Concurrent Object-Oriented Programming (Part Two)

Concurrent and Parallel Programming

14 marzo 2022

Construction, publication and sharing

- Correctly encapsulating and managing the state of objects, by possibly making them thread-safe, is not sufficient to create multi-threaded object oriented programs.
- For correct concurrent OOP, robust techniques are also required to:
  - construct the objects,
  - make the objects accessible (publication),
  - share the use of the objects.

SUPSI

Dipartimento tecnologie innovative

Safe construction

- If a reference to an object escapes while the object is under construction, it may happen that the object gets shared between the threads, when it is still not fully and correctly constructed (therefore in an inconsistent state).
- In particular, care has to be taken to avoid the escape of the "this" reference during construction time. For example, it might happen if a thread is started by the constructor code.

Example

```
public class EventListener {
    public EventListener(EventSource eventSource) {
        // do our initialization
        // ...

        // register ourselves with the event source
        eventSource.registerListener(this);
    }

    public void onEvent(Event e) {
        // handle the event
    }
}
```

The EventSource is managed by an independent thread

```
public class RecordingEventListener extends EventListener {
    private final List<Event> list;

    public RecordingEventListener(EventSource eventSource) {
        super(eventSource);
        list = Collections.synchronizedList(new ArrayList<>());
    }

    public void onEvent(Event e) {
        list.add(e);
        super.onEvent(e);
    }

    public Event[] getEvents() {
        return list.toArray(new Event[list.size()]);
    }
}
```

Safe construction

- Obtaining the safe construction of a class, might be difficult because of inheritance. To avoid mistakes it has to be avoided that extensions of the class unintentionally violate the safe construction constraint.
- One of the possible solutions is to limit inheritance, by specifying the class as final, or by using a private constructor and a public factory method.

Example

```
class AccumulatorWithFM {
    static int initialValue = 0;
    private int value = 0;

    static AccumulatorWithFM createAccumulator(final int value) {
        if (value > initialValue)
            return new AccumulatorWithFM(value);
        else if (value > 0)
            return new AccumulatorWithFM(initialValue);
        else
            return null;
    }

    private AccumulatorWithFM(final int value) {
        this.value = value;
    }

    // ...
}

class AccumulatorExtension extends AccumulatorWithFM {
    public AccumulatorExtension(final int value) {
        super(value);
    }
}
```

Error: the constructor is not visible!

Publication of objects

- In multi-threaded applications it is important to analyze the state that a class owns (state ownership), in particular in terms of referenced objects.
- State ownership is an element of the class design. If a class references an *HashMap*, the state might include the *Map*, all Map.*Entry* instances, and potentially all the keys and objects contained in the *Map*.

Publication of objects

- Publishing an object means making it accessible to the code outside the scope of the class (or the method, in case of local variables), that owns it in a specific moment.

- Attention: the publication of an object is performed by a thread that makes the object available to other threads.

Example

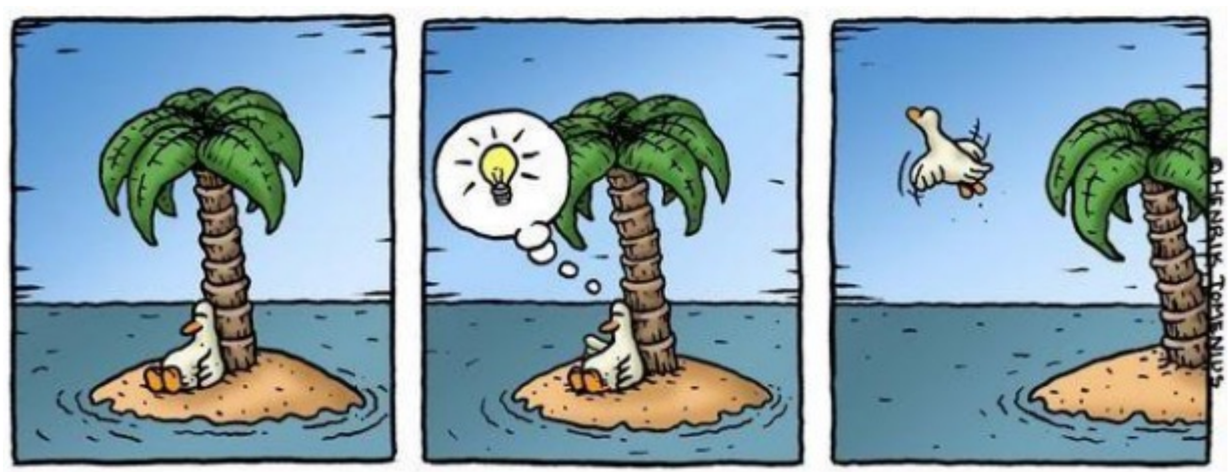
```java
class Printer implements Runnable {
    @Override
    public void run() {
        while (PubblicationExample.list == null) {
        }
        for (final String string : PubblicationExample.list) {
            System.out.print(string);
        }
    }
}

public class PubblicationExample {
    static List<String> list = null;

    public static void main(final String[] args) {
        // Starts thread here …
        List<String> tempList = new ArrayList<String>();
        tempList.add("Goodmorning");
        tempList.add("students,");
        tempList.add("welcome ");
        tempList.add("to the ");
        tempList.add("course.");
        list = tempList;
    }
}
```

Publication of objects

- Publishing the internal state of a class can compromise its encapsulation and make it difficult to preserve its invariants.
- An object that is wrongly published (when it shouldn't), is called an escaped object.
- Any published object might be used inappropriately by other threads. For this reason it is important to limit the publication of objects, by profiting from confinement techniques as much as possible.

Confinement and publication

- If an object is confined to a context (e.g. a class by means of instance confinement, or a method by means of stack confinement), publishing and allowing the object to escape has to be considered a BUG!



Indirect and hidden publication

- The publication of an object might result in the indirect publication of other objects (objects that can be reached by following a chain of references).
- The publication of objects can be hidden.

- From the point of view of a class, methods with a behavior that is not fully specified by the class (such as overridden methods) are a danger.
- If a confined object is provided to an overridden method, it has to be considered as being published.

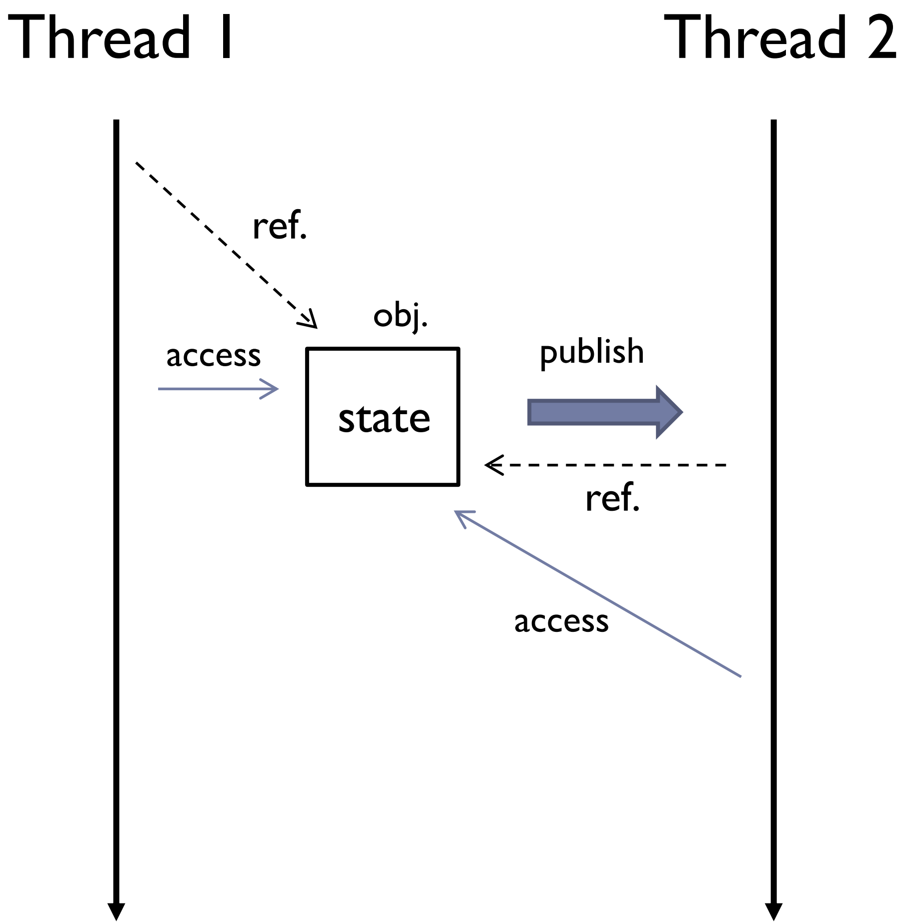
Safe publication

- *For the moment let's focus on the publication, not on any subsequent share and modification of the published objects!*

- Safe publication of an object means that the reference to the object and its state are made accessible simultaneously (in an atomic way) => this can only be done by using synchronization tools.
Visibility problems must be avoided!

- All mutable objects must be safely published.
Synchronization is required on both sides: the publisher, as well as the recipient of the publication.

Safe publication



Safe publication

- A safely constructed object can be safely published:
  - by initializing a static reference with default initialization or a static initializer (*public static MyObject obj = new MyObject(42);*)
  - by storing the object reference in a volatile variable (or in an AtomicReference)
  - by storing the object reference in a final field of a safely constructed object
  - by storing the object reference in a lock-protected field
  - by storing the object reference in a thread-safe collection (will be discussed later)

SUPSI

Dipartimento tecnologie innovative

Example

Stack confinement to avoid successive inconsistencies

Safe publication

```
class Printer implements Runnable {
    @Override
    public void run() {
        while (PubblicationExample.list == null) {
        }
        List<String> tempList = PubblicationExample.list;
        for (final String string : tempList) {
            System.out.print(string);
        }
    }
}

public class PubblicationExample {
    volatile static List<String> list = null;

    public static void main(final String[] args) {
        // Starts thread here …
        List<String> tempList = new ArrayList<String>();
        tempList.add("Hello");
        tempList.add("students, ");
        tempList.add("welcome ");
        tempList.add(«to ");
        tempList.add("the course.");
        list = tempList;
    }
}
```

Effectively immutable objects

- A safely published object can be accessed in a read-only form without synchronization, even if the object is not immutable. In this case, we are talking about effectively immutable objects.
- Why? Remember: the visibility effect of volatile variables, atomic variables, and locks, also extends to any other state (including the internal state of published objects) after using the volatile/atomic variable or the lock.
- At a practical level, it is possible to take advantage from effectively immutable objects, for example, in combination with the "one writer - many readers" technique.

Publication and mutable objects

- For all objects that are modified after their publication, safe publication only guarantees correct visibility of the published state.
- After publication, mutable objects have then to always be accessed safely. Therefore, either they are thread-safe objects or all accesses have to be protected by locks in the client code.

SUPSI

Dipartimento tecnologie innovative

Publication and thread-safe objects

- If an object owned by a class is thread safe, does not participate in any invariant (e.g. dependencies with other variables), then it can be published (safely) without difficulties.
- Otherwise, ... care has to be taken by introducing additional synchronization, to avoid later inappropriate use of the object.

Example

- Attention: even if an object is thread-safe, it might still be needed to use synchronization in the client code.

```
public static Object getLast(final List list) {
    synchronized(list) {
        final int lastIndex = list.size() - 1;
        return list.get(lastIndex);
    }
}

public static void deleteLast(final List list) {
    synchronized(list) {
        final int lastIndex = list.size() - 1;
        list.remove(lastIndex);
    }
}
```

A thread-safe List is used

Possible publications of objects

- To summarize:
  - Immutable objects: can be published by any means, but it is better to publish them safely (for correct visibility of the reference).
  - Effectively immutable objects: must be published safely.
  - Mutable objects: must be published safely and must be thread-safe or protected by synchronization when used.

Possible object shares

- To summarize:
  - Thread-confined objects: are accessed (read and write) only by the owning thread.
  - Shared read-only objects: can be read concurrently without synchronization. Include immutable objects as well as effectively immutable objects.
  - Shared thread-safe objects: perform synchronization internally. Can be accessed (read and write) concurrently without synchronization in the client code.
  - Shared objects protected by locks: can only be accessed (read and write) when holding the associated lock.

SUPSI

Dipartimento tecnologie innovative

Immutable Holder of Objects

- You might have the impression that immutable objects limit the possibility of updating the program states.
- But this is not true.
- Immutable objects can be replaced with new instances containing the new values, when needed.

Immutable Holder of Objects

- Immutable objects can be exploited to provide atomicity: every time a group of values/objects has to be used in an atomic way, an immutable holder of these values/objects can be created.
- Then, if the values/objects need to be modified, a new holder can be created to replace the previous one. This operation has to be the direct responsibility of the holder (copy-on-write).
- All threads that still have a reference to the old holder, continue to see a consistent (but old) state.

Immutable Holder of Objects

- To ensure correct visibility, this approach can be complemented with a volatile (or atomic) reference to the holder instance for performing safe-publication each time the holder instance is replaced with a new one.
- Attention: any race-condition such as check-then-act or read-modify-write has to be avoided.

- An effective solution to share the immutable holder is to use the "one writer - many readers" technique.

Example

```java
//Immutable Holder
final class Holder {
    private final int n;
    private final int m;

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

    public Holder incrementValues(final int delta) {
        return new Holder(n + delta, m + delta);
    }

    //…
}
```

```java
class Foo {
    private volatile Holder holder;

    public Holder getHolder() {
        return holder;
    }

    public void incrementValues(final int delta) {
        holder = holder.incrementValue(delta);
    }
}
```

In this case the substitution is not atomic.

Example

```java
final class Foo {
    private final AtomicReference<Holder> holderRef
        = new AtomicReference<Holder>();

    public Holder getHolder() {
        return holderRef.get();
    }

    public void incrementValue(final int delta) {
        holderRef.set(holderRef.get().incrementValue(delta));
    }
}
```

Also in this case the substitution is not atomic. It could eventually be made atomic by implementing the CAS idiom.

Example

```
final class Foo {
    private Holder holder;

    public synchronized Holder getHolder() {
        return holder;
    }

    public synchronized void setValue(final int n) {
        holder = new Holder(n);
    }

    public synchronized void incrementValue(final int delta) {
        holder = holder.incrementValue(delta);
    }
}
```

Locks and performances

- We know that declaring all methods of a class synchronized is a simple solution to obtain thread-safety.
- However, this technique can compromise the performances of the program.
- Extreme situations occur when only one thread at a time performs large synchronized operations, blocking all other threads.
- Mutexes can be very ineffective because threads may get suspended with consequent context-switching.

Uncontended synchronization

- When developing a concurrent application, the difference between contended and uncontended synchronization has to be taken into consideration.
- Don't worry about uncontended synchronization. The Java compiler has optimization support for it (locks removal, escape analysis, ...).

- But even if the compiler is not able optimize, uncontended synchronization is not expensive because threads are not frequently blocked at the lock.

Highly contended locks

- The main difficulties arise when there are highly contended exclusive-locks.
- The serialization effect caused by excessive synchronization reduces the concurrent execution potential. When a concurrent application runs serially, performances might be worse than for its mono-thread version.
- Two factors produce highly contended locks: the number of requests and the duration of the lock.
- In extreme cases, if many threads are waiting, computing resources might be underused even if there is a lot of work to do.

Poor concurrency

- We speak of poor concurrency, when the number of simultaneous invocations is limited by the program structure (instead of the available processing resources).

Locks and concurrency

- To increase concurrency, the size of the synchronized code blocks (or protected by other locking solutions) has to be reduced.
- Balance between simplicity (synchronized on the whole method) and concurrency (synchronized on small parts of code) has to be found.
- The acquisition of locks involves overheads, therefore splitting a synchronized code block into multiple ones is not always the advisable solution.
- Compromises have to be found on the different aspects of the application design.

Reducing the lock contention

There are 3 ways to reduce the lock contention:

- Reduce the lock duration
  - less synchronized code
  - lock splitting
- Reduce the lock request frequency
- Replace exclusive locks with other synchronization approaches (immutability, volatile variables, atomic variables, ...).

SUPSI

Dipartimento tecnologie innovative

Example: lock splitting

```java
public class ServerStatus {
    private final Set<String> users;
    private final Set<String> queries;

    public ServerStatus() {
        users = new HashSet<String>();
        queries = new HashSet<String>();
    }

    public synchronized void addUser(String u) {
        users.add(u);
    }

    public synchronized void addQuery(String q) {
        queries.add(q);
    }

    public synchronized void removeUser(String u) {
        users.remove(u);
    }

    public synchronized void removeQuery(String q) {
        queries.remove(q);
    }
}
```

SUPSI
Dipartimento tecnologie innovative

Example: lock splitting

```java
public class ServerStatus {
    private final Set<String> users;
    private final Set<String> queries;

    public ServerStatus() {
        users = new HashSet<String>();
        queries = new HashSet<String>();
    }

    public void addUser(String u) {
        synchronized (users) {
            users.add(u);
        }
    }

    public void addQuery(String q) {
        synchronized (queries) {
            queries.add(q);
        }
    }

    public void removeUser(String u) {
        synchronized (users) {
            users.remove(u);
        }
    }

    public void removeQuery(String q) {
        synchronized (queries) {
            queries.remove(q);
        }
    }
}
```

Hot fields

- A program that takes advantage of lock splitting is a program that has contended locks but uncontended data.
- The granularity of the lock cannot be reduced if there are variables required for each operation (hot fields).
- If hot fields are present, the only option is to modify the design of the program, by e.g. considering alternative solutions to the lock.

Alternatives to mutexes

- Volatile variables
  - for status variables (e.g. stop flags)
- Immutable objects
- Various techniques for thread safety
  - example: thread-confinement, stack-confinement, ...
- ReadWrite locks
  - acquire the lock only at the time of writing
  - very good if lot of reads and just a few writes are performed
- Atomic Variables
  - good for counters, management of dynamic data structures, ...
- Concurrent collections and synchronizers

Summary of topics

- Safe construction
- Publication of objects
- Safe publication
- Effectively immutable objects
- Publication and mutable/thread-safe objects
- Sharing of objects after publication
- Technique of the immutable holder
- Locks and performances
- Possible locks optimizations