# 上級プログラミング１（第１１回）

工学部　情報工学科
杉本　徹

# 今日のテーマ

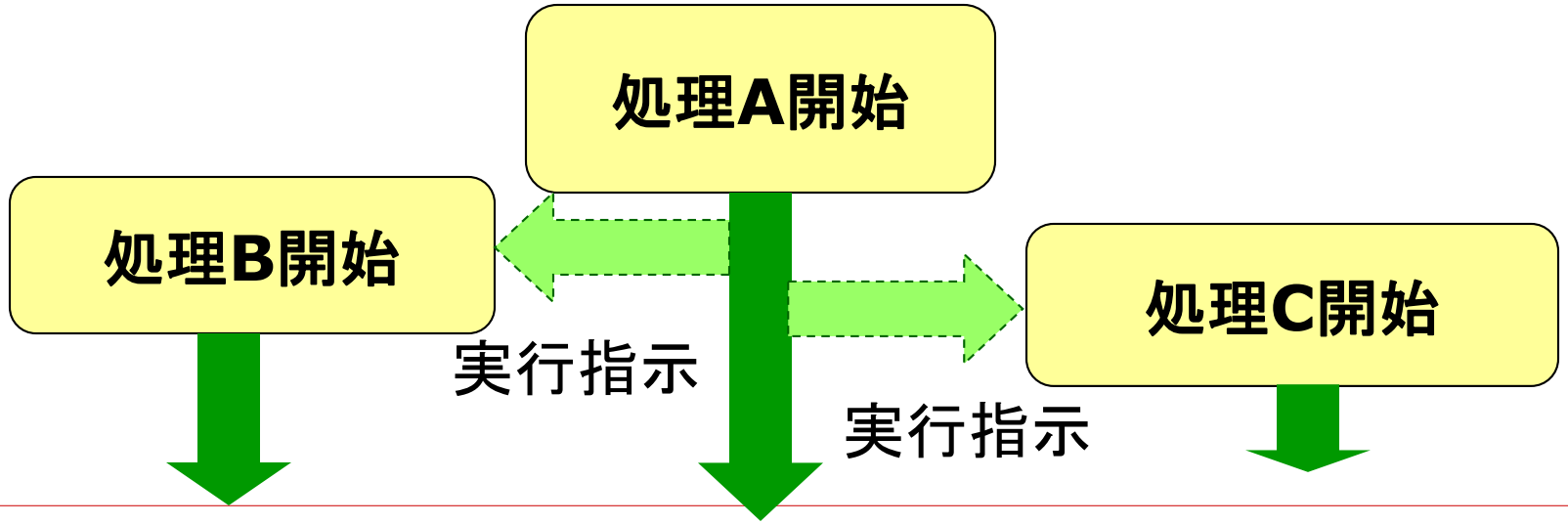
- スレッドの復習
- スレッドの同期
  - 銀行口座にたとえて

- リスト処理（Collectionライブラリ）

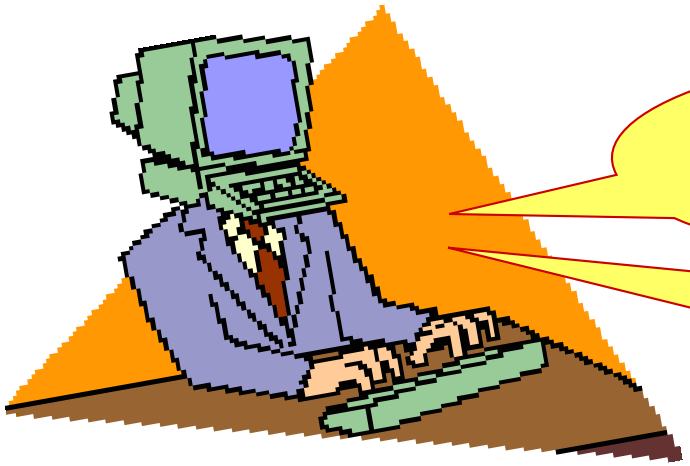
# スレッドの復習

# マルチスレッド

- 1つのプログラム（正確にはプロセス）の中で、処理の流れ（スレッド）を同時に複数実行するしくみ
- 複数のスレッドが同時に複数実行されるプログラムをマルチスレッドプログラムと呼ぶ（普通のプログラムはシングルスレッドプログラム）



# 例えていうなら



メール送ろうっと！

ああっ、レポートも
つくらなきゃ！

メール作成開始

別のスレッドを実行

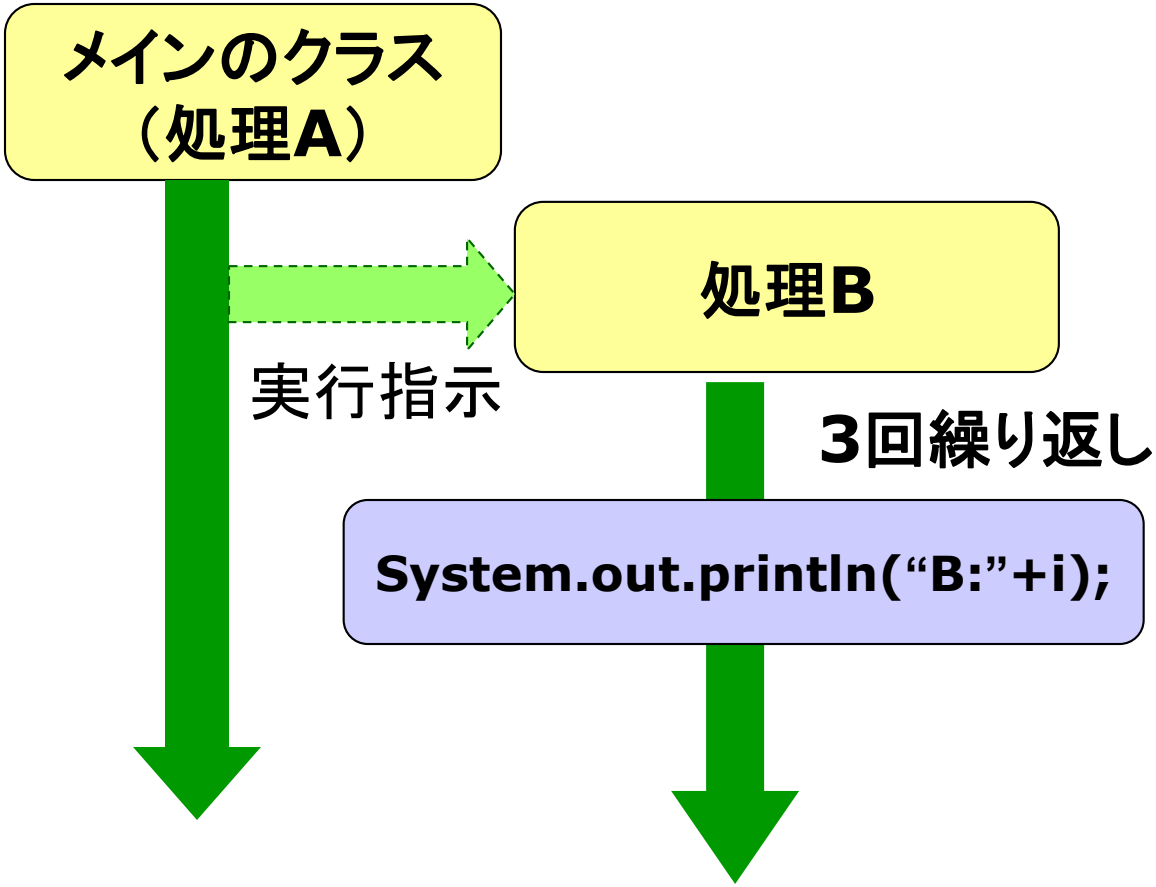
レポート作成開始

複数の処理を
同時進行で行う

# マルチスレッドを実現するクラスの作り方

- Threadクラスを継承する方法
  - Threadクラスを継承したクラスのオブジェクトを生成
- Runnableインターフェイスを実装したクラスを利用する方法
  - そのオブジェクトをコンストラクタの引数としてThreadクラスのオブジェクトを生成

# マルチスレッドプログラムの簡単な例



# ソースプログラム(1)　処理B

```
class ThreadSmpl extends Thread {
        public void run(){
                for(int i=0; i<3; i++){
                        System.out.println("B:"+i);
                }
        }
}
```

① Threadクラスを継承する
② run()メソッドをオーバーライドし、同時実行させる処理を記述する。
run()メソッドはstart()メソッドから呼び出される

# ソースプログラム(2)　処理A

```
class ThreadMain {
        public static void main(String[] args) {
            ThreadSmpl shoriB = new ThreadSmpl();
            shoriB.start();
            for(int i=0; i<3; i++){
                        System.out.println("A:"+i);
            }
            System.out.println("Main is ended.");
        }
}
```

処理Bを新規スレッドとしてstart()メソッドで実行後、A:･･･を表示

**startメソッド：** 新たにスレッドを生成し，そのrunメソッドを呼び出す

# スレッドを作るもう一つの方法
# ソースプログラム(1’)　処理B

```
class ThreadSmpl implements Runnable {
        public void run(){
                for(int i=0; i<3; i++){
                        System.out.println("B:"+i);
                }
        }
}
```

① Runnableインターフェイスを実装する
② run()メソッドをオーバーライドし、同時実行させる処理を記述する。

# ソースプログラム(2') 処理A

```
class TreadMain {
        public static void main(String[] args) {
            ThreadSmpl smpB = new ThreadSmpl();
            Thread shoriB=new Thread(smpB);
            shoriB.start();
            for(int i=0; i<3; i++){
                        System.out.println("A:"+i);
            }
            System.out.println("Main is ended.");
        }
}
```

# スレッドの休止

**Threadクラスのsleepメソッドを使う**

```
class ThreadSmpl extends Thread {
    public void run(){
        for(int i=0; i<3; i++){
          try{
                Thread.sleep(1000); //単位はミリ秒
          }catch(Exception  e){
                e.printStackTrace();
          }
          System.out.println("B:"+i);
        }
    }
}
```

# スレッドの同期

# 二人で使っている銀行口座にお金を預けたい

銀行口座

残金＝¥3000

¥5000

takashi

¥5000

¥5000

naoto

¥5000

# 次のようにクラスを定義する

```
class Bank{
   int  okane=3000;
   void addOkane(int i){
      int work=okane;
      try{
         Thread.sleep(1000)
      }catch(Exception e){}
        okane=work+i;
   }
   int getOkane(){
        return okane;
   }
}
```

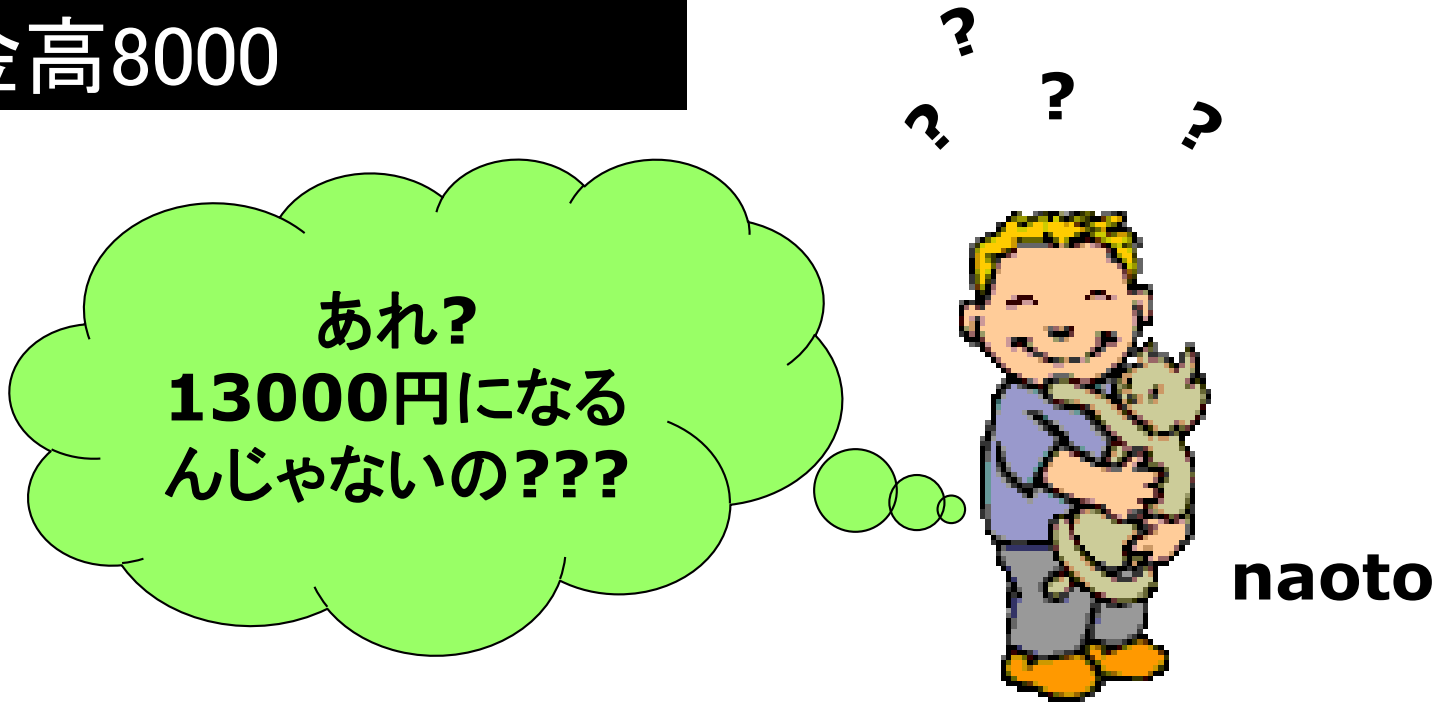
```
class Boy extends Thread{
    Bank b;
    Boy(Bank b){
          this.b=b;
    }
    public void run(){
       b.addOkane(5000);
    }
}
```

# 単純に考えるとこれでよさそうだが

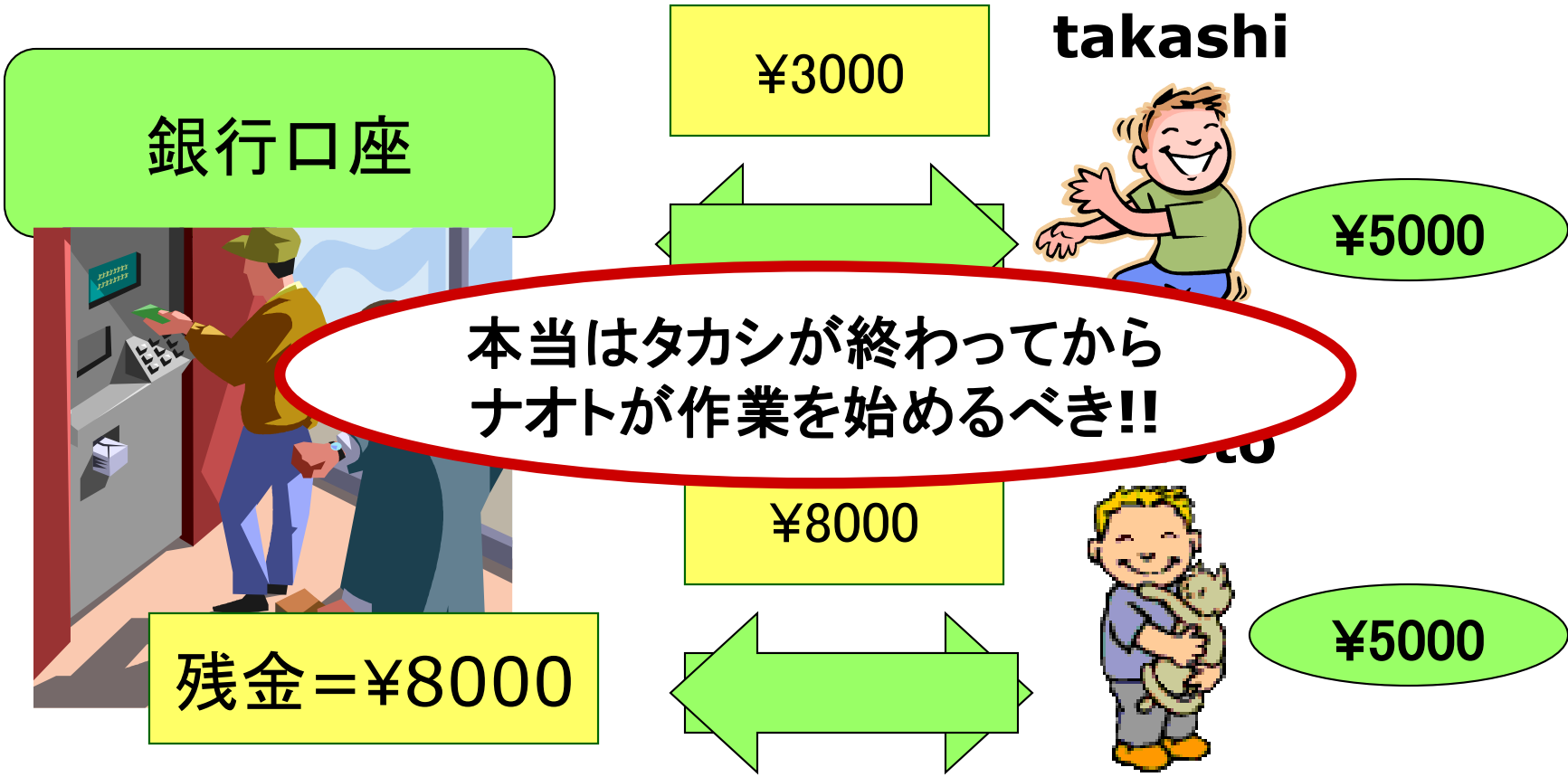
```
class NonSync{
        public static void main(String[] args){
            Bank b = new Bank();
            Boy takashi = new Boy(b);
            Boy naoto= new Boy(b);
            takashi.start();   //タカシがお金を預ける
            naoto.start();     //ナオトがお金を預ける
            try{        takashi.join();    //二人が振り込み
                        naoto.join();      //終わるまで待つ
            }catch(Exception e){e.printStackTrace();}
            System.out.println("預金高"+b.getOkane());
        }
}
```

# 実行してみると・・・

```
C:¥>java NonSync
預金高8000
```



# 原因は？


銀行口座
¥3000
takashi
¥5000
本当はタカシが終わってから
ナオトが作業を始めるべき!!
¥8000
¥5000
残金＝¥8000

# スレッドセーフ

- もとのプログラムをマルチスレッド化しても問題が起こらないこと、もしくはそのためのスレッドの仕組み
  - 単一スレッドを動かすだけでは問題は顕在化しないことが多い
  - オブジェクトや変数を複数のスレッドで共有し、変更を加える場合などで問題が発生する場合がある
- この例は、明らかにスレッドセーフではない

# どうしたらいい？

- タカシの操作が終わるまで、ナオトの処理を待たせればよい
- すなわち、スレッド間の同期をとればよい
  - 同期をとるべきメソッドにsynchronized演算子を付ける
  - 同期をとるべき部分をsynchronizedブロックで囲む

# メソッドにsynchronized修飾子をつける方法（ほかは変えない）

```
class Bank{
        int  okane=3000;
        synchronized void addOkane(int i){
                int work=okane;
                try{ Thread.sleep(1000);
                }catch(Exception e){}
                okane=work+i;
        }
        int getOkane(){
                return okane;
        }
}
```

あるスレッドがこの処理を終えるまでロックがかかり他のスレッドは待たされる

# 同期を取る部分をsynchronizedブロックで囲む方法（ほかは変えない）

```
class Boy extends Thread{
    int i=0;    Bank b;
    Boy(Bank b){
       this.i=b.getOkane();
       this.b=b;
    }
    public void run(){
        synchronized(b){
                b.addOkane(5000);
        }
    }
}
```

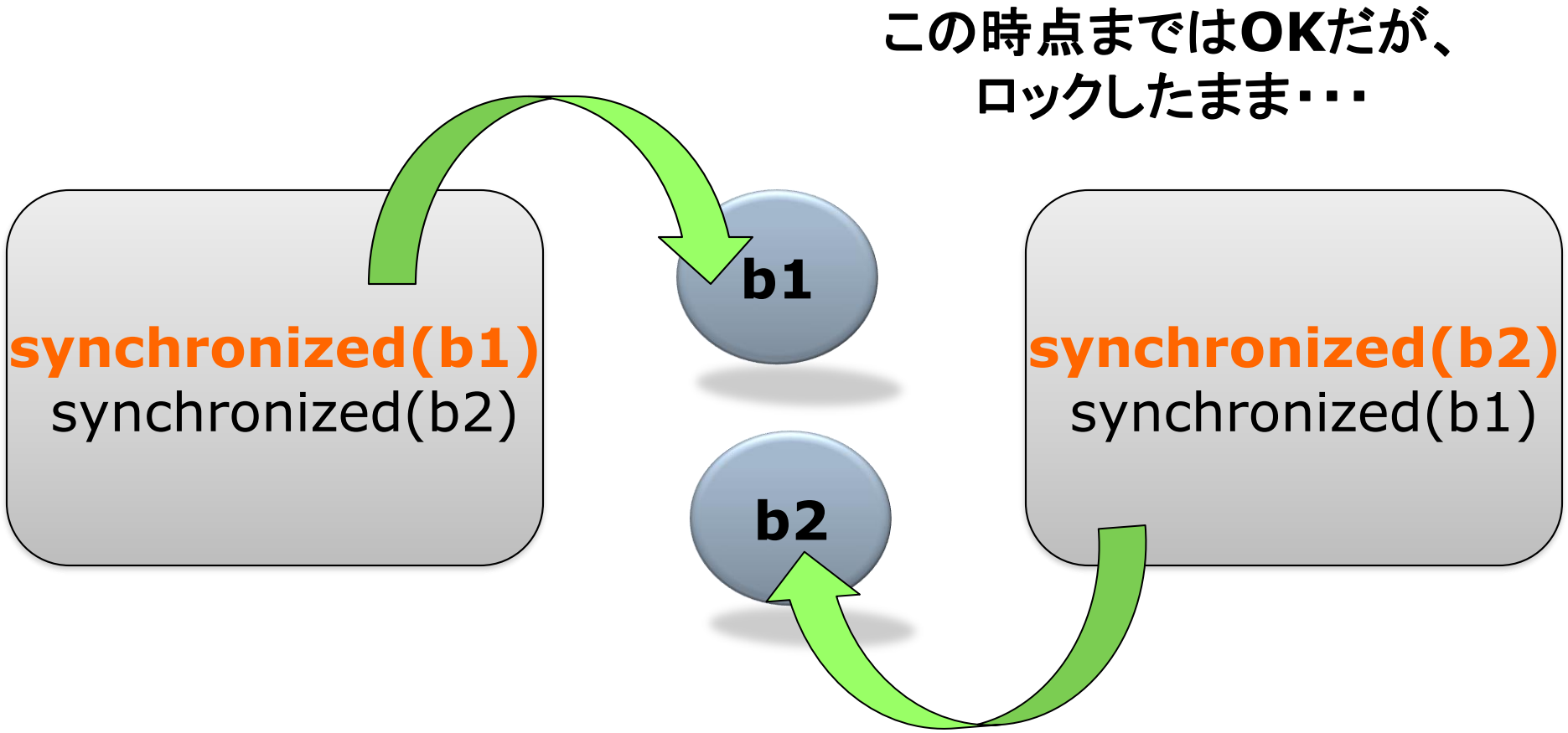
**引数のオブジェクト（今の場合b）がアクセスされている間は他のスレッドはアクセスできない（ロックされる）**

# 注意しなければならない点

あるオブジェクトをロックしながら、
別のオブジェクトをロックすると問題が起こる場合がある

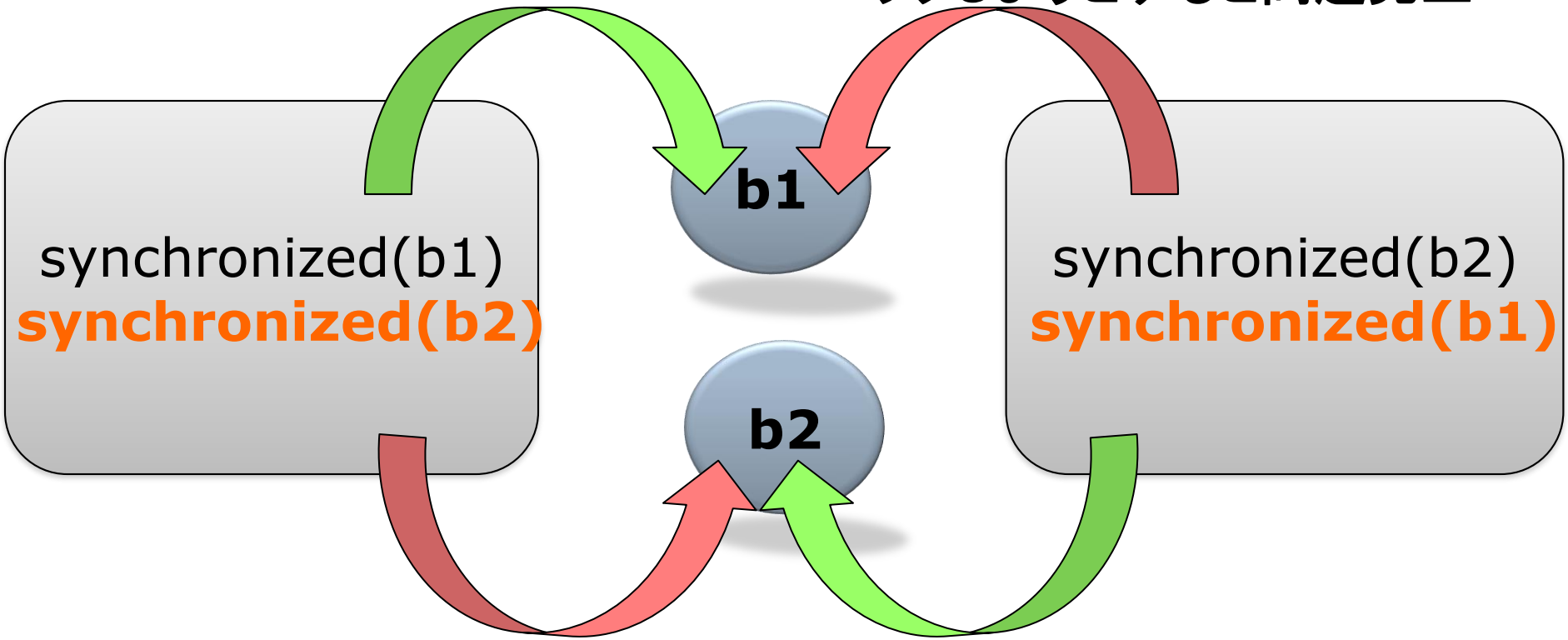
```
synchronized(b1){
        ...
        ...
        synchrnoized(b2){
                ...
                ...
        }
}
```

# こんなことが起こりうる(1)



# こんなことが起こりうる(2)
## ・・・・デッドロック

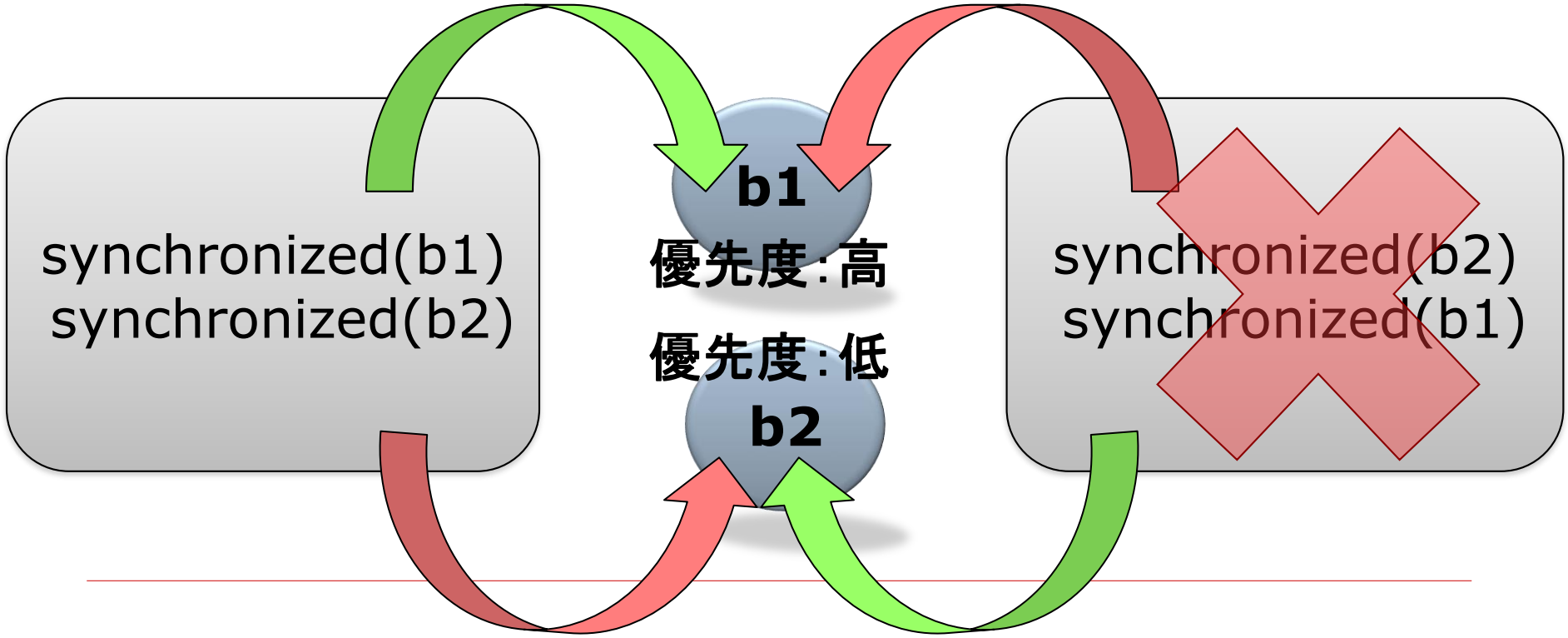
ロックしたまま次のオブジェクトを
ロックしようとすると問題発生




すでにロックされているオブジェクトを互いに待ち続ける

# デッドロックの解消法

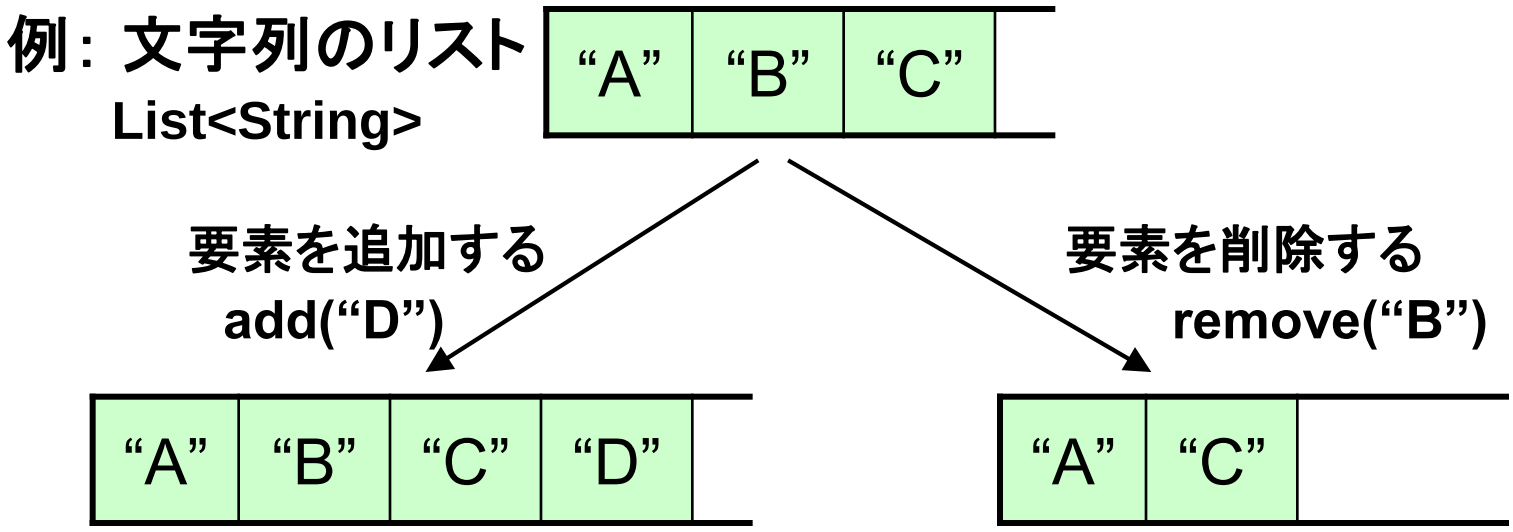
- ロックをかけられるオブジェクトに優先順位をつけて、必ずその順番でロックをかける



# リスト処理（Collectionライブラリ）

# リスト（List）とは？

- （同じ型の）オブジェクトを並べたもの
- 配列と似ているが，後から要素を自由に追加したり削除したりできる点が異なる ⇒ 便利！



# リストと配列の比較

| | リスト | 配列 |
|---|---|---|
| 要素の型 | 最初に指定（オブジェクト型のみ） ※注 | 最初に指定（基本型，オブジェクト型） |
| 要素の個数 | 後から自由に要素の追加・削除が可能 | 固定（配列オブジェクト初期化時に指定） |
| 使用の準備 | import java.util.* | — |
| 型名 | 例： List<String> | 例： String[] |
| 初期化 | new ArrayList<String>() | new String[10] |
| 要素の参照 | 例： l.get(3) | 例： a[3] |
| 要素数を求める | 例： l.size() | 例： a.length |
| 主なメソッド | 要素の追加・削除，検索，ソートなど | 検索，ソートなど（Arrays クラス） |

※ 注： 基本型（intなど）の値を要素として代入することは可能

例： List<Integer> l = new ArrayList<Integer>(); l.add(123);

# List は（クラスではなく）インタフェース

- 復習：インタフェース
  - どんなメソッドが必要かという宣言のみ
    （例：add, remove, get, sizeなどのメソッドが
    必要だと宣言している）
  - 各メソッドの実現方法の定義は含まないため，
    直接インタスタンスを作ることはできない
- Listインタフェースを実装するクラス
  - ArrayListクラス
  - LinkedListクラス　など

# リストの使い方（例）

- 宣言，初期化：List<String> l = new ArrayList<String>();
- 要素の追加： l.add(“A”); l.add(挿入位置, “A”)
- 要素の削除： l.remove(“B”); l.remove(削除位置);
- 値の参照： l.get(参照したい位置);
- リストの全要素にわたる繰り返し処理
  - 方法1（配列風）：
    for (int i = 0; i < l.size(); i++) { ･･･ l.get(i) ･･･ }
  - 方法2（Java 5 で導入された拡張for文を使用）
    for (String s : l) { ･･･ s ･･･ }

# その他の便利なメソッド

- java.util.List に用意されたメソッド
  - ある値がリストに含まれるか検査　l.contains(値)
  - ある値の出現位置を求める　l.indexOf(値)

- java.util.Collections に用意されたメソッド
  - 要素のソート　Collections.sort(リスト)
  - 他に，要素の置換，並べ替え，最大値抽出，出現回数カウントなどのメソッドが用意されている

# 演習：List の使い方に慣れる

```
import java.util.*;
public class Test6 {
	public static void main(String[] args) {
		List<String> list = new ArrayList<String>();
		list.add(“this");
		list.add("is");
		list.add("a");
		list.add("pen");
		System.out.println(list.size());
		for (int i = 0; i < list.size(); i++) {
			System.out.println(list.get(i));
		 }
	}
}
```

1. リストlistから “pen” を削除し，”book” を追加（remove, add メソッド）
2. リストlistに “is” という要素が含まれているか検査（containsメソッド）
3. リストlistの全要素を順に出力する処理を拡張for文を使うものへ書き換え
4. リストlistをアルファベット順にソート（Collections.sortメソッド）

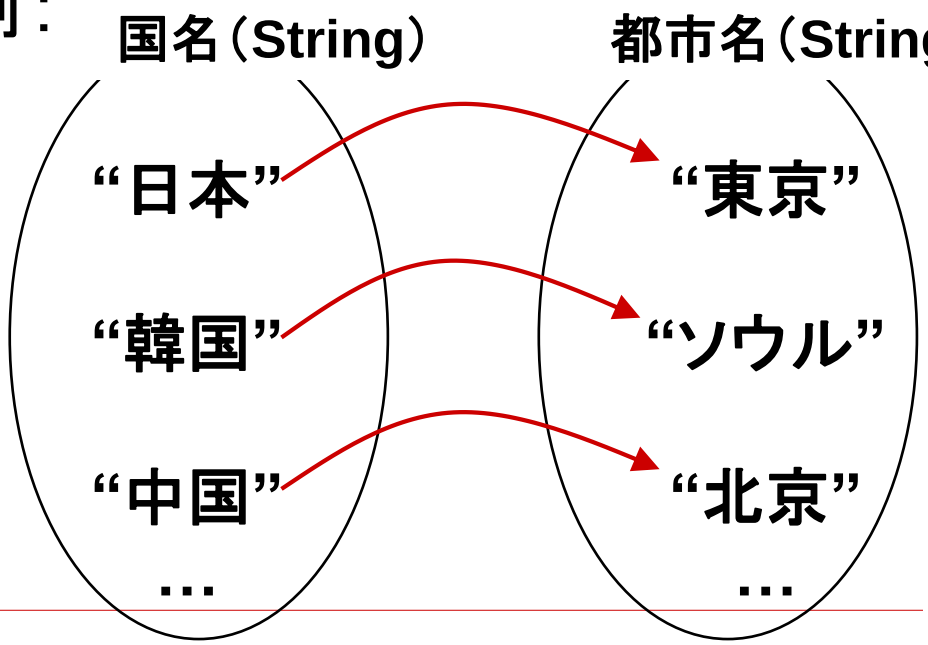
# 参考：歴史的なメモ

- Vector クラス
  - 古くからある．最近はあまり使われない
- Collection ライブラリ（List, Set, Map など）
  - Java 1.2 から導入された強力なライブラリ
- Java 5 からの新機能
  - List の要素の型を指定できるようになった
    - 以前は単に「List」⇒ Java 5 からは「List<String>」
  - 拡張 for文
  - auto boxing/unboxing（基本型も使いやすく）

# 参考：マップ（Map）の利用

- Map<型1, 型2>　型1のオブジェクトと型2のオブジェクトの対応関係（写像）を表す

- 例：




**メソッド put(key, value)**

**指定したキー key に，指定した値 value を対応づける**

**メソッド get(key)**

**指定したキー key に対応づけられた値を返す**

# 参考：Mapを使ったプログラム例

```
import java.util.*;

public class Test {
    public static void main(String[] args) {
        Map<String, String> map = new  HashMap<String, String>();
        map.put("日本", "東京");
        map.put("韓国", "ソウル");
        map.put("中国", "北京");
        System.out.println("日本の首都は" + map.get("日本"));
        for (String str : map.keySet()) {
            System.out.println(str + "の首都は" + map.get(str));
        }
    }
}
```

注：**Map**はインタフェース，**HashMap**は**Map**を実装したクラス

# 来週の予定

- 1限：期末試験
  - 資料持ち込み不可
  - 初回～今回の授業で説明した概念，用語，キーワード，考え方などを広く復習しておくこと
- 2限：演習
  - 最終課題（要レポート提出）を出題