

応用Javaプログラミング

第8回

1. マルチスレッド(上級編)
2. デッドロック(deadlock)
3. ロック・オブジェクト
4. オブジェクトの固有ロック

前回設計した BankAccount2クラスの withdrawメソッド

// 貯金を引き落とすメソッド

@Override // コンパイラへの指示

protected void withdraw(long amount) {

// 他のスレッドに割り込まれたくない部分に入る前に鍵をかける
balanceChangeLock.lock();

// 他のスレッドに割り込まれたくない部分を tryブロックに入れる
// finallyブロックで 鍵を外す.

try {

super.withdraw(amount) ;

} finally {

balanceChangeLock.unlock() ; // 鍵を外す

}

}

BankAccount2.java

同期ブロックとtry-finally

前回設計した BankAccount2クラス のwithdrawメソッド(展開された形)

// 引き落としを行うメソッド

protected void withdraw(long amount) {

// 他のスレッドに割り込まれたくない部分に入る前に鍵をかける

balanceChangeLock.lock();

// 他のスレッドに割り込まれたくない部分を **try**ブロックに入れる

// **finally**ブロックで 鍵を外す.

try {

if (amount > balance) {

System.out.println("insufficient balance"); // 残高不足を表示

} else {

System.out.print("Withdrawing " + amount); // 引き出し額を表示

long newBalance = balance - amount; // 新残高を計算

System.out.println(", new balance is " + newBalance); // 新残高を表示

balance = newBalance; // 残高を更新

}

} finally {

balanceChangeLock.unlock(); // 鍵を外す

}

}

引き落としたい金額が
残高より大きいとき、
引き落とすことを
あきらめている。

withdrawメソッドでしたい 新しい処理

引き落とししたい金額が
残高より大きいとき、
残高が十分に増えて
くるまで待つ。

// 引き落としを行うメソッド

```
protected void withdraw(long amount) {
```

```
// 他のスレッドに割り込まれたくない部分に入る前に鍵をかける  
balanceChangeLock.lock( );
```

```
// 他のスレッドに割り込まれたくない部分を tryブロックに入れる  
// finallyブロックで 鍵を外す.
```

```
try {
```

```
    while (amount > balance) {
```



```
        // 残高が、引き落とし金額以上になるまで待つ
```

```
    }
```

```
    System.out.print("Withdrawing " + amount); // 引き出し額を表示
```

```
    long newBalance = balance - amount; // 新残高を計算
```

```
    System.out.println(", new balance is " + newBalance); // 新残高を表示
```

```
    balance = newBalance; // 残高を更新
```

```
    } finally {
```

```
        balanceChangeLock.unlock( ); // 鍵を外す
```

```
    }
```

```
}
```

残高が増えてくるまで待つためのだめな方法

// 引き落としを行うメソッド

```
protected void withdraw(long amount) throws InterruptedException {
```

```
// 他のスレッドに割り込まれたくない部分に入る前に鍵をかける
```

```
balanceChangeLock.lock();
```

```
// 他のスレッドに割り込まれたくない部分を try ブロックに入れる
```

```
// finally ブロックで 鍵を外す.
```

```
try {
```

```
    while (amount > balance) {
```



```
        Thread.sleep(1000); // (たとえば, 1秒) 待ってからまた残高をチェックしに行く
```

```
    }
```

```
    System.out.print("Withdrawing " + amount); // 引き出し額を表示
```

```
    long newBalance = balance - amount; // 新残高を計算
```

```
    System.out.println(", new balance is " + newBalance); // 新残高を表示
```

```
    balance = newBalance; // 残高を更新
```

```
} finally {
```

```
    balanceChangeLock.unlock(); // 鍵を外す
```

```
}
```

```
}
```

残高が増えてくるまで待つための だめな方法の問題点

引き落としを担当するスレッド(**withdraw**メソッドを実行するスレッド)が残高不足の間にずっと

文 **Thread.sleep(1000);**

を実行して休むが、**休んでいる間ロック(鍵)を解除しない。**

残高を増やす(別の)スレッドは、**deposit**メソッドを実行する必要がある。

withdrawメソッドと**deposit**メソッドで同じロック・オブジェクトを使っているので、残高を増やす(別の)スレッドは**deposit**メソッドを実行するのに、引き落としを担当するスレッドがかけた鍵の解除が必要である。

しかし、引き落としを担当するスレッド(**withdraw**メソッドを実行するスレッド)が残高不足の間にずっと**鍵を解除しない。**

∴ 2つのスレッドがお互いに相手を待つ。これは**デッドロック**という。

java.util.concurrent.locks.Condition インターフェース

デッドロックを防ぐためによく使われるインターフェース

主なメソッド

public abstract void await()

信号が送信されるか、または、割り込みが発生するまで現在のスレッドを待機させる。

public abstract void signalAll()

待機中のすべてのスレッドに信号を送る。待っている各スレッドは、awaitから復帰する前にロックを再取得する必要がある。

残高が増えてくるまで待つための

よい方法: ロックから**Condition**インスタンスを取得して使う。

あるロックから**Condition**インスタンスを取得して使うと、下記のことを可能にできる:

- ① そのロックを持っているスレッドが一時的にロックを外し;
- ② 別のスレッドがロックを取得して仕事に進んでそれを終えてからロックを外してそのロックオブジェクトを待っているスレッドに通知し;
- ③ 元のスレッドが後ほどロックを再取得.

ロックから**Condition**インスタンスを取得する方法:

まず、**ロック・オブジェクトはコンストラクタで生成すること**を思い出そう.

(同じコンストラクタで)その後すぐに、そのロック・オブジェクトの**newCondition**メソッドを呼び出してそのロック・オブジェクトから**Condition**インスタンスを取得する.

ロックから取得した Conditionインスタンスの使い方

待ちたいスレッド側で,

銀行口座の例では, `amount > balance`

```
while ( 待ち条件 ) {  
    (取得したConditionインスタンス).await(); //待つ  
}
```

進んで仕事を済ますスレッド側で,

ロックをかける;

```
try {  
    仕事を済ます ;  
    (取得したConditionインスタンス).signalAll();  
} finally { ロックを外す; }
```

例：銀行口座

// **Lock**インターフェース, **ReentrantLock**クラス, **Condition**クラスを使うので次の1行が必要
import java.util.concurrent.locks.* ;

public class BankAccount5 extends BankAccount {

// このクラスの1つ以上のメソッドをスレッドセーフにしたいとき, このクラスのフィールド
// として, 次の行のように **Lock**インターフェース(を実装したクラス)型の変数を追加する

private Lock balanceChangeLock;

private Condition sufficientBalanceCondition; // 待ち条件のフィールド

// コンストラクタ(新しい口座を開く)

public BankAccount5() {

// コンストラクタで**ReentrantLock**クラスのインスタンス(**ロック・オブジェクト**)を生成

balanceChangeLock = new **ReentrantLock**();

sufficientBalanceCondition = balanceChangeLock.**newCondition**(); //条件取得

}

次頁に続く...

BankAccount5.java



Conditionインスタンスを使ったwithdrawメソッド

// 引き落としを行うメソッド

```
protected void withdraw(long amount) {  
    // 他のスレッドに割り込まれたくない部分に入る前に鍵をかける  
    balanceChangeLock.lock( );  
  
    // 他のスレッドに割り込まれたくない部分を tryブロックに入れる  
    // finallyブロックで 鍵を外す.  
    try {  
        try {  
            while (amount > balance)  
                sufficientBalanceCondition.await( ); // (sleepの代わりに)待つ  
            System.out.print("Withdrawing " + amount); // 引き出し額を表示  
            long newBalance = balance - amount; // 新残高を計算  
            System.out.println(", new balance is " + newBalance); // 新残高を表示  
            balance = newBalance; // 残高を更新  
            sufficientBalanceCondition.signalAll( );  
        } finally {  
            balanceChangeLock.unlock( ); // 鍵を外す  
        }  
    } catch ( InterruptedException e) { Thread.currentThread().interrupt( ); }  
}
```

Conditionインスタンスを使った depositメソッド

// 貯金を預けるメソッド

```
protected void deposit(long amount) {  
    System.out.print("Depositing " + amount); // 預け金額を表示  
  
    // 他のスレッドに割り込まれたくない部分に入る前に鍵をかける  
    balanceChangeLock.lock();  
  
    // 他のスレッドに割り込まれたくない部分を tryブロックに入れる  
    // finallyブロックで 鍵を外す.  
    try {  
        super.deposit(amount);  
        sufficientBalanceCondition.signalAll(); // 待っているスレッド全員に通知  
    } finally {  
        balanceChangeLock.unlock(); // 鍵を外す  
    }  
}
```



Conditionインスタンスを使った setBalanceメソッド

// 残高を更新するメソッド

@Override // コンパイラへの指示

protected void setBalance(long newBalance) {

// 他のスレッドに割り込まれたくない部分に入る前に鍵をかける
balanceChangeLock.lock();

// 他のスレッドに割り込まれたくない部分を tryブロックに入れる
// finallyブロックで 鍵を外す.

try {

super.setBalance(newBalance);

sufficientBalanceCondition.signalAll(); // 待っているスレッド全員に通知

} finally {

balanceChangeLock.unlock(); // 鍵を外す

}

}

BankAccount5.java

getNumber()とgetBalance()は元のままでよい.

Agent5クラス

```
public class Agent5 {  
    private static void testAccount(BankAccount account, long amount, int times,  
                                    int delay ) {  
        // タスク(預け入れという仕事)を1つ生成し, そのタスクを担うスレッドを2つ生成する  
        DepositTask dTask = new DepositTask(account, amount, times, delay) ;  
        Thread dAgent 1= new Thread(dTask) ;  
        Thread dAgent 2= new Thread(dTask) ;  
  
        // タスク(引き落としという仕事)を1つ生成し, そのタスクを担うスレッドを2つ生成する  
        WithdrawTask wTask = new WithdrawTask(account, amount, times, delay) ;  
        Thread wAgent1 = new Thread(wTask) ;  
        Thread wAgent2 = new Thread(wTask) ;  
  
        // 先ほど生成した4つのスレッドを起動する  
        wAgent1.start( ) ;  
        wAgent2.start( ) ;  
        dAgent1.start( ) ;  
        dAgent2.start( ) ;  
    }  
  
    public static void main(String[ ] args ) {  
        testAccount(new BankAccount5(), 100, 1000, 1) ;  
    }  
}
```

インスタンス固有のロックを使用したときは？

```
public class BankAccount3 extends BankAccount {  
    // 貯金を預けるメソッド  
    @Override // コンパイラへの指示  
    protected synchronized void deposit(long amount) {  
        super.deposit(amount);  
    }  
  
    // 貯金を下すメソッド  
    @Override // コンパイラへの指示  
    protected synchronized void withdraw(long amount) {  
        super.withdraw(amount);  
    }  
  
    // 残高を更新するメソッド  
    @Override // コンパイラへの指示  
    protected synchronized void setBalance(long amount) {  
        super.setBalance(amount);  
    }  
}
```



インスタンス固有のロックを使用したときは？

// 貯金を預けるメソッド(同期メソッド)

```
protected synchronized void deposit(long amount) {  
    System.out.print("Depositing " + amount); // 預け金額を表示  
    long newBalance = balance + amount; // 新しい残高を計算  
    System.out.println(", new balance is " + newBalance); // 新残高を表示  
    balance = newBalance; // 貯金残高を更新  
}
```



// 貯金を引き出すメソッド(同期メソッド)


```
protected synchronized void withdraw(long amount) {  
    if (amount > balance) {  
        System.out.println("insufficient balance"); // 残高不足を表示  
    } else {  
        System.out.print("Withdrawing " + amount); // 引き出し額を表示  
        long newBalance = balance - amount; // 新残高を計算  
        System.out.println(", new balance is " + newBalance); // 新残高を表示  
        balance = newBalance; // 残高を更新  
    }  
}
```




引き落としたい金額が残高より大きいとき、引き落とすことをあきらめている。

インスタンス固有のロックを使用したとき


各インスタンスに**固有のロック**が1つあり, 前頁のプログラムは下記の(擬似)プログラムと等価である.



```
protected void deposit(long amount) { // 貯金を預けるメソッド(同期メソッド)
    this.固有ロック.lock( ); // 擬似文
    System.out.print("Depositing " + amount); // 預け金額を表示
    long newBalance = balance + amount; // 新しい残高を計算
    System.out.println(" , new balance is " + newBalance); // 新残高を表示
    balance = newBalance; // 貯金残高を更新
    this.固有ロック.unlock( ); // 擬似文
}
```



```
protected void withdraw(long amount) { // 貯金を引き出すメソッド(同期メソッド)
    this.固有ロック.lock( ); // 擬似文
    if (amount > balance) {
        System.out.println("insufficient balance"); // 残高不足を表示
    } else {
        System.out.print("Withdrawing " + amount); // 引き出し額を表示
        long newBalance = balance - amount; // 新残高を計算
        System.out.println(" , new balance is " + newBalance); // 新残高を表示
        balance = newBalance; // 残高を更新
    }
    this.固有ロック.unlock( ); // 擬似文
}
```



インスタンス固有のロックを使用したときにも 残高が増えてくるまで待ちたい

// 貯金を預けるメソッド(同期メソッド)

```
protected synchronized void deposit(long amount) {  
    System.out.print("Depositing " + amount); // 預け金額を表示  
    long newBalance = balance + amount; // 新しい残高を計算  
    System.out.println(", new balance is " + newBalance); // 新残高を表示  
    balance = newBalance; // 貯金残高を更新  
}
```



// 貯金を引き出すメソッド(同期メソッド)

```
protected synchronized void withdraw(long amount) {  
    while (amount > balance) {  
        // 残高が、引き落とし金額以上になるまで待つ  
    }  
    System.out.print("Withdrawing " + amount); // 引き出し額を表示  
    long newBalance = balance - amount; // 新残高を計算  
    System.out.println(", new balance is " + newBalance); // 新残高を表示  
    balance = newBalance; // 残高を更新  
}
```



if-else文を
while文に変更

BankAccount6.java の一部

インスタンス固有のロックを使用したとき

インスタンス固有のロックを使用したとき 残高が増えてくるまで待つための**方法**は？

答え: **Object**クラスのインスタンスメソッド **wait** と **notifyAll** を使う。

Objectクラスの主なメソッド

public final void wait()

他のスレッドがこのオブジェクトの**notifyAll**または**notify**メソッドを呼び出すまで、現在のスレッドを待機させる。

public final void notifyAll()

このオブジェクトの固有ロックを待っているすべてのスレッドに信号を送る。待っている各スレッドは、waitから復帰する前にロックを再取得する必要がある。

直感的な理解: 各インスタンスの固有ロックに固有の条件が1つあり、
wait() は **固有条件.await()** と等価、
notifyAll() は **固有条件.signalAll()** と等価。

インスタンス固有のロックを使用したとき 残高が増えてくるまで待つための方法は？

// 貯金を預けるメソッド(同期メソッド)

```
protected synchronized void deposit(long amount) {  
    System.out.print("Depositing " + amount); // 預け金額を表示  
    long newBalance = balance + amount; // 新しい残高を計算  
    System.out.println(", new balance is " + newBalance); // 新残高を表示  
    balance = newBalance; // 貯金残高を更新  
    notifyAll(); // このオブジェクトのロックを待っているスレッドに信号を送る.  
    // 固有条件.signalAll() と等価.  
}
```



// 貯金を引き出すメソッド(同期メソッド)

```
protected synchronized void withdraw(long amount) {  
    while (amount > balance)  
        wait(); // 残高不足の間 待つ. 固有条件.await() と等価.  
    System.out.print("Withdrawing " + amount); // 引き出し額を表示  
    long newBalance = balance - amount; // 新残高を計算  
    System.out.println(", new balance is " + newBalance); // 新残高を表示  
    balance = newBalance; // 残高を更新  
    notifyAll(); // このオブジェクトのロックを待っているスレッドに信号を送る.  
}
```



インスタンス固有のロックを使用したとき

インスタンス固有のロックを使用したとき 残高が増えてくるまで待つための方法は？

// 貯金を預けるメソッド(同期メソッド)

```
protected synchronized void setBalance(long newBalance) {  
    super.setBalance(newBalance); // 預け金額を表示  
    notifyAll(); // このオブジェクトのロックを待っているスレッドに信号を送る.  
    // 固有条件.signalAll() と等価.  
}
```



インスタンス固有のロックを使用したとき

スレッドのライフサイクル(復習)

生きている

