Mathias Hölzl  
Mat. Nr.: 1234567  
Gruppe: 6  
Gruppenmitglieder: Tischler Florian, Mathias Hölzl  
gelöste Aufgaben: 3/3  
resultierende Punkte: 100%

# Aufgabe 1

## a)

Mithilfe von **ReentrantLock**.**lock**/**unlock**() und **Contition**.**await**() wartet der Consumer bis der Producer ein Objekt bereitstellt. Erst wenn er von jedem Producer ein Objekt erhalten hat kann der Consumer fortfahren.

## b)

Mithilfe von **ReentrantLock** mit param **fair** true wird sichergestellt, dass kein thread bei wait verhungert. Alternative: wait/notifyAll, wobei die Resourcenvergabe zufällig und nicht fair ist.

Anmerkung: die main-Methode befindet sich in der Klasse Main

# Aufgabe 2

Als erstes wurde die Receive Klasse um Getter/Setter für das message Feld erweitert. Im Getter wird dabei das message Feld beim Lesen wieder auf null zurückgesetzt. Das hängt mit der Synchronisierung in der Empfangsmethode zusammen und wird später erläutert. Die receiveMessage Funktion wurde so implementiert:

String receiveMessage(**int** i)  
{  
 **synchronized** (**receive**[i])  
 {  
 String msg = **receive**[i].getMessage();  
 **while**(msg == **null**)  
 {  
 **try** {  
 **receive**[i].wait();  
 }  
 **catch** (InterruptedException e) {  
 e.printStackTrace();  
 }  
  
 msg = **receive**[i].getMessage();  
 }  
  
 **return** msg;  
 }  
}

Zuerst wird das betroffene Receive Objekt gesperrt und geprüft ob eine Nachricht vorliegt (mit der Überprüfung ob die Nachricht null ist). Ist noch keine Nachricht eingetroffen wartet der aufrufende Thread auf das Senden der Nachricht. Sobald er benachrichtigt wurde speichert er die Nachricht und gibt sie zurück. Damit dieses Verhalten bei weiteren Senden auch eintritt, wird im Getter des Receive Objekts die Nachricht auf null zurückgesetzt.

Die Sendefunktion sendMessage sieht wie folgt aus:

**public void** sendMessage(String m, **int** pid)  
{  
 **synchronized** (**receive**[pid])  
 {  
 **receive**[pid].setMessage(m);  
 **receive**[pid].notify();  
 }  
}

Dabei wird nur das benötigte Receive Objekt gesperrt und mithilfe des Setters die Nachricht gespeichert. Anschließend wird ein eventuell wartender Thread aufgeweckt (es wartet immer maximal ein Thread, deswegen notify und nicht notifyAll).

Um sicherzustellen, dass alle Prozesse erzeugt und registriert wurden bevor p0 ihnen ihre Nachbaren zuweist wurde ein CountdownLatch verwendet. Dieser wird von p0 mit der Anzahl der Threads auf die gewartet werden muss (n-1) initialisiert. In der Run Methode wird nun das Thread.Sleep() durch latch.await() ersetzt. Da es vorkommen kann, dass der Referenzvektor bereits gesetzt wurde bevor der Prozess den wait() Block in der Run Methode erreicht, wurde hier noch eine Abfrage eingebaut welche sicherstellt, dass ein Prozess der seinen Referenzvektor bereits erhalten hat den wait() Block nicht betritt.

**if** (**pid**==0) {**try** {  
 **latch**.await();  
 } **catch** (InterruptedException e) {  
 e.printStackTrace();  
 }  
  
 *// p0 sends the reference vector to the others* **p**[**pid**] = (Process)**this**;  
 **for** (**int** i=1; i<**p**.**length**; i++) {  
 **try** {  
 **p**[i].neighbours(**p**);  
 } **catch** (Exception e) {  
 System.***err***.println(**"init exception:"**);  
 e.printStackTrace();  
 }  
 }  
}  
**else** {  
 *// wait until they got the reference vector* **synchronized** (**this**) {  
 **if**(**p** == **null**){  
 **try** {  
 **this**.wait();  
 }  
 **catch** (InterruptedException e) {}  
 }  
 }  
}

# Aufgabe 3

## a)

Für das Testszenario wurde der Konstruktor des ExpensiveObjects um eine Thread.sleep() erweitert um eine zeitintensive Operation zu simulieren. Rufen nun mehrere Threads gleichzeitig getInstance() auf so erscheint für jeden Thread die Bedingung if(instance==null) als wahr und alle erzeugen eine neue Instanz. Um dieses Verhalten zu erzielen werden in der TestLazyInit der Threads erzeugt die getInstance auf demselben LazyInitRaceCondition Objekt aufrufen. Hierzu wurde aus der LazyInitRaceCondition ein Interface extrahiert damit später die anderen Implementierungen mit derselben Methode getestet werden können.

Anmerkung: abhängig davon wie die Threads gescheduled werden kann es sein, dass ein, zwei oder alle Threads schlussendlich die gleiche Instanz erhalten. Dieses Verhalten ist jedoch nicht deterministisch.

## b)

Um dieses Verhalten zu vermeiden gibt es unter anderen folgende Ansätze:

### Synchronized

Dabei wird die getInstance–Methode mit dem synchronized Keyword ausgestattet um sicherzustellen, dass maximal ein Thread den Methodenkörper betritt. Somit wird sichergestellt, dass nur eine Instanz erstellt wird.

### Atomic

Eine Möglichkeit wäre die instance Variable als AtomicReference Feld anzulegen. Dies allein reicht jedoch nicht um getInstance Thread-Safe zu machen, da damit nur die Abfrage und die Zuweisung atomar wird, jedoch nicht beide zusammen. Um dies zu erreichen kann man die Methode compareAndSet(expected, value) verwenden, die dem Objekt, abhängig davon ob das Objekt gleich expected ist oder nicht, den Paramter value zuweist.

Beide Lösungsansätze wurden implementiert und bezüglich des Testszenarios aus a) als korrekt befunden.

Anmerkung: die main-Methode befindet sich in der Klasse Main