

Visibility
Publication and escape
Thread confinement
Immutable and final
Safe publication

Sharing Objects

Ferd van Odenhoven
Pieter van den Hombergh

Fontys Hogeschool voor Techniek en Bedrijfsmanagement

26 november 2009



Visibility
Publication and escape
Thread confinement
Immutable and final
Safe publication

- 1 Visibility
- 2 Publication and escape
- 3 Thread confinement
- 4 Immutable and final
- 5 Safe publication



<p>Visibility Publication and escape Thread confinement Immutable and final Safe publication</p>	<p>Memory visibility state data locking and visibility</p>
--	--

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 synchronization in library classes
- See what can go wrong in NoVisibility.java example on next slide



Visibility Publication and escape Thread confinement Immutable and final Safe publication	Memory visibility stale data locking and visibility
--	--

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!



FvO,PvdH/FHTBM

Sharing Objects

26 november 2009

4/34

Visibility Publication and escape Thread confinement Immutable and final Safe publication	Memory visibility stale data locking and visibility
--	--

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- *Stale data*: an out of date data element.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 it is correct data: *out-of-thin-air safety*.
- This does not apply to 64 bit numeric variables (**double** and **long**).
- Because a 64 bit read(or write) operation consists of two 32 bit read(or write) operations.
- So declare these as **volatile** or guard them by a lock.



FvO,PvdH/FHTBM

Sharing Objects

26 november 2009

5/34

Visibility Publication and escape Thread confinement Immutable and final Safe publication	Memory visibility stale data locking and visibility
--	--

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;}
}
```



FvO,PvdH/FHTBM

Sharing Objects

26 november 2009

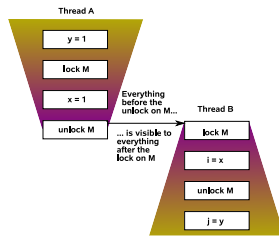
6/34

Visibility
 Publication and escape
 Thread confinement
 Immutable and final
 Safe publication

Memory visibility
 stale data
locking and visibility

Synchronization guarantees visibility

Locking is not only mutual exclusion but also affects *memory visibility!*
 Locking and unlocking ensures memory 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.*



Visibility
 Publication and escape
 Thread confinement
 Immutable and final
 Safe publication

Memory visibility
 stale data
locking and visibility

Volatile

- *Volatile variables* 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
 Publication and escape
 Thread confinement
 Immutable and final
 Safe publication

Memory visibility
 stale data
locking and visibility

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
 Publication and escape
 Thread confinement
 Immutable and final
 Safe publication

Memory visibility
 stale data
locking and visibility

Visibility

- 
- Locking can guarantee both visibility and atomicity; volatile variables can only guarantee 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 insure 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).



FvO,PvdH/FHTBM

Sharing Objects

26 november 2009

10/34

Visibility
 Publication and escape
 Thread confinement
 Immutable and final
 Safe publication

Publishing

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>();
}
```



FvO,PvdH/FHTBM

Sharing Objects

26 november 2009

11/34

Visibility
 Publication and escape
 Thread confinement
 Immutable and final
 Safe publication

Publishing

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;
    }
}
```



FvO,PvdH/FHTBM

Sharing Objects

26 november 2009

12/34

Visibility
Publication and escape
 Thread confinement
 Immutable and final
 Safe publication

Publishing

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!



FvO,PvdH/FHTBM

Sharing Objects

26 november 2009

13/34

Visibility
Publication and escape
 Thread confinement
 Immutable and final
 Safe publication


Publishing

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);
            }
        });
    }
}
```




FvO,PvdH/FHTBM

Sharing Objects

26 november 2009

14/34

Visibility
Publication and escape
 Thread confinement
 Immutable and final
 Safe publication

Publishing

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;
    }
}
```



FvO,PvdH/FHTBM

Sharing Objects

26 november 2009

15/34

Visibility
Publication and escape
Thread confinement
Immutable and final
Safe publication

Keeping the information on one thread.
Ad-hoc thread confinement
Stack confinement
ThreadLocal

Do no let the beasties out



FvO,PvdH/FHTBM

Sharing Objects

26 november 2009

16/34

Visibility
Publication and escape
Thread confinement
Immutable and final
Safe publication

Keeping the information on one thread.
Ad-hoc thread confinement
Stack confinement
ThreadLocal

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** subsystems 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.



FvO,PvdH/FHTBM

Sharing Objects

26 november 2009

17/34

Visibility
Publication and escape
Thread confinement
Immutable and final
Safe publication

Keeping the information on one thread.
Ad-hoc thread confinement
Stack confinement
ThreadLocal

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;
}
```



FvO,PvdH/FHTBM

Sharing Objects

26 november 2009

18/34

Visibility
Publication and escape
Thread confinement
Immutable and final
Safe publication

Keeping the information on one thread.
Ad-hoc thread confinement
Stack confinement
ThreadLocal

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 confinement 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.



FvO,PvdH/FHTBM

Sharing Objects

26 november 2009

19/34

Visibility
Publication and escape
Thread confinement
Immutable and final
Safe publication

Keeping the information on one thread.
Ad-hoc thread confinement
Stack confinement
ThreadLocal

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 **do not** have to be threadsafe. For instance the JDBC specification does not require the `Connection` objects to be threadsafe. See the book for a solution.



FvO,PvdH/FHTBM

Sharing Objects

26 november 2009

20/34

Visibility
Publication and escape
Thread confinement
Immutable and final
Safe publication

Keeping the information on one thread.
Ad-hoc thread confinement
Stack confinement
ThreadLocal

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.



FvO,PvdH/FHTBM

Sharing Objects

26 november 2009

21/34

Visibility
Publication and escape
Thread confinement
Immutable and final
Safe publication

immutable is threadsafe
3 Stooges?
No, a final string set.
Final as good practice
volatile to increase vivibility

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.)



FvO,PvdH/FHTBM

Sharing Objects

26 november 2009

22/34

Visibility
Publication and escape
Thread confinement
Immutable and final
Safe publication

immutable is threadsafe
3 Stooges?
No, a final string set.
Final as good practice
volatile to increase vivibility

The 3 Stooges, once famous



The middle one is named "Curly" of course.



FvO,PvdH/FHTBM

Sharing Objects

26 november 2009

23/34

Visibility
Publication and escape
Thread confinement
Immutable and final
Safe publication

immutable is threadsafe
3 Stooges?
No, a final string set.
Final as good practice
volatile to increase vivibility

```
@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> stooges = new Vector<String>();
        stooges.add("Moe");
        stooges.add("Larry");
        stooges.add("Curly");
        return stooges.toString();
    }
}
```



FvO,PvdH/FHTBM

Sharing Objects

26 november 2009

24/34

Visibility	immutable is threadsafe
Publication and escape	3 Stooges?
Thread confinement	No, a final string set.
Immutable and final	Final as good practice
Safe publication	volatile to increase vivibility

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.



FvO,PvdH/FHTBM

Sharing Objects

26 november 2009

25/34

Visibility	immutable is threadsafe
Publication and escape	3 Stooges?
Thread confinement	No, a final string set.
Immutable and final	Final as good practice
Safe publication	volatile to increase vivibility

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);
    }
}
```



FvO,PvdH/FHTBM

Sharing Objects

26 november 2009

26/34

Visibility	immutable is threadsafe
Publication and escape	3 Stooges?
Thread confinement	No, a final string set.
Immutable and final	Final as good practice
Safe publication	volatile to increase vivibility

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);
    }
}
```



FvO,PvdH/FHTBM

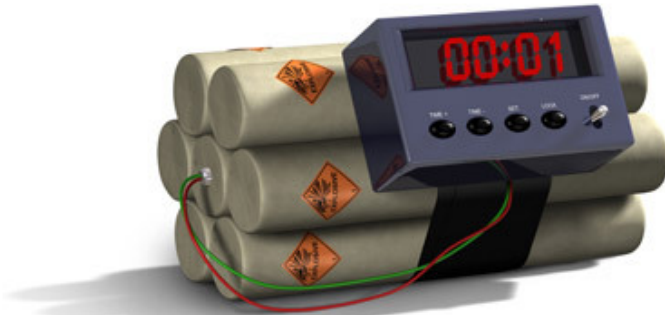
Sharing Objects

26 november 2009

27/34

Visibility	Improper publication example
Publication and escape	Immutable is safe
Thread confinement	Safe publication idioms
Immutable and final	Effectively immutable objects
Safe publication	Mutable objects

You are not supposed to be a code terrorist



FvO,PvdH/FHTBM

Sharing Objects

26 november 2009

28/34

Visibility	Improper publication example
Publication and escape	Immutable is safe
Thread confinement	Safe publication idioms
Immutable and final	Effectively immutable objects
Safe publication	Mutable objects

- 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.



FvO,PvdH/FHTBM

Sharing Objects

26 november 2009

29/34

Visibility	Improper publication example
Publication and escape	Immutable is safe
Thread confinement	Safe publication idioms
Immutable and final	Effectively immutable objects
Safe publication	Mutable objects

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.");
    }
}
```



FvO,PvdH/FHTBM

Sharing Objects

26 november 2009

30/34

Visibility
 Publication and escape
 Thread confinement
 Immutable and final
Safe publication

Improper publication example
Immutable is safe
 Safe publication Idioms
 Effectively immutable objects
 Mutable objects

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.



Visibility
 Publication and escape
 Thread confinement
 Immutable and final
Safe publication

Improper publication example
 Immutable is safe
Safe publication Idioms
 Effectively immutable objects
 Mutable objects

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.**



Visibility
 Publication and escape
 Thread confinement
 Immutable and final
Safe publication

Improper publication example
 Immutable is safe
 Safe publication Idioms
Effectively immutable objects
 Mutable objects

Effectively immutable is immutable by use.

Objects that are 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 (a class-library design mistake probably).
 If your application uses this in way that a stored date is not changed anymore, the synchronisation of the collection is sufficient:

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



