Threads

(1.) (a.) Nehmen Sie das Beispiel 2 von der Folie und ändern Sie dies ab, so dass sie z.B. ein Array von 100 Threads starten können! Testen Sie aber zunächst ihren Kode mit einem Array von 2 Threads!
Nehmen Sie als Vorlage:

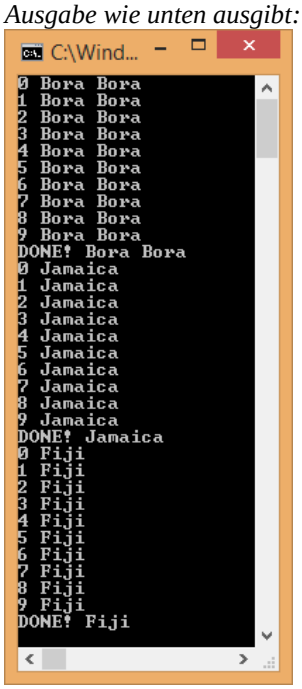
```
class CCount2Ten implements Runnable
{
   @Override
   public void run()
   {
      //Task of this thread is to print multiplication of successive numbers up to 1000 numbers
      for(int i = 0; i < 10; i++)
      {
         System.out.println(i);
      }
   }
}

public class TestCCount
{
  public static void main(String[] args)
  {
    CCount2Ten myCount = new CCount2Ten();
    Thread threadCount = new Thread(myCount);
    threadCount.start();
  }
}
```

(b.) Ändern Sie obiges Programm wie folgt ab:
Erweitern Sie das Programm wie folgt: Im Hauptprogramm soll der Name von jedem Thread „Anna Nass Nr. X“ lauten! Ferner soll nach dem Starten der Name von „Anna Nass Nr. X“ in „Lee Monade Nr. X“ ändern! X= Zahl von 0 bis ?! Ermitteln Sie auch: Wie lauten die Namen von Threads, wenn noch keine Name vom Programmierer vergeben ist?
Geben Sie in der Klasse CCount den Namen des Threads aus!
(c.) Ändern Sie das Programm so ab, dass 10 Threads gestartet werden, wobei der 1. Thread mit Priorität 10, der 2. Thread mit Priorität 9 usw. laufen soll!
(d.) Starten Sie 5 Threads die von 1 bis 10 zählen und lassen Sie den 5. Thread 1 Sekunde, den 4. Thread 2 Sekunden usw. schlafen während der Zahlen 5 und 6 beim Hochzählen von 1 bis 10! Dazu sind notwendigerweise Änderungen in der Klasse CCounter notwendig!
(e.) Lassen Sie die 5 Threads hintereinander von 1 bis 10 zählen, wobei der 5. anfängt, von 1 bis 10 zählt, dann der 4. von 1 bis 10 usw.! Hinweis: Verwenden Sie NICHT synchronized, versuchen Sie selbst, beim Eintritt eines Threads in die run-Methode diese mit einem Lock zu sperren und am Ende wieder zu entsperrren!

(2.) Laden Sie das Programm aus dem Template Order und machen Sie sich mit der Wirkungsweise vertraut! Lassen Sie jede später kodierte Variante mehrfach laufen, um die Wirkung zu verizifieren!
(a.) Beschreiben Sie verbal das Problem des Buchungsablaufes!
(b.) Lösen Sie das Problem durch zufügen des Schlüsselwortes „synchronized“ an einer geeigneten Stelle!

(3.) Gegeben sind die Dateien (a.)-(g.)in dem Ordner 03!
Erklären Sie die Ausgabe!
Lassen Sie jedes Hauptprogramm 5 mal laufen! „Experimentieren(=Verändern)“ Sie gegebenenfalls den Quellkode, um die Wirkung zu verstehen!
zu (b.): Löschen Sie die sleep-Methode zwischen den beiden Blöcken, führen Sie das Programm ***nochmals*** *5-mal aus und interpretieren Sie erneut das Ergebnis!*
zu (c.): Ersetzen Sie
synchronized(Thread.class)
durch
synchronized(getclass())
und machen Sie eine Vorhersage, bevor Sie den modifizierten Quellkode 5-mal starten und das erwartetet Ergebnis vergleichen und evtl. Abweichungen erklären!

(4.) Gegeben ist ein Quellkode in dem Ordner 04\Template!
(a.) Bestimmen Sie alle Möglichkeiten, dass Ihnen das Programm bei ausgewählten Programmdurchläufen die Ausgabe wie unten ausgibt:

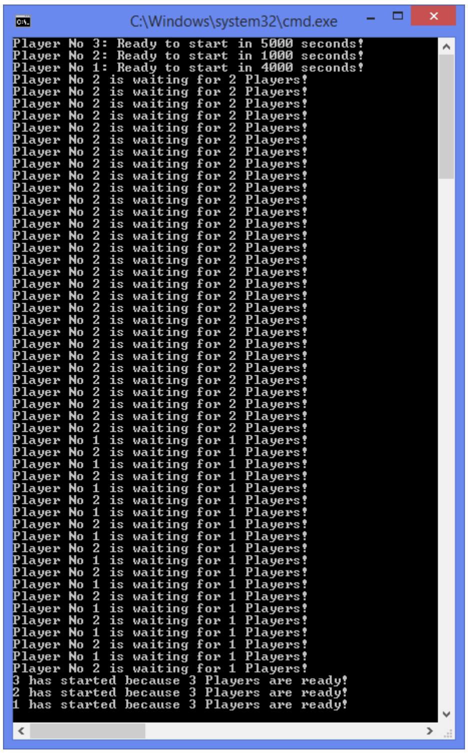


(b.) Welche Möglichkeit gibt es, das Hauptprogramm so zu verändern(OHNE die Methode run zu verändern), dass Bora Bora IMMER zuletzt ausgegeben wird?
Hinweis: Verwenden sie im aufrufenden Programm die Methode „Join" des Objektes Thread nachdem sie die Methode start aufgerufen haben(Join braucht zwingende „try-catch")!
Join: „Blockiert den aufrufenden Thread, bis der durch diese Instanz dargestellten Thread beendet wird."

(5.) Gegeben ist ein Quellkode in dem Ordner 05\Template!
Vereinfachen Sie den Quellkode und bestimmen Sie alle Möglichkeiten, dass Ihnen das Programm bei ausgewählten Programmdurchläufen die Ausgabe wie unten ausgibt:

(6.) Lassen Sie alle Programme des Ordners 06 laufen, recherchieren Sie warum die Ausgaben so erfolgen und Dokumentieren Sie ihre Ergebnisse in einer Powerpoint-Datei „14b - Threads – Advanced Features.pptx“. Verwenden Sie meine Folien als Vorlage!

(7.) In einem Multiplayer-Spiel kann nicht begonnen werden, bis eine Anzahl n von Spieler bereit ist. Simulieren Sie das Warten und lassen sie die Threads nach dem Start eine Meldung ausgeben:



Kodieren Sie eine Klasse, die dies simuliert!
Kodierung der aufrufenden Klasse:

```
public class TestPlayers
{
   //Main Thread
  public static void main(String[] args)
  {
    final int noOfPlayers=3;
    //Neues Array/Feld von Threads
    Thread[] trdPlayer = new Thread[noOfPlayers];

    for (int i = 0; i < noOfPlayers; i++) {
      // Für das neu erzeugte Array müssen wir in JEDEM Element Speicher für die Klasse reservieren!
      CPlayers clsPlayer = new CPlayers(noOfPlayers);
      // Klasse zufügen
      trdPlayer[i] = new Thread(clsPlayer);
      // Namen auf Nr. setzen
      trdPlayer[i].setName(String.valueOf(i+1));
      // Thread starten
      trdPlayer[i].start();
    }
  }
}
```

(8.) Ein Prozess arbeitet dabei als Produzent, der Fließkommazahlen »herstellt«, und ein anderer als Konsument, der die produzierten Daten »verbraucht«. Die Kommunikation zwischen beiden erfolgt über ein gemeinsam verwendetes Vector-Objekt, das die produzierten Elemente zwischenspeichert und als Monitor für die wait-/notify-Aufrufe dient. Da die Wartezeit zufällig ausgewählt wird, kann es durchaus dazu kommen, dass der Produzent eine größere Anzahl an Elementen anhäuft, die der Konsument noch nicht abgeholt hat. Erläutern Sie das Programm speziell die neuen Befehle!

(9.) Gegeben ist nachfolgender Quellkode:

```
class MyPrinter {
  public  void print1To3(int value) {
    for (int i = 1; i <= 3; i++) {
      System.out.print(value);
    }
    System.out.println("");
  }
}

class MyThread extends Thread {
  int no;
  MyPrinter p;
  MyThread(int no) {
    this.no = no;
    p = new MyPrinter();
  }
  public void run() {
    for (int i=1; i <= 10; i++) {
      p.print1To3(i);
    }
  }
}

class MyThreadDemo {
  public static void main(String args[]) {
    MyThread t1 = new MyThread(1);
    MyThread t2 = new MyThread(2);
    MyThread t3 = new MyThread(3);
    t1.start();
    t2.start();
    t3.start();
    System.out.print("Main ends");
  }
}
```

Ändern Sie den Quellkode, so dass die Ausgabe wie folgt lautet:

Die Idee ist, dass jeder Thread das gleiche Printer Objekt bekommt!

Threads - Lösungen

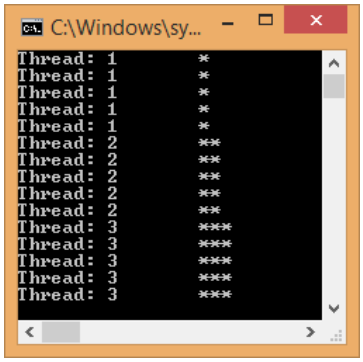
(1.)siehe Order

(2.)siehe Order

(3.)
(a.) Mit getClass() wird die gesamte Klasse gesperrt, so dass die einzelnen Threads(der gleichen Klasse) warten müssen, bis der aktive Thread beendet ist.

```
synchronized(getClass())
{
  for(int i=1; i<=5; i++)
  {
    System.out.println("Thread: "+text);
    try{
      Thread.sleep(200);
    }
    catch(Exception e)
    {};
  }
```

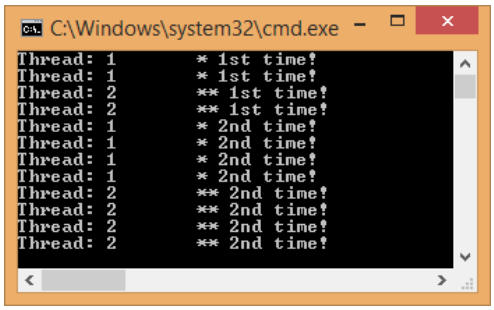
Im klassenweise synchronisierten Block wird die Wiederholungsanweisung bis zum Ende ausgeführt. Hier kann auch nicht ein anderer Thread die Ausführung starten, da bei dem Schlafen des aktiven Threads diese nicht den Programmblock freigibt



(b.) Mit getClass() wird die gesamte Klasse gesperrt allerdings immer nur für bestimmte Blöcke. Zwischen den synchronisierten Blöcken kann ein anderer Thread einen der beiden Programmkodeblöcke ausführen.

```
synchronized(getClass())
{
  for(int i=1;i<=2;i++){
    System.out.println("Thread: "+text+" 1st time!");
    try{Thread.sleep(200);} catch(Exception e){};
  }
}
try{
  Thread.sleep(10);
}
catch(Exception e)
{};
synchronized(getClass())
{
  for(int i=1;i<=4;i++){
    System.out.println("Thread: "+text+" 2nd time!");
    try{Thread.sleep(200);} catch(Exception e){};
  }
}
```

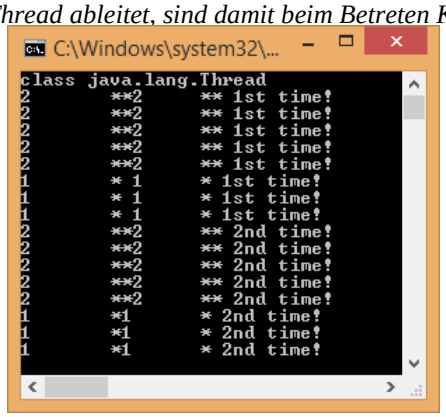
Zwischen den klassenweise synchronisierten Blöcken kann der nächste wartende Thread weiter ausgeführt werden. Die sleep-Methode ist nicht nötig, im anderen Fall passiert der Wechsel sonst aber selten(->ausprobieren!!!)!



(c.)

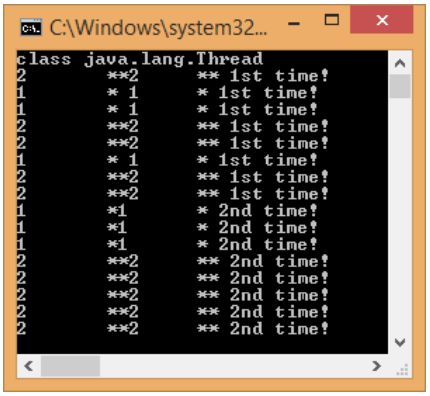
```
synchronized(Thread.class)
{
  for(int i=1;i<=5;i++){
    System.out.println("Thread: "+text+" 1st time!");
    try{Thread.sleep(200);} catch(Exception e){};
  }
}
try{
  Thread.sleep(10);
}
catch(Exception e)
{};
synchronized(Thread.class)
{
  for(int i=1;i<=5;i++){
    System.out.println("Thread: "+text+" 2nd time!");
    try{Thread.sleep(200);} catch(Exception e){};
  }
}
```

Bei Thread.class bezieht sich die Synchronisation auf die Klasse Thread! Da jedes Thread-Objekt sich von Thread ableitet, sind damit beim Betreten Kodes des Threads alle anderen Thread gesperrt!



(d.) Bei getClass() findet die Synchronisation nur für jede Klasse(klassenweise) statt! Da beide Threads Objekte unterschiedlicher Klassen verwenden, gibt es also keine Synchronisation zwischen diesen beiden Klassen!

```
synchronized(getClass())
{
  for(int i=1;i<=5;i++){
    System.out.println("Thread: "+text+" 1st time!");
    try{Thread.sleep(200);} catch(Exception e){};
  }
}
try{
  Thread.sleep(10);
}
catch(Exception e)
{};
synchronized(getClass())
{
  for(int i=1;i<=5;i++){
    System.out.println("Thread: "+text+" 2nd time!");
    try{Thread.sleep(200);} catch(Exception e){};
  }
}
```



(4.) (a.)
(b.) setPriority setzt nur die Verteilung der Zeitscheiben auf eine Priorität, garantiert aber nicht, dass zwingend ein Thread als erstes startet!

(7.) Eigene Lösung!