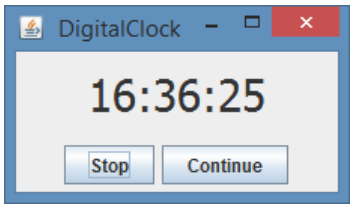
1. Implementieren Sie das *„Dining Philosophers"-Problem* laut besprochener Vorgabe, und vermeiden Sie den *Deadlock* indem Sie die vorgestellte Lösungsvariante mit der Klasse `ForkControl` implementieren.



2. Programmieren Sie die Klasse `JDigitalClock` abgeleitet von `JLabel`, welche das Interface `Runnable` implementiert und im Ausgabetext der Basisklasse `JLabel` im *Sekundentakt* die *aktuelle Uhrzeit* ausgibt. Diese Klasse soll die Methoden `setStopped(boolean stopped)` und `getStopped()` zur Verfügung stellen, durch welche die Aktualisierung der Uhrzeitausgabe *angehalten* werden kann. Programmieren Sie die Klasse so, dass im Zustand Angehalten keine Prozessorleistung benötigt wird!

   Implementieren Sie noch die *Benutzerschnittstelle*, welche Ihre Klasse `JDigitalCock` verwendet.

3. Ein *archäologisch bedeutsamer Tunnel* soll für die Öffentlichkeit zugänglich gemacht werden:

   a. Der Tunnel besitzt *zwei Eingänge* durch welche die Besucher nur in *Gruppen* – begleitet durch einen *Führer* – den Tunnel besichtigen können.

   b. An jedem *Eingang* befinden sich *vier Führer*.

   c. Aus Sicherheitsgründen dürfen *maximal 50 Besucher* gleichzeitig im Tunnel sein.

   d. Ein Führer begleitet eine Gruppe von seinem Eingang aus in den Tunnel und kehrt mit der Gruppe zu seinem Eingang wieder zurück bevor er mit einer weiteren Gruppe zur Besichtigung startet.

   Sie sollen eine *Netzwerkanwendung* entwickeln, welches an den zwei Eingängen den Zugang in den Tunnel verwaltet, so dass die obigen Randbedingungen eingehalten werden.

   Teilen Sie die einzelnen Programmfunktionalitäten client- und serverseitig folgendermaßen auf *Klassen* auf (Dokumentation wird mitgeliefert):

net.tfobz.tunnel.client	net.tfobz.tunnel.server
ClientForm	ServerMain
ClientThread	ServerThread
GuidesMonitor	VisitorsMonitor

`ClientForm`
Diese Klasse erstellt die *Benutzerschnittstelle* und den `GuidesMonitor` zur Verwaltung der Gruppenführer pro Eingang. Sie enthält auch die Ereignisbehandlungsmethoden für die beiden Knöpfe. In diesen Methoden werden die Objekte vom Typ `ClientThread` zur Behandlung der Clientanfragen angelegt und die Threads gestartet.

`ClientThread`
Jede Anfrage um Start einer Besichtigung oder Beendigung einer solchen muss in einem eigenen Thread durchgeführt werden, da insbesondere bei nicht Vorhandensein eines Führers oder bei nicht verfügbarem Besucherkontingent eine solche Anfrage längere Zeit warten und deshalb das `ClientForm` blockiert würde.
Damit der Thread seine Aufgabe durchführen kann, muss er einerseits den `GuidesMonitor` als Referenz enthalten, um die Führeranforderung zu stellen. Andererseits muss er `ClientForm` kennen, um die Ausgaben und Anpassungen an der Benutzerschnittstelle vornehmen zu können.
Der Thread erhält die *Besucheranzahl*. Ist diese *positiv*, so fordert er zuerst beim `GuidesMonitor` einen Führer für die Gruppe an. Erhält er diese, so wird über eine *Netzwerkverbindung* mit dem `ServerMain`-Programm Verbindung aufgenommen und um die eingegebene Anzahl von Besuchern angefragt. Ist die

Besucheranzahl *negativ*, so bedeutet dies, dass die Führung beendet wird, der Führer dem GuidesMonitor zurückgegeben und beim Server ebenfalls die Besucheranzahl zurückgegeben wird. Ist die Besucheranzahl *gleich 0*, so wird der Thread anweisen beim Server die Anzahl der verfügbaren Besucher nachzufragen, die noch im Tunnel Platz haben.

GuidesMonitor
An ihm kann ein *Führer angefordert* aber auch ein solcher zurückgegeben werden. Dieser muss eine *Referenz* auf ClientForm haben, damit die Statusmeldungen dort angezeigt werden können.

ServerMain
In dieser *Konsolenanwendung* wird zuerst ein VisitorsMonitor angelegt, und dann wartet das Programm in einer *Endlosschleife* auf Clientanfragen. Erreicht ihm eine solche, so wird diese in einem Thread vom Typ ServerThread abgearbeitet. Dadurch dass jede Anfrage in einem eigenen Thread abgearbeitet wird, können mehrere Anfragen gleichzeitig bearbeitet werden.

ServerThread
Der Thread liest vom *Socket* die *Anzahl*, und dabei werden die drei Fälle – *größer 0*, *kleiner 0* oder *gleich 0* – unterschieden und entsprechen am VisitorsMonitor die Anfragen gestellt. Das Ergebnis wird an den Client zurück geschickt. Der ServerThread erhält den Socket des Clients und eine *Referenz* auf VisitorsMonitor.

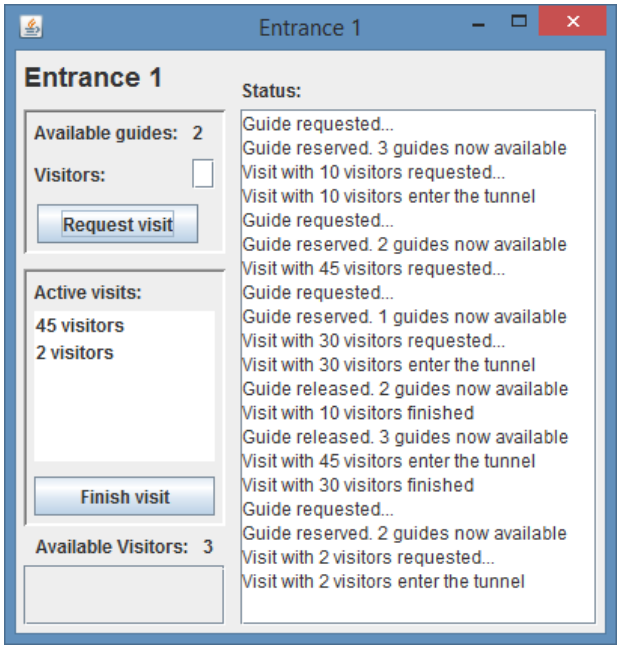
VisitorsMonitor
Dieser verwaltet die *verfügbaren Besucher*, welche eingelassen werden können.

Die *Benutzerschnittstelle* für die *beiden Clients* soll wie abgebildet gestaltet werden:

e. Es soll eingegeben werden, aus *wie vielen Besuchern* die in den Tunnel einzulassende Gruppe besteht. Dann soll es möglich sein, vom Server die Erlaubnis zu erhalten, die Gruppe in den Tunnel einzulassen.

f. In einer Liste werden alle momentan von diesem Eingang aus eingelassenen Gruppen geführt. Wenn eine Gruppe nach der Besichtigung den Tunnel mit ihrem Führer verlässt, so soll es möglich sein, die *Besichtigung zu beenden* und dem Server die Besucheranzahl zurück zu geben.

g. Weiters soll periodisch – jede Sekunde – überprüft werden, wie viele Besucher insgesamt noch in den Tunnel eingelassen werden können. Die Anzahl wird unter *verfügbare Besucher* ausgegeben.



h. Rechts im *Statusfeld* werden alle Operationen mit protokolliert.

Die Benutzerschnittstelle für die *Serveranwendung* soll durch eine einfache Konsolenanwendung realisiert werden, und lediglich die einzelnen Serveraktionen mit protokollieren.

```
S E R V E R
===========
50 available visitors
/127.0.0.1 requests 10 visitors
/127.0.0.1 receives 10 visitors. 40 visitors available
/127.0.0.1 requests 45 visitors
/127.0.0.1 requests 30 visitors
/127.0.0.1 receives 30 visitors. 10 visitors available
/127.0.0.1 releases 10 visitors. 20 visitors available
/127.0.0.1 releases 30 visitors. 50 visitors available
/127.0.0.1 receives 45 visitors. 5 visitors available
/127.0.0.1 requests 2 visitors
/127.0.0.1 receives 2 visitors. 3 visitors available
```