

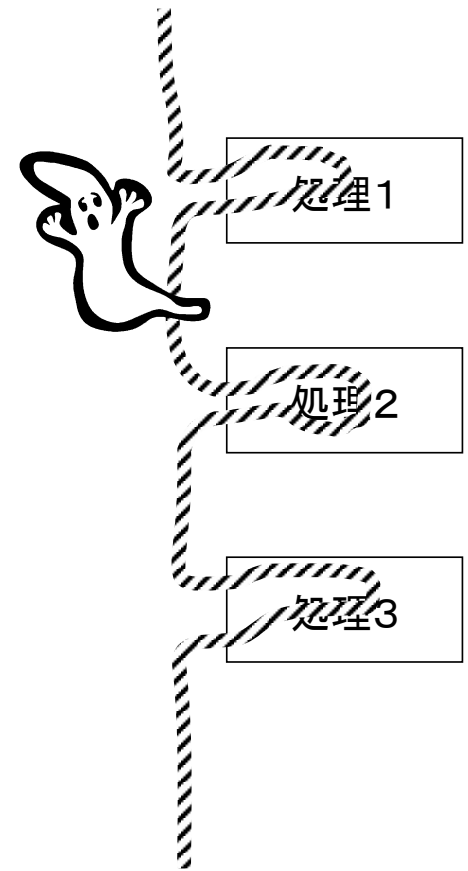
Javaプログラミング (9)

スレッド(1)

成蹊大学理工学部
情報科学科

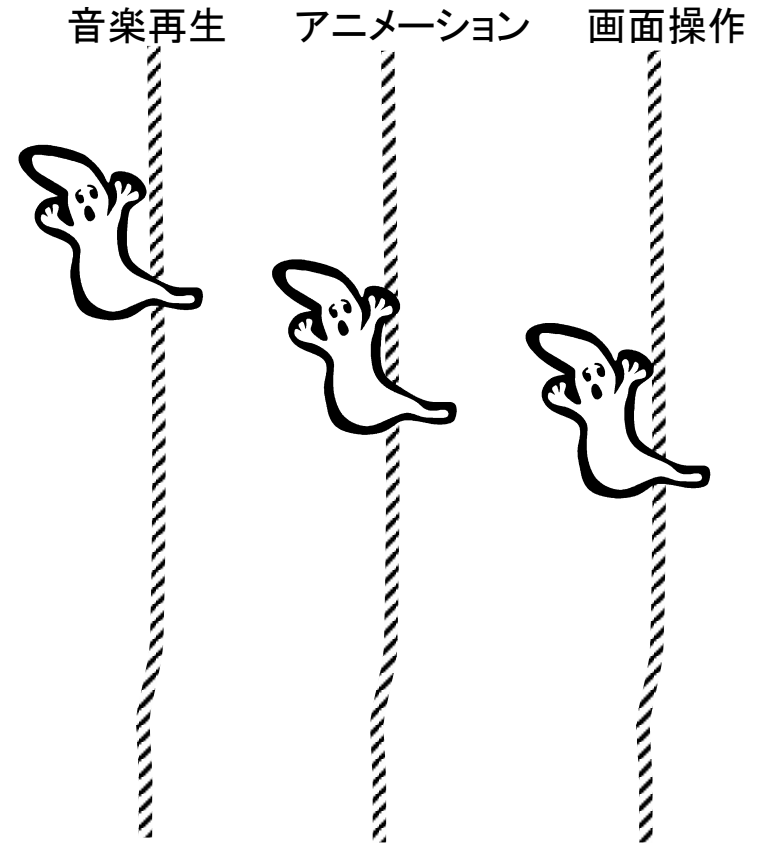
スレッドとは

- スレッド(Thread) : 英語で「糸」または「筋道」という意味. コンピュータプログラムでは, 処理を実行している主体をスレッドという.
- シングルスレッドのプログラム
 - これまで作成してきたプログラムは, スレッドが1本しか存在しないので, 「シングルスレッド (single thread)」のプログラムである.
 - シングルスレッドのプログラムでは, 1つの処理が終わると次の処理というように, 処理が順々に行われる. 従って, プログラムの実行開始から終了までの処理が, 1本の糸である.



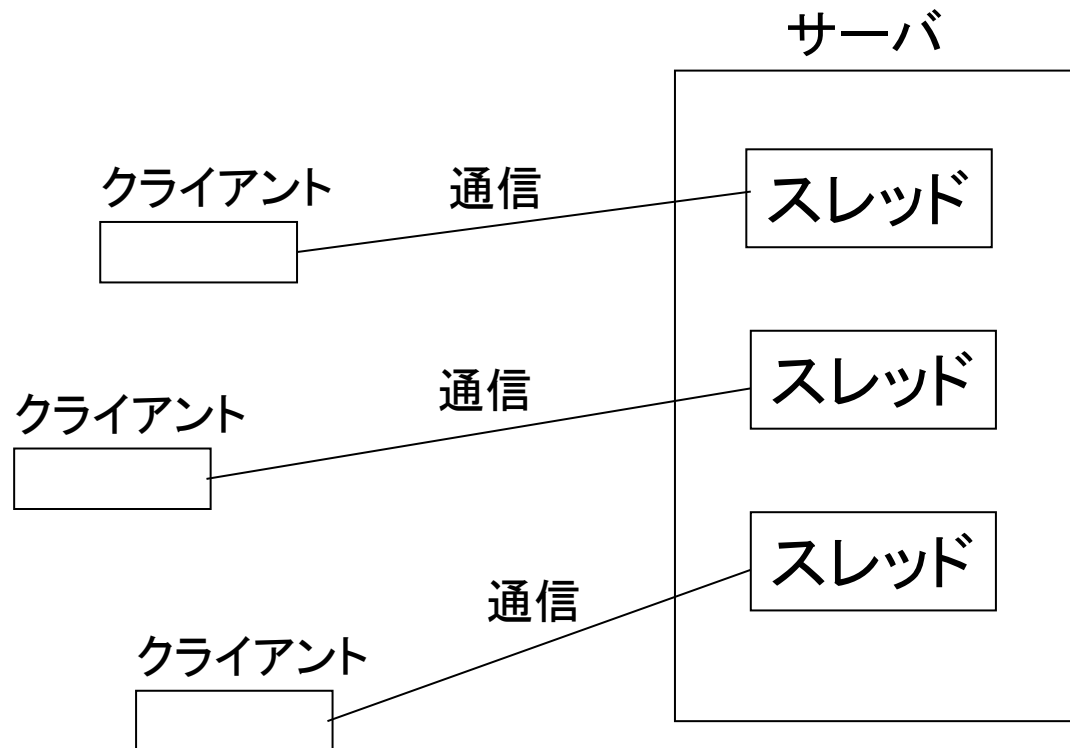
マルチスレッド

- マルチスレッドのプログラム
 - 1つのプログラム上でいくつもの処理を同時に実行しているプログラムをマルチスレッドプログラムという.
 - (例)
 - (1) Windows MediaPlayer: 音楽の再生, アニメーションの再生, 画面操作を平行して行う
 - (2) ネットワークのサーバ: 複数のクライアントに同時に応答

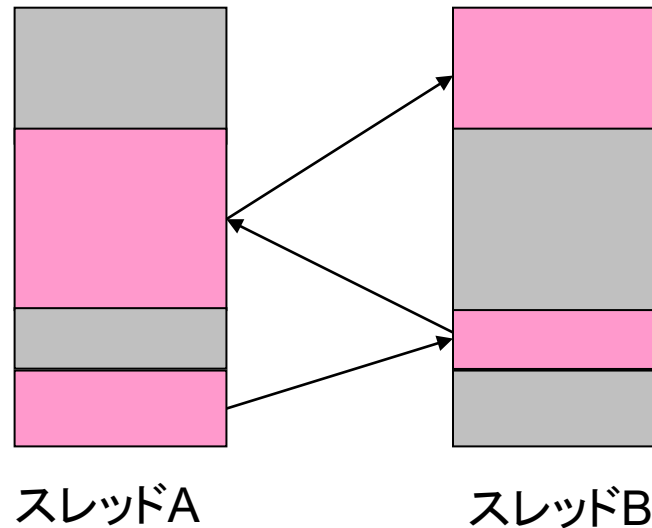


マルチスレッド

- 複数のクライアントを処理するサーバ



マルチスレッドの仕組み



- シングルコアのCPUが1つだけの場合、本当には、同時に複数のスレッドを実行することはできない。アプリケーション（JavaではJava仮想マシン）が、CPUの処理時間を非常に短い単位に分割し、複数のスレッドに順番に割り当てることによって、複数の処理があたかも同時に動いているかのように見せている。
- マルチプロセスやマルチタスクは、OS が管理する大がかりな並列処理機構であり、各プロセスが固有のメモリ空間を持つ。

Javaによるスレッドの実装方法

- (方法A) Threadクラスの拡張クラスを作る
- (方法B) Runnableインタフェースを実装したクラスを作る

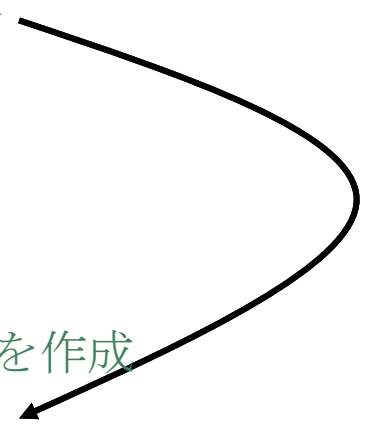
Threadクラスの拡張

(A) Threadクラスの拡張クラスを作る方法

- (1) Threadクラスの拡張クラスを作成する
- (2) そのクラスのrunメソッドを宣言する (オーバーライド)
- (3) そのクラスのインスタンスを作る
- (4) startメソッドを呼び出す

Threadクラスを用いた例

```
public class CountTestA {  
    public static void main(String[] args) {  
        CountA threadA = new CountA(); // (3) Threadのインスタンス生成  
        threadA.start(); // (4) startメソッドの呼び出し  
        for (int i=0; i<10; i++) {  
            System.out.println("main: "+i);  
        }  
    }  
}  
  
class CountA extends Thread { // (1) Threadクラスの拡張クラスを作成  
    public void run() { // (2) runメソッドのオーバーライド  
        for (int i=0; i <= 10; i++) {  
            System.out.println("Thread: " + i*10);  
        }  
    }  
}
```



Threadクラスを用いた例(Cont.)

- CountAというThreadクラスを拡張したクラスを作成
- CountAクラスの中で、Threadクラスのrun()をオーバーライドするrun()を作成
- CountTestAクラスのmainメソッドの中で、CountAクラスのインスタンスを作成.
- このインスタンスにstart()メソッドを適用して、スレッドをスタートさせる
- start()メソッドは実際にスレッドを起動するためのメソッドである(ここが新たなスレッドが生まれる瞬間).
- start()はrun()メソッドを呼び出す. これによってスレッドが動き出す.
- Countクラスのインスタンスを作成した時点では、まだスレッドは動き出していない.

実行結果

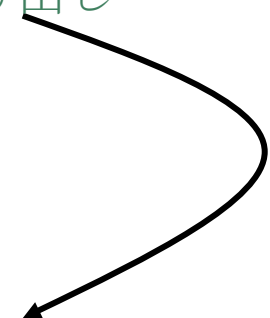
```
main: 0  
Thread: 0  
main: 1  
Thread: 10  
Thread: 20  
Thread: 30  
Thread: 40  
Thread: 50  
main: 2  
Thread: 60  
main: 3  
Thread: 70  
main: 4  
Thread: 80  
main: 5  
Thread: 90  
main: 6  
Thread: 100  
main: 7  
main: 8  
main: 9
```

- いつも同じ結果ではないことに注目！
- 2つのスレッドは独立に動いている
- どちらがどのタイミングで実行されるかは決まっていない

mainメソッドを持つクラスのスレッド化

//(1) mainメソッドを持つクラスをThreadの拡張クラスとして作成

```
public class CountTestA2 extends Thread {  
    public static void main (String[] args) {  
        //CountTestA2のインスタンス  
        CountTestA2 threadA2 = new CountTestA2();  
        threadA2.start(); // (4) startメソッドの呼び出し  
        for (int i=0; i<10; i++) {  
            System.out.println("main: "+i);  
        }  
    }  
  
    public void run() { //(2) runメソッドのオーバーライド  
        for (int i=0; i <= 10; i++) {  
            System.out.println("Thread: " + i*10);  
        }  
    }  
}
```



mainメソッドを持つクラスのスレッド化 (Cont.)

- CountTestA2自身をThreadクラスの変換クラスとして宣言
- 自分のインスタンスを作成(つまり, Threadクラスのインスタンス)
- 作成したインスタンスに対し, start()メソッドを呼び出し
- 実行結果は, 前と同じ

Runnableインタフェースの実装

(B) Runnableインタフェースを実装したクラスを作る方法

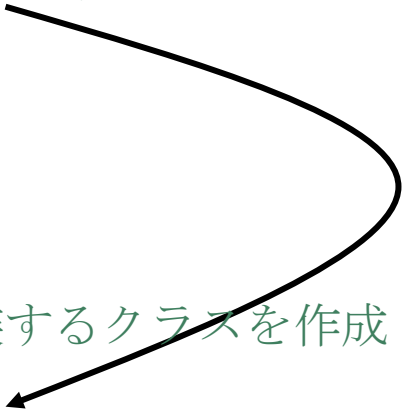
- (1) Runnableを実装するクラスを作成する
- (2) Runnableインタフェース中のrun()メソッドをそのクラスで実装
- (3) そのクラスのインスタンスを作る
- (4) これを引数にしたThreadオブジェクトを作る
- (5) startメソッドを呼び出す

- Runnableインタフェースは、run()メソッド持っている
- Runnableインタフェースを実装した(つまり、それに含まれるrun()メソッドも実装されている)クラスを作成.

Runnableインタフェース実装例

```
public class CountTestB {  
    public static void main(String[] args) {  
        CountB ctB = new CountB(); // (3) インスタンスを作成  
        Thread threadB = new Thread(ctB); //(4) Threadオブジェクトを  
        threadB.start(); // (5) startメソッドの呼び出し  
        for (int i=0; i<10; i++) {  
            System.out.println("main: "+i);  
        }  
    }  
}  
  
class CountB implements Runnable { //(1) Runnableを実装するクラスを作成  
    public void run() { //(2) runメソッドの実装  
        for (int i=0; i <= 10; i++) {  
            System.out.println("Thread: " + i*10);  
        }  
    }  
}
```

作成



2つの方法の比較

- クラス宣言の違い
 - CounteTestA→**class** CountA **extends** Thread
 - CounteTestB→**class** CountB **implements** Runnable
- スレッド開始時間の違い
 - CounteTestA→CountA threadA = **new** CountA();
threadA.start();
 - CounteTestB→CountB ctB = **new** CountB();
Thread threadB = **new** Thread(ctB);
threadB.start();

スレッド実装方法まとめ

	Threadクラスの拡張 (方法A)	Runnableインタフェースの実装 (方法B)
用途	簡単なクラス階層向け	複雑なクラス階層向け
スーパークラス	Threadクラス	なんでも良い
クラスの宣言	Threadクラスの拡張 class MyClass extends Thread { } }	Runnableインタフェースの実装 class MyClass implements Runnable { } }
スレッドの起動	startメソッドを呼ぶ MyClass mc = new MyClass; mc.start();	Thread経由でstartメソッドを 呼ぶ MyClass mc = new MyClass(); Thread th = new Thread(mc); th.start();
スレッド開始	runメソッド	runメソッド

・不必要に複雑なクラス階層にしないで良いインタフェースを使ったやり方の方が望ましいという考えもある

スレッドの排他制御

- 2つのスレッドが同じフィールドに代入しようとする
と問題が生じる
- そのためスレッドの順番待ちをさせる必要がある

2箇所のATMから同時に入金する

```
public class BankTest {  
    public static void main (String[] args) {  
        Account testAccount = new Account();  
        AccountManager cam1 = new AccountManager("cam1",  
testAccount);  
        AccountManager cam2 = new AccountManager("cam2",  
testAccount);  
        cam1.start(); //2つのスレッドを同時にスタート  
        cam2.start();  
    }  
}
```

入金を行うAccountManagerを2つスレッドで立ち上げる

スレッドの排他制御 (Cont.)


```
class AccountManager extends Thread { //Threadクラスの拡張クラスを作成
    String name; //フィールド
    Account targetAccount; //フィールド
    public AccountManager (String name, Account ac) { //コンストラクタ
        this.name = name;
        this.targetAccount=ac;
    }

    public void run () {
        for (int i =0 ;i<10; i++) {
            targetAccount.deposit(1000); //1000円入金
        }
    }
}
```

入金を行うAccountManagerをスレッドで実装
Accountに1回に1000円ずつ入金する

スレッドの排他制御 (Cont.)

```
class Account {  
    private int value = 0; //預金残高をフィールドとする  
  
    public void deposit (int money) {  
        int currentValue = value; //現在の預金残高  
        System.out.println(Thread.currentThread().getName()+  
            ":入金がありました");  
        value += money; //現在の預金残高に預金額を加算  
        //新規の預金残高から入金前の預金残高を引いて預金額を計算  
        System.out.println(Thread.currentThread().getName()+  
            ":預金額:"+(value-currentValue)+" "+"新規残額:"+value);  
        System.out.println(Thread.currentThread().getName()+  
            ":預金を完了しました");  
    }  
}
```



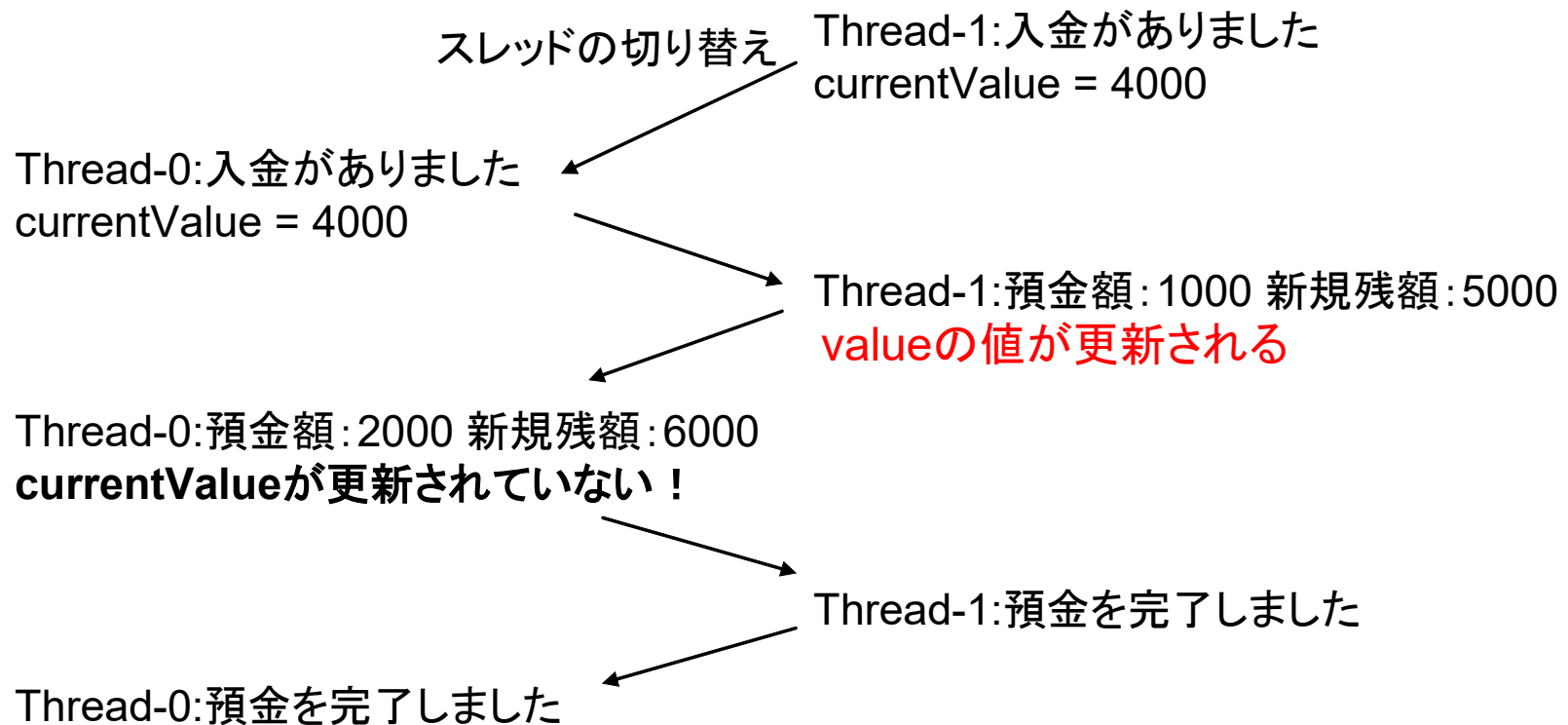
残高を管理するAccountクラス
入金があるとメッセージを表示する

実行結果

Thread-0:入金がありました
Thread-0:預金額：1000 新規残額：1000
Thread-0:預金を完了しました
Thread-0:入金がありました
Thread-0:預金額：1000 新規残額：2000
Thread-0:預金を完了しました
Thread-0:入金がありました
．．． 中略
Thread-0:入金がありました
Thread-1:入金がありました
Thread-0:預金額：1000 新規残額：5000
Thread-1:預金額：2000 新規残額：6000
Thread-0:預金を完了しました
Thread-1:預金を完了しました
Thread-0:入金がありました
Thread-1:入金がありました
Thread-0:預金額：1000 新規残額：7000
Thread-1:預金額：2000 新規残額：8000
Thread-0:預金を完了しました
Thread-1:預金を完了しました
．．． （ただし、毎回動作は異なる）

1回に2000円振り込んだことになっているのに、
残高は1000円しか増えていない！

なぜこのようなことが起こるのか



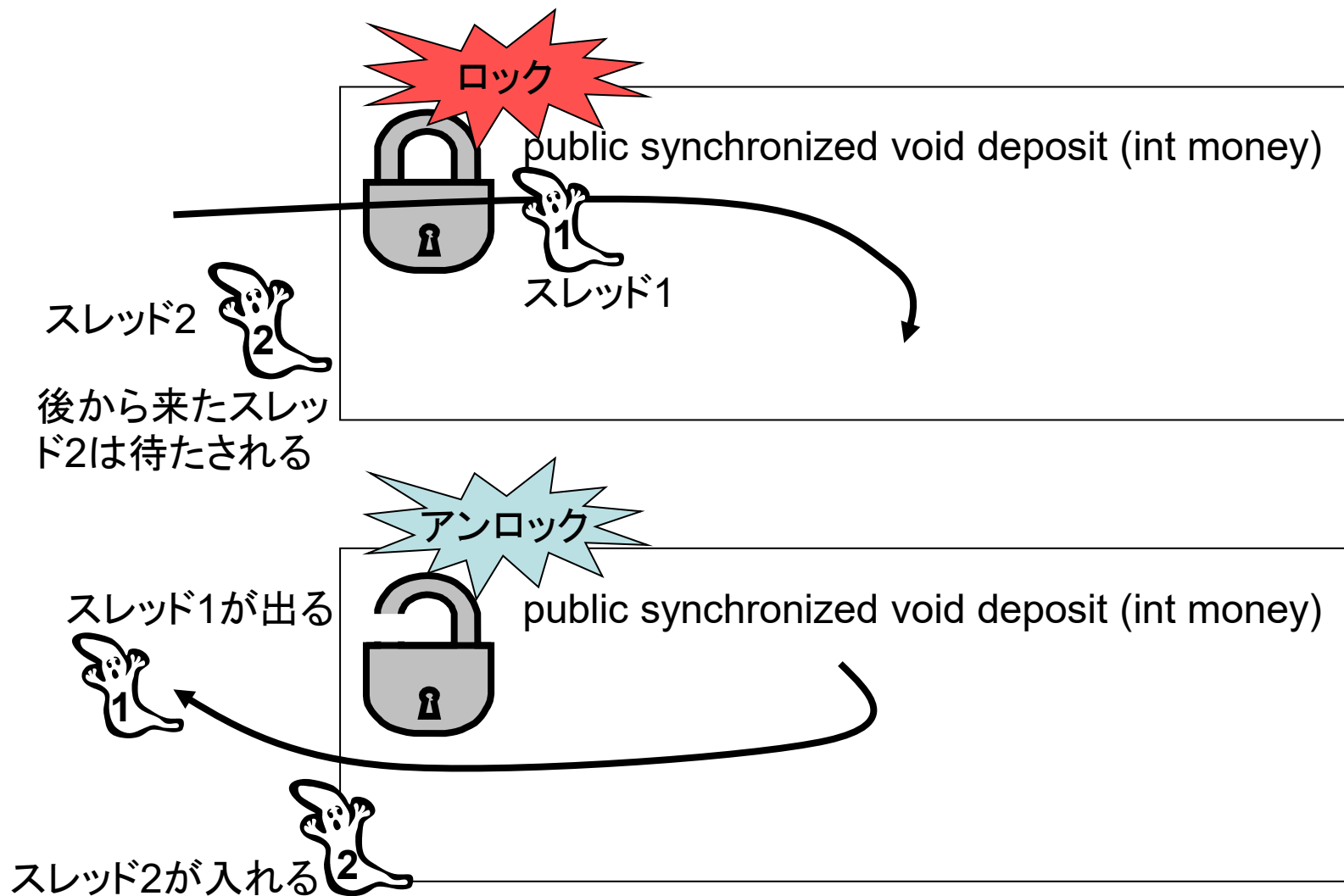
synchronizedメソッド

depositメソッドの宣言に” synchronized”をいれると排他制御ができる

```
public synchronized void deposit (int money)
```

- メソッドに”synchronized”をつけると、1つのスレッドがそのメソッドを実行中である間は、他のスレッドは同時に実行できない(ロックがかかった状態).
- 実行中のスレッドがそのメソッドの処理を終了するまで、他のスレッドは待たされる.

synchronizedメソッドの仕組み



実行例

Thread-0:入金がありました
Thread-0:預金額：1000 新規残額：1000
Thread-0:預金を完了しました
Thread-1:入金がありました
Thread-1:預金額：1000 新規残額：2000
Thread-1:預金を完了しました
Thread-1:入金がありました
Thread-1:預金額：1000 新規残額：3000
Thread-1:預金を完了しました
Thread-0:入金がありました
Thread-0:預金額：1000 新規残額：4000
Thread-0:預金を完了しました
Thread-1:入金がありました
Thread-1:預金額：1000 新規残額：5000
Thread-1:預金を完了しました
Thread-0:入金がありました
Thread-0:預金額：1000 新規残額：6000
Thread-0:預金を完了しました

Thread-1:入金がありました
Thread-1:預金額：1000 新規残額：7000
Thread-1:預金を完了しました
Thread-0:入金がありました
Thread-0:預金額：1000 新規残額：8000
Thread-0:預金を完了しました

正しく預金管理されている

synchronizedブロック

- メソッド全体ではなく、メソッドの一部だけにロックをかけることもできる.

synchronizedブロック
synchronized (ロックをかけるオブジェクト) { ...ロックしたい処理の内容 }

Synchronizedブロックを用いた実装

```
public void deposit (int money) {  
    synchronized (this) {  
        int currentValue = value; //現在の預金残高  
        System.out.println(Thread.currentThread().getName()+  
            ":入金がありました");  
        value += money; //現在の預金残高に預金額を加算  
        //新規の預金残高から入金前の預金残高を引いて預金額を計算  
        System.out.println(Thread.currentThread().getName()+  
            ":預金額:"+(value-currentValue)+" "+"新規残額:"+value);  
        System.out.println(Thread.currentThread().getName()+  
            ":預金を完了しました");  
    }  
}
```

スレッドの休止

スレッドを休止させる

```
Thread.sleep(long msec);
```

- スレッドを休止させたいとき
 - スレッドの動作が速すぎて不都合なとき
 - スレッドの動きに時間間隔をあけたいとき
- このメソッドは static 宣言されたクラスメソッドなので、オブジェクト・メソッドとしてではなく、Thread クラスのメソッドとして呼び出す
- Thread.sleepメソッドはInterruptedExceptionを投げる可能性があるので、try...catchでくくっておく

Thread休止の例

```
public class ThreadTest {  
    public static void main (String[] args) {  
        System.out.println("開始");  
        try {  
            Thread.sleep(5000);  
        } catch (InterruptedException e) {  
            e.printStackTrace();  
        }  
        System.out.println("終了");  
    }  
}
```