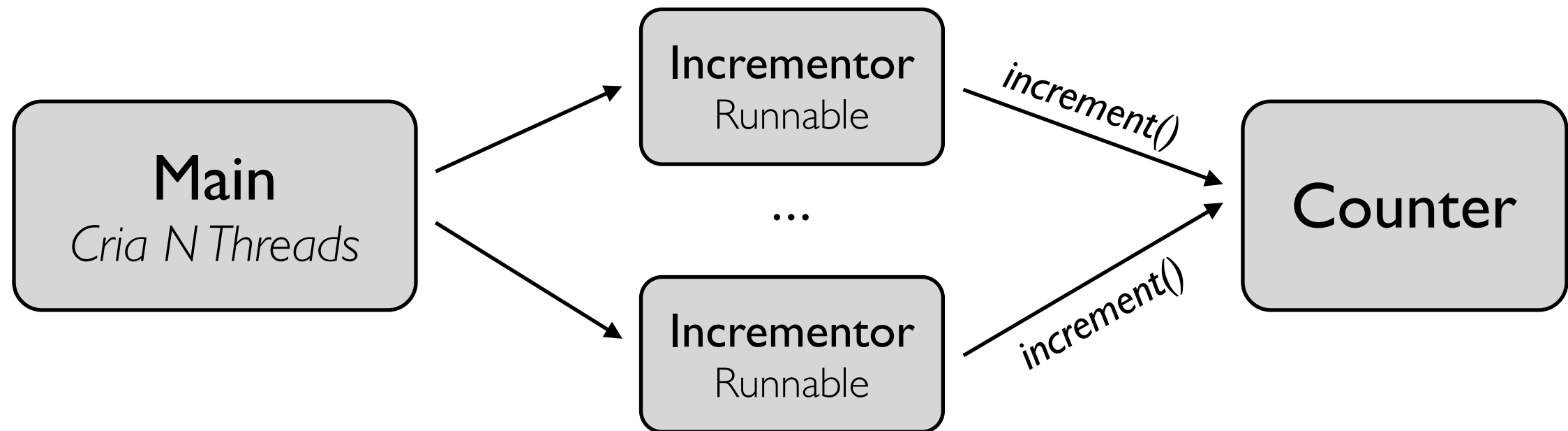


# Exclusão Mútua

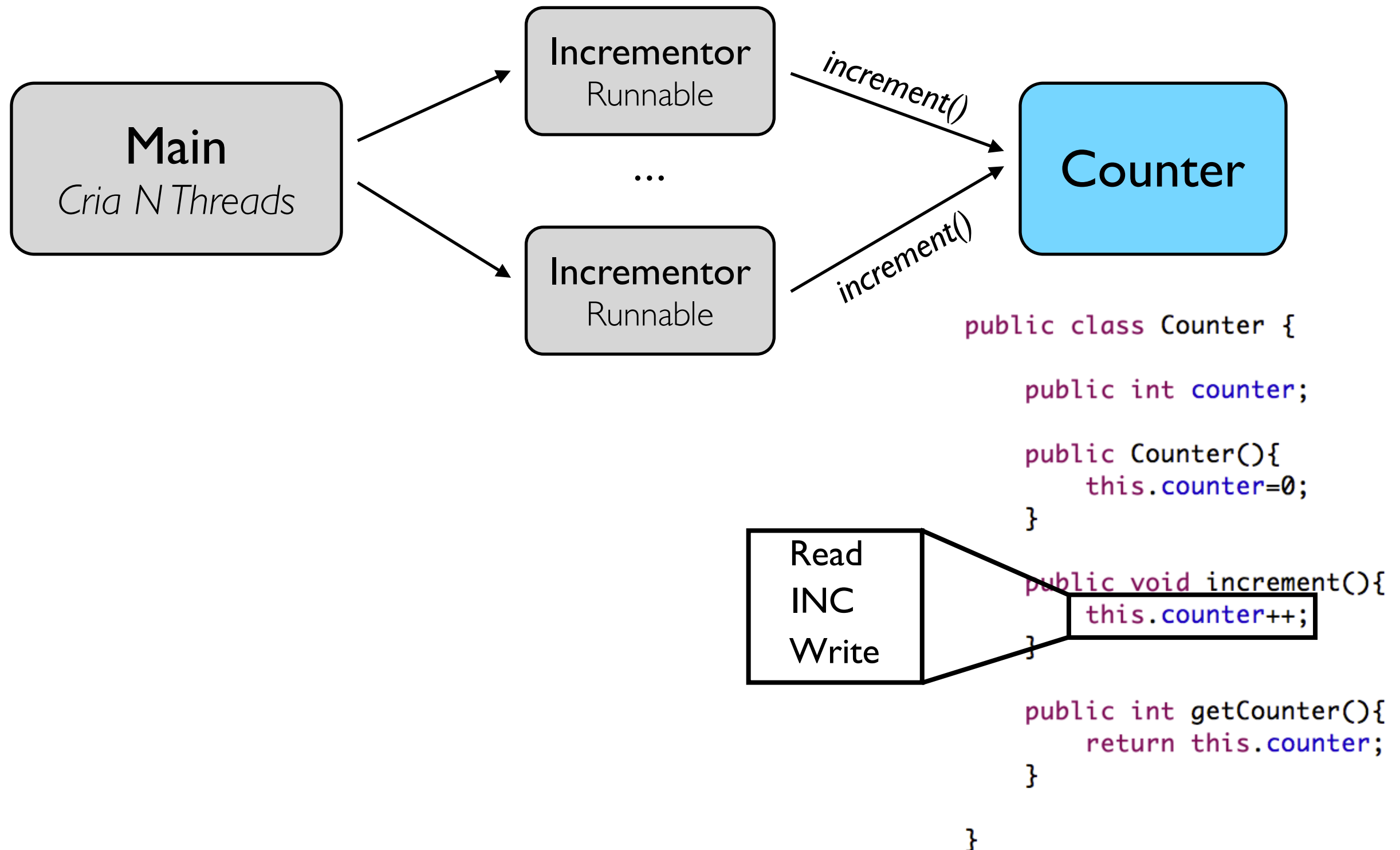
## Última aula:

Programa com **N** threads que têm acesso a um único objecto de uma classe **Counter**. Cada thread deverá incrementar **I** vezes o contador. Thread Main imprime o valor do contador no final.



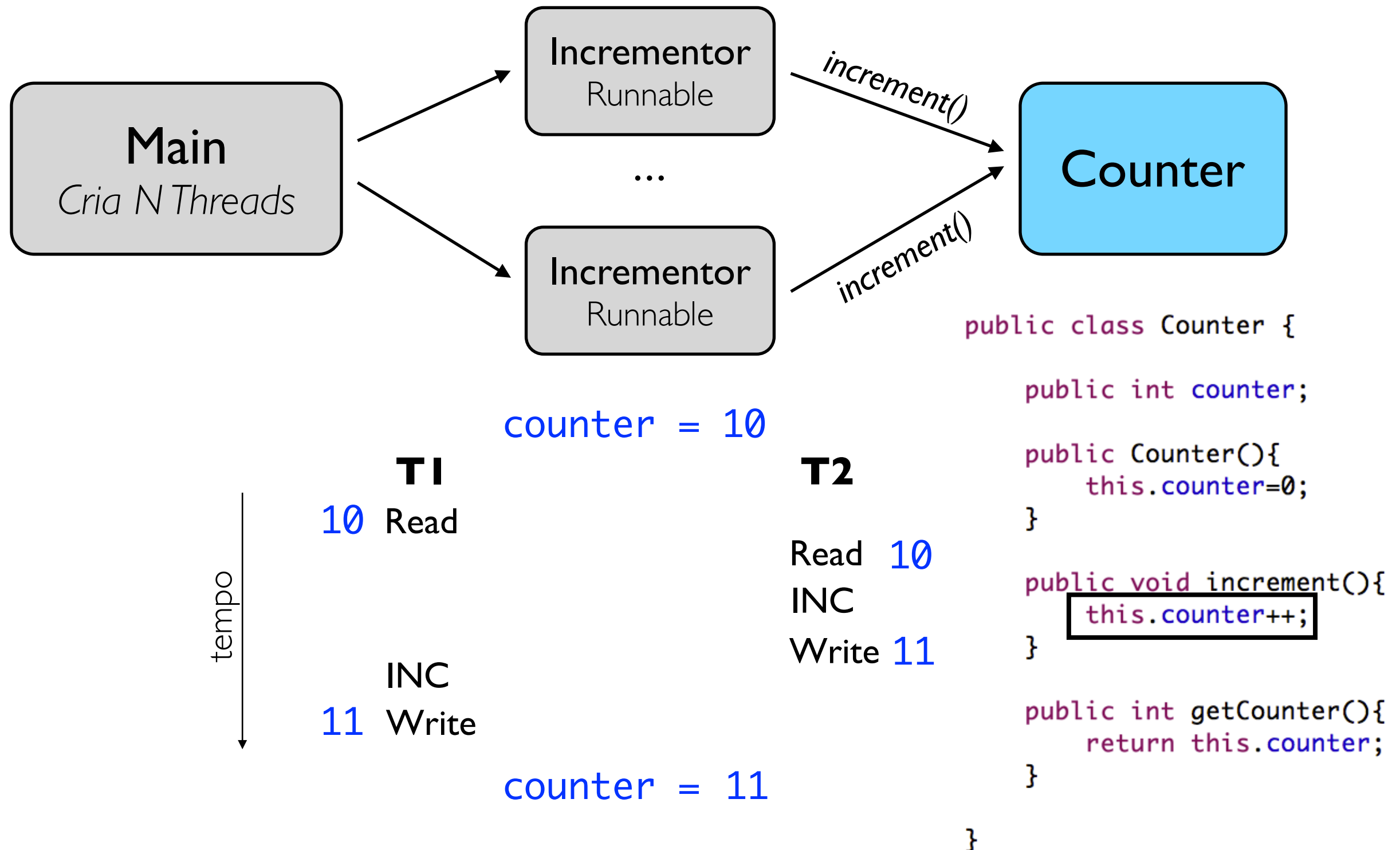
## Última aula:

Programa com **N** threads que têm acesso a um único objecto de uma classe **Counter**. Cada thread deverá incrementar **I** vezes o contador. Thread Main imprime o valor do contador no final.



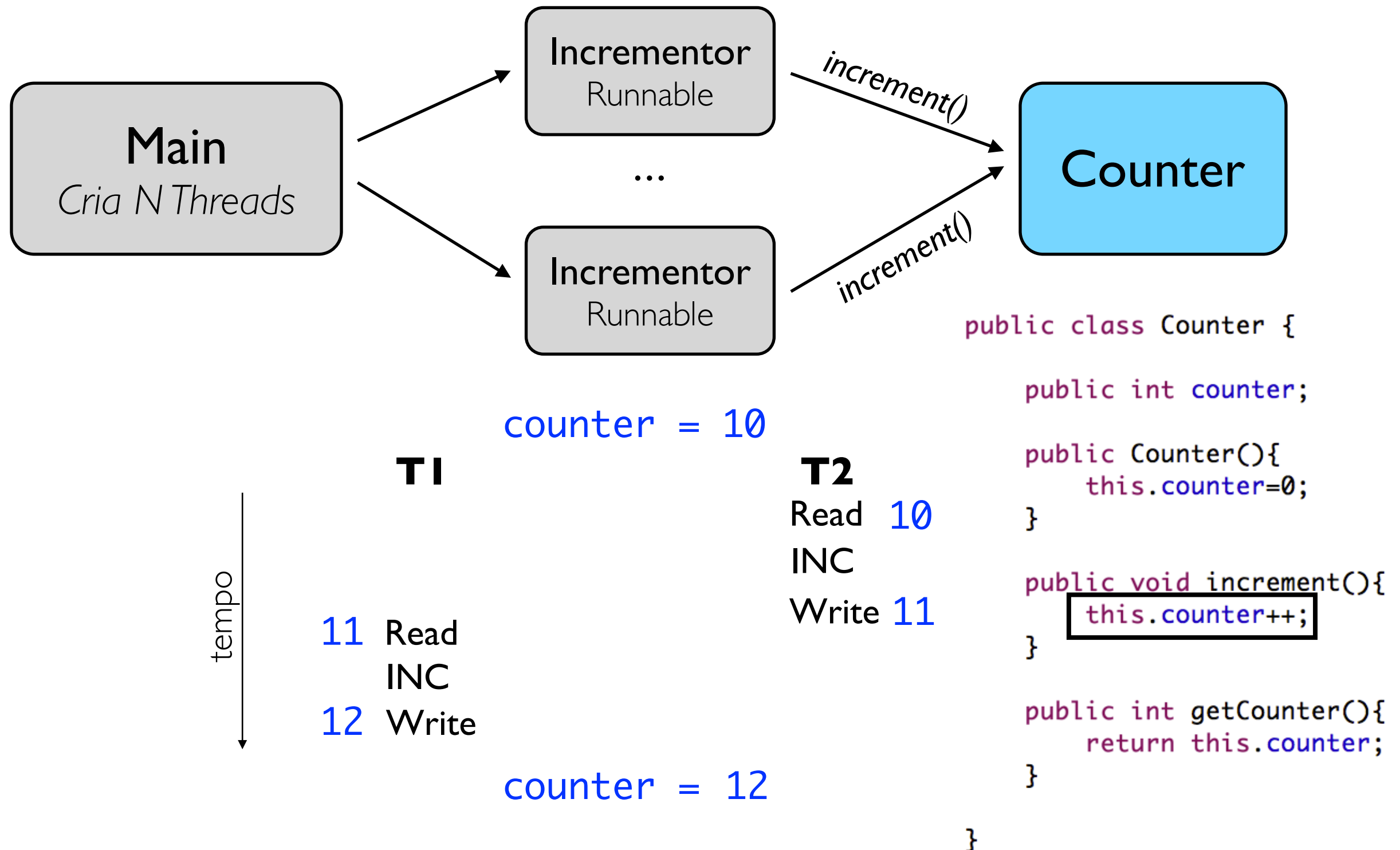
## Última aula:

Programa com **N** threads que têm acesso a um único objecto de uma classe **Counter**. Cada thread deverá incrementar **I** vezes o contador. Thread Main imprime o valor do contador no final.



## Última aula:

Programa com **N** threads que têm acesso a um único objecto de uma classe **Counter**. Cada thread deverá incrementar **I** vezes o contador. Thread Main imprime o valor do contador no final.



- **Exclusão mútua:** propriedade que garante que dois processos ou threads não acedem simultaneamente a um recurso partilhado.
- Proteger **secções críticas** do código.

- **Exclusão mútua:** propriedade que garante que dois processos ou threads não acedem simultaneamente a um recurso partilhado.
- Proteger **secções críticas** do código.

```
public class Counter {  
    public int counter;  
  
    public Counter(){  
        this.counter=0;  
    }  
  
    public void increment(){  
        this.counter++;  
    }  
}
```

secção crítica

- Em JAVA:
  - cada objecto tem um monitor/lock interno associado (herdado da class Object)
  - exclusão mútua através do mecanismo **synchronized**:

**Método:** (usa lock de this) `public synchronized void metodo() {  
i++;  
}`

**Bloco:** (usa lock de objecto) `public void metodo() {  
x++;  
synchronized (objecto){  
i++;  
}  
}`



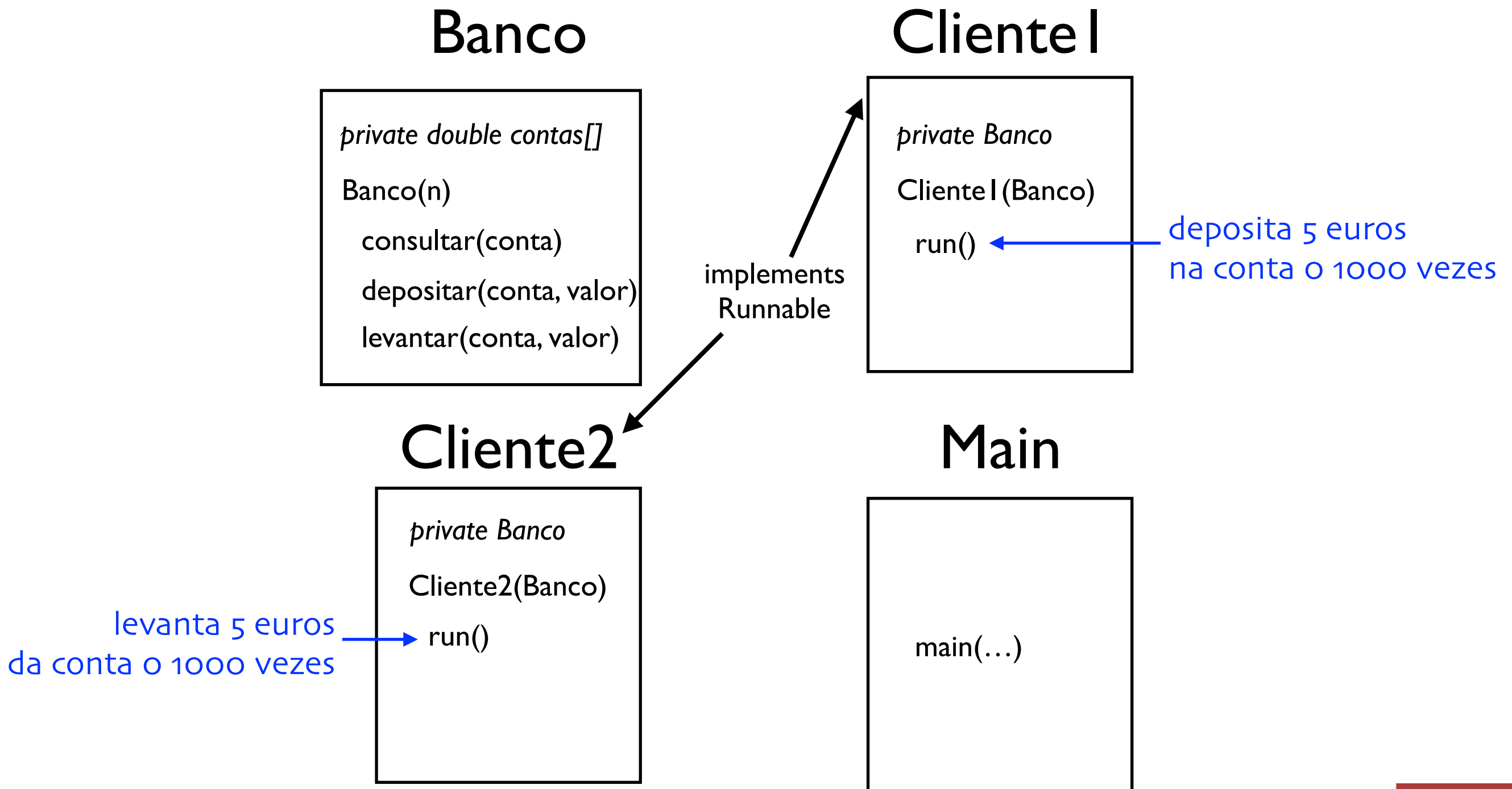
# Exercícios

- 1) Modifique o exercício anterior — incremento concorrente de um contador partilhado — de modo a garantir a execução correcta do programa.

# Exercícios

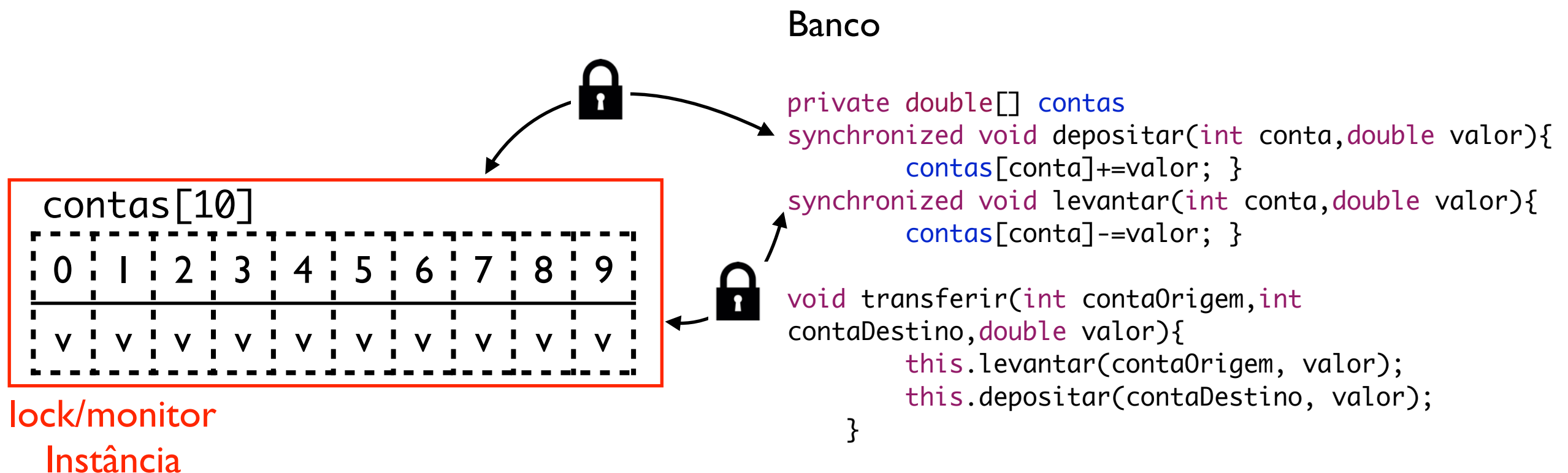
- 2) Implemente uma classe **Banco** que ofereça os métodos de consulta, crédito e débito de valores sobre um número fixo de contas (com saldo inicial nulo). Utilize exclusão mútua ao nível do objecto Banco.
- **Teste:** Adicionar dois clientes distintos (threads):
    - Cliente 1 - deposita 5 euros na conta o 1000 vezes
    - Cliente 2 - levanta 5 euros da conta o 1000 vezes

# Exercício 2



# Exercícios

- 3) Acrescente o método transferir à classe Banco como composição das operações de débito e crédito de um valor sobre duas contas.
- **Teste:** Iniciar conta 0 com 1000€ e conta 1 com 0€. Adicionar dois clientes distintos (threads):
    - Cliente 1 - transfere 1000€ de 0 para 1
    - Cliente 2 - levanta 1000€ de 1



```
void transferir(int contaOrigem, int contaDestino, double valor){  
    this.levantar(contaOrigem, valor);  
    this.depositar(contaDestino, valor);  
}
```

**Cenário:** conta0 = 1000, conta1 = 0

**Cliente1:** banco.transferir(conta0, conta1, 1000)

**Cliente2:** banco.levantar(conta1, 1000)

## Cliente1

## Cliente2

lock(Banco)  
levantar(conta0, 1000)  
unlock(Banco)

saldo conta0 = 0; saldo conta1 = 0!

lock(Banco)  
levantar(conta1, 1000)  
unlock(Banco)

lock(Banco)  
depositar(conta1, 1000)  
unlock(Banco)

```
void synchronized transferir(int contaOrigem,int contaDestino,double valor){  
    this.levantar(contaOrigem, valor);  
    this.depositar(contaDestino, valor);  
}
```


**Cenário:** conta0 = 1000, conta1 = 0

**Cliente1:** banco.transferir(conta0, conta1, 1000)

**Cliente2:** banco.levantar(conta1, 1000)

## Cliente1

## Cliente2

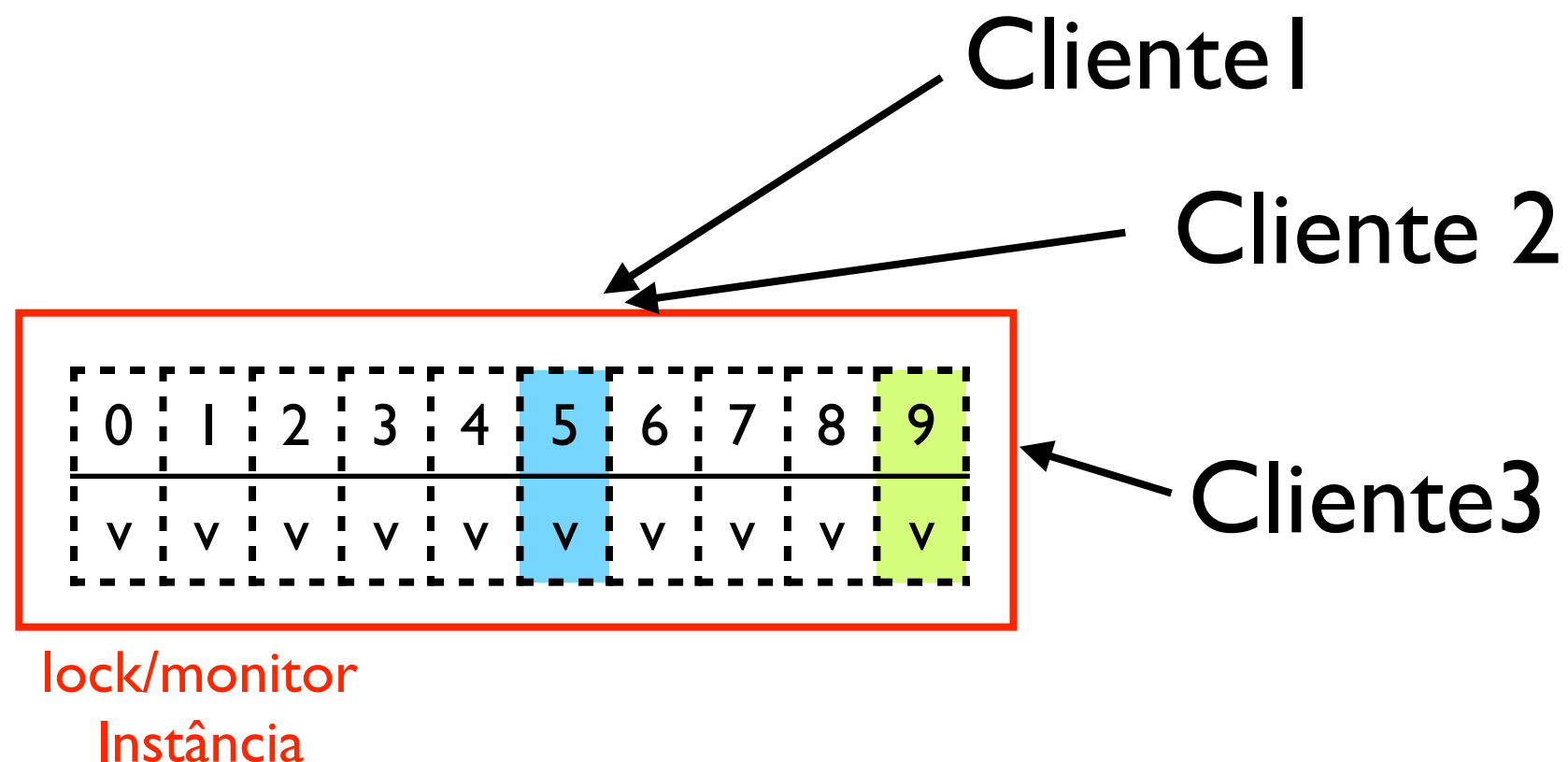


```
lock(Banco)  
levantar (conta0, 1000)  
depositar (conta1, 1000)  
unlock(Banco)
```

```
saldo conta0 = 0; saldo conta1 = 1000
```

```
lock(Banco)  
levantar(conta1, 1000)  
unlock(Banco)
```

- Exclusão mútua ao nível do objecto Banco pode ser ineficiente:
- Cliente 1 e 2 alteram o valor da conta 5, o mesmo recurso.
- Cliente 3 altera o valor da conta 9, um recurso diferente.
- Cliente 3 acede ao mesmo lock que Cliente 1 e 2.





# Exercícios

4) Reimplemente a classe Banco utilizando exclusão mútua ao nível das contas individuais. A classe Banco tem que disponibilizar os seguintes métodos:

- void depositar(int conta,double valor)
- void levantar(int conta,double valor)
- double consultar(int conta)
- void transferir(int contaOrigem,int contaDestino, double valor)
- **Dica:** Adicionar ao exercício anterior uma classe Conta, que disponibilize os métodos:
  - void depositar(double valor)
  - void levantar(double valor)
  - double consultar()

# Aula 2: Exercício 4

- Exclusão mútua ao nível das Contas
- Tentativa 1: Synchronized nos métodos da classe Conta

**Cenário:** conta0 = 1000, conta1 = 0

**Cliente1:** banco.transferir(conta0, conta1, 1000)

**Cliente2:** banco.consultar(conta1)

Cliente1

lock(conta0)  
conta0.levantar(1000)  
unlock(conta0)

lock(conta1)  
conta1.depositar(1000)  
unlock(conta1)

Cliente2

saldo conta0 = 0; saldo conta1 = 0!

lock(conta1)  
conta1.consultar()  
unlock(conta1)

# Aula 2: Exercício 4

- Exclusão mútua ao nível das Contas
- Tentativa 2: Lock contas sem ordem na classe Banco

```
public void transferir(conta0, conta1, valor)
{
    synchronized (conta0){
        synchronized (conta1){
            conta0.levanta(valor)
            conta1.deposita(valor)
        }
    }
}
```

# Aula 2: Exercício 4

- Exclusão mútua ao nível das Contas
- Tentativa 2: Lock contas sem ordem na classe Banco

**Cenário:** conta0 = 1000, conta1 = 0

**Cliente1:** banco.transferir(conta0, conta1, 1000)

**Cliente2:** banco.transferir(conta1, conta0, 1000)

## Cliente1

```
lock(conta0)
lock(conta1)
levanta (conta0, 1000)
deposita(conta1, 1000)
unlock(conta1)
unlock(conta0)
```

## Cliente2

```
lock(conta1)
lock(conta0)
levanta (conta1, 1000)
deposita(conta0, 1000)
unlock(conta0)
unlock(conta1)
```

# Aula 2: Exercício 4

- Exclusão mútua ao nível das Contas
- Tentativa 2: Lock contas sem ordem na classe Banco

**Cenário:** conta0 = 1000, conta1 = 0

**Cliente1:** banco.transferir(conta0, conta1, 1000)

**Cliente2:** banco.transferir(conta1, conta0, 1000)

Cliente1

Cliente2

lock(conta0)

lock(conta1)

lock(conta1) *bloqueia...*

lock(conta0) *bloqueia...*

levanta (conta0, 1000)  
deposita(conta1, 1000)  
unlock(conta1)  
unlock(conta0)

**Deadlock!**

levanta (conta1, 1000)  
deposita(conta0, 1000)  
unlock(conta0)  
unlock(conta1)



# Aula 2: Exercício 4

- Exclusão mútua ao nível das Contas
- Tentativa 3: Lock contas com ordem na classe Banco

```
public void transferir(conta0, conta1, valor){  
    conta_menor_id = Math.min(conta0, conta1)  
    conta_maior_id = Math.max(conta0, conta1)  
    synchronized (contas[conta_menor_id]){  
        synchronized (contas[conta_maior_id]){  
            conta0.levanta(valor)  
            conta1.deposita(valor)  
        }  
    }  
}
```

# Aula 2: Exercício 4

- Exclusão mútua ao nível das Contas
- Tentativa 3: Lock contas com ordem na classe Banco

**Cenário:** conta0 = 1000, conta1 = 0

**Cliente1:** banco.transferir(conta0, conta1, 1000)

**Cliente2:** banco.transferir(conta1, conta0, 1000)

## Cliente1

```
lock(conta_min)
lock(conta_max)
levanta (conta0, 1000)
deposita(conta1, 1000)
unlock(conta_max)
unlock(conta_min)
```

## Cliente2

```
lock(conta_min)
lock(conta_max)
levanta (conta1, 1000)
deposita(conta0, 1000)
unlock(conta_max)
unlock(conta_min)
```

# Aula 2: Exercício 4


- Exclusão mútua ao nível das Contas
- Tentativa 3: Lock contas com ordem na classe Banco

**Cenário:** conta0 = 1000, conta1 = 0

**Cliente1:** banco.transferir(conta0, conta1, 1000)

**Cliente2:** banco.transferir(conta1, conta0, 1000)

## Cliente1



```
lock(conta_min) lock(conta0)
lock(conta_max) lock(conta1)
levanta (conta0, 1000)
deposita(conta1, 1000)
unlock(conta_max)
unlock(conta_min)
```

## Cliente2

```
lock(conta_min) lock(conta0)
lock(conta_max) lock(conta1)
levanta (conta1, 1000)
deposita(conta0, 1000)
unlock(conta_max)
unlock(conta_min)
```