
        }
        System.out.println("test1(i) executed " + test1_i + " times");

        stop = TEST_DURATION + System.currentTimeMillis();
        int test1_b = 0;
        while(((test1_b % 1000) != 0)
            || (System.currentTimeMillis() < stop)) {
            test1_b++;
        }
    }
}

```

```

        res = test1(b);
    }
System.out.println("test1(b) executed " + test1_b + " times");
System.out.println("test1(i) was " + (test1_i / test1_b)
                  + " times faster");

stop = TEST_DURATION + System.currentTimeMillis();
int test2_i = 0;
while(((test2_i % 1000) != 0)
      || (System.currentTimeMillis() < stop)) {
    test2_i++;
    res = test2(i);
}
System.out.println("test2(i) executed " + test2_i + " times");

stop = TEST_DURATION + System.currentTimeMillis();
int test2_b = 0;
while(((test2_b % 1000) != 0)
      || (System.currentTimeMillis() < stop)) {
    test2_b++;
    res = test2(b);
}
System.out.println("test2(b) executed " + test2_b + " times");
}
}

```

When I run this test on JDK 1.4.1_01 with my Pentium 4m 1.7GHz, I get the following values under client hotspot:

```

test1(i) executed 22700000 times
test1(b) executed 126000 times
test1(i) was 180 times faster
test2(i) executed 52420000 times
test2(b) executed 53015000 times

```

Oops, it appears that logic was incorrect! `test2()` is about twice as fast as the best `test1()`! How does this look with the server hotspot compiler?

```
java -server TryCatchCostTest
```

```
test1(i) executed 50310000 times
test1(b) executed 43000 times
test1(i) was 1170 times faster
test2(i) executed 51915000 times
test2(b) executed 51952000 times
```

The difference between `test1()` and `test2()` is not as pronounced as with the client hotspot, but `test2()` is still marginally faster. However, the version of `test1()` that causes the exception has become three times slower!

Under what circumstances did this idea proliferate? My suspicion is that it used to be true for the interpreted version of Java, probably around JDK 1.0. Have a look at what happens when we run this test with the `-Xint` switch. It appears that `test1(i)` is indeed faster!

```
java -Xint TryCatchCostTest
test1(i) executed 19153000 times
test1(b) executed 180000 times
test1(i) was 106 times faster
test2(i) executed 14522000 times
test2(b) executed 18719000 times
```

The **book** is great in that for all important statistics, it compares performance results for Sun JDK 1.1.8, 1.2.2, 1.3.1, 1.3.1 -server, 1.4.0, 1.4.0 -server and 1.4.0 -Xint. This is probably the most useful part of the book, and also lends credibility to the statements that Jack makes. If Jack makes a statement without proof, you cannot assume that it is true. However, when the statement is backed up with values, you can be assured that it is probably correct.

Can you paint yourself into a corner?

Kent Beck is famous for his phrase: "Make it run, make it right, make it fast."

Unless we can compile it, we do not need to try tune the performance. Once we get it to compile, we have to make sure that it is correct. What is the point of having a blazingly fast algorithm that is incorrect? I can sort any collection of elements in $O(1)$ if it does not have to be correct.

In my doctoral thesis, I examined how one could analytically evaluate the performance of a distributed system written in the Specification and Description Language (SDL). Since there were few analytical techniques for performance evaluation of SDL systems, I converted SDL to a new form of Petri net based on the Queuing Petri Net invented by the University of

Dortmund. There are many correctness and performance testing techniques for Petri nets, so the first step was to determine whether the net contained deadlocks, was unbounded or had livelocks. Once the correctness had been determined, I could apply the numerous performance techniques to the net and then convert the results back to the original SDL system.

Due to the complexity of the Markov Chain that was generated from the Petri net, we could only analyse small systems, but then, it was a doctoral thesis, not a commercial product. You are allowed to dream a bit in theory when you do a PhD, infact, a PhD is where you investigate everything about nothing.

In the real world, we usually stop after the first test: "does it compile?" Most software in the world has not been tested before being delivered. Why? Because testing that it works correctly is a tedious job that takes time and programmers do not make mistakes anyway. I am yet to walk into a company and ask whether they have unit tests, without getting a nervous reaction. When I then ask what they are doing about performance I get an even more nervous reaction.

When the system then performs badly, they have to call in trouble-shooters like Jack and myself to tune their applications. Performance involves both CPU times and memory sizes. A few newsletter ago, I casually asked whether you wanted to know how to detect OutOfMemoryErrors ahead of time. More than 100 readers responded that they were in desperate need of such a tool. Jack and I do not mind this state of affairs at all - that is a good way to earn our keep. However, you will have a problem when you need Jack and me but you cannot afford our rates. Then you will kick yourself for putting performance last.

I asked Jack about the philosophy of "Make it run, make it right, make it fast" and Jack mentioned to me that there are situations where this does not work. There are ways in which you can paint yourself in a corner so that the only alternative is to throw away the application and start again. This piqued my curiosity, and I asked Jack for an example.

"An example of painting yourself in the corner is when you do not use the *Page-by-Page Iterator Design Pattern* from the beginning"

As I thought about the troubled projects I had consulted for, I realised that Jack was correct. You can get a better optimising compiler, you can reduce the number of Strings that get created, you can change the collection you use, but if you do not have the Page-by-Page Iterator Design Pattern in your design from the beginning, you will be p*ssing against the wind. This applies for 3-tier and for 2-tier applications. The P-b-P Iterator can be added to your system after-the-fact, but it will require plenty of work and cause errors.

Conclusion

Should you purchase the **Java Performance Tuning Book**? There is no easy answer to that question. If you do not have the US\$ 44.95 to pay for the book, it will be unlikely that you will

be able to afford to pay for Jack or me to help you tune your system. Basically, you would be on your own. You can also browse [Jack's website](#) or look at [our newsletter archive](#). Otherwise, if you are **serious** about writing good Java code that is well-tuned, you would do well to purchase a copy of Jack's book, and read through it over a period of a few months.

That is all I have to say on this topic. The book could be converted into about 30 newsletters, that is how much information is contained therein.

Kind regards

Heinz

Issue 064 - Disassembling Java Classes

Author: Dr. Heinz M. Kabutz

Date: 2003-02-14

Category: Performance

Java Versions:

Abstract:

Welcome to the 64th edition of **The Java(tm) Specialists' Newsletter** sent to 5869 Java Specialists in **94 countries**.

I stepped off the plane last Tuesday, having come via London from Estonia, and was hit in the face by 31 degree Celsius heat. I was still wearing my warm clothes that were necessary in the -17 degree Celsius in Estonia (travel report to follow), but very quickly, I adapted again to shorts and T-shirt.

While I was in London, I had the opportunity to meet Jack Shirazi, the author of the **best Java performance tuning website**. Basically, when I don't know or understand something about Java performance, I go look at the website. Still coming is a review of Jack's book.

Disassembling Java Classes

A few months ago, on our local Java User Group discussion forum, I casually asked the question what was faster: `i++`, `++i` or `i+=1`. I also mentioned that the answer might surprise them. Wanting to encourage thinking, I never gave the answer ;-)

Yesterday, one of my readers, a bright young student at the Cape Town Technical University, sent me an answer based on a thorough investigation through microbenchmarks. His conclusion was correct, but the approach to getting to the conclusion led to a lot of effort being spent on his part.

The easiest way to decide which is fastest is actually to *disassemble* the Java Class. Note that I am saying disassemble, not decompile. Let's look at the following class:

```
public class Increment {  
    public int preIncrement() {  
        int i = 0;  
    }  
}
```

```

    ++i;
    return i;
}
public int postIncrement() {
    int i = 0;
    i++;
    return i;
}
public int negative() {
    int i = 0;
    i=-1;
    return i;
}
public int plusEquals() {
    int i = 0;
    i += 1;
    return i;
}
}
}

```

I may ask you: which method is the fastest, which is the slowest? C programmers would probably say that the fastest are `i++` and `++i`. When you try running these methods many times, you will notice slight differences between them, based on which you run first, how soon the hotspot kicks in, whether you use client or server hotspot, whether you are busy playing an MP3 on your machine while running the test and the phase of the moon.

Instead of measuring the performance, why not investigate what the Java compiler did? You can disassemble a class with the standard `javap` tool available in your `JAVA_HOME\bin` directory:

```
javap -c Increment
```

Note that the class must already be compiled. The result is the following:

```

Compiled from Increment.java
public class Increment extends java.lang.Object {
    public Increment();
    public int preIncrement();
    public int postIncrement();
}

```

```

    public int negative();
    public int plusEquals();
}

Method Increment()
0 aload_0
1 invokespecial #9 <Method java.lang.Object()>
4 return

Method int preIncrement()
0 iconst_0
1 istore_1
2 iinc 1 1
5 iload_1
6 ireturn

Method int postIncrement()
0 iconst_0
1 istore_1
2 iinc 1 1
5 iload_1
6 ireturn

Method int negative()
0 iconst_0
1 istore_1
2 iinc 1 1
5 iload_1
6 ireturn

Method int plusEquals()
0 iconst_0
1 istore_1
2 iinc 1 1
5 iload_1
6 ireturn

```

Now when we look at the different methods of incrementing the local variable, we can see that they are actually identical!

Next time you are thinking of measuring performance using `System.currentTimeMillis()`, think of also looking at the generated byte code. It might save you a lot of time.

I want to thank Daniël Maree for inspiring me to write this newsletter through his thorough research as to which increment method was faster.

Kind regards

Heinz

P.S. I hope you all enjoy Valentine's day. Remember to leave work early today - i.e. before 8pm ;-)

Issue 062 - The link to the outer class

Author: Dr. Heinz M. Kabutz

Date: 2002-12-27

Category: Language

Java Versions:

Abstract:

Welcome to the 62nd edition of **The Java(tm) Specialists' Newsletter** sent to 5450 Java Specialists in **91 countries**.

Here I am, sitting in my shorts on a beautiful starry night on our balcony, listening to the Guinea Fowls in our trees, trying to churn out yet another newsletter to appeal to your cerebral impulses. Guinea Fowls make a very strange sound, which is extremely annoying to some, and music to others. Fortunately for me (and the Guinea Fowls), I like their squawking. From the sunburnt-southern-hemispherer to all eggnog-drinking-northern-hemispherers: Relax, the days are getting longer again :-)

Don't you just love those Out-Of-Office messages? Boy, am I going to get a lot of those with this newsletter...

The link to the outer class

Consider the following classes:

```
public abstract class Insect {  
    public Insect() {  
        System.out.println("Inside Insect() Constructor");  
        printDetails();  
    }  
    public void printDetails() {  
        System.out.println("Just an insect");  
    }  
}  
  
public class Beetle extends Insect {  
    private final int legs;  
    public Beetle(int legs) {  
        this.legs = legs;  
    }  
    public void printDetails() {  
        System.out.println("A beetle with " + legs + " legs");  
    }  
}
```

```
System.out.println("Inside Beetle() Constructor");
this.legs = legs;
}
public void printDetails() {
    System.out.println("The beetle has " + legs + " legs");
    if (legs < 6) {
        System.out.println("Ouch");
    }
}
}

public class BeetleTest {
    public static void main(String[] args) {
        Beetle sad_bug = new Beetle(5); // lost one leg in an
                                         // argument with his wife
        Beetle happy_bug = new Beetle(6); // the wife bug ;-
    }
}
```

Stop for a moment and think of what the effect would be of running BeetleTest. Don't read further until you have decided what would happen.

I hope you didn't peep :-) Here is the output:

```
Inside Insect() Constructor
The beetle has 0 legs
Ouch
Inside Beetle() Constructor
Inside Insect() Constructor
The beetle has 0 legs
Ouch
Inside Beetle() Constructor
```

Yes, even though `legs` was **final**, we were able to access it before it was initialised. What is more, we are able to call the subclass' methods from the constructor of the superclass,

before the subclass had been initialised! This should come as no surprise to you, since by being subscribed to **The Java(tm) Specialists' Newsletter** you would be classed as a Java Specialist. Yes? No ... ?

But, the plot thickens. Look at the following class:

```
public class NestedBug {
    private Integer wings = new Integer(2);
    public NestedBug() {
        new ComplexBug();
    }
    private class ComplexBug extends Insect {
        public void printDetails() {
            System.out.println(wings);
        }
    }
    public static void main(String[] arguments) {
        new NestedBug();
    }
}
```

When we run this code, we get a NullPointerException:

```
Inside Insect() Constructor
java.lang.NullPointerException
    at NestedBug.access$0(NestedBug.java:2)
    at NestedBug$ComplexBug.printDetails(NestedBug.java:8)
    at Insect.<init>(Insect.java:4)
    at NestedBug$ComplexBug.<init>(NestedBug.java:6)
    at NestedBug.<init>(NestedBug.java:4)
    at NestedBug.main(NestedBug.java:12)
Exception in thread "main"
```

A friend of mine once had this problem, so my first thought was that somehow, because `wings` was null, we ended up with a NullPointerException when printing it. That explanation did not make sense, because calling `toString()` on a null pointer is supposed to just return `null`. My friend changed his code to the following:

```

public class NestedBug2 {
    private Integer wings = new Integer(2);
    public NestedBug2() {
        new ComplexBug();
    }
    private class ComplexBug extends Insect {
        public void printDetails() {
            if (wings != null) { // line 8
                System.out.println(wings);
            }
        }
    }
    public static void main(String[] arguments) {
        new NestedBug2();
    }
}

```

Sadly, this does not make the NullPointerException go away:

```

Inside Insect() Constructor
java.lang.NullPointerException
    at NestedBug2.access$0(NestedBug2.java:2)
    at NestedBug2$ComplexBug.printDetails(NestedBug2.java:8)
    at Insect.<init>(Insect.java:4)
    at NestedBug2$ComplexBug.<init>(NestedBug2.java:6)
    at NestedBug2.<init>(NestedBug2.java:4)
    at NestedBug2.main(NestedBug2.java:14)
Exception in thread "main"

```

But wait! The line with the NullPointerException is the line that simply checks whether wings is null? Can the mere act of checking whether wings is `null` itself cause a NullPointerException? In this case it appears that it can! Or is the actual mistake on line 2? What is this `access$0` method?

To understand this strange behaviour, we have to understand what the constructor of the inner class consists of. The easiest way of doing this is to use JAD and decompile the class file with the `-noinner` option:

```
jad -noinner NestedBug2$ComplexBug.class

class NestedBug2$ComplexBug extends Insect {
    NestedBug2$ComplexBug(NestedBug2 nestedbug2) {
        this$0 = nestedbug2;
    }
    public void printDetails() {
        if (NestedBug2.access$0(this$0) != null)
            System.out.println(NestedBug2.access$0(this$0));
    }
    private final NestedBug2 this$0; /* synthetic field */
}
```

We also need to look at NestedBug2.class:

```
jad -noinner NestedBug2.class

public class NestedBug2 {
    public NestedBug2() {
        wings = new Integer(2);
        new NestedBug2$ComplexBug(this);
    }
    public static void main(String arguments[]) {
        new NestedBug2();
    }
    static Integer access$0(NestedBug2 nestedbug2) {
        return nestedbug2.wings;
    }
    private Integer wings;
}
```

Again, I must urge you to spend a few minutes thinking about this. Step through what would happen in your head or scribble on a piece of paper, but try to understand. When `printDetails()` is called by `Insect>`, the handle to outer class (`this$0`) has not yet been set in the `ComplexBug` class, so that class passes `null` to the `access$0` method, which of course causes a `NullPointerException` when it tries to access `nestedbug2.wings` as we would expect.

I bet that many developers have run into this problem, but being under typical time pressure would have just coded around it until they got it working, without taking the time to think

about what is happening.

This experience leads us to some questions:

- Is `this$0` a new type of keyword you did not know about?
- What happens when your inner class already contains a variable `this$0`?
- Is calling a method from a constructor a good idea in the first place?
- When, if ever, are inner classes a good idea?

I will leave you with these questions to answer yourself over the festive season. For those of you who are actually working, I hope this newsletter has cheered up your workday and that you will be motivated to seek new nuggets of information inside the fascinating Java language.

All the best for the new year. May 2003 bring you new opportunities to learn and grow, not just in the material world :-)

Heinz

Issue 056 - Shutting down threads cleanly

Author: Dr. Heinz M. Kabutz

Date: 2002-09-16

Category: Language

Java Versions:

Abstract:

Welcome to the 56th edition of **The Java(tm) Specialists' Newsletter** sent to 4609 Java Specialists in **85 countries**. Whenever I think I wrote a "killer" newsletter, I get a lukewarm response from my readers, and when I think I wrote a flop, I get accolades, like with my last newsletter about Oak. I expected rotten tomatoes in the form of "here are 29 other newsletters about Oak and modern Java". Perhaps I should *try* and write flops, then I will have more successes ;-)

When I was in Germany from June to August 2000, we lived in a village about 45 minutes by train from **Infor AG**. Every day I had 1.5 hours to read books about various topics. Over the course of 3 months, I got through a lot of reading, I even read **The Java Virtual Machine Specification!** Much like the Oak specification, it is a *must* for any serious Java developer to read. The more I read in that spec, the more I realised that the hope of "write once, run anywhere" was a wishful dream. Another book that I devoured is **Concurrent Programming in Java, 2nd Ed** by Doug Lea, one of the writers of `java.util.HashMap`. My thoughts on shutting down threads is based on ideas I gleaned from those two books, so before you shoot me down, please read the books. The **Java 2 Performance and Idiom Guide** is also an excellent book to read for the reason that it got me thinking about performance the way that no other book has. It is quite dated, which cannot be avoided with a book about a language that changes so quickly. That book also agrees with my approach of shutting down threads.

How to shutdown threads cleanly

I remember starting off with JDK 1.0, and playing with Threads. I would start a Thread, and then to stop it, I simply called `stop()`. It was the most obvious thing to do, with disastrous consequences. If we refer back to the **Oak Newsletter**, that was the same as having *all* your code unprotected, and receiving an **asynchronous** exception in your Thread. The exception that is thrown asynchronously is `java.lang.ThreadDeath`. Needless to say, the more I think about it, the more I am baffled that they allowed that in the first place, especially in the absence of the `protect` keyword that they had in Oak. Using `stop()` is incredibly dangerous, as it will kill your thread even if it is in the middle of something important. There is no way to protect yourself, so if you spot code that uses `stop()`, you should frown.

So, how do you shutdown a thread cleanly? The developers of Java have actually left the

protect mechanism in place for us to use, it is just named differently. First of all, we should never use `stop()`. Ever. Period. Do not even think of using it. Sun should remove it from `java.lang.Thread` as soon as possible. Don't. No. Nooooooo. Doooown. Secondly, the only place where we are allowed to receive an exception to tell us that the thread is being shutdown is while the thread is blocked. Getting a shutdown notification at *any* other time would be dangerous and nondeterministic. The way this works is via the `java.lang.InterruptedException`. I admit that `InterruptedException` is not the most obvious choice of *name* for indicating that another thread is trying to shutdown your thread.

I have seen code that uses a boolean flag to indicate whether the thread is running or not. However, Java already provides that flag in the form of the interrupted flag, so why duplicate effort? Usually, the code would work something like this:

```
public class UsingFlagToShutdownThread extends Thread {
    private volatile boolean running = true;
    public void run() {
        while (running) {
            System.out.print(".");
            System.out.flush();
            try {
                Thread.sleep(1000);
            } catch (InterruptedException ex) {}
        }
        System.out.println("Shutting down thread");
    }
    public void shutdown() {
        running = false;
    }
    public static void main(String[] args)
        throws InterruptedException {
        UsingFlagToShutdownThread t = new UsingFlagToShutdownThread();
        t.start();
        Thread.sleep(5000);
        t.shutdown();
    }
}
```

What is so bad with that code? This example is not too bad, since the longest we would wait unnecessarily would be one second. However if we normally sleep for 30 seconds, then it could take a while before your program is completely shut down. This is especially true if you have a lot of threads and you `join()` each one to make sure that it does finish.

Java has another mechanism that you should rather use: simply interrupt the thread. The

code would then look like this:

```
public class UsingInterruptToShutdownThread extends Thread {
    public void run() {
        while (true) {
            System.out.print(".");
            System.out.flush();
            try {
                Thread.sleep(1000);
            } catch (InterruptedException ex) {
                Thread.currentThread().interrupt(); // very important
                break;
            }
        }
        System.out.println("Shutting down thread");
    }

    public static void main(String[] args)
        throws InterruptedException {
        Thread t = new UsingInterruptToShutdownThread();
        t.start();
        Thread.sleep(5000);
        t.interrupt();
    }
}
```

I must admit that I have not seen many programmers handle InterruptedExceptions correctly, i.e. using my way ;-) Most of the time, programmers view InterruptedException as an irritating checked exception that they have to catch, but which they usually ignore:

```
while (true) {
    // ... do something
    try {
        Thread.sleep(30000);
    } catch (InterruptedException ex) {}
}
```

Why do we have to interrupt the thread again?

In my example, after I caught the InterruptedException, I used `Thread.currentThread().interrupt()` to immediately interrupted the thread again.

Why is this necessary? When the exception is thrown, the interrupted flag is cleared, so if you have nested loops, you will cause trouble in the outer loops. Consider the following code:

```
public class NestedLoops extends Thread {
    private static boolean correct = true;
    public void run() {
        while (true) {
            System.out.print(".");
            System.out.flush();
            for (int i = 0; i < 10; i++) {
                System.out.print("#");
                System.out.flush();
                try {
                    Thread.sleep(100);
                } catch (InterruptedException ex) {
                    if (correct) Thread.currentThread().interrupt();
                    System.out.println();
                    System.out.println("Shut down inner loop");
                    break;
                }
            }
            try {
                Thread.sleep(1000);
            } catch (InterruptedException ex) {
                if (correct) Thread.currentThread().interrupt();
                System.out.println();
                System.out.println("Shut down outer loop");
                break;
            }
        }
        System.out.println("Shutting down thread");
    }
    private static void test() throws InterruptedException {
        Thread t = new NestedLoops();
        t.start();
        Thread.sleep(6500);
        t.interrupt();
        t.join();
        System.out.println("Shutdown the thread correctly");
    }
    public static void main(String[] args)
        throws InterruptedException {
        test();
        correct = false;
        test();
    }
}
```

```
}
```

When you run this code, you will see something like this:

```
.#####.#####.#####.#####
Shut down inner loop

Shut down outer loop
Shutting down thread
Shutdown the thread correctly
.#####.#####.#####.#####
Shut down inner loop
.#####.#####.#####.#####.etc.
```

Herein lies the danger with this approach: if some library incorrectly handles `InterruptedException` then your code will not shut down correctly.

From a purely theoretical view, you should use the interrupt mechanism of threads to shut them down. However, you have to be very careful that you use that mechanism throughout your code, otherwise you will not be able to shut down all your threads.

What about threads blocked on IO?

Threads can be blocked on `wait()`, `sleep()`, waiting to enter a synchronized block or waiting on some IO to complete. We cannot shut down a thread waiting to enter a synchronized block, so if you have a livelock or deadlock you will not be able to shut down your system cleanly. `wait()` and `sleep()` both throw an `InterruptedException`, as does `join()`. But, what about when you're blocked on IO? There is an exception called `java.io.InterruptedIOException`, which is supposed to cover the situation where you interrupt a thread that is waiting on some IO to complete. As you might have guessed, it is not implemented consistently. It works for piped streams, but none of the others seem to have that effect.

If you want to stop a thread waiting on a socket, you will have to unfortunately close the socket underneath the thread. Fortunately, the `interrupt()` method is not `final`, so you can override it to also close the socket. Inside the `catch` clause of `java.io.IOException` you can then check whether the thread has been interrupted or not:

```

import java.io.IOException;
import java.io.InputStream;
import java.io.InterruptedIOException;

public class BlockedOnIO extends Thread {
    private final InputStream in;
    public BlockedOnIO(InputStream in) {
        this.in = in;
    }
    public void interrupt() {
        super.interrupt();
        try {
            in.close();
        } catch (IOException e) {} // quietly close
    }
    public void run() {
        try {
            System.out.println("Reading from input stream");
            in.read();
            System.out.println("Finished reading");
        } catch (InterruptedException e) {
            Thread.currentThread().interrupt();
            System.out.println("Interrupted via InterruptedException");
        } catch (IOException e) {
            if (!isInterrupted()) {
                e.printStackTrace();
            } else {
                System.out.println("Interrupted");
            }
        }
        System.out.println("Shutting down thread");
    }
}

```

For shutting down threads reading from sockets, we would do something like this:

```

import java.io.IOException;
import java.net.ServerSocket;
import java.net.Socket;

public class BlockedOnSocketIO {
    public static void main(String[] args)
        throws IOException, InterruptedException {
        ServerSocket ss = new ServerSocket(4444);

```

```

Socket socket = new Socket("localhost", 4444);
System.out.println("Made socket, now reading from socket");
Thread t = new BlockedOnIO(socket.getInputStream());
t.start();
Thread.sleep(5000);
t.interrupt();
}
}

```

When we run our code, we see the following:

```

Made socket, now reading from socket
Reading from input stream
Interrupted
Shutting down thread

```

Alternatively, when we use Pipes:

```

import java.io.IOException;
import java.io.PipedInputStream;
import java.io.PipedOutputStream;

public class BlockedOnPipedIO {
    public static void main(String[] args)
        throws IOException, InterruptedException {
        PipedInputStream in =
            new PipedInputStream(new PipedOutputStream());
        Thread t = new BlockedOnIO(in);
        t.start();
        Thread.sleep(5000);
        t.interrupt();
    }
}

```

When we run that code, we see the following:

```
Reading from input stream  
Interrupted via InterruptedException  
Shutting down thread
```

Unfortunately, the IO library in Java is not consistent, so you have to cater for both possibilities in your shutdown methods.

I hope that this newsletter will be as useful to you as it has been to me. Shutting down threads cleanly is unfortunately not as easy as it should be, but the mechanism in this newsletter is superior to calling `stop()` (and thereby using an **asynchronous exception**) and it is also better than using a flag to indicate whether the thread is supposed to carry on running or not.

The only problem with my approach is that if you use some library that does not handle `InterruptedException` correctly, you will have problems shutting down your thread. You might have to have a separate thread that calls `join()` with a timeout and repeatedly interrupts the thread until it is shut down.

That's the end of the newsletter. The birds are singing to celebrate spring, my baby sister is here to visit, so we are now going to celebrate life with a Cuban cigar :-)

Heinz

Issue 042 - Speed-kings of inverting booleans

Author: Dr. Heinz M. Kabutz

Date: 2002-02-23

Category: Performance

Java Versions:

Abstract:

Welcome to the 42nd edition of **The Java(tm) Specialists' Newsletter**, sent to 2779 Java experts in over 75 countries. I'm writing this newsletter about 30'000 feet above mother earth on my way to Mauritius. Yes, eventually we managed to sort out all the bureaucratic problems, and had to just shift the Design Patterns course by one week :-))

Speed-kings of inverting booleans

About 10 days ago, I was chatting on ICQ to **Roman Porotnikov**, the best Java programmer in the Ukraine according to Brainbench, when he posed an interesting question:

"What's more quick variant for `flag = !flag;? :` (one guy said `f1g = !f1g;` is an answer ;))"

I didn't really know the answer, so I guessed: "Probably `flag = flag ? false : true;`"

Being the avid programmer that I am, I quickly wrote a test program:

```
public class NotTest1 {
    public static void main(String[] args) {
        boolean flag = true;
        long start;
        start = -System.currentTimeMillis();
        for (int i=0; i<100000000; i++) {
            flag = !flag;
        }
        start += System.currentTimeMillis();
        System.out.println("time for flag = !flag: " + start + "ms");
    }
}
```

```

start = -System.currentTimeMillis();
for (int i=0; i<100000000; i++) {
    flag = flag?false:true;
}
start += System.currentTimeMillis();
System.out.println("time for flag = flag?false:true: " + start + "ms");
}
}

```

Imagine my glee when I saw the following performance results. Roman might be the best Java programmer in the Ukraine, but I am the best Java programmer on this airplane!

```

time for b = !b: 1712ms
time for b = b?false:true: 1132ms

```

I was still puzzling over this as I could not understand how that could possibly be faster, when Roman piped up:

"The answer is actually `flag ^= true;`"

Hmmmm - XOR on a bitwise level - sneaky! I added his "way" to my test to see if it really was faster, although I did believe that bitwise manipulation should be faster, but you never know with Java ;-)

```

public class NotTest2 {
    public static void main(String[] args) {
        boolean flag = true;
        long start;
        start = -System.currentTimeMillis();
        for (int i=0; i<100000000; i++) {
            flag = !flag;
        }
        start += System.currentTimeMillis();
        System.out.println("time for flag = !flag: " + start + "ms");

        start = -System.currentTimeMillis();
        for (int i=0; i<100000000; i++) {
            flag = flag?false:true;
        }
    }
}

```

```

        }
        start += System.currentTimeMillis();
        System.out.println("time for flag = flag?false:true: " + start + "ms");

        for (int i=0; i<100000000; i++) {
            flag ^= true; // XOR
        }
        start += System.currentTimeMillis();
        System.out.println("time for flag ^= true: " + start + "ms");
    }
}

```

And of course, Roman was right, as you can see from the figures below.

```

time for flag = !flag: 1722ms
time for flag = flag?false:true: 1162ms
time for flag ^= true: 781ms

```

Interesting figures. It proves that my version is 32% faster and that Roman's version is 55% faster.

I mentioned this strange idea to [Paul van Spronsen](#) and he suggested we look at the generated bytecode. You can disassemble Java bytecode with the `javap` tool that forms part of the JDK. [HK: at this point of writing, we hit some turbulence and our food was being served so I thought it best to wait until the [hotel](#). I must just add that this is the best hotel I've stayed at in all my travels and we are planning another bunch of courses in May - will let you know next newsletter. Back to the newsletter ...] In order to be able to compare the bytecode easily, I've split the cases into `Normal.java`, `Faster.java` and `Fastest.java`.

```

public class Normal {
    public void test() {
        boolean flag = true;
        flag = !flag;
    }
}

```

Compiling this class and running the command `javap -c Normal` produced the following

for method test (comments are mine):

```
Method void test()
  0 iconst_1    // push constant "true"
  1 istore_1    // store in location 1 (flag)
  2 iload_1     // load value in location 1
  3 ifne 10     // if value is false goto bytecode 10
  6 iconst_1    // push constant "true"
  7 goto 11     // goto location 11
 10 iconst_0    // push constant "false"
 11 istore_1    // store value on stack in location 1
 12 return      // duh - this is obvious
                // don't you just LOVE assembler comments?
```

Ok, that was fairly optimal... Let's look at the next case and see how it differs.

```
public class Faster {
    public void test() {
        boolean flag = true;
        flag = flag?false:true;
    }
}
```

The resultant bytecodes were:

```
Method void test()
  0 iconst_1    // push constant "true"
  1 istore_1    // store in location 1 (flag)
  2 iload_1     // load value in location 1
  3 ifne 10     // if value is true goto bytecode 10
  6 iconst_0    // push constant "false"
  7 goto 11     // goto location 11
 10 iconst_1    // push constant "true"
 11 istore_1    // store value on stack in location 1
 12 return      //
```

Identical? Yep, pretty much identical. The only difference is in one case we are testing for "equal" and in the other we are testing for "not equal". Surely that could not make such a big difference? (I'll leave the decompiling and understanding of the XORoman way as an exercise to the reader ;-)

```
public class Fastest {
    public void test() {
        boolean flag = true;
        flag ^= true;
    }
}
```

What happened? I have to assume that some part of the hotspot kicked in after some iterations and that the second example was only faster because it was second, so I ran the examples longer:

```
public class Not {
    public static void test() {
        boolean flag = true;
        long start;

        start = -System.currentTimeMillis();
        for (int i=0; i<1000000000; i++) {
            flag ^= true;
        }
        start += System.currentTimeMillis();
        System.out.println("time for flag ^= true: " + start + "ms");

        start = -System.currentTimeMillis();
        for (int i=0; i<1000000000; i++) {
            flag = !flag;
        }
        start += System.currentTimeMillis();
        System.out.println("time for flag = !flag: " + start + "ms");

        start = -System.currentTimeMillis();
        for (int i=0; i<1000000000; i++) {
            flag = flag?false:true;
        }
        start += System.currentTimeMillis();
        System.out.println("time for flag = flag?false:true: " + start + "ms");
    }
}
```

```

    }
    public static void main(String[] args) throws Exception {
        test();
        Thread.sleep(1);
        test();
    }
}

```

Letting it run longer certainly shows more truth:

```

time for flag ^= true: 12397ms
time for flag = !flag: 11356ms
time for flag = flag?false:true: 11326ms
time for flag ^= true: 5697ms
time for flag = !flag: 11326ms
time for flag = flag?false:true: 11326ms

```

We can learn two lessons from this:

1. `flag ^= true` is faster than `flag = !flag`
2. Never trust Java performance statistics.

Don't forget that an intelligent compiler could've recognised what you were doing and done it on a bit level. There are many factors that affect Java performance: architecture, compiler, hotspot compiler, hardware, etc. and these all play a role when it comes to determining performance.

That's all for tonight - even the mosquitos are asleep already so I better sign off.

Heinz

P.S. If you write to me, please feel free to address me as "Heinz" - we are casually formal in South Africa ;-)

P.P.S. Roman Porotnikov's ICQ number is 76669875 and he has an interesting webpage at <http://ejb.how.to>

Issue 025 - Final Newsletter

Author: Dr. Heinz M. Kabutz

Date: 2001-07-12

Category: Language

Java Versions:

Abstract:

Welcome to the 25th issue of **The Java(tm) Specialists' Newsletter**. I hope that for at least *some* of you, your heart sank when you saw the title of this newsletter. No, your mailbox is not getting lighter, I just thought I'd write a bit about how I use the "final" keyword.

Incidentally, on Monday we broke through the 1000th reader barrier (on an upward trend) so thanks to all of you who promoted this newsletter and sent it to friends and colleagues.

Please remember to forward this newsletter to as many Java enthusiasts as you know who might be interested in receiving such a newsletter.

The last few newsletters were quite heavy, so this week I would like to look at style, specifically on uses of the "final" keyword. It is not a newsletter on how to abuse the "final" keyword, which might surprise some of the more loyal readers of this newsletter.

I would like to thank Carl Smotricz from Germany for hosting an archive of my newsletters at <http://www.smotricz.com/kabutz>. Please let me know if you would like to include an archive on your website. I am currently in discussion with a designer to put together a website for my company, which will include the newsletters and information about the type of work my company does.

Final

The keyword "final" in Java is used in different ways depending on the context. We can have final methods, final classes, final data members, final local variables and final parameters. A final class implicitly has all the methods as final, but not necessarily the data members. A final class may not be extended, neither may a final method be overridden.

Final primitive data members cannot be changed once they are assigned, neither may final object handle data members (Vector, String, JFrame, etc.) be reassigned to new instances, but if they are mutable (meaning they've got methods that allow us to change their state), their contents may be changed. Since String is immutable, once a handle to it is final, we *could* consider it as a constant, if we ignore the effects of newsletter 14. So how do we use this construct in the real world?

Final Methods

I personally try to avoid making a method final, unless there is a very good reason for it to be final. There are typically two reasons to make a method final, performance and design. Let's look at performance first:

When a method is final, it may be inlined. Before HotSpot compiling (JDK 1.1.x), these methods were usually inlined at compile time, whereas with HotSpot they are inlined at runtime, unless the compiler can guarantee that the inlined method will always be compiled together with the code that uses it.

```
// somebody else's class
public class A {
    public static final void f() {
        System.out.println("A's f()");
    }
}

// our class
public class B {
    public void g() {
        A.f();
    }
}
```

In the past, at compile time our class would be turned into:

```
// compiled class
public class B {
    public void g() {
        System.out.println("A's f()");
    }
}
```

The effect of this was that we had to make one less method call, and since method calls produce extra overhead, we saved some clock cycles. The disadvantage of this, made clear to me by an old (ok, experienced) COBOL programmer during one of my courses, was that whenever somebody else's class changed we would have to remember to recompile our class!

In JDK 1.[234].x with HotSpot(tm), Sun changed this so that the methods were no longer

inlined at compile time, but rather by the HotSpot compiler at run time, IFF the performance measurements suggested that it would improve the overall performance of our code.

There are quite a few factors which will affect whether a method will be inlined or not, and we cannot assume that just because we make something final that it will definitely be inlined. As we saw in newsletter 21, it is a good idea anyway to always recompile all your code when you get a new version of someone else's library, so this is not necessarily a reason to NOT use final methods.

When you make a method final, no-one else will be able to override it again. You thus limit extensibility of your code by choosing to make the method or, even worse, your class final. I have been utterly frustrated in the past when I wanted to extend code where the developer had tried to add optimizations in the form of final. I thus never make a method or class final unless I specifically want to stop others from overriding them.

So, when do I use final methods? If I have performance values that prove that final makes a difference then I would consider using it for performance reasons, otherwise I would only ever use it for design reasons.

This is all old hat for you I'm sure, so let's look at final data members:

Final data members

One of the difficulties in programming is coming up with good names for "things". (there, that just proves my point, doesn't it?) I remember an experienced (ok, old) C programmer who was programming in Java and decided to use very long names for everything, for example:

```
public class SessionConnectorWithRetryAtLeastThreeTimes {
    private String connectionNameReceivedFromInternet;
    private int numberOfTimesThatWeShouldRetryAtLeast;
}
```

Alright, I'm exaggerating a little bit, but I hope you get the idea. The beauty of good names is that comments become very easy to write, sometimes even partly redundant. In Java, we can then write a constructor that takes the state for the object and assigns the correct data members. For example:

```
public class SessionConnectorWithRetryAtLeastThreeTimes {
```

```

private String connectionNameReceivedFromInternet;
private int numberOfTimesThatWeShouldRetryAtLeast;
public SessionConnectorWithRetryAtLeastThreeTimes(
    String c, int n) {
    connectionNameReceivedFromInternet = c;
    numberOfTimesThatWeShouldRetryAtLeast = n;
}
}

```

The problem with our constructor is that we have to explain in our documentation what c and n represent. It would be much better to use the same names in the parameters of the constructor as we use for the data members, as it reduces the confusion. The standard way in Java of solving this problem is to use the same names for the parameters as we do for the data members and then to explicitly specify what we are referring to, using the "this" keyword.

```

public class SessionConnectorWithRetryAtLeastThreeTimes {
    private String connectionNameReceivedFromInternet;
    private int numberOfTimesThatWeShouldRetryAtLeast;
    public SessionConnectorWithRetryAtLeastThreeTimes(
        String connectionNameReceivedFromInternet,
        int numberOfTimesThatWeShouldRetryAtLeast) {
        this.connectionNameReceivedFromInternet =
            connectionNameReceivedFromInternet;
        this.numberOfTimesThatWeShouldRetryAtLeast =
            numberOfTimesThatWeShouldRetryAtLeast;
    }
}

```

The above code will compile and run, but not correctly. Take a few minutes to figure out what could possibly be wrong with it...

I hope you didn't find the mistake. The first parameter of the constructor is spelt differently to the data member, thanks to a simple spelling mistake. When we thus say

```
this.connectionNameReceivedFromInternet =
    connectionNameReceivedFromInternet;
```

both the names refer to the data member, so the data member will always null!

This is the reason why some companies try to persuade their staff to augment their data members with strange characters (m_ or _) to differentiate them from parameters, rather than use the "this" trick. I know of at least two companies where such coding standards are used. The effect is that either the data members look ugly, or the parameters look ugly.

A simple way of preventing such mistakes, besides learning to touch type 100% correctly, is to make the data members final, where possible. When we do that, the code below will no longer compile, and we can find our mistakes much easier.

```
public class SessionConnectorWithRetryAtLeastThreeTimes {
    private final String connectionNameReceivedFromInternet;
    private final int numberoftimesThatWeShouldRetryAtLeast;
    public SessionConnectorWithRetryAtLeastThreeTimes(
        String connectionNameReoeivedFromInternet,
        int numberoftimesThatWeShouldRetryAtLeast) {
        this.connectionNameReceivedFromInternet =
            connectionNameReceivedFromInternet;
        this.numberoftimesThatWeShouldRetryAtLeast =
            numberoftimesThatWeShouldRetryAtLeast;
    }
}
```

As a matter of habit, I make all data members final wherever that is possible. Mistakes that would have taken me days to find now pop out at the next compile.

Final local variables

There are two reasons I know for making a local variable or a parameter final. The first reason is that you don't want your code changing the local variable or parameter. It is considered by many to be bad style to change a parameter inside a method as it makes the

code unclear. As a habit, some programmers make all their parameters "final" to prevent themselves from changing them. I don't do that, since I find it makes my method signature a bit ugly.

The second reason comes in when we want to access a local variable or parameter from within an inner class. This is the actual reason, as far as I know, that final local variables and parameters were introduced into the Java language in JDK 1.1.

```
public class Access1 {
    public void f() {
        final int i = 3;
        Runnable runnable = new Runnable() {
            public void run() {
                System.out.println(i);
            }
        };
    }
}
```

Inside the run() method we can only access i if we make it final in the outer class. To understand the reasoning, we have to look at what the compiler does. It produces two files, Access1.class and Access1\$1.class. When we decompile them with JAD, we get:

```
public class Access1 {
    public Access1() {}
    public void f() {
        Access1$1 access1$1 = new Access1$1(this);
    }
}
```

and

```
class Access1$1 implements Runnable {
    Access1$1(Access1 access1) {
        this$0 = access1;
    }
}
```

```

public void run() {
    System.out.println(3);
}
private final Access1 this$0;
}

```

Since the value of i is final, the compiler can "inline" it into the inner class. It perturbed me that the local variables had to be final to be accessed by the inner class until I saw the above.

When the value of the local variable can change for different instances of the inner class, the compiler adds it as a data member of the inner class and lets it be initialised in the constructor. The underlying reason behind this is that Java does not have pointers, the way that C has.

Consider the following class:

```

public class Access2 {
    public void f() {
        for (int i=0; i<10; i++) {
            final int value = i;
            Runnable runnable = new Runnable() {
                public void run() {
                    System.out.println(value);
                }
            };
        }
    }
}

```

The problem here is that we have to make a new local data member each time we go through the for loop, so a thought I had today while coding, was to change the above code to the following:

```

public class Access3 {
    public void f() {
        Runnable[] runners = new Runnable[10];
        for (final int[] i={0}; i[0]<runners.length; i[0]++) {

```

```

runners[i[0]] = new Runnable() {
    private int counter = i[0];
    public void run() {
        System.out.println(counter);
    }
};

for (int i=0; i<runners.length; i++)
    runners[i].run();
}

public static void main(String[] args) {
    new Access3().f();
}
}

```

We now don't have to declare an additional final local variable. In fact, is it not perhaps true that `int[] i` is like a common C pointer to an `int`? It took me 4 years to see this, but I'd like to hear from you if you have heard this idea somewhere else.

I always appreciate any feedback, both positive and negative, so please keep sending your ideas and suggestions. Please also remember to take the time to send this newsletter to others who are interested in Java.

Heinz

Errata

In newsletter 23, I relied on hearsay for some information without checking it out myself. Yes, the amount of memory used by each Thread for its stack was incorrect. According to new measurements, it seems that each stack takes up approximately 20KB, not the 2MB stated in the original newsletter, so for 10000 threads we will need about 200MB. Most operating systems cannot handle that many threads very well anyway, so we want to avoid creating that many. Thanks to Josh Rehman for pointing out this mistake.

It seems that with JDK 1.1.8 each thread takes up 145KB, I'm not sure whether the stacks grow dynamically. Under the JDK 1.2.2 that comes with JBuilder 3, the stacks grow, so I managed to have 100 threads take up roughly 100 MB of memory, or 1MB each. After 1 MB is used up, the VM mysteriously returns without any message so I had to experiment a bit to get this information. Under JDK 1.3.0 I got DrWatsons when I tried to make the stack of each thread grow to any real size. It is unrealistic to expect our program to have stack depths of 10000 method calls, so we could probably quite safely use 50KB as a realistic stack size for each thread.

Issue 016 - Blocking Queue

Author: Dr. Heinz M. Kabutz

Date: 2001-04-11

Category: Performance

Java Versions:

Abstract:

Welcome to the 16th issue of **The Java(tm) Specialists' Newsletter**, written in a dreary-weathered-Germany. Since I'm a summer person, I really like living in South Africa where we have 9 months of summer and 3 months of sort-of-winter. It's quite difficult to explain to my 2-year old son the concepts of snow, snow-man, snow-ball, etc. Probably as difficult as explaining to a German child the concepts of cloudless-sky, beach, BSE-free meat, etc.

Next week I will again not be able to post the newsletter due to international travelling (Mauritius), but the week after that I will demonstrate how it is possible to write type-safe enum types in Java using inner classes and how it is possible to "switch" on their object references. Switch statements should never be used, but it is nevertheless fascinating to watch how the Java language constructs can be abused...

Blocking Queues for inter-thread communication

This week I want to speak about a very useful construct that we use for inter-thread communication, called a blocking queue. Quite often in threaded applications we have a producer-consumer situation where some threads want to pop jobs onto a queue, and some other worker threads want to remove jobs from the queue and then execute them. It is quite useful in such circumstances to write a queue which blocks on pop when there is nothing on the queue. Otherwise the consumers would have to poll, and polling is not very good because it wastes CPU cycles.

I have written a very simple version of the `BlockingQueue`, a more advanced version would include alarms that are generated when the queue reaches a certain length.

Warning Advanced:

When I write pieces of code which are synchronized, I usually avoid synchronizing on "this" or marking the whole method as synchronized. When you synchronize on "this" inside the class, it might happen that other code outside of your control also synchronize on the handle to your object, or worse, call notify on your handle. This would severely mess up your well-written BlockingQueue code. I therefore as a habit always use private data members as locks inside a class, in this case I use the private queue data member.

Another disadvantage of indiscriminately synchronizing on "this" is that it is very easy to then lock out parts of your class which do not necessarily have to be locked out from each other. For example, I might have a list of listeners in my BlockingQueue which are notified when the list gets too long. Adding and removing such listeners from the BlockingQueue should be synchronized, but you do not have to synchronize in respect of the push and pop operations, otherwise you limit concurrency.

```
//: BlockingQueue.java
import java.util.*;
public class BlockingQueue {
    /**
     * It makes logical sense to use a linked list for a FIFO queue,
     * although an ArrayList is usually more efficient for a short
     * queue (on most VMs).
     */
    private final LinkedList queue = new LinkedList();
    /**
     * This method pushes an object onto the end of the queue, and
     * then notifies one of the waiting threads.
     */
    public void push(Object o) {
        synchronized(queue) {
            queue.add(o);
            queue.notify();
        }
    }
    /**
     * The pop operation blocks until either an object is returned
     * or the thread is interrupted, in which case it throws an
     * InterruptedException.
     */
    public Object pop() throws InterruptedException {
        synchronized(queue) {
            while (queue.isEmpty()) {
                queue.wait();
            }
            return queue.removeFirst();
        }
    }
    /** Return the number of elements currently in the queue. */
    public int size() {
        return queue.size();
    }
}
```

Now we've got a nice little test case that uses the blocking queue for 10 worker threads which will each pull as many tasks as possible from the queue. To end the test, we put one poison pill onto the queue for each of the worker threads, which, when executed, interrupts the current thread (evil laughter).

```
//: BlockingQueueTest.java
public class BlockingQueueTest {
    private final BlockingQueue bq = new BlockingQueue();
    /**
     * The Worker thread is not very robust. If a RuntimeException
     * occurs in the run method, the thread will stop.
     */
    private class Worker extends Thread {
        public Worker(String name) { super(name); start(); }
        public void run() {
            try {
                while(!isInterrupted()) {
                    ((Runnable)bq.pop()).run();
                }
            } catch(InterruptedException ex) {}
            System.out.println(getName() + " finished");
        }
    }
    public BlockingQueueTest() {
        // We create 10 threads as workers
        Thread[] workers = new Thread[10];
        for (int i=0; i<workers.length; i++)
            workers[i] = new Worker("Worker Thread " + i);
        // We then push 100 commands onto the queue
        for (int i=0; i<100; i++) {
            final String msg = "Task " + i + " completed";
            bq.push(new Runnable() {
                public void run() {
                    System.out.println(msg);
                    // Sleep a random amount of time, up to 1 second
                    try { Thread.sleep((long)(Math.random()*1000)); } catch(InterruptedException ex) {} }
            });
        }
        // We then push one "poison pill" onto the queue for each
        // worker thread, which will only be processed once the other
        // tasks are completed.
        for (int i=0; i<workers.length; i++) {
            bq.push(new Runnable() {
                public void run() {
```

```
        Thread.currentThread().interrupt();
    }
})
}
// Lastly we join ourself to each of the Worker threads, so
// that we only continue once all the worker threads are
// finished.
for (int i=0; i<workers.length; i++) {
    try {
        workers[i].join();
    } catch(InterruptedException ex) {}
}
System.out.println("BlockingQueueTest finished");
}
public static void main(String[] args) throws Exception{
    new BlockingQueueTest();
}
}
```

The concepts in the newsletter can be expanded quite a bit. They could, for example, be used as a basis for implementing a ThreadPool, or otherwise you can implement an "ActiveQueue" which performs callbacks to listeners each time an event is pushed onto the queue via a Thread running inside the ActiveQueue.

It is also possible to use PipedInputStream and PipedOutputStream to send messages between threads, but then you have to set up a whole protocol, and if you want to exchange objects you have to use ObjectOutputStream which will be alot slower than just passing handles.

Until next week, and please remember to forward this newsletter in its entirety to as many Java users as you know.

Heinz

Issue 015 - Implementing a SoftReference based HashMap

Author: Dr. Heinz M. Kabutz

Date: 2001-03-28

Category: Performance

Java Versions:

Abstract:

Welcome to the 15th issue of **The Java(tm) Specialists' Newsletter**, this week we are looking at an extremely useful addition to the Java language, namely soft and weak references. In a nutshell, the main difference between the two seems to be that the SoftReference has some notion of remembering when last it was accessed, which may be used by the GC (if it feels like it) to release little-used SoftReferences first.

Next week I will be doing some training in Germany, so I will not be able to send out a newsletter, as I have not decided on a list server yet. Hopefully the week after that you'll have your normal fix again. Please continue forwarding this newsletter to your local JUG, friends and family who are interested in Java. If you do send the newsletter to a closed user list, kindly tell me, I like having an indication of how many souls are reading my newsletter. Similarly, if you quote from my newsletter, be so kind as to attribute the source.

Many thanks to Sydney Redelinghuys for his input and corrections in this newsletter.

Once again I have seen the value in actually testing my code, as there was a serious error in my code which I only found through my "unit" test.

Implementing a SoftReference based HashMap

Java is slow. Java is a memory hog.

But, if I have to write a network application, I would not hesitate for a second to use Java. Why? Because Java shines in threaded, network-based applications and because over the years, I have recognised the weaknesses of Java and have learnt (the hard way) what I should do, or not do, to avoid running into too serious problems. Recognising the problems takes a while to learn, which is why most Java jobs require you to have at least 2 years of Java programming experience.

One of the most common places of memory wastage is in hash maps, so SUN have provided a WeakHashMap to minimize memory usage in caches implemented using maps. A

WeakHashMap stores the keys using WeakReference objects, which means in layman's terms that as soon as the key is not referenced from somewhere else in your program, the entry may be removed and is available for garbage collection. (Have a look at the JavaDocs for the java.util.WeakHashMap and java.lang.ref.WeakReference for more information) Say you write a middle tier that provides an OO view of your database via an application server. What you really want is for the values to be automatically released, rather than the keys. If you put all the objects into a normal HashMap, you can easily run out of memory when you access many different objects from different clients. But if you store the objects in a WeakHashMap they are cleared as soon as your clients is not referencing them anymore. What you really want, however, is to only have the objects released when the VM is running low on memory, since that is when you get problems.

Enter SoftReferences. As far as I understand, a SoftReference will only be garbage collected when the VM is running low on memory and the object it is pointing to is not accessed from a normal (hard) reference. This is probably a better option to use for the HashMap values than a WeakReference, but the default JDK collections don't include a GC-friendly HashMap based on values and neither does it provide a SoftReference based HashMap.

Before I show you a SoftHashMap implementation, based on ideas by Sydney (what's up doc?) Redelinghuys, I need to explain some ideas which will make our SoftHashMap more optimal.

1. Each time we change the map (put, remove, clear) or ask for its size, we first want to go through the map and throw away entries for which the values have been garbage collected. It is quite easy to find out which soft references have been cleared. You can give the SoftReference a ReferenceQueue to which it is added when it is garbage collected.
2. We don't want our hash map to be bullied by the garbage collector, so we provide the option of the map itself keeping a hard link to the last couple of objects (typically 100).
3. The SoftHashMap will use a variant of the Decorator pattern to add this functionality to an internally kept java.util.HashMap. I'm busy working on a Design Patterns course based on the GOF book, let me know if you want further information.

Without further ado, here comes the SoftHashMap:

```
//: SoftHashMap.java
import java.util.*;
import java.lang.ref.*;

public class SoftHashMap extends AbstractMap {
    /** The internal HashMap that will hold the SoftReference. */
    private final Map hash = new HashMap();
    /** The number of "hard" references to hold internally. */
}
```

```

private final int HARD_SIZE;
/** The FIFO list of hard references, order of last access. */
private final LinkedList hardCache = new LinkedList();
/** Reference queue for cleared SoftReference objects. */
private final ReferenceQueue queue = new ReferenceQueue();

public SoftHashMap() { this(100); }
public SoftHashMap(int hardSize) { HARD_SIZE = hardSize; }

public Object get(Object key) {
    Object result = null;
    // We get the SoftReference represented by that key
    SoftReference soft_ref = (SoftReference)hash.get(key);
    if (soft_ref != null) {
        // From the SoftReference we get the value, which can be
        // null if it was not in the map, or it was removed in
        // the processQueue() method defined below
        result = soft_ref.get();
        if (result == null) {
            // If the value has been garbage collected, remove the
            // entry from the HashMap.
            hash.remove(key);
        } else {
            // We now add this object to the beginning of the hard
            // reference queue. One reference can occur more than
            // once, because lookups of the FIFO queue are slow, so
            // we don't want to search through it each time to remove
            // duplicates.
            hardCache.addFirst(result);
            if (hardCache.size() > HARD_SIZE) {
                // Remove the last entry if list longer than HARD_SIZE
                hardCache.removeLast();
            }
        }
    }
    return result;
}

/** We define our own subclass of SoftReference which contains
not only the value but also the key to make it easier to find
the entry in the HashMap after it's been garbage collected. */
private static class SoftValue extends SoftReference {
    private final Object key; // always make data member final
    /** Did you know that an outer class can access private data
     members and methods of an inner class? I didn't know that!
     I thought it was only the inner class who could access the
     outer class's private information. An outer class can also
     access private members of an inner class inside its inner
}

```

```

    class. */
private SoftValue(Object k, Object key, ReferenceQueue q) {
    super(k, q);
    this.key = key;
}

/** Here we go through the ReferenceQueue and remove garbage
collected SoftValue objects from the HashMap by looking them
up using the SoftValue.key data member. */
private void processQueue() {
    SoftValue sv;
    while ((sv = (SoftValue)queue.poll()) != null) {
        hash.remove(sv.key); // we can access private data!
    }
}
/** Here we put the key, value pair into the HashMap using
a SoftValue object. */
public Object put(Object key, Object value) {
    processQueue(); // throw out garbage collected values first
    return hash.put(key, new SoftValue(value, key, queue));
}
public Object remove(Object key) {
    processQueue(); // throw out garbage collected values first
    return hash.remove(key);
}
public void clear() {
    hardCache.clear();
    processQueue(); // throw out garbage collected values
    hash.clear();
}
public int size() {
    processQueue(); // throw out garbage collected values first
    return hash.size();
}
public Set entrySet() {
    // no, no, you may NOT do that!!! GRRR
    throw new UnsupportedOperationException();
}
}

```

And here comes some test code that demonstrates to a certain degree that I'm not talking complete nonsense. Soft and weak references are quite difficult to experiment with as there is a lot of freedom left to the writers of the JVM of how they must implement them. I wish the implementation would hold back longer before removing these references, that the JVM would wait until it is really running low on memory before clearing them, but unfortunately I am not the one who wrote the JVM. I have tried to question the authors of the java.lang.ref

package to find out what the strategy is for references in future versions, but have not had any response yet.

```
//: SoftHashMapTest.java
import java.lang.ref.*;
import java.util.*;

public class SoftHashMapTest {
    private static void print(Map map) {
        System.out.println("One=" + map.get("One"));
        System.out.println("Two=" + map.get("Two"));
        System.out.println("Three=" + map.get("Three"));
        System.out.println("Four=" + map.get("Four"));
        System.out.println("Five=" + map.get("Five"));
    }
    private static void testMap(Map map) {
        System.out.println("Testing " + map.getClass());
        map.put("One", new Integer(1));
        map.put("Two", new Integer(2));
        map.put("Three", new Integer(3));
        map.put("Four", new Integer(4));
        map.put("Five", new Integer(5));
        print(map);
        byte[] block = new byte[10*1024*1024]; // 10 MB
        print(map);
    }
    public static void main(String[] args) {
        testMap(new HashMap());
        testMap(new SoftHashMap(2));
    }
}
```

Until next week, and please remember to forward this newsletter and send me your comments.

Heinz

Issue 001 - Deadlocks

Author: Dr. Heinz M. Kabutz

Date: 2000-11-30

Category: Performance

Java Versions:

Abstract:

Java deadlocks can be tricky to discover. In this very first Java Specialists' Newsletter, we look at how to diagnose such an error and some techniques for solving it.

Welcome to **The Java(tm) Specialists' Newsletter**, a low-volume newsletter exploring experiences of using Java "in the field" (not only `java.lang.reflect.Field`, but that also). The reason I have sent this email to you is because you were either on a Java course presented by me, or you are a colleague, or I thought the topic might simply interest you.

Just to clear things up, as I always tell my students, my company's name is Maximum Solutions and our logo is **The Java(tm) Specialists' Newsletter**, hence the title for the newsletter. This does not mean we are good at Java, it simply means that this is all we do (at the moment). Perhaps in future I will change my logo to "The XML Specialists", or, heaven forbid "The VB Specialists", but then I would have to be really hard-up ;-) The logo is therefore not a display of my arrogance (those of you who know me better would assure you that arrogance is definitely not one of my characteristics, neither is a bad haircut nor rolled up jeans) but is supposed to give me focus in the IT industry.

A warning I feel I have to give you is that the tricks I have discovered in Java are not always "kosher" and are used solely at your own risk. Whenever I mention something that is non-kosher I will also attach a warning label so the more experienced of you can use it and the rest not. Always make sure that your unit tests run before and after a change to a new JDK version, the days of JDK 1.1.x were extremely interesting because each version screwed up the South Africa Locale in a different way. For the past 3.5 years I've been programming exclusively in Java and I can say that it is a fascinating language. Once you dig below the surface of Java it becomes increasingly more fun and interesting to work with, and you end up wanting to exploit the limits of the JDK (last week I made a piece of code I had written with someone else 2 years ago 1000x faster). The JDK 1.3.0 source code contained in the `src.jar` file when you install the JDK contains a lot of Java code, 574253 lines of Java code to be exact, which is slightly less than the Java program written in Stellenbosch in South Africa where I gained most of my Java programming experience.

Deadlocks in Java

While travelling overseas I had the privilege of spending some time helping a Java program in dire straits. It was a graphical interface that was supposed to emulate a C++ program running on a server, bla bla bla. Anyway, this application would occasionally have a thread

deadlock with the result that the screen would stop refreshing. Imagine the despair when facing an unfamiliar source base of several 10'000 lines of Java code with a deadlock SOMEWHERE in there?!? The code was well written, but even though, the task was scary. Luckily I had heard somewhere about the secret CTRL+BREAK trick that SUN smuggled into the JDK without telling anybody. If you press CTRL+BREAK while your Java program is running in a command prompt you get a stack trace of exactly what each thread is doing and by looking at the code you easily find which thread is waiting for which lock!

When you get a deadlock in your program you want to be able to reproduce it reliably as quickly as possible, so we got half the team to just concentrate on finding deadlocks that they could reproduce. In the meantime, the other half would look at the stack traces and figure out why the code was deadlocking. In the process of looking we discovered something which I had heard about second-hand but had not encountered myself. Last week I heard of the same problem occurring at another company which took a lot of effort to clear up, so if your program has any GUI in it, especially Swing, it might pay to listen to this:

In Swing, all GUI components have to be changed from within the Swing thread.

This means that you cannot execute `jLabel1.setText("blabla")` from within any thread besides the Swing thread.

If you have change any GUI from another thread you should rather say:

```
SwingUtilities.invokeLater(new Runnable() {
    public void run() {
        jLabel1.setText("blabla");
    }
})
```

This will do the change that you requested as soon as it can, which is usually within a few microseconds. There is another call `invokeAndWait` which I have hardly ever used except in race-conditions.

I was under the impression that failure to use `invokeLater` would cause some refresh problems of a kind where some parts of the screen get refreshed and others not, I did not realise it could cause a thread deadlock (or maybe that is what everyone was talking about - refresh problem - hmpf - more like a system meltdown). Luckily you don't always have to call `invokeLater` whenever you change a GUI component because in some cases you already are in the Swing GUI thread, for example if a button is pressed then the `ActionListener.actionPerformed` method will be called from the Swing thread.

However, if you provide a callback class to another class that is not part of the AWT/Swing group you will never know what thread it is coming from, so it is safest to invokeLater.

Remember that any work you do in invokeLater is going to hold up the Swing thread from repainting the window, so please don't do big database queries inside invokeLater but rather only call invokeLater for the parts of the code that are genuinely graphics related. It might pay off to bunch all the GUI related lines in your method by refactoring your code a bit. For information on refactoring look at the book with that title by Martin Fowler.

Warning Advanced:

A small optimisation is to have a class that figures out if the current thread is the Swing thread (`SwingUtilities.isEventDispatchThread()`) and if it is not calls `SwingUtilities.invokeLater(Runnable)`, otherwise it calls the runnable code directly. This way you can finish everything you need to do without interference by other threads. Reason for the Non-Kosher label is that it seems you can have several event dispatch threads, I don't know under what circumstances so if you know, please enlighten me. However, it almost seems like only one of them can be active at any one time.

This is the end of my first newsletter, I appreciate any feedback!

Kind regards

Heinz