v1.0
BETA

Sistemi Concorrenti e di Rete LS

II Facoltà di Ingegneria - Cesena

a.a 2008/2009

[module lab 2.1]
THREAD SAFETY

THREAD SAFETY DEFINITION

- A central aspect of concurrent programming is writing thread-safe code / **thread-safe** classes
  - a class is thread-safe if it continues to behave correctly when accessed from multiple threads, regardless of the scheduling or interleaving of the execution of those threads by the runtime environment, and with no additional synchronization or other coordination on the part of the calling code
- *Correctness* means that a class conforms to its specification.
  - a good specification defines
    - *invariants* constraining an object's state
    - *post-conditions* describing the effects of its operations

> Thread-safe classes encapsulate any needed synchronization so that clients need not to provide their own

SHARED MUTABLE STATE

- Writing thread-safe code is – at its core – about *managing access to state and in particular to shared, mutable state*:
  - **shared**: variable or object could be accessed by multiple threads
  - **mutable**: its value could change durint its lifetime
- if multiple threads access the same mutable state variable without appropriate synchronization, your program is *broken*.
- There are three ways to fix it:
  - don't share state variable across threads
  - make the state variable immutable
  - use synchronization whenever accessing the state variable

THREAD-SAFETY AND OO PRINCIPLES

- It is far easier to design a class to be thread-safe than to retrofit it for thread-safety later
  - > OO techniques and principles – encapsulation, immutability, clear specification of invariants – are very important also for designing and developing thread-safe code
- ”Good practice first to make your code right and clean, and then make it fast”
  - – if this well-known rule is important in programming in general, it is even more important in concurrent programming
  - – *first safety, then performance*
    - ...with some exceptions, of course...

STATELESS OBJECT

- Stateless objects are *always thread-safe*
  - actions of a thread accessing a stateless object cannot affect the correctness of operations in other threads
- As an example, let's consider a simple servlet [*]

```
@ThreadSafe
public class StatelessFactorizer implements Servlet {
  public void service(ServletRequest req, ServletResponse resp){
    BigInteger i = extractFromRequest(req);
    BigInteger[] factors = factor(i);
    encodeIntoResponse(resp,factors);
  }
  private BigInteger extractFromRequest(ServletRequest req){...}
  private BigInteger[] factor(BigIntger x){...};
  private void encodeIntoResponse(ServletResponse resp,BigInteger[] v){...};
}
```

- This servlet – as most of the servlets – is stateless: it has no fields and references no fields from other classes.
  - this class is thread-safe.

[*] about servlets: the Servlet Framework – as almost all the other Java frameworks, such as RMI or Swing – create threads and call your components from those threads, leaving you the responsibility of making your components thread-safe

STATE-FULL OBJECTS & ATOMICITY

- **Atomicity** is the key aspect in thread safety for state-full objects.
  - example: extension of the servlet with a notion of state
    - functioning as 'hit counter'

```
@NotThreadSafe
public class UnsafeCountingFactorizer implements Servlet {
  private long count = 0;
  public void service(ServletRequest req, ServletResponse resp){
    BigInteger i = extractFromRequest(req);
    BigInteger[] factors = factor(i);
    count++;
    encodeIntoResponse(resp,factors);
  }
  public long getCount() { return count; }
  ...
}
```

- The class is **not** thread-safe
  - lost update problem in this case, in particular
  - reason: count++ is not atomic
  - count++ is an example of *read-modify-write* operation
  - needs to be made *atomic*

RACE CONDITIONS

- The concurrent execution of non-atomic sequence of statements that should be considered atomic generate *race conditions*
  - a race condition occurs when the correctness of a computation depends on the relative timing or interleaving of multiple threads by the runtime
    - ...and getting the right answer relies on lucky timing..
- `UnsafeCountingFactorizer` servlet has several race conditions making its results unreliable

CHECK-AND-ACT

- Most common race condition type: *check-and-act*
  - when a potentially stale observation is used to make a decision on what to do next
- Example:

```
...
If (file X doesn't exist) -- check
  then create file X      -- act
...
```

- Since check+act are not atomic, *the state can change after check and before act*.

EXAMPLE: LAZY INITIALIZATION PATTERN (SINGLETON PATTERN)

- This class has race conditions (on the field instance) that undermine its correctness.

```
@NotThreadSafe
public class LazeInitRace {

  private ExpensiveObject instance = null;

  public ExpensiveObject getInstance(){
    if (instance == null){
      instance = new ExpensiveObject();
    }
    return instance;
  }
}
```

COMPOUND ATOMIC ACTIONS

- *check-and-act* and *read-modify-write* are examples of compound actions
  - sequences of operations that must be executed atomically in order to remain thread-safe

USING ATOMIC VARIABLES TO SOLVE RACE CONDITIONS

- Atomic variable classes in `java.util.concurrent.atomic` package
  - for effecting atomic state transitions on numbers and objects references

```
@ThreadSafe
public class CountingFactorizer implements Servlet {

  private final AtomicLong count = new AtomicLong(0);

  public void service(ServletRequest req, ServletResponse resp){
    BigInteger i = extractFromRequest(req);
    BigInteger[] factors = factor(i);
    count.incrementAndGet();
    encodeIntoResponse(resp,factors);
  }

  public long getCount() { return count.get(); }
  ...
}
```

ATOMIC VARIABLES ARE NOT ENOUGH

- When *multiple variables participate in an invariant*, atomic access on the individual variable is not enough
  - for instance compound actions on different objects
- Example: unsafe servlet with caching

```
@NotThreadSafe
public class UnsafeCachingFactorizer implements Servlet {
  private final AtomicReference<BigInteger> lastNumber =
                                              new AtomicReference<BigInteger>();
  private final AtomicReference<BigInteger[]> lastFactors =
                                              new AtomicReference<BigInteger[]>();
  public void service(ServletRequest req, ServletResponse resp){
    BigInteger i = extractFromRequest(req);
    if (i.equals(lastNumber.get())){
       encodeIntoResponse(resp,lastFactors.get());
    } else {
      BigInteger[] factors = factor(i);
      lastNumber.set(i);
      lastFactors.set(factors);
      encodeIntoResponse(resp,factors);
    }
  }
}
```

ATOMIC COMPOUND ACTIONS IN JAVA : SYNCHRONIZED BLOCKS

- Compound-actions - and atomic statement blocks - in Java can be realised by means of synchronized blocks or methods

```
synchronized(lock){
   statement
   statement
   statement
}
```

- A synchronized block has 2 parts
  - a reference to an object that will serve as the lock
  - block of code to be guarded by the lock

INTRINSIC LOCK AND ENTRY SET

- Atomic blocks work by exploiting the **lock** embedded in each Java object (more on this in next modules)
  - called **intrinsic lock** or monitor lock
  - functioning as a *guard* for the block
- The lock is automatically acquired and then released by a thread respectively when entering and exiting the block
  - if the lock is already acquired, the thread is blocked (suspended) and added to the entry set
  - when a thread exited the block , one thread of the entry set is selected and re-activated
  - no ordering policy is specified
  - if the lock is not released by the thread inside the block, threads in the entry set are blocked forever (starvation)
- For static methods and fields, the lock is associated to the related Class object
- For **synchronized** methods, the object serving as lock is **this**

REENTRANCY (1/2)

- Java intrinsic locks are reentrant:
  - if a thread tries to acquire a lock that it already holds, the request succeeds
- def: *lock reentrancy*
  - when locks are acquired on a **per-thread** basis
    - vs. per-invocation basis
    - per-invocation basis is adopted instead as default locking behaviour for Pthreads (POSIX threads) mutex-es

REENTRANCY (2/2)

- Reentrancy facilitates encapsulation of locking behaviour and thus simplify the development of OO concurrent code

```
public class Widget {
  public synchronized doSomething(){...}
}

public class LoggingWidget extends Widget {
   public synchronized void doSomething(){
     System.out.println(toString()+": calling doSomething");
     super.doSomething();
   }
}
```

- Without reentrancy the above example would lead to a deadlock

GUARDING STATE WITH LOCKS

- *Guarding a state* – possibly composed by more than one variables -- means making its access atomic
  - taking into the account that it could be accessed by multiple (different) compound actions
- "Golden rules" for keeping safety:
  - for each mutable state variable that may be accessed by more than one thread, all accesses to that variable must be performed with *the same lock* held (variables guarded by locks)
  - every shared, mutable variable should be guarded by exactly one lock
    - common pattern: encapsulating all mutable state within an object and then protected it through its intrinsic lock
    - Towards the monitor abstraction (next module)
  - for every invariant that involves more than one variables, all the variables involved in that invariant must be guarded by the same lock

PERFORMANCE: POOR CONCURRENCY PROBLEM (1/2)

- The misuse of atomic blocks can lead to performance problems.
- Example
  - caching servlet with atomic block implemented by synchronized method:

```
@ThreadSafe
public Class SynchronizedFactorizer implements Servlet {
  private  BigInteger lastNumber;
  private  BigInteger[] lastFactors;
  public synchronized void service(ServletRequest req, ServletResponse resp){
    BigInteger i = extractFromRequest(req);
    if (i.equals(lastNumber)){
       encodeIntoResponse(resp,lastFactors);
    } else {
      BigInteger[] factors = factor(i);
      lastNumber.set(i);
      lastFactors.set(factors);
      encodeIntoResponse(resp,factors);
    }
  }
}
```

PERFORMANCE: POOR CONCURRENCY PROBLEM (2/2)

- The class is thread-safe, but it inhibits multiple clients for using the factoring servlets simultaneously at all, resulting in unacceptably poor responsiveness!
  - the intended use of the servlet framework is subverted, since servlets have been conceived to handle multiple requests simultaneously
  - as a result, the web application exhibits poor concurrency
  - the number of simultaneous invocations is limited not by the availability of processing resources, but by the structure of the application itself
- The problem can be solved by good engineering, understanding which parts need to be atomic blocks

A MORE EFFICIENT SOLUTION

```
@ThreadSafe
public Class CachedFactorizer implements Servlet {
  private  BigInteger lastNumber;
  private  BigInteger[] lastFactors;
  private  long hits;
  private  long cachedHits;
  public void service(ServletRequest req,ServletResponse resp){
    BigInteger i = extractFromRequest(req);
    BigInteger[] factors = null;
    synchronized(this){
      hits++;
      if (i.equals(lastNumber)){
        cachedHits++;
        factors = lastFactors.clone();
      }
    }
    if (factors == null){
      BigInteger[] factors = factor(i);
      synchronized(this){
        lastNumber = i;
        lastFactors = factors.clone();
      }
    }
    encodeIntoResponse(resp,factors);
  }
}
```

NOTE

- When implementing a policy for atomic blocks, resist the temptation to prematurely sacrifice simplicity (potentially compromising safety) for the sake of performance
- Avoid holding locks during lengthy computations or operations at risk of not completing quickly such as network or console I/O

THREAD-CONFINEMENT

- Accessing shared, mutable data requires synchronization: one way to avoid this requirement is to *not share*
  - if data is accessed from a single thread, no synchronization is needed
- *Thread confinement* technique
  - confining the creation and use of an object to a thread
- Example: **Swing** uses thread confinement extensively
  - Swing visual components and data model objects are not thread-safe
  - safety is achieved by confining them to the *Swing event dispatch thread*
  - > *code running in threads other than the Swing dispatcher thread should not access this objects!*
- Thread confinement is an element of program's design
  - no direct language support
  - must be enforced by the implementation

STACK-CONFINEMENT

- **Stack-confinement** is a special case of thread confinement in which an object can only be reached through *local* variables
  - it makes it easier to confine objects to a thread
- Example:

```
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 SpecialGenderComparator());
  animals.addAll(candidates);
  for (Animal a: animals){
    if (candidate == null || !condidate.isPotentialMate(a)){
      candidate = a;
    } else {
      ark.load(new AnimalPair(candidate,a));
      numPairs++;
      candidate = null;
    }
  }
}
```

  - there is exactly one reference to the set (TreeSet), held in local variable and then confined to the executing thread

WRAP UP: POLICIES FOR USING AND SHARING OBJECTS IN A CONCURRENT PROGRAM

- **Thread-confined**
  - a thread-confined object is owned exclusively by and confined to one thread, and can be modified by its owning thread
- **Shared read-only (no updates)**
  - a shared-only object can be accessed concurrently by multiple threads without additional synchronization
    - but cannot be modified by any thread
- **Shared thread-safe**
  - a thread-safe object performs synchronization internally, so multiple threads can freely access it through its public interface without further synchronization
- **Guarded**
  - a guarded object can be accessed only with a specific lock held