Visibility

Publication and escape

Thread confinement

Immutable and final

Safe publication

Memory visibility is visibility by other threads.

- *Memory Visibility:* Make sure that other threads can see the changes made by this thread.
- If one thread modifies the *state of an object* and another thread reads that state, there is no guarantee that the latter will see (and thus can use) the changes in that object! (Unsafe publication.)
- *Safe publishing* can be obtained through synchronization:
  - by explicitly using synchronization
  - by taking advantage of synchronzation in library classes
- See what can go wrong in NoVisibility.java example on next slide

NoVisibility code example

```
public class NoVisibility {
    private static boolean ready;
    private static int number;

    private static class ReaderThread extends Thread {
        public void run() {
            while (!ready)
                Thread.yield();
            System.out.println(number);
        }
    }

    public static void main(String[] args) {
        new ReaderThread().start();
        number = 42;
        ready = true;
    }
}
```

Don't do this!

Stale data

- *Stale data:* an out of date data element. Use synchronization for every access of a variable.
- Access(!), that is setters and **getters**!
- Data might be stale, but was correct at some time: out-of-thin-air safety.
- But even that does not apply to 64 bit numeric primitive variables (`double` and `long`).
  - Because a 64 bit read(or write) operation consists of two 32 bit read(or write) operations.
  - So **always** declare such shared 64 bit primitive as `volatile` or guard them by a lock.

Mutable Integer holder: code example

```
@NotThreadSafe
public class MutableInteger {
    private int value;

    public int get() { return value; }

    public void set(int value) { this.value = value; }
}
```

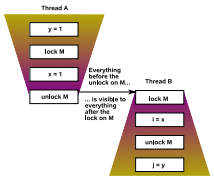
```
@ThreadSafe
public class SynchronizedInteger {
   @GuardedBy("this")
   private int value;

   public synchronized int get(){return value;}

   public synchronized void set(int value){this.value = value;}
}
```


Locking affects visibility

Locking is not only mutual exclusion but also affects *memory visibility*!

Locking and unlocking ensures memory visibility.

Synchronization guarantees visibility



- **A** synchronized block is guarded by a lock on object **M**. Everything done by thread A before the unlock on **M** is visible to thread **B** after the lock on **M**.
- *Without synchronization there is no such guarantee.*

Volatile

- *Volatile variable*s are a weaker form of synchronization.
- Declaring a field `volatile` assure that compiler and runtime don't mix the operations on that variable with other memory operations.
- No caching is allowed in this case, so a *read* always returns the most recent value.

```
public class CountingSheep {
    volatile boolean asleep;

    void tryToSleep() {
        while (!asleep)
            countSomeSheep();
    }

    void countSomeSheep() {
        // One, two, three...
    }
}
```


Visibility

Proper use of volatile

- Use `volatile` variables only when they simplify implementing and verifying your synchronisation policy
- Avoid volatile (instead of locked accessed) variables when verifying correctness would require substantial reasoning about visibility
- Good use of volatile variables includes ensuring visibility of their own state, that of the object they refer to, or indicating that an important lifecycle event (such as initialization or shutdown) has occurred

Visibility

- Locking can guarantee both visibility and atomicity; volatile variables can only guaranteee visibility.
- You **can** use volatile if the following criteria are met:
  - Writes to the variable do NOT depend on its current value (like in a ++ operation) **OR**
  - you can ensure that only a single thread ever writes the value;
  - The variable does not participate in invariants with the other state variables; and
  - Locking is not required for any other reason while the variable is being accessed (read or written).

public provides an escape

Publishing is a door to escape

`public` allows escape routes.
Publishing an object means making it available outside its current scope. Publishing sounds like an activity (it is a verb), but in fact the `public` keyword on a member is sufficient to name that something published.

```
    public static Set<Secret> knownSecrets;

    public void initialize() {
        knownSecrets = new HashSet<Secret>();
    }
```


This is how *Papillon* got out. . .

And do not do something like this, because you do not know what the caller will do to your private parts.

```
class UnsafeStates {
    private String[] states = new String[]{
        "AK", "AL" /*...*/
    };

    public String[] getStates() {
        return states;
    }
}
```

Indecent exposure

Do not expose yourselves until fully dressed. In objects that is: fully constructed. (Design pattern hint: factories.)

- During construction an object is fragile, because its invariants may not yet hold.
- Its like putting an unborn onto the streets.

Do not allow the this reference to escape *during* construction.

It is not (yet) safe out there!

Often made mistake with listeners

Don't do this in a constructor!

```
public class ThisEscape {
    public ThisEscape(EventSource source) {
        source.registerListener(new EventListener() {
            public void onEvent(Event e) {
                doSomething(e);
            }
        });
    }
```

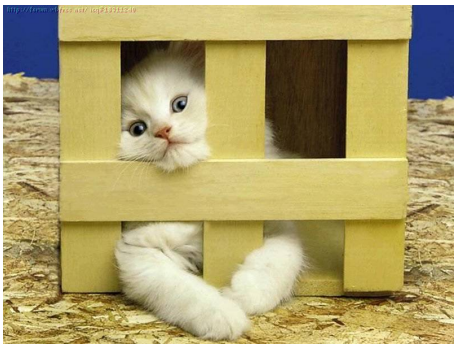
Safe listener attachment

Maybe you should consider to make more constructors private.

⇒ **Factory Method** design pattern.

```
public class SafeListener {
    private final EventListener listener;

    private SafeListener() {
        listener = new EventListener() {
            public void onEvent(Event e) {
                doSomething(e);
            }
        };
    }

    public static SafeListener newInstance(EventSource source) {
        SafeListener safe = new SafeListener();
        source.registerListener(safe.listener);
        return safe;
    }
```


Do no let the beasties out



Ad-hoc thread confinement is not a technique

But rather a lack of the use of that (like language constructs etc).

- Ad hoc thread confinement is often not a confinement but rather a agreement or a usage convention (hopefully properly documented in the API documentation).
- It is allowable for **single threaded** *sub*systems and often the "technique" of choice for GUI frameworks for performance reasons.
- If you can ensure the ***writing*** to a volatile member only takes place from **one** thread, than that is safe, since you confined the *modification* to one thread.
- Use ad hoc thread confinement sparingly because of its fragility.

Method local variables exist only on the stack

The runtime stack is by definition thread-local.

```
public int loadTheArk(Collection<Animal> candidates) {
    SortedSet<Animal> animals;
    int numPairs = 0;
    Animal candidate = null;

    // animals confined to method, don't let them escape!
    animals = new TreeSet<Animal>(new SpeciesGenderComparator());
    animals.addAll(candidates);
    for (Animal a : animals) {
        if (candidate == null || !candidate.isPotentialMate(a))
            candidate = a;
        else {
            ark.load(new AnimalPair(candidate, a));
            ++numPairs;
            candidate = null;
        }
    }
    return numPairs;
}
```


Stack confinement

- Stack confinement is easily achieved by using method local variables. These are not allocated on the heap but on the stack.
- Since there is one stack per thread, the local variables are automatically confined to that stack and thread.
- For primitive types (`int` `boolean`, etc) this confinment cannot be broken.
- Beware of Objects, for they **are** allocated on the heap and only the reference will be stored on the stack.
- So do not hand out (return, pass it to alien methods) that reference, or escape still is possible.
- If you stick to these rules confining a nonthreadsafe object to the stack **is** threadsafe.

Use of java.lang.ThreadLocal

The java.lang.ThreadLocal class binds its member to one thread. You could imagine it as a hashmap with the thread as key and member as value.

```
private static ThreadLocal<Connection> connectionHolder
        = new ThreadLocal<Connection>() {
            public Connection initialValue() {
                return DriverManager.getConnection(DB_URL);
            };
        };
public Connection getConnection() {
    return connectionHolder.get();
}
```

And as long as you take care those ThreadLocal members do not escape, these members **don't** have to be threadsafe. E.g. the JDBC specification does not require the `Connection` objects to be threadsafe. See the book for a solution.

Use of ThreadLocal II

- If you port a single threaded application to a multi threaded environment, you can use ThreadLocal to preserve thread safety by converting globals (like Singletons) into ThreadLocals, semantics permitting.
- ThreadLocal is often used in frameworks, like **J(2)EE** for instance.
- Note that over-use of ThreadLocal can detract from reusability and introduce hidden couplings.

Immutable is threadsafe

Immutable objects are always threadsafe

An object is immutable if:

- Its state cannot be modified after construction;
- all its fields are final; and
- It is *properly constructed* (this did not escape during construction.)

The 3 Stooges, once famous

The middle one is named "Curly" of course.


```java
@Immutable
public final class ThreeStooges {
    private final Set<String> stooges = new HashSet<String>();

    public ThreeStooges() {
        stooges.add("Moe");
        stooges.add("Larry");
        stooges.add("Curly");
    }

    public boolean isStooge(String name) {
        return stooges.contains(name);
    }

    public String getStoogeNames() {
        List<String> stoogesL = new Vector<String>();
        stoogesL.add("Moe");
        stoogesL.add("Larry");
        stoogesL.add("Curly");
        return stoogesL.toString();
    }
}
```


Final as a good practice

Just as it is a good practice to make all fields `private` unless they *need* greater visibility, it is good practice to make all fields final unless they *need* to mutable.

- Even if an object is mutable, making some fields immutable simplifies the reasoning about threadsafety, because the number of possible states is reduced.
- It also documents to the **maintainers** of the class that the fields are not changed.

Using `volatile` to publish immutable object, Cache

```
@Immutable
public class OneValueCache {
    private final BigInteger lastNumber;
    private final BigInteger[] lastFactors;

    public OneValueCache(BigInteger i,
                         BigInteger[] factors) {
        lastNumber = i;
        lastFactors = Arrays.copyOf(factors, factors.length);
    }

    public BigInteger[] getFactors(BigInteger i) {
        if (lastNumber == null || !lastNumber.equals(i))
            return null;
        else
            return Arrays.copyOf(lastFactors, lastFactors.length);
    }
```

Usage:

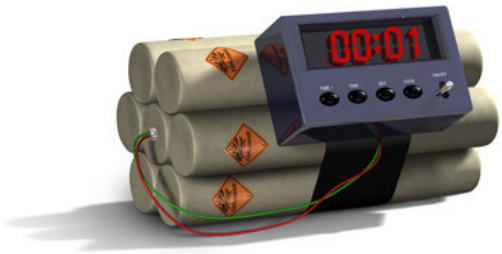
```
    private volatile OneValueCache  cache = new OneValueCache(BigInteger{1},
                                    new BigInteger[]{new BigInteger(1)});
    public void set(BigInteger i, BigInteger[] factors){
        cache = new OneValueCache(i, factors);
    }
```


Use `volatile` to publish

Publish the immutable cache object through a `volatile` reference:

```
@ThreadSafe
public class VolatileCachedFactorizer extends GenericServlet
   implements Servlet {
    private volatile OneValueCache cache =
        new OneValueCache(null, null);

    public void service(ServletRequest req, ServletResponse resp) {
        BigInteger i = extractFromRequest(req);
        BigInteger[] factors = cache.getFactors(i);
        if (factors == null) {
            factors = factor(i);
            cache = new OneValueCache(i, factors);
        }
        encodeIntoResponse(resp, factors);
    }
```


You are not supposed to be a code terrorist



- Of course sometimes you will have to share information.
- Cooperation is in some way the essence of a multithreaded application.
- But this publication should be done safely.

Unsafe publication:

```
public Holder holder;

public void initialize() {
    holder = new Holder(42);
}
```

- This kind of improper publication could allow another thread to observe a partially constructed object.

Failure risk if not properly published

```
public class Holder {
    private int n;

    public Holder(int n) {
        this.n = n;
    }

    public void assertSanity() {
        if (n != n)
            throw new AssertionError("This statement is false.");
    }
}
```

Safe publication

Immutable objects can be safely published

Immutable objects can be used safely by any thread without additional synchronization, even when synchronization is **not** used to publish them, since any copy in memory or cache will have the same value.

Four ways to publicize safely

It is an idiom (how do I say this in Java)

To publish an object safely, **both** *reference to* **and** *object state* must be made visible at the same time to other threads. A **properly constructed** object can be safely published by either:

1. Initializing an object reference from a `static` **`initializerMethod()`**;
2. Storing a reference to it into a volatile field or `AtomicReference`;
3. Storing a reference to it into a `final` field of a **properly constructed object**; or
4. Storing a reference to it into a field that is properly guarded by a **lock**.

Effectively immutable is immutable by use.

Objects thate ar not **technically** immutable, but whose state is not modified after publication, are called **effectively immutable**.

Effectively immutable can be safe (enough)

Safely published *effectively immutable* objects can be used safely by any thread without additional synchronization.

Example: Date is mutable[1]. If your application uses this in way that a stored date is not changed anymore, the synchronisation of the collection is sufficient:

```
public Map<Srting, Date> lastLogin =
    Collections.synchronizedMap(new HashMap<String, Date>());
```

[1] probably a class-library design mistake

Ensure correct visibility of changes.

With mutable objects proper publication only ensures the correct visibility of the as-Published state. Synchronisation must also be used for every access, to ensure proper visibility of following modifications.

The publication requirements depend on its mutability

- Immutable objects can be published through any mechanism.
- Effectively immutable objects must be safely published.
- Mutable objects must be safely published and must be either threadsafe themselves or guarded by a lock.