発展プログラミング

第8回：排他制御

宮田章裕 <miyata.akihiro@nihon-u.ac.jp>

前回講義の復習

マルチスレッドプログラミング

- 方法1：Threadクラスを継承する
- 方法2：Runnableインタフェースを実装する

Threadクラスの継承による マルチスレッドプログラミング (1/4)

手順

[1] Threadクラスを継承したクラスを作成する
[2] そのクラス内でrun()をオーバーライドする
[3] そのクラスのインスタンス経由でstart()を呼び出す

MyThread1.java のアウトライン

```
                    [1]
public class MyThread1 extends Thread {
  [2]
   public void run() {

       // Do something.

   }

}
```

Main.java のアウトライン

```
public class Main {

    public static void main(String[] args) {
        MyThread1 thread = new MyThread1();
  [3]   thread.start();

        // Do something.
    }

}
```

これらの処理が
並行して行われる

Threadクラスの継承による マルチスレッドプログラミング (2/4)

MyThread1.java

```java
public class MyThread1 extends Thread {

    private final static long SLEEP_LEN_MSEC = 1000;


    public void run() {
        for(int i = 0; i < 10; i++) {
            System.out.println("MyThread1: " + i);
            try {
                Thread.sleep(SLEEP_LEN_MSEC);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }

}
```

Main.java

```java
public class Main {

    private final static long SLEEP_LEN_MSEC = 1000;


    public static void main(String[] args) throws InterruptedException {
        MyThread1 thread = new MyThread1();
        thread.start();

        for(int i = 0; i < 10; i++) {
            System.out.println("Main: " + i);
            Thread.sleep(SLEEP_LEN_MSEC);
        }
    }

}
```

- run()
  Threadクラスで定義されているメソッド。
  サブクラスでオーバーライドして利用する。

- Thread.sleep()
  当該スレッドの処理を指定時間止めるメソッド。
  本質的には不要であるが，実行結果の視認性を高めるために便宜上利用している。

- InterruptedException
  一定時間スリープするThread.sleep()を実行する際に生じうる例外。
  MyThread1のrun()はThreadクラスからオーバーライドしたものであり，
  サブクラスで定義を変更してthrowsを付けることができないため，
  try・catchで例外処理を実装している。

```
% javac Main.java
% java Main
Main: 0
MyThread1: 0
Main: 1
MyThread1: 1
MyThread1: 2
Main: 2
. . .
```

実行するたびに表示順が異なる

Threadクラスの継承による マルチスレッドプログラミング (3/4)

MyThread1.java

```java
public class MyThread1 extends Thread {

    private final static long SLEEP_LEN_MSEC = 1000;

    public void run() {
        for(int i = 0; i < 10; i++) {
            System.out.println("MyThread1: " + i);
            try {
                Thread.sleep(SLEEP_LEN_MSEC);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }

}
```

D

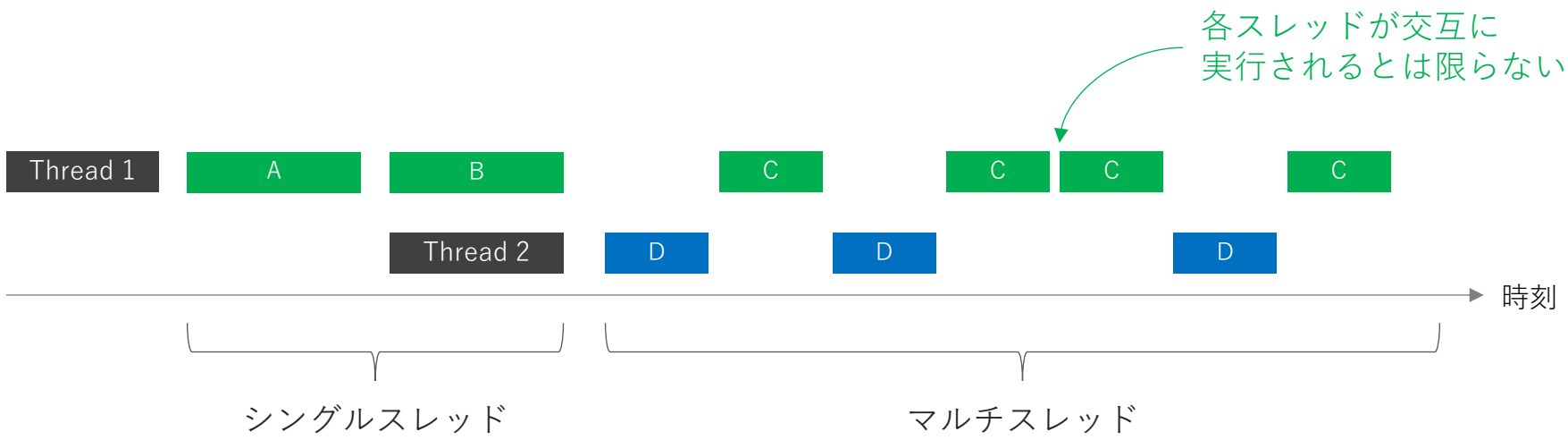
Main.java

```java
public class Main {

    private final static long SLEEP_LEN_MSEC = 1000;

    public static void main(String[] args) throws InterruptedException {
        MyThread1 thread = new MyThread1();
        thread.start();

        for(int i = 0; i < 10; i++) {
            System.out.println("Main1: " + i);
            Thread.sleep(SLEEP_LEN_MSEC);
        }
    }

}
```

A

B

C



Threadクラスの継承による マルチスレッドプログラミング (4/4)

- Threadクラスを継承する方法の問題点
  - Javaでは多重継承が許されていない（1つのクラスしか継承できない）ため Threadクラスを継承してしまうと他のクラスを継承できない
  - 公式ドキュメントにおいてもこの方法は相対的に一般的ではなく，制約があるので，もう一方の方法（後述の方法2）の利用を推奨している
    https://docs.oracle.com/javase/tutorial/essential/concurrency/runthread.html

MyThread1.java

```java
public class MyThread1 extends Thread {

    private final static long SLEEP_LEN_MSEC = 1000;


    public void run() {
        for(int i = 0; i < 10; i++) {
            System.out.println("MyThread1: " + i);
            try {
                Thread.sleep(SLEEP_LEN_MSEC);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }

}
```

マルチスレッドプログラミング

- 方法1：Threadクラスを継承する
- 方法2：Runnableインタフェースを実装する

Runnableインタフェースの実装による マルチスレッドプログラミング (1/2)

手順

[1] Runnableインタフェースを実装したクラスを作成する
[2] そのクラス内でrun()を実装する
[3] そのクラスのインスタンスを引数にしてThreadクラスのインスタンスを作成する
[4] Threadクラスのインスタンス経由でstart()を呼び出す

MyThread3.java のアウトライン

```
                    [1]
public class MyThread3 implements Runnable {
  [2]
   public void run() {
       // Do something.
   }
}
```

Main.java のアウトライン

```
public class Main {

    public static void main(String[] args) {
  [3]   MyThread3 mt = new MyThread3();
        Thread thread = new Thread(mt);
  [4]   thread.start();

        // Do something.
    }
}
```

これらの処理が
並行して行われる

Runnableインタフェースの実装による マルチスレッドプログラミング (2/2)

MyThread3.java

```java
public class MyThread3 implements Runnable {

    private final static long SLEEP_LEN_MSEC = 1000;


    public void run() {
        for(int i = 0; i < 10; i++) {
            System.out.println("MyThread3: " + i);
            try {
                Thread.sleep(SLEEP_LEN_MSEC);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }

}
```

Main.java

```java
public class Main {

    private final static long SLEEP_LEN_MSEC = 1000;


    public static void main(String[] args) throws InterruptedException {
        MyThread3 mt = new MyThread3();
        Thread thread = new Thread(mt);
        thread.start();

        for(int i = 0; i < 10; i++) {
            System.out.println("Main: " + i);
            Thread.sleep(SLEEP_LEN_MSEC);
        }
    }

}
```

```
% javac Main.java
% java Main
Main: 0
MyThread3: 0
Main: 1
MyThread3: 1
MyThread3: 2
Main: 2
. . .
```

実行するたびに表示順が異なる

スレッドの待ち合わせ

スレッドの待ち合わせ (1/3)

- 単純な実装では各スレッドが独立して動いてしまう
  - Thread2・3終了後に，Thread1で特定の処理を実行，ということができない

MyThread1.java

```java
public class MyThread1 implements Runnable {

    private final static long SLEEP_LEN_MSEC = 1000;

    private int id;


    public MyThread1(int id) {
        this.id = id;
    }

    public void run() {
        for(int i = 0; i < 10; i++) {
            System.out.println("MyThread #" + id + ": " + i);
            try {
                Thread.sleep(SLEEP_LEN_MSEC);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }

}
```

C

Main.java

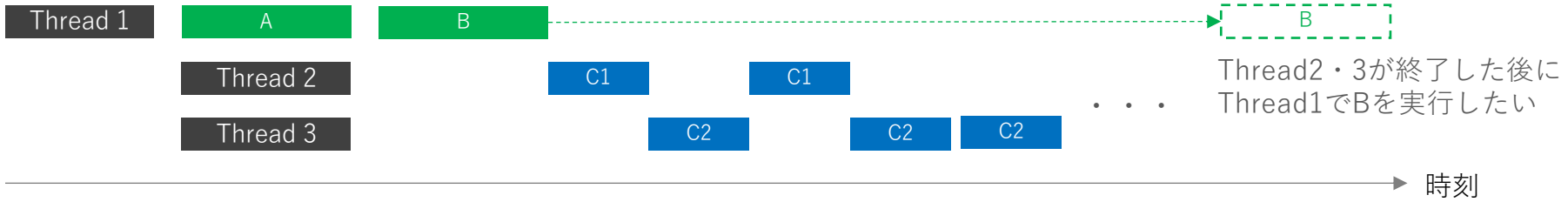
```java
public class Main {

    private final static int THREAD_COUNT = 2;

    public static void main(String[] args) {
        for(int i = 0; i < THREAD_COUNT; i++) {
            MyThread1 mt = new MyThread1(i);
            Thread thread = new Thread(mt);
            thread.start();
        }

        System.out.println("FINISH");
    } Bが最後に実行されることを
}     期待するが，実際はそうならない
```

A

B

実行結果

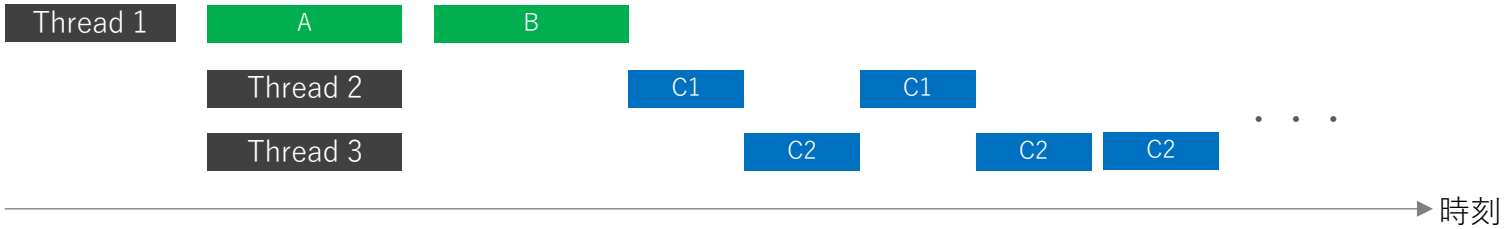
```
FINISH
MyThread #1: 0
MyThread #0: 0
MyThread #1: 1
MyThread #0: 1
MyThread #1: 2
MyThread #0: 2
MyThread #1: 3
MyThread #0: 3
MyThread #1: 4
MyThread #0: 4
MyThread #1: 5
MyThread #0: 5
MyThread #1: 6
MyThread #0: 6
MyThread #1: 7
MyThread #0: 7
MyThread #1: 8
MyThread #0: 8
MyThread #1: 9
MyThread #0: 9
```



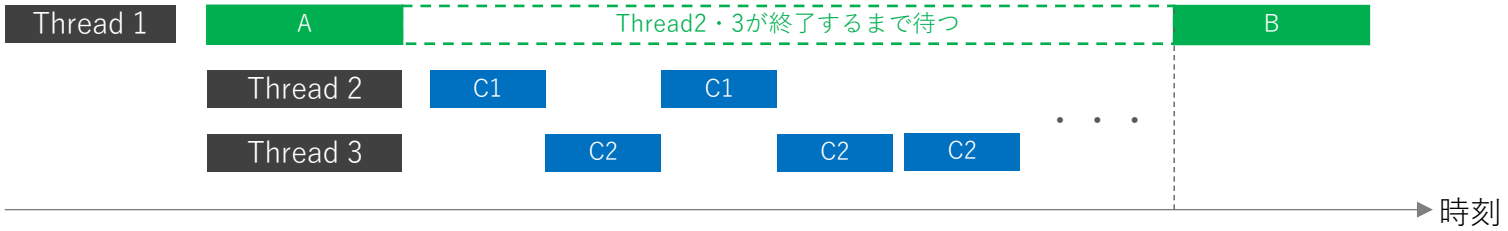
スレッドの待ち合わせ (2/3)

- join()で特定のスレッドの終了を待つことができる

join()を使わない場合（各スレッドが独立して動く）



Thread1でjoin()を使う場合（Thread2・3の終了を待ってからThread1が動く）



スレッドの待ち合わせ (3/3)

Counter.java

```java
public class Counter implements Runnable {

    private final static long SLEEP_LEN_MSEC = 1000;

    private int id;

    public Counter(int id) {
        this.id = id;
    }

    public void run() {
        for(int i = 0; i < 5; i++) {
            System.out.println("Counter #" + id + ": " + i);
            try {
                Thread.sleep(SLEEP_LEN_MSEC);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }

}
```

Main.java

```java
public class Main {

    public static void main(String[] args) throws InterruptedException {
        Counter c = new Counter(0);
        Thread thread = new Thread(c);
        thread.start();

        thread.join();

        System.out.println("FINISH");
    }

}
```

thread.join()の実行に必要

ここで，threadインスタンスで
開始したスレッドが終了するのを待つ
（この行をコメントアウトして挙動の変化を確認してみよう）

実行結果

```
Counter #0: 0
Counter #0: 1
Counter #0: 2
Counter #0: 3
Counter #0: 4
FINISH
```

狙い通り，他スレッドが終了した後で
System.out.println(“FINISH”); が実行される

作業準備

- 本日の作業ディレクトリの作成・移動
  - mkdir -p SOMEWHERE/2021_ap/08
  - 以降，SOMEWHERE/2021_ap/08をWORK_DIRとする

- 作業ディレクトリの作成・移動
  - cd WORK_DIR
  - Bb > 08: Multithreading 2 > Code > thread_test_2.zip をWORK_DIRにダウンロード
  - unzip thread_test_2.zip
  - cd thread_test_2

IR08-1

❖空欄を埋めて，(1)(2)のそれぞれの挙動を実現するプログラムを完成させよ。

Counter.java

```java
public class Counter implements Runnable {

    private final static long SLEEP_LEN_MSEC = 1000;
    private int id;

    public Counter(int id) {
        this.id = id;
    }

    public void run() {
        for(int i = 0; i < 5; i++) {
            System.out.println("Counter #" + id + ": " + i);
            try {
                Thread.sleep(SLEEP_LEN_MSEC);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }

}
```

Main.java

```java
public class Main {

    public static void main(String[] args) throws InterruptedException {
        Counter c0 = new Counter(0);
        Thread thread0 = new Thread(c0);

        Counter c1 = new Counter(1);
        Thread thread1 = new Thread(c1);

        空欄

        System.out.println("FINISH");
    }

}
```

(1)の挙動

```
Counter #1: 0
Counter #0: 0
Counter #1: 1
Counter #0: 1
Counter #1: 2
Counter #0: 2
Counter #1: 3
Counter #0: 3
Counter #1: 4
Counter #0: 4
FINISH
```

ここの表示順は問わない
(#0と#1が交互でなくてよい)

FINISHを最後に表示する

(2)の挙動

```
Counter #0: 0
Counter #0: 1
Counter #0: 2
Counter #0: 3
Counter #0: 4
Counter #1: 0
Counter #1: 1
Counter #1: 2
Counter #1: 3
Counter #1: 4
FINISH
```

Counter #0の表示が終了してから
Counter #1の表示を開始する

FINISHを最後に表示する

共有資源へのアクセス

共有資源へのアクセス (1/3)

- 各スレッドは共有資源にアクセスできる

Shared.java

```java
public class Shared {

    private int value = 0;

    public int getValue() {
        return value;
    }

}
```

MyThread.java

```java
public class MyThread implements Runnable {

    private Shared shared;

    public MyThread(Shared shared) {
        this.shared = shared;
    }

    public void run() {
        System.out.println(shared.getValue());
    }

}
```

Main.java

```java
public class Main {

    public static void main(String[] args) {

        Shared shared = new Shared();

        Thread[] threads = new Thread[3];
        for(int i = 0; i < threads.length; i++) {
            threads[i] = new Thread(new MyThread(shared));
            threads[i].start();
        }
    }

}
```

同じSharedクラスの
インスタンスを
各スレッドに渡している

実行結果 (3つのスレッドは同じ値を取得できる)

```
0
0
0
```

作業準備

- 作業ディレクトリの作成・移動
  - cd WORK_DIR
  - Bb > 08: Multithreading 2 > Code > shared_test_1.zip をWORK_DIRにダウンロード
  - unzip shared_test_1.zip
  - cd shared_test_1

IR08-2

❖shared_test_1に次の改造を行い，結果を予測してから実行せよ。おそらく予期しない結果となるので，その結果に至った理由を推測せよ。

- Sharedクラスに，valueの値を1つ増やすメソッドを実装する。
- 上記メソッドを3本のスレッド（MyThread）から10000回ずつ呼び出す。
- 上記3スレッド全ての処理が終了したらSharedクラスのvalueの値をmainメソッド内で標準出力する。

共有資源へのアクセス (2/3)

- 単純に見える作業も実は複数ステップから構成されている

Shared.java

```java
public class Shared {

    private int value = 0;

    public int getValue() {
        return value;
    }

    public void changeValue() {
        value++;
    }

}
```

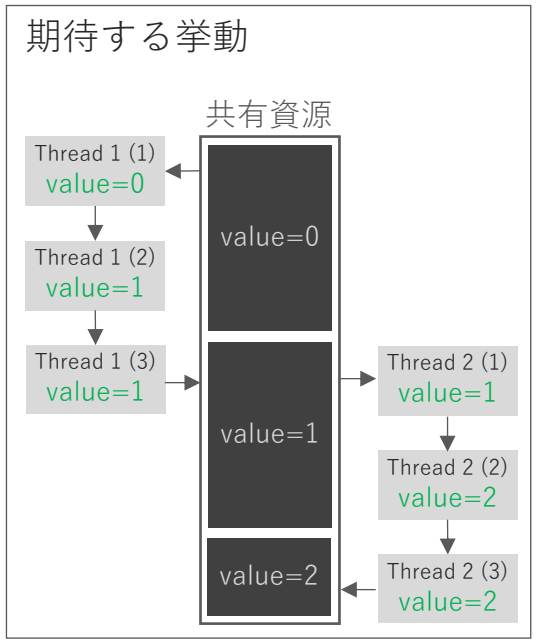
MyThread.java

```java
package shared;

public class MyThread implements Runnable {

    private Shared shared;

    public MyThread(Shared shared) {
        this.shared = shared;
    }

    public void run() {
        for(int i = 0; i < 10000; i++) {
            shared.changeValue();
        }
    }

}
```

各スレッドがshared.changeValue()を実行するとき
具体的には下記ステップが実行されている

(1) 自スレッドのメモリにvalueの値をコピー
(2) 自スレッドのメモリでvalueの値を増やす
(3) (2)を共有インスタンスのvalueの値に反映

期待する挙動



Thread 1が
共有資源を読み込んで
作業をしている間は
Thread 2には待っていてほしい。

共有資源へのアクセス (3/3)

- 各スレッドは共有資源を独自のタイミングで利用してしまう

Shared.java

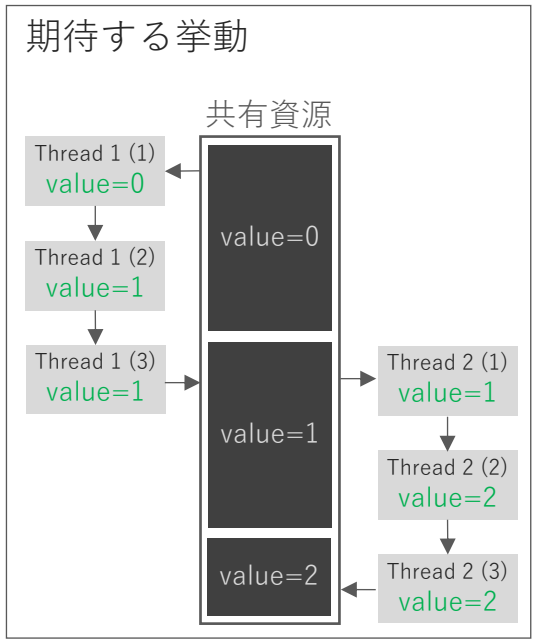
```java
public class Shared {

    private int value = 0;

    public int getValue() {
        return value;
    }

    public void changeValue() {
        value++;
    }

}
```

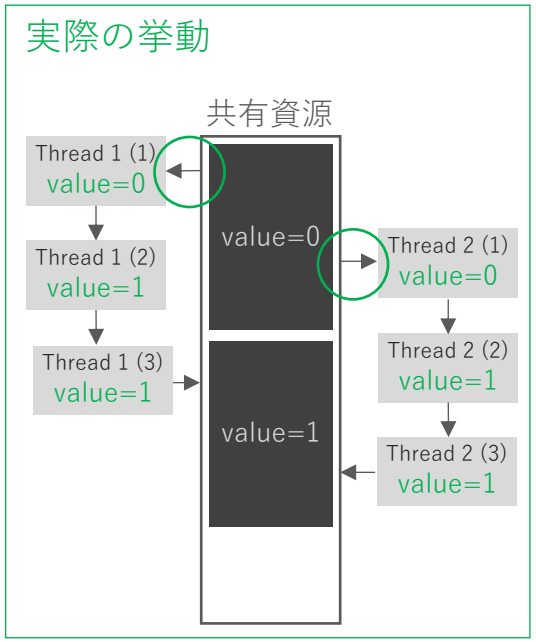
MyThread.java

```java
package shared;

public class MyThread implements Runnable {

    private Shared shared;

    public MyThread(Shared shared) {
        this.shared = shared;
    }

    public void run() {
        for(int i = 0; i < 10000; i++) {
            shared.changeValue();
        }
    }

}
```

各スレッドがshared.changeValue()を実行するとき
具体的には下記ステップが実行されている

(1) 自スレッドのメモリにvalueの値をコピー
(2) 自スレッドのメモリでvalueの値を増やす
(3) (2)を共有インスタンスのvalueの値に反映





排他制御

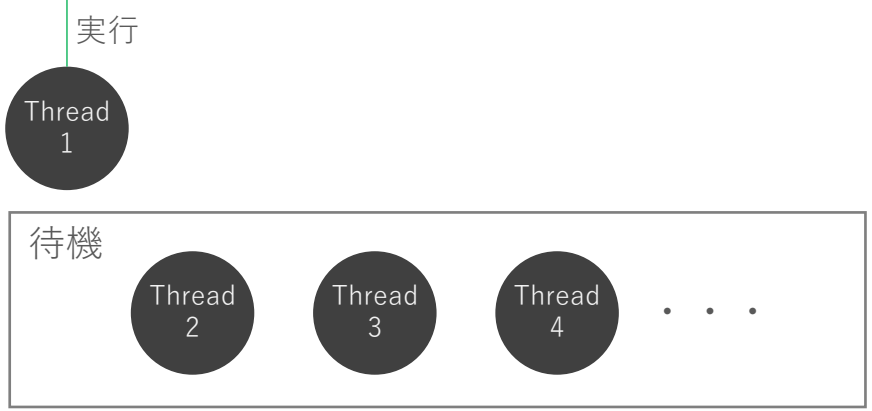
排他制御 (1/3)

Synchronization

- 共有資源の不整合を回避する手段
- 特定コードを同時に実行できるスレッドを1つに限定する

```
public class Shared {

    private int value = 0;

    public int getValue() {
        return value;
    }

    public void changeValue() {
        value++;
    }
}
```

この部分を同時に実行できるスレッドを
1つに限定できれば不整合が生じない



排他制御 (2/3)

Synchronization

- 方法1：Synchronized statements
  - 1スレッドだけに実行させたい範囲をsynchronizedで囲む
  - その範囲を実行するためのロックを指定する
  - ロックは全スレッド中で唯一のものである必要があり，典型的には
    (1) その範囲を実行するのに必要なインスタンスか
    (2) 専用に用意したロック用のObject型のインスタンス変数を用いる

(1)の書き方

```
public class Shared {

    private int value = 0;

    public int getValue() {
        return value;
    }

    public void changeValue() {

        synchronized(this) {
            value++;
        }

    }
}
```

(2)の書き方

```
public class Shared {

    private Object lock = new Object();

    private int value = 0;

    public int getValue() {
        return value;
    }

    public void changeValue() {

        synchronized(lock) {
            value++;
        }

    }
}
```

排他制御 (3/3)

Synchronization

- 方法2：Synchronized method
  - 1スレッドだけに実行させたいメソッドにsynchronizedの指定を行う

```
public class Shared {

    private int value = 0;

    public int getValue() {
        return value;
    }

    public synchronized void changeValue() {
        value++;
    }
}
```

排他制御を行ったIR08-2の解答例

Shared.java

```java
public class Shared {

    private int value = 0;

    public int getValue() {
        return value;
    }

    public void changeValue() {
        synchronized(this) {
            value++;
        }
    }

}
```

MyThread.java

```java
package shared;

public class MyThread implements Runnable {

    private Shared shared;

    public MyThread(Shared shared) {
        this.shared = shared;
    }

    public void run() {
        for(int i = 0; i < 10000; i++) {
            shared.changeValue();
        }
    }

}
```

Main.java

```java
public class Main {

    public static void main(String[] args) throws InterruptedException {
        Shared shared = new Shared();

        Thread[] threads = new Thread[3];
        for(int i = 0; i < threads.length; i++) {
            threads[i] = new Thread(new MyThread(shared));
            threads[i].start();
        }

        for(int i = 0; i < threads.length; i++) {
            threads[i].join();
        }

        System.out.println(shared.getValue());
    }

}
```

実行結果

```
30000
```

作業準備

- 作業ディレクトリの作成・移動
  - cd WORK_DIR
  - Bb > 08: Multithreading 2 > Code > deposit_1.zip をWORK_DIRにダウンロード
  - unzip deposit_1.zip
  - cd deposit_1

IR08-3

- 下記プログラムは，Aliceの口座に3本の振込スレッドを用いて合計3,000円の振込を行うことを意図したものである。DepositThreadクラスのrun()内に排他制御を導入し，この意図を満たすようにせよ。他の部分は変更してはいけない。

Account.java

```java
public class Account {

    private String owner;
    private int balance;

    public Account(String owner) {
        this.owner = owner;
        balance = 0;
    }

    public void deposit(int amount) {
        balance += amount;
    }

    public void displayBalance() {
        System.out.println(owner + "'s balance: " + balance);
    }
}
```

DepositThread.java

```java
public class DepositThread implements Runnable {

    private final static int DEPOSIT_COUNT = 1000;
    private final static int DEPOSIT_AMOUNT= 1;

    private Account account;

    public DepositThread(Account account) {
        this.account = account;
    }

    public void run() {
        for(int i = 0; i < DEPOSIT_COUNT; i++) {
            account.deposit(DEPOSIT_AMOUNT);
        }
    }
}
```

Main.java

```java
public class Main {

    private final static int DEPOSIT_THREAD_COUNT = 3;

    public static void main(String[] args) throws InterruptedException {
        Account aliceAccount = new Account("Alice");

        Thread[] depositThreads = new Thread[DEPOSIT_THREAD_COUNT];
        for(int i = 0; i < depositThreads.length; i++) {
            depositThreads[i] = new Thread(new DepositThread(aliceAccount));
            depositThreads[i].start();
        }

        for(int i = 0; i < depositThreads.length; i++) {
            depositThreads[i].join();
        }

        aliceAccount.displayBalance();
    }
}
```

ここに排他制御を導入しても期待する挙動は実現できるが，
本問題ではここは変更してはいけないものとする。

ここに排他制御を導入する。
Accountクラス等，他の場所を変更してはいけない。

実行結果例
（実行するたび結果が変わる）

```
Alice's balance: 2394
```

```
Alice's balance: 1029
```

本日のまとめ

❖講義内容

- join()によるスレッドの待ち合わせ
- 共有資源へのアクセス
- 排他制御

❖授業内課題提出

- 各授業内課題（IR）の解答を記載せよ。
- 「講義内容のまとめ」の解答欄に
  上記「講義内容」の各項目について文章で説明を記載せよ。