Banco - Criar Conta

- Método Banco.criarConta
  - <u>Abordagem:</u> Usar lockBanco para proteger alterações ao HashMap contas

```
// Usado para gerar IDs de Conta
private int lastId = 0;

public int criarConta(double saldo){
    this.lockBanco.lock();
    int id = lastId++;
    Conta c = new Conta(id, saldo);
    this.contas.put(id, c);
    this.lockBanco.unlock();

    return id;
}
```

Banco - Criar Conta

- Método Banco.criarConta
  - <u>Abordagem:</u> Usar lockBanco para proteger alterações ao HashMap contas

novo id tem de ser obtido dentro da secção crítica, senão duas threads podem criar 2x a mesma conta

```
// Usado para gerar IDs de Conta
private int lastId = 0;

public int criarConta(double saldo){
    this.lockBanco.lock();
    int id = lastId++;
    Conta c = new Conta(id, saldo);
    this.contas.put(id, c);
    this.lockBanco.unlock();

    return id;
}
```

Banco - Fechar Conta

- Método Banco.fecharConta
  - Abordagem 1: Verificar se conta existe e depois obter lockBanco para proteger alterações ao HashMap contas

```
public double fecharConta(int id) {
    if(contas.containsKey(id)){
          this.lockBanco.lock();
          Conta c = this.contas.get(id);
          this.contas.remove(id);
          this.lockBanco.unlock();
          return c.consultar();
    } else
        return -1;
}
```

Banco - Fechar Conta

- Método Banco.fecharConta
  - Abordagem 1: Verificar se conta existe e depois obter lockBanco para proteger alterações ao HashMap contas

Cliente1: `banco.fechar(0)`
Cliente2: `banco.fechar(0)`

Cliente1

```
if(contas.containsKey(id)){  True

this.lockBanco.lock();
Conta c = this.contas.get(id);
this.contas.remove(id);
this.lockBanco.unlock();
```

Cliente2

```
if(contas.containsKey(id)){  True



this.lockBanco.lock();
Conta c = this.contas.get(id);
double saldo = c.consultar();
this.contas.remove(id);
this.lockBanco.unlock();
```

NullPointerException: Conta 0 já não existe!

Banco - Fechar Conta

- Método Banco.fecharConta
  - Abordagem 1: Obter lockBanco para verificar se a conta existe e para proteger alterações ao HashMap contas

```
public double fecharConta(int id) {
   this.lockBanco.lock();
   Conta c = this.contas.remove(id);
   if(c == null) {
     this.lockBanco.unlock();
     return -1;
   }

   this.lockBanco.unlock();
   return c.consultar();
}
```

Banco - Fechar Conta

- Método Banco.fecharConta
  - <u>Abordagem 1:</u> Obter lockBanco para verificar se a conta existe e para proteger alterações ao HashMap contas

Cliente1: banco.fechar(0)
Cliente2: banco.fechar(0)

Cliente1

```
this.lockBanco.lock();
Conta c = contas.remove(id);
if(c == null){False
this.lockBanco.unlock();
return c.consultar();
```

Cliente2

```
this.lockBanco.lock();
Conta c = contas.remove(id);
if(c == null) {True
  this.lockBanco.unlock();
  return -1;
}
```

Banco - Fechar Conta

- Método Banco.fecharConta
  - Abordagem 1: Obter lockBanco para verificar se a conta existe e para proteger alterações ao HashMap contas

Cliente1: `banco.fechar(0)`
Cliente2: `banco.depositar(0,10)`

Cliente1

```
this.lockBanco.lock();
Conta c = contas.remove(id);
if(c == null){…}
this.lockBanco.unlock();
return c.consultar();
```

False

Saldo é retornado enquanto há um depósito em curso. Dinheiro "perde-se".

Cliente2

```
this.lockBanco.lock();
Conta c = this.contas.get(id);
if(c == null){…}
c.lock();
this.lockBanco.unlock();
```

False

Conta 0 existe!

```
c.depositar(10);
c.unlock();
```

Conta 0 já não existe! Operação "depositar" numa conta inexistente.

Banco - Fechar Conta

- Método Banco.fecharConta
  - Abordagem 2: Obter lockBanco para verificar se a conta existe e para proteger alterações ao HashMap contas. Obter lockConta para impedir que se remova uma conta quando outra thread está a meio de uma operação sobre essa conta.

```
public double fecharConta(int id) {
   this.lockBanco.lock();
   Conta c = this.contas.remove(id);
   if(c == null) {
     this.lockBanco.unlock();
     return -1;
   }
   c.lock();
   this.lockBanco.unlock();
   double saldo = c.consultar();
   c.unlock();
   return saldo;
}
```

Banco - Fechar Conta

- Método Banco.fecharConta
  - Abordagem 2: Obter lockBanco para verificar se a conta existe e para proteger alterações ao HashMap contas. Obter lockConta para impedir que se remova uma conta quando outra thread está a meio de uma operação sobre essa conta.

Cliente1: `banco.fechar(0)`
Cliente2: `banco.depositar(0,10)`

Cliente1

```
this.lockBanco.lock();
Conta c = this.contas.remove(id);
if(c == null) {…}
```
False
```
c.lock();
this.lockBanco.unlock();
double saldo = c.consultar();
c.unlock();
return saldo;
```

Saldo já contempla o depósito efetuado e a conta só foi fechada após a operação "depósito".

Cliente2

```
this.lockBanco.lock();
Conta c = this.contas.get(id);
if(c == null){…}
```
False

Conta 0 existe!
```
c.lock();
this.lockBanco.unlock();
c.depositar(10);
c.unlock();
```

Banco - Balanço Total

- Método Banco.consultarTotal
  - Abordagem 1: Para cada conta, obter lockBanco para verificar se a conta existe e obter lockConta para impedir que outra thread remova a conta durante a consulta do saldo.

```
public double consultarTotal(int[] contasInput) {
        double saldoTotal = 0;
    for(int i = 0; i < contasInput.length; i++){
       int id = contasInput[i];
       this.lockBanco.lock();
       if(contas.containsKey(id)){
          contas.get(id).lock();
          this.lockBanco.unlock();
          saldoTotal += contas.get(id).consultar();
          contas.get(id).unlock();
       }
       else this.lockBanco.unlock();
    }
    return saldoTotal;
}
```

Banco - Balanço Total

- Método Banco.consultarTotal
  - Abordagem 1: Para cada conta, obter lockBanco para verificar se a conta existe e obter lockConta para impedir que outra thread remova a conta durante a consulta do saldo.

Cenário: `conta0 = conta1 = conta2 = 10`
Cliente1: `banco.consultarTotal([0,1,2])`
Cliente2: `banco.transferir(0,2,10)`

Cliente1

```
this.contas.get(0).lock();
saldoTotal += this.contas.get(0).consultar(); saldoTotal = 10
this.contas.get(0).unlock();

this.contas.get(1).lock();
saldoTotal += this.contas.get(1).consultar(); saldoTotal = 20
this.contas.get(1).unlock();
this.contas.get(2).lock();
saldoTotal += this.contas.get(2).consultar(); saldoTotal = 40
this.contas.get(2).unlock();
```

Cliente2

obtém locks das contas 0 e 2 e faz a transferência

```
banco.transferir(0,2,10)
```

Banco só tem 30€!

Banco - Balanço Total

- Método Banco.consultarTotal
  - <u>Abordagem 2:</u> Obter lockBanco para obter todos os lockConta das contas válidas desejadas. Consultar saldo total só após ter todos os locks das contas.

```
public double consultarTotal(int[] contasInput) {
        double saldoTotal = 0;
        ArrayList<Integer> contasLocked = new ArrayList(contasInput.length);
        this.lockBanco.lock();
        for(int i = 0; i < contasInput.length; i++){
            int id = contasInput[i];
            if(contas.containsKey(id)){
                this.contas.get(id).lock();
                contasLocked.add(id);
            }
        }
        this.lockBanco.unlock();
        for(int id : contasLocked){
            saldoTotal += contas.get(id).consultar();
            contas.get(id).unlock();
        }
        return saldoTotal;
    }
```

obter os locks das contas

consultar o saldo total

Banco - Balanço Total

- Método Banco.consultarTotal
  - <u>Abordagem 2:</u> Obter lockBanco para obter todos os lockConta das contas válidas desejadas. Consultar saldo total só após ter todos os locks das contas.

Cenário: `conta0 = conta1 = conta2 = 10`
Cliente1: `banco.consultarTotal([0,1,2])`
Cliente2: `banco.transferir(0,2,10)`

Cliente1

```
this.contas.get(0).lock();
this.contas.get(1).lock();
this.contas.get(2).lock();
(…)
saldoTotal += this.contas.get(0).consultar(); saldoTotal = 10
this.contas.get(0).unlock();
saldoTotal += this.contas.get(1).consultar(); saldoTotal = 20
this.contas.get(1).unlock();
saldoTotal += this.contas.get(2).consultar(); saldoTotal = 30
this.contas.get(2).unlock();
```

Cliente2

obtém locks das contas 0 e 2 e faz a transferência

```
banco.transferir(0,2,10)
```

ReentrantReadWriteLock

- Mecanismo utilizado permite mais concorrência sem detrimento da exclusão mútua, expondo um lock para leitura e outro para escrita.
- O lock de **leitura** é de acesso **partilhado**: múltiplas threads com este lock têm acesso à secção crítica para fins de leitura.
- O lock de **escrita** é de acesso **exclusivo**: apenas uma thread tem acesso à secção crítica para fins de leitura e/ou escrita.
- Métodos:
  - `ReentrantReadWriteLock() // construtor`
  - `readLock().lock() // adquirir lock de leitura`
  - `readLock().unlock() // libertar lock de escrita`
  - `writeLock().lock() // adquirir lock de escrita`
  - `writeLock().unlock() // libertar lock de escrita`

https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/locks/ReentrantReadWriteLock.html

ReentrantReadWriteLock

- Compatibilidade de locks:

	Lock Leitura	Lock Escrita
Lock Leitura	✅	❌
Lock Escrita	❌	❌

https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/locks/ReentrantReadWriteLock.html