Programação Concorrente
(Inverno 2018/2019)
1ª Lista de Exercícios de Preparação

1. Usando monitores Java ou CLI, implemente o sincronizador *future holder* usado para hospedar dados resultantes de cálculos demorados. A interface pública do sincronizador é apresentada de seguida em Java.

```
public class FutureHolder<T> whwre T: class {
  public void setValue(T value);
  public T getValue(long timeout);
  public boolean isValueAvailable();
}
```

   A operação `getValue` bloqueia as *threads* invocantes até que os dados sejam disponibilizados através da chamada a `setValue`. Como as instâncias desta classe são de utilização única, chamadas subsequentes a `setValue` produzem excepção (`IllegalStateException`). A operação `getValue` retorna os dados hospedados, ou `null` caso ocorra *timeout*. O sincronizador suporta cancelamento das *threads* em espera.

2. Usando os monitores disponíveis nas linguagens C# ou Java , implemente o sincronizador `AutoResetEvt`, cuja interface pública, em Java , e a semântica de sincronização se descrevem a seguir.

```
class AutoResetEvt {
  public boolean await(long timeout) throws InterruptedException;
  public void signal();
  public void pulseAll();
}
```

   O sincronizador é semelhante ao evento de *reset* automático do Windows. O método `signal`: coloca o estado do evento como sinalizado no caso de não existir nenhuma *thread* em espera ou liberta a *thread* há mais tempo em espera (FIFO). O método `pulseAll` só tem efeito se o estado do evento for não sinalizado e existirem *threads* em espera. Nesse caso acorda todas as *threads* em espera voltando de imediato ao estado não sinalizado. A chamada ao método `await` retorna de imediato se o evento estiver sinalizado fazendo *reset* ao estado de sinalização ou bloqueia a *thread* invocante até que: (a) ocorra uma notificação por invocação do método `signal` ou do método `pulseAll`; (b) expire o limite de tempo de espera especificado, ou; o bloqueio da *thread* seja interrompido.

3. Implemente em C# ou Java, com base nos monitores implícitos, o sincronizador `advertising panel` que suporta a afixação de mensagens publicitárias pelas *threads* editoras, que ficarão expostas para consumo, por parte das *threads* consumidoras, durante o intervalo de tempo especificado,. A interface pública deste sincronizador em C# é a seguinte.

```
public class AdvertisingPanel<M> where M : class {
  public void Publish(M message, int exposureTime);
  public M Consume(int timeout); // throws ThreadInterruptedException
}
```

A operação `Publish` publica uma mensagem publicitária definindo os respectivos conteúdo e tempo de exposição (em milissegundos). O painel pode ter apenas uma mensagem afixada de cada vez, pelo que a publicação de uma mensagem pode substituir a mensagem exposta anteriormente. Sempre que é publicada uma mensagem, esta tem que ser obrigatoriamente entregue a todas as *threads* que se encontrem bloqueadas, mesmo quando o tempo de exposição da mensagem for zero (mensagens transitórias). As *threads* que pretendam consumir mensagens publicitárias invocam o método `Consume`, cuja execução poderá terminar: (1) devolvendo a instância do tipo `M` que contém uma mensagem publicitária válida; (2) devolvendo `null`, se expirar o intervalo de tempo especificado pelo argumento timeout, ou; (3) lançando `ThreadInterruptedException`, se a espera da *thread* for interrompida.

4. Implemente em C# a classe `ExpirableLazy` com a seguinte interface pública

```
public class ExpirableLazy where T:class {
   public ExpirableLazy(Func provider, TimeSpan timeToLive);
   public T Value {get;} // throws InvalidOperationException,ThreadInterruptedException
}
```

Esta classe implementa uma versão da classe `System.Lazy`, pertencente à plataforma .NET, thread-safe e com limitação no tempo de vida, especificado através do parâmetro `timeToLive`, do valor calculado. O acesso à propriedade `Value` deve ter o seguinte comportamento:
(a) caso o valor já tenha sido calculado e o seu tempo de vida ainda não tenha expirado, retorna esse valor;(b) caso o valor ainda não tenha sido calculado ou o tempo de vida já tenha sido ultrapassado, inicia o cálculo chamando `provider` na própria *thread* invocante e retorna o valor resultante; (c) caso já exista outra thread a realizar esse cálculo, espera até que o valor esteja calculado; (d) lança `ThreadInterruptedException` se a espera da *thread* for interrompida.
Caso a chamada a provider resulte numa excepção: (a) a chamada a `Value` nessa *thread* deve resultar no lançamento dessa excepção; (b) se existirem outras *threads* à espera do valor, deve ser seleccionada uma delas para a retentativa do cálculo através da função `provider`. Não existe limite no número de retentativas.
O tempo de vida inicia-se quando o valor é retornado da função `provider`.

5. Implemente em C# o sincronizador exchanger `Exchanger<T>` que permite a troca, entre pares de *threads*, de mensagens definidas por instâncias do tipo T. A classe que implementa o sincronizador deve definir, pelo menos, o método `bool Exchange(T mine, int timeout, out T yours)`, que é chamado pelas *threads* para oferecer uma mensagem (parâmetro mine) e receber a mensagem oferecida pela *thread* com que emparelham (parâmetro yours). Quando a troca de mensagens não pode ser realizada de imediato (não existe já uma *thread* bloqueada), a *thread* corrente fica bloqueada até que outra *thread* invoque o método `Exchange`, seja interrompida ou expire o limite de tempo, especificado através do parâmetro timeout.