



iscte

INSTITUTO
UNIVERSITÁRIO
DE LISBOA



emprego
digital

Módulo 4: Laboratório de Programação

Aula 3

Cadeados, Barreiras, Semáforos e outros sistemas de controlo



Na aula de hoje

Agora que já sabemos o que são threads, como podemos fazer a sua coordenação e sincronização, avancemos para exemplos mais avançados:

- Problemas habituais de programação concorrente
- Cadeados externos
- Semáforos
- Barreiras
- BlockingQueue (Filas bloqueantes)

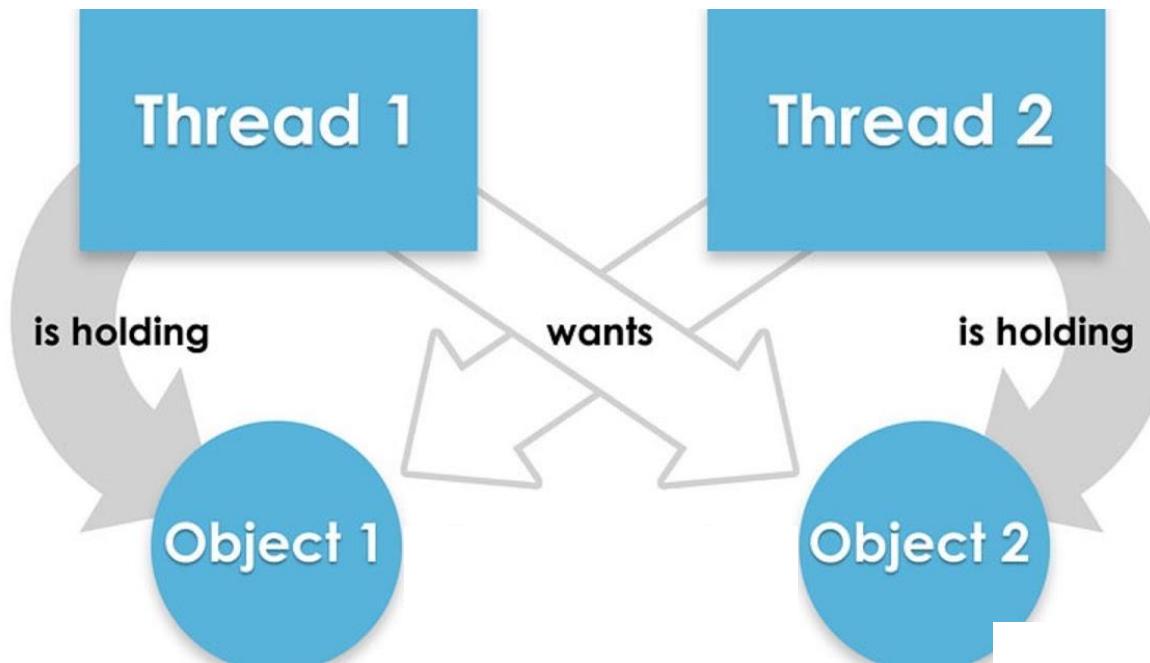
Problemas de Sincronização

- A programação concorrente acarreta alguns desafios, que precisam de ser tidos em conta durante o desenvolvimento do programa. Os mais comuns são:
 - **Deadlock**
 - **Livelock**
 - **Starvation**

Deadlock

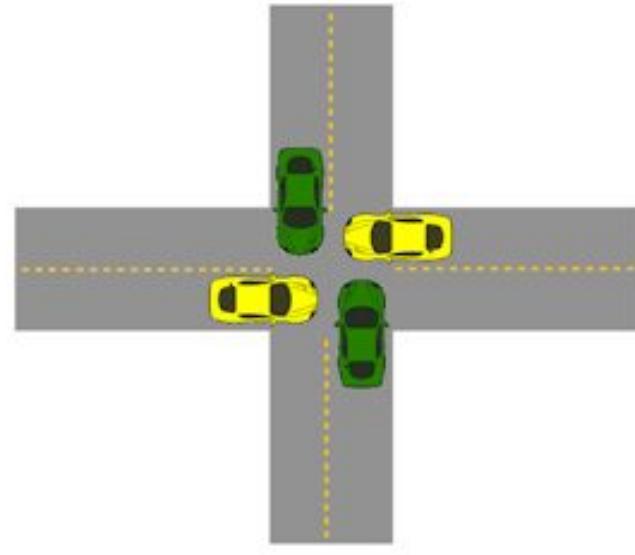
- Imagine que tem uma **thread A**, que aguarda por uma **thread B**, que por sua vez aguarda pela **thread A**. O que acontece? Ambas ficam à espera uma da outra, para sempre.
 - **A isto chamamos Deadlock**
- O impasse acontece quando um pedido se encontra num estado onde duas ou mais threads estão cada uma à espera da outra para poderem libertar um recurso.
- Este é dos principais e mais difíceis problema para resolver em programação concorrente.

Deadlock



Livelock

- Um livelock é **semelhante a um deadlock** excepto que as threads estão em estado ativo em vez de estarem em espera.
- Em contraste ao Deadlock, os recursos partilhados saltam de “mão em mão” mas **sempre num estado que impossibilita o avanço do programa.**



Starvation

- A Starvation acontece quando uma thread não é capaz de obter acesso aos recursos partilhados (fica sempre para trás, outras threads passam sempre à frente).
- Isto pode acontecer quando as seções mais críticas são grandes, forçando outras threads a esperar e reduzir o nível de concorrência.

Vamos conhecer outras estruturas de sincronização e coordenação.

Para além das estruturas base de sincronização e coordenação (`wait()`, `notify()`, `notifyAll()`, `join(...)`) o Java possui objetos de utilização simples, que permitem sincronizar e coordenar threads de forma mais complexa.

Cadeados externos - **ReentrantLock**

- ReentrantLock é uma classe que fornece sincronismo aos métodos enquanto acede aos recursos partilhados.
- O código que manipula o recurso partilhado deve estar envolvido de chamadas para bloquear e desbloquear o método.
- Isto permite bloquear todas as threads que pretendam aceder ao recurso partilhado excepto aquela que estiver a correr no momento.
- **Pode ser usado em alternativa aos cadeados intrínsecos do Java (keyword synchronized).**

ReentrantLock - Métodos

- **lock()**: este método aumenta a contagem da espera em 1 e **obtém** o cadeado se o recurso partilhado estiver livre. Caso não esteja, fica em **wait**.
- **unlock()**: diminui a contagem de espera em 1. Quando esta contagem chega a zero, o recurso é libertado.
- **tryLock()**: Se o recurso não for mantido por qualquer outra thread, então a chamada para tryLock() retorna verdadeiro e a contagem de retenção é incrementada em um. Se o recurso não estiver livre, então o método retorna falso e a thread não é bloqueada, mas sai.

ReentrantLock - Exemplo

```
ReentrantLock l = new ReentrantLock();  
  
l.lock();  
//Código a executar  
l.unlock();
```

Apesar de existir alguma utilidade na utilização deste tipo de cadeados, o ReentrantLock habitualmente não é muito usado, em detrimento das **soluções base de sincronismo do Java**, que permitem resolver o mesmo problema.

Semáforos

- Um Semáforo é uma estrutura que guarda um conjunto de **autorizações**. Sempre que alguém tenta entrar numa zona protegida por um Semáforo, isso apenas é possível se existir uma **autorização disponível**.
- Ao entrar, o número de autorizações disponíveis é **diminuído em 1**.
- Ao sair, o número de autorizações disponíveis **aumenta em 1**.
- Ao criar o Semáforo, podemos escolher com quantas autorizações este começa.

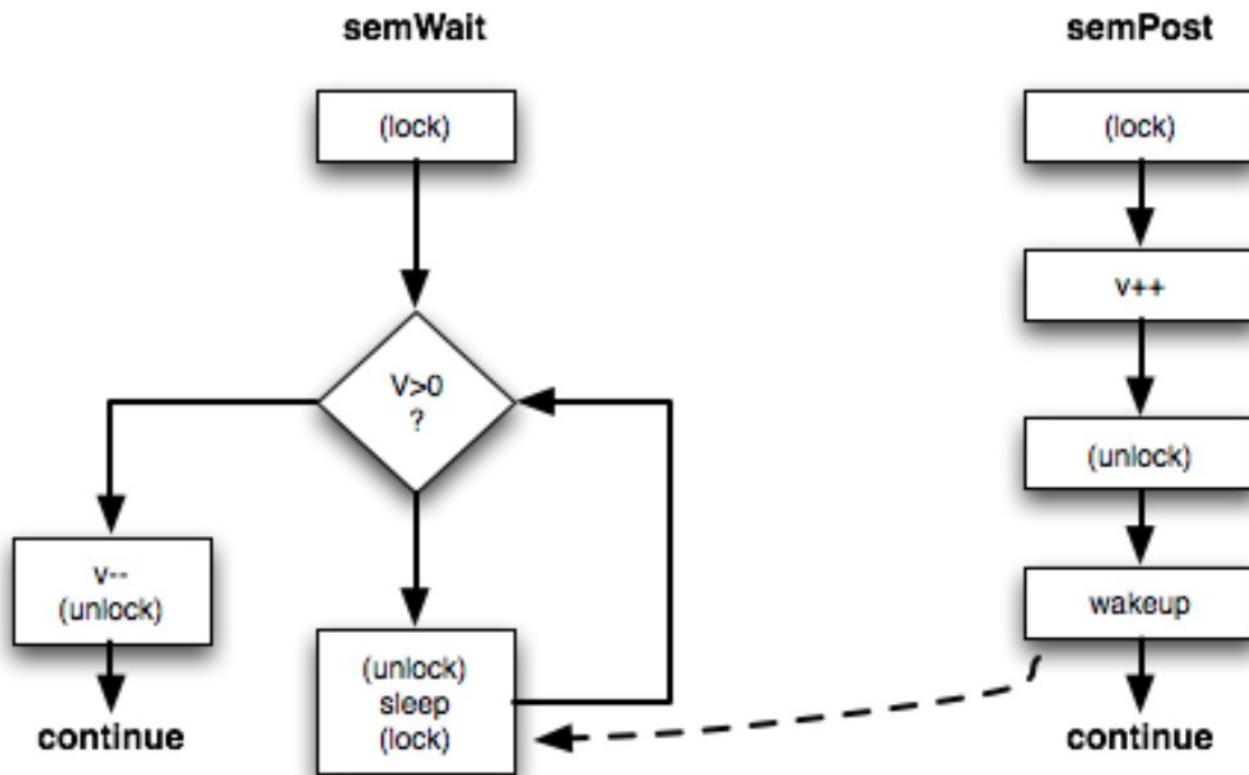
Em Java, o semáforo é implementado
pela Classe **Semaphore**.

<https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/Semaphore.html>

Semaphore - Métodos

- **semWait()**: uma thread tenta obter uma autorização do semáforo. Se existirem atualizações disponíveis, o método tem retorno. Se a contagem for zero, a thread vai esperar até que outra thread incremente a contagem do semáforo.
- **semPost()**: aumenta o número de autorizações do semáforo e desperta as threads que estiverem à espera, se existirem. Pode ser utilizado ao sair da zona protegida pelo semáforo, **mas não só!**

Semaphore



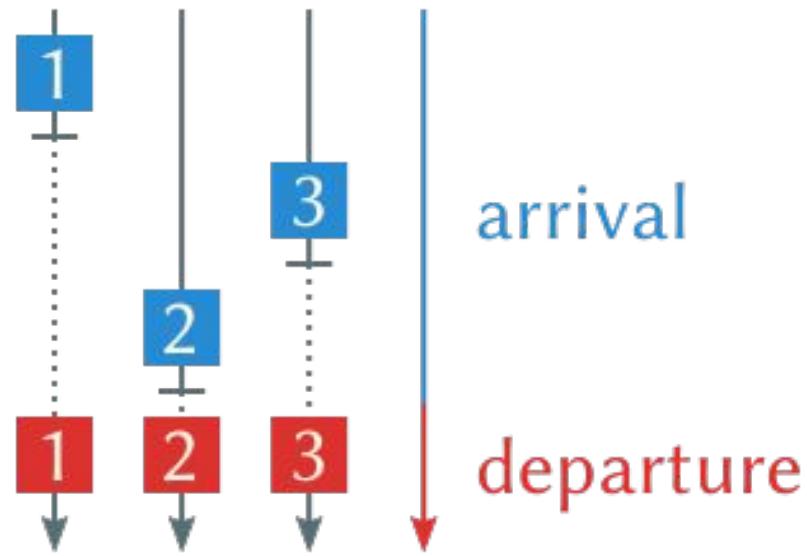
Como funciona o **Semaphore**?

```
public class Semaphore {  
    ...  
    public synchronized void semWait(){  
        boolean interrupted = false;  
        while (count == 0) {  
            try { wait(); }  
            catch (InterruptedException ie) {interrupted = true;}  
        }  
        count --;  
        if (interrupted) Thread.currentThread().interrupt();  
    }  
    public synchronized void semPost(){  
        count ++;  
        notifyAll();  
    }  
}
```

Barreiras

- Uma barreira permite que um conjunto de threads esperem umas pelas outras. Esta coordenação é conseguida chamando um método da barreira.
- A barreira é inicializada com o número de threads a bloquear (N).
- As threads são bloqueadas na barreira **até todas as N threads terem chegado**. Nesse momento **são todas libertadas**.
- A ideia é definir um **ponto de coordenação** onde um conjunto de threads só avança após todas as N threads terem chegado a esse ponto.

Barreiras



Em Java, a Barreira é implementada pela Classe **CyclicBarrier**.

<https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/CyclicBarrier.html>

CyclicBarrier

- É a implementação de Barreiras em Java.
- São bastante utilizadas em problemas que utilizem um conjunto fixo de threads que esperam umas pelas outras.
- **Tem o nome de cíclica pois a Barreira pode ser reutilizadas depois das threads serem libertadas.**

CyclicBarrier

- Pode receber um runnable Runnable opcional que é executado uma vez por ponto de barreira, após a chegada da última thread, mas antes que esta seja lançada.
- **Esta ação de barreira é útil para atualizar o estado partilhado antes de qualquer uma das partes continuar!**

Como funciona a CyclicBarrier?

```
class CyclicBarrier {  
    private int numberWaiters;  
    private int currentWaiters = 0;  
    private int passedWaiters = 0;  
  
    public synchronized void barrierWait() throws InterruptedException {  
        currentWaiters++;  
  
        while (currentWaiters < numberWaiters) {  
            wait();  
        }  
  
        if(passedWaiters == 0)  
            notifyAll();  
        passedWaiters++;  
  
        if(passedWaiters == numberWaiters){  
            passedWaiters = 0;  
            currentWaiters = 0;  
        }  
    }  
}
```

O primeiro a ser libertado notifica os restantes.

Todos assinalam a sua passagem.

Se já todos passaram pela barreira, esta fecha novamente.

Filas Bloqueantes - **BlockingQueue**

- Uma BlockingQueue é uma fila que coloca em wait as threads se a queue estiver vazia ou se não existir mais capacidade para colocar um novo elemento.
- Disponibiliza os seguintes métodos:
 - **put(e)**: permite adicionar um elemento “e” à queue. Caso a queue esteja cheia, a thread fica em **wait** até que seja retirado um elemento.
 - **take()**: retira o primeiro elemento da queue. Caso a queue esteja vazia, a thread fica em wait até que um novo elemento seja colocado.

Filas Bloqueantes - **BlockingQueue**

- Em Java, a interface BlockingQueue é implementada pelas seguintes classes:
 - **ArrayBlockingQueue<E>**: com as características de uma ArrayList.
 - **LinkedBlockingQueue<E>**: com as características de uma LinkedList.
 - **PriorityBlockingQueue<E>**: com as características de uma PriorityQueue, definido uma ordem de acordo com um comparador

Exercício 1 - “Jantar de Filósofos”

Cinco filósofos que representam threads de interação, vêm à mesa e executam o seguinte processo:

```
while(true){  
    pensar();  
    pegarGarfo();  
    comer();  
    pousarGarfo();  
}
```



Exercício 1 - “Jantar de Filósofos”

Os garfos representam recursos que as threads têm de segurar exclusivamente para progredir. Apenas existem 5 garfos para os 5 filósofos. A coisa que torna o problema interessante, irrealista e pouco higiénico, é que os filósofos precisam de dois garfos para comer, por isso um filósofo esfomeado pode ter de esperar que um vizinho pouse um garfo.

- Apenas um filósofo pode segurar um garfo de cada vez.
- Não deve ser possível a ocorrência de deadlocks.
- Não deve ser possível um filósofo morrer à fome à espera por um garfo.
- Deve ser impossível que mais do que um filósofo comer ao mesmo tempo.

Resolução Exercício 1 - Garfo

```
class Fork {  
  
    private int id;  
    private boolean inUse;  
  
    public Fork(int id){  
        this.id=id;  
        this.inUse=false;  
    }  
  
    public synchronized void up(){  
        while (inUse) {  
            try {  
                wait();  
            } catch (InterruptedException e) {}  
        }  
        inUse = true;  
    }  
  
    public synchronized void down(){  
        inUse=false;  
        notifyAll();  
    }  
}
```

Resolução Exercício 1 - Filósofo

```
public class Philosopher extends Thread {  
    int id, times_eat=0;  
    Fork leftFork, rightFork;  
    public Philosopher(int id,Fork leftFork,Fork rightFork){  
        this.id=id;  
        this.leftFork=leftFork;  
        this.rightFork=rightFork;  
    }  
    public void run(){  
        try {  
            while (times_eat != 5) {  
                thinking(); //sleep durante 500ms (ou outro valor)  
                leftFork.up();  
                rightFork.up();  
                eating(); //sleep durante 500ms (ou outro valor)  
                leftFork.down();  
                rightFork.down();  
            }  
        } catch (InterruptedException e){}  
    }  
}
```

Resolução Exercício 1 - Main

```
public static void main(String[] args){  
    Fork f1=new Fork(1);  
    Fork f2=new Fork(2);  
    Fork f3=new Fork(3);  
    Fork f4=new Fork(4);  
    Fork f5=new Fork(5);  
  
    new Philosopher(1,f1,f2).start();  
    new Philosopher(2,f2,f3).start();  
    new Philosopher(3,f3,f4).start();  
    new Philosopher(4,f4,f5).start();  
    new Philosopher(5,f5,f1).start();  
}
```

Exercício 1 - Notas

Este exercício merece uma reflexão aprofundada. Ele representa um problema conceptual, formulado em 1965, e é habitualmente utilizado para demonstrar os problemas que podem acontecer em programação concorrente.

Particularmente, o problema foi desenhado para ilustrar o desafio de evitar uma situação de **deadlock**, onde o sistema se encontra bloqueado, não conseguindo sair desse estado.

Para demonstrar um cenário onde isso pode acontecer, considere uma solução na qual cada filósofo se comporta da forma descrita no slide seguinte.

Exercício 1 - Deadlock

Imaginemos que os filósofos se comportam da seguinte forma:

- Pegar no garfo esquerdo;
- Pegar no garfo direito;
- Comer;
- Pousar o garfo direito;
- Pousar o garfo esquerdo;
- Repetir

Ora, esta sequência de acções levaria-nos a um estado de deadlock. Este é um estado em que cada filósofo pegou o garfo da esquerda e espera que o garfo da direita fique disponível. Assim, pode ocorrer um deadlock e cada filósofo aguardará eternamente por outro (o da direita) para liberar um garfo, o que não acontecerá pois esse filósofo também estará a aguardar pelo garfo à sua direita.

Exercício 1 - Deadlock

Imag

Como resolver?

- P
 - P
 - C
 - P
 - P
 - P
 - P
 - P
- Por exemplo, se os filósofos “pares” pegarem primeiro no garfo esquerdo e só depois no garfo direito e os filósofos “ímpares” fizerem o contrário, conseguimos impedir este problema.
- Esta é a razão pela qual o main está implementado desta forma:

Ora,

m estado

em d

eita fique

dispo

amente

por ou

filósofo

também estará a aguardar pelo garfo à sua direita.

Exercício 2 - Banquete

Num banquete de javalis assados existem vários cozinheiros e glutões concorrentes e uma mesa com capacidade limitada (só cabem na mesa 10 javalis).

Cada cozinheiro repetidamente produz um javali assado e coloca-o na mesa. Caso encontre a mesa cheia fica à espera que a mesa fique com espaço disponível para depositar o seu javali.

Cada glutão repetidamente retira da mesa um javali e consome-o. Caso encontre a mesa vazia fica à espera que a mesa contenha de novo algum javali. Sincronize o recurso mesa por forma a coordenar devidamente os vários cozinheiros e glutões. Utilize ainda mensagens apropriadas para observação do comportamento dos vários atores no banquete.

Cada Javali deve ser identificado pelo cozinheiro que o produziu e por um número de ordem.

Resolução Exercício 2 - Javali

```
public class Javali {  
  
    private static int LATEST_ID = 0;  
    private Cozinheiro cozinheiro;  
    private int id;  
  
    public Javali(Cozinheiro cozinheiro){  
        id = getAndIncrementLastestId();  
        this.cozinheiro = cozinheiro;  
    }  
  
    public int getID() {  
        return id;  
    }  
  
    public synchronized static int getAndIncrementLastestId(){  
        return LATEST_ID++;  
    }  
}
```

Resolução Exercício 2 - Mesa

```
public class Mesa {  
  
    private List<Javali> javalis = new ArrayList<Javali>();  
    private int maxSize;  
  
    public Mesa(int maxSize){  
        this.maxSize = maxSize;  
    }  
  
    public synchronized void colocaJavali(Javali j) throws InterruptedException {  
        while(javalis.size()>=maxSize){  
            wait();  
        }  
        javalis.add(j);  
        notifyAll();  
    }  
  
    public synchronized Javali retiraJavali() throws InterruptedException {  
        while(javalis.size()==0){  
            wait();  
        }  
        Javali j = javalis.remove(0);  
        notifyAll();  
        return j;  
    }  
}
```

Resolução Exercício 2 - Glutão

```
public class Glutao extends Thread{  
  
    private String nome;  
    private Mesa mesa;  
  
    public Glutao(String nome, Mesa mesa) {  
        this.nome = nome;  
        this.mesa = mesa;  
    }  
  
    public String getNome() {  
        return nome;  
    }  
  
    @Override  
    public void run() {  
        while(true){  
            try {  
                Javali j = mesa.retiraJavali();  
                sleep(1000); //comer  
                System.out.println(nome + " comeu o Javali " + j.getID());  
            } catch (InterruptedException e) {}  
        }  
    }  
}
```

Resolução Exercício 2 - Cozinheiro

```
public class Cozinheiro extends Thread{  
  
    private String nome;  
    private Mesa mesa;  
  
    public Cozinheiro(String nome, Mesa mesa){  
        this.nome = nome;  
        this.mesa = mesa;  
    }  
  
    @Override  
    public void run() {  
        while(true){  
            try {  
                sleep(1000); //cozinha  
                Javali j = new Javali(this);  
                System.out.println(nome + " cozinhou o Javali " + j.getID());  
                mesa.colocaJavali(j);  
            } catch (InterruptedException e) {}  
        }  
    }  
}
```

Resolução Exercício 2 - Main

```
public static void main(String[] args) {  
    Mesa m = new Mesa(10);  
  
    Cozinheiro c1 = new Cozinheiro("Cozinheiro 1", m);  
    Cozinheiro c2 = new Cozinheiro("Cozinheiro 1", m);  
    Cozinheiro c3 = new Cozinheiro("Cozinheiro 1", m);  
  
    Glutao g1 = new Glutao("Glutao 1", m);  
    Glutao g2 = new Glutao("Glutao 1", m);  
    Glutao g3 = new Glutao("Glutao 1", m);  
  
    c1.start();  
    c2.start();  
    c3.start();  
  
    g1.start();  
    g2.start();  
    g3.start();  
}
```

Exercício 3 - Banquete II

Resolva novamente o exercício anterior, desta vez utilizando uma Blocking Queue.

Resolução Ex.3 - MesaBlockingQueue

```
public class MesaBlockingQueue extends Mesa {  
    private BlockingQueue<Javali> javalis;  
    public MesaBlockingQueue(int maxSize){  
        super(maxSize);  
        javalis = new ArrayBlockingQueue<Javali>(maxSize);  
    }  
    public synchronized void colocaJavali(Javali j) throws InterruptedException {  
        javalis.put(j);  
    }  
    public synchronized Javali retiraJavali() throws InterruptedException {  
        return javalis.take();  
    }  
}
```

Resolução Exercício 3 - Main

```
public static void main(String[] args) {  
    MesaBlockingQueue m = new MesaBlockingQueue(10);  
  
    Cozinheiro c1 = new Cozinheiro("Cozinheiro 1", m);  
    Cozinheiro c2 = new Cozinheiro("Cozinheiro 1", m);  
    Cozinheiro c3 = new Cozinheiro("Cozinheiro 1", m);  
  
    Glutao g1 = new Glutao("Glutao 1", m);  
    Glutao g2 = new Glutao("Glutao 1", m);  
    Glutao g3 = new Glutao("Glutao 1", m);  
  
    c1.start();  
    c2.start();  
    c3.start();  
  
    g1.start();  
    g2.start();  
    g3.start();  
}
```

Exercício 4 - Transportadora

Uma transportadora é responsável por fazer o envio de um conjunto de encomendas. Por questões de eficiência, as encomendas têm de ser enviadas em conjuntos de 3. Ou seja, apenas quando são efetuadas 3 encomendas, a transportadora faz o envio de todas, para os respetivos destinatários. O cliente deve aguardar que a encomenda seja enviada.

Assuma que existem 3 clientes (threads) a efetuar encomendas em intervalos aleatórios entre 3 e 6 segundos. Sugestão: utilize uma barreira para representar o sistema de envio da transportadora.

Resolução Exercício 4 - Transportadora

```
public class Transportadora {  
  
    List<Encomenda> encomendasEmEspera = new ArrayList<Encomenda>();  
    CyclicBarrier barrier;  
  
    public Transportadora(){  
        barrier = new CyclicBarrier(3);  
    }  
  
    public void envia(Encomenda e) throws BrokenBarrierException, InterruptedException {  
        barrier.await();  
        //se a barreira libertou foi porque 5 threads fizeram o await  
        System.out.println("A encomenda " + e.getID() + " foi enviada.");  
    }  
}
```

Resolução Exercício 4 - Encomenda

```
class Encomenda {  
  
    private static int LATEST_ID = 0;  
    private int id;  
  
    public Encomenda(){  
        id = getAndIncrementLastestId();  
    }  
  
    public int getID() {  
        return id;  
    }  
  
    public synchronized static int getAndIncrementLastestId(){  
        return LATEST_ID++;  
    }  
}
```

Resolução Exercício 4 - Cliente

```
class Cliente extends Thread {  
    private String nome;  
    private Transportadora transportadora;  
    public Cliente (String nome, Transportadora transportadora){  
        this.nome = nome;  
        this.transportadora = transportadora;  
    }  
    @Override  
    public void run() {  
        while(true){  
            Random random = new Random();  
            int time = random.nextInt(3000) + 3000;  
            try {  
                sleep(time);  
            } catch (InterruptedException e) {}  
            Encomenda e = new Encomenda();  
            try {  
                System.out.println(nome + " envia encomenda " + e.getID());  
                transportadora.envia(e);  
            } catch (...) { ... }  
        }  
    }  
}
```

Resolução Exercício 4 - Main

```
public static void main(String[] args) {  
  
    Transportadora t = new Transportadora();  
  
    Cliente c1 = new Cliente("Cliente 1", t);  
    Cliente c2 = new Cliente("Cliente 2", t);  
    Cliente c3 = new Cliente("Cliente 3", t);  
  
    c1.start();  
    c2.start();  
    c3.start();  
  
}
```

Exercício 5 - Server Room

Uma empresa de informática possui na sua sede uma sala de acesso restrito, onde se encontram os servidores com os sistemas dos seus clientes. Por questões de logística, essa sala só pode ser acedida por 2 pessoas em simultâneo. Assim, num momento inicial existem 3 permissões de entrada, sendo retirada uma permissão sempre que alguém entrar na sala e adicionada uma permissão sempre que alguém sair, neste caso permitindo a entrada de mais uma pessoa.

Assuma que existem 3 funcionários (threads) a tentar aceder constantemente à sala do servidor. Lá dentro, a execução do seu trabalho deve durar entre 5 e 8 segundos. Sugestão: utilize um semáforo para representar o sistema de controlo de acesso à sala do servidor.

Resolução Exercício 5 - Sala

```
public class Sala {  
    private Semaphore semaphore;  
  
    public Sala(){  
        semaphore = new Semaphore(2);  
    }  
  
    public void entrar() throws InterruptedException {  
        semaphore.acquire();  
    }  
  
    public void sair(){  
        semaphore.release();  
    }  
}
```

Resolução Exercício 5 - Funcionário

```
public class Funcionario extends Thread {  
    private String nome;  
    private Sala sala;  
    public Funcionario(String nome, Sala sala){  
        this.nome = nome;  
        this.sala = sala;  
    }  
    @Override  
    public void run() {  
        while(true){  
            try {  
                System.out.println(nome + " aguarda entrada na sala...");  
                sala.entrar();  
                System.out.println(nome + " entrou na sala e começou a trabalhar...");  
                Random r = new Random();  
                sleep(r.nextInt(5000) + 8000);  
                System.out.println(nome + " terminou o trabalho e irá agora sair da sala...");  
                sala.sair();  
            } catch (InterruptedException e) {}  
        }  
    }  
}
```

Resolução Exercício 5 - Main

```
public static void main(String[] args) {  
    Sala sala = new Sala();  
  
    Funcionario f1 = new Funcionario("Funcionario 1", sala);  
    Funcionario f2 = new Funcionario("Funcionario 2", sala);  
    Funcionario f3 = new Funcionario("Funcionario 3", sala);  
  
    f1.start();  
    f2.start();  
    f3.start();  
}
```



O futuro profissional começa aqui

