Prof. Dr. Oscar Nierstrasz
Pascal Gadient, Leonel Merino

Solution
Series 04 — 11.10.2017 – v1.0a
Safety Patterns

Exercise 1 (3 Points)

Please answer the following questions:

a. Why are immutable classes inherently safe? **Answer:**

Because the state of an instance of an immutable class cannot be changed after its creation. This prevents the object from ever getting into an inconsistent state. To create an immutable class you must ensure that the instance fields are never changed after the construction e.g. by declaring them final.

b. What is "balking"? **Answer:**

Balking describes a design pattern that is used in conjunction with state-dependent actions. In practice, balking consists in returning a failure (or rather a refusal to do something) if some preconditions fail, i.e. some object fails to satisfy the class invariant. A requesting object is then informed, e.g. by raising an IllegalStateException. This pattern is very common in data access classes such as URLConnection, where the connection needs to be in a certain state before data can be exchanged.

c. When is partial synchronization better than full synchronization? **Answer:**

Partial synchronization should be preferred over full synchronization when

- *Objects encapsulate both mutable and immutable data, i.e contain changeable and unchangeable instance variables*
- *Methods can be organized such that parts of them deal with critical sections and other parts do not*

For these cases only the "critical section"-parts need synchronization and this reduces largely the resulting overhead.

d. How does containment avoid the need for synchronization? **Answer:**

By embedding unsynchronized objects inside other objects that have at most one active thread at a time. The "contained" objects can be seen as exclusively held resources. Since such objects do not reveal their identity to other objects (i.e. not accessible via public methods etc.), they do not need to be synchronized.

e. What liveness problems can full synchronization introduce? **Answer:**

For instance, when one method contains two separate critical sections a potential deadlock may occur. Moreover, the nested monitor lockout problem can arise as well. Two layers of synchronization locks may lead to nested monitor lockouts when:

- *A thread releases the outer lock for the resource*
- *Another thread just started waiting at the inner lock ...*
- *... and this thread also possesses the exclusive access right (lock) to release the inner lock (other threads are not able to release the inner lock)*

The noticeable impact of both problems is identical: Threads might wait forever for the demanded resource to be unlocked.[1]

f. When is it legitimate to declare only some methods as synchronized? **Answer:**

When methods exist that access only immutable data. These methods do not have to be declared as synchronized. See partial synchronization.

Exercise 2 (2 Points)

The dining savages: A tribe of savages eats communal dinners from a large pot that can hold M servings of stewed missionary. When a savage wants to eat, he helps himself from the pot unless it is empty in which case he waits for the pot to be filled. If the pot is empty the cook refills the pot with M servings. The behavior of the savages and the cook are described by:

```
SAVAGE = (getserving -> SAVAGE).
COOK   = (fillpot -> COOK).
```

Model the behavior of the pot as an FSP process. **Answer:**

```
const M  = 5
SAVAGE   = (getserving -> SAVAGE).
COOK     = (fillpot -> COOK).
POT                    = SERVINGS[0],
SERVINGS[i:0..M]       = (when (i==0) fillpot -> SERVINGS[M]
                         |when (i>0 ) getserving -> SERVINGS[i-1]
                         ).
||SAVAGES = (SAVAGE || COOK || POT).
```

Exercise 3 (2 Points)

Please consider the FSP below and answer the questions:

```
    property LIFTCAPACITY = LIFT[0],
    LIFT[i:0..8]      = (enter -> LIFT[i+1]
                        |when(i>0) exit -> LIFT[i-1]
                        |when(i==0)exit -> LIFT[0]
                        ).
```

[1] http://tutorials.jenkov.com/java-concurrency/nested-monitor-lockout.html

a. Which values can the variable `i` take? **Answer:**

   The variable `i` can either be one of the values $\{0, 1, 2, 3, 4, 5, 6, 7, 8\}$

b. What kind of property is used and what does it guarantee? **Answer:**

   A safety property is used and it enforces the variable to be within the predefined range (i = 0..8), i.e. people can enter the lift up to the maximum load, but not beyond.

c. Provide an action trace that violates the provided property. **Answer:**

   enter, enter, enter, enter, enter, enter, enter, enter, enter (9 times)

d. Provide an action trace that does not violate the provided property. **Answer:**

   enter, enter, enter, exit, exit, exit

Exercise 4 (3 Points)

Implement a thread-safe MessageQueue class in Java using one of the safety patterns. You have to justify your choice of the safety pattern.

The MessageQueue has the following specifications:

- It provides FIFO (first-in-first-out) access
- Messages are strings
- The queue has a fixed capacity that is set at construction time
- There are two methods: `add(String msg)` and `remove()`
- Method `add(String msg)` inserts a message to the end of the queue. If the queue is full, it waits until a message is removed
- Method `remove()` returns the (first) message at the beginning of the queue and then removes it from the queue
- Write a unit test for your MessageQueue to demonstrate the thread-safety of your implementation

Please consider also the order of importance for development:
1. Logic (Does the implementation work?)
2. Concurrency (Do safety- and liveness-property hold?)
3. Scalability (Does the approach scale?)

Answer:

Please have a look at the attached Java source files.