

応用Javaプログラミング

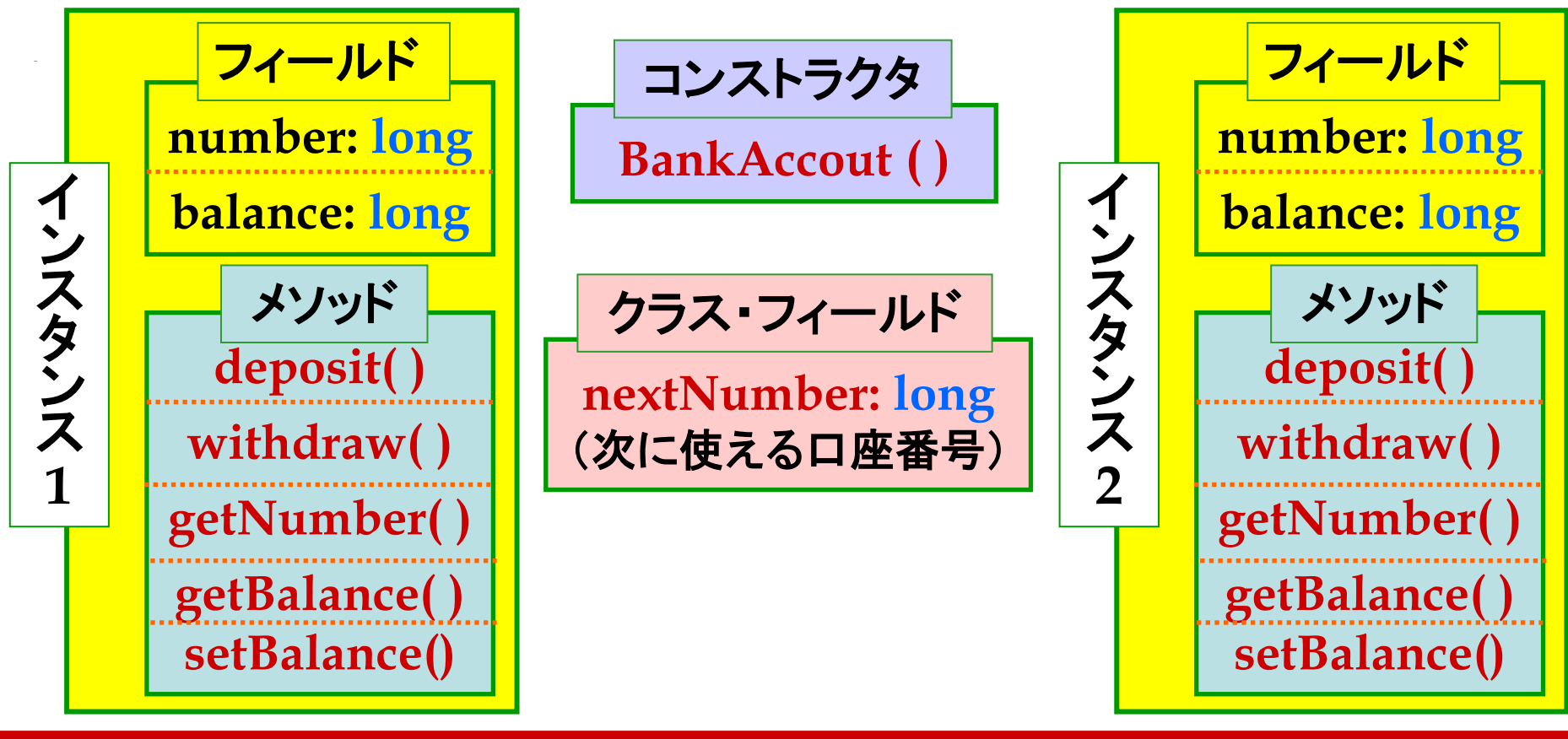
第7回

1. マルチスレッド(中級編)
2. 例(だめな銀行口座クラス)
3. 例(よい銀行口座クラス)
4. **Builder**デザインパターン

銀行口座を管理する簡単なクラス

銀行口座は本当は複雑で、Builderデザインパターンで構築すべき。

BankAccount クラス



number: 口座番号を覚える
deposit: 預け入れを行う
getNumber: 口座番号を取得

balance: 貯金残高を覚える
withdraw: 引き出しを行う
getBalance: 貯金残高を取得

BankAccount クラス

```
public class BankAccount {  
    private static volatile long nextNumber ; // 次に使える口座番号(クラス・フィールド)  
    private long number ; // 口座番号を覚えるインスタンス・フィールド  
    private long balance ; // 貯金残高を覚えるインスタンス・フィールド  
  
    // コンストラクタ(新しい口座を開く)  
    public BankAccount( ) {  
        number = nextNumber ; // 口座番号を(次に使える番号に)設定  
        nextNumber++ ; // 次に使える番号を増やす  
        balance = 0 ; // 貯金残高を初期化  
    }  
  
    // 貯金を預けるメソッド  
    public 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  
    }  
}
```

次頁に続く...

BankAccount.java

BankAccount クラス(続き)

// 貯金を引き出すメソッド

```
public 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  
    }  
}
```

```
public long getNumber( ) // 口座番号を返すメソッド  
{ return number; }
```

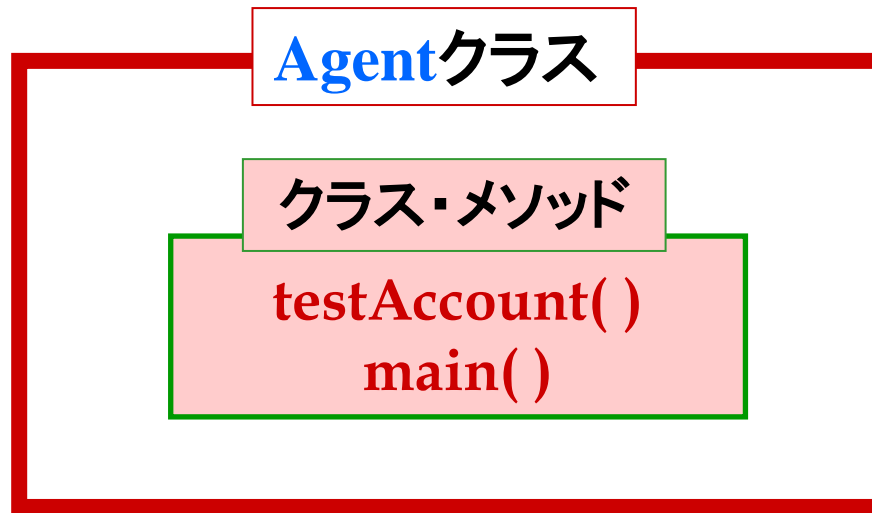
```
public long getBalance( ) // 残高を返すメソッド  
{ return balance; }
```

```
public void setBalance(long newBalance) // 残高を返すメソッド  
{ balance = newBalance; }  
} // BankAccountクラスの終わり
```

BankAccount.java

テスト・クラス

(出し入れを代行する業者を表す**クラス**?)



testAccountメソッドで行う処理:

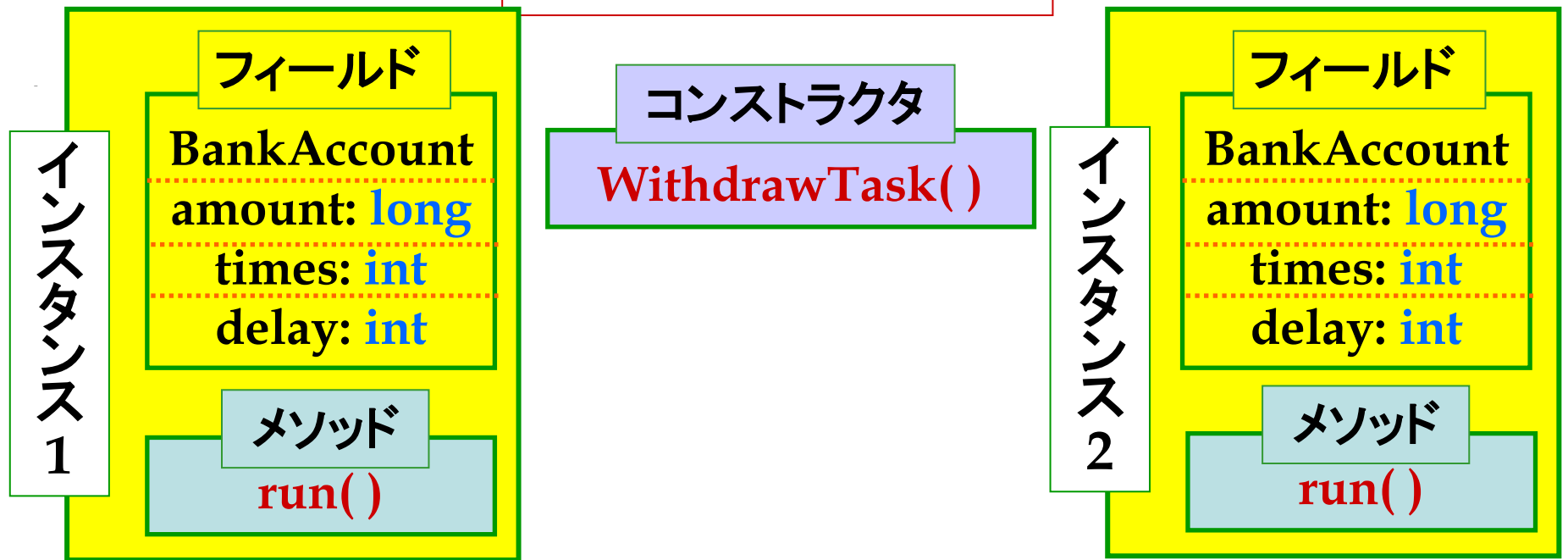
引数を通して取引口座, 回数, 間隔, 額を受け取り, その口座への預け入れを代行するagent(スレッド) および その口座からの引き落としを代行するagent(スレッド)をそれぞれ1つ生成して起動する.

mainメソッドで行う処理:

銀行口座を1つ作り, **testAccount**メソッドを呼び出す.

引き落としを表す簡単なタスク・クラス

WithdrawTask クラス



BankAccount: 引き落とし先の口座番号を覚える。

amount: 引き落とし金額を覚える

times: 引き落としの回数を覚える

run: 引き落としを行うメソッド

```

public class WithdrawTask implements Runnable {
    private int delay ; //引き落とし間隔(定数)を覚えるインスタンス・フィールド
    private BankAccount account; // 引き落とし先口座番号を覚えるインスタンス・フィールド
    private long amount ; // 引き落とし金額を覚えるインスタンス・フィールド
    private int times ; // 引き落としの回数を覚えるインスタンス・フィールド

    // コンストラクタ(引き落とし先の口座, 金額, 回数を初期化する)
    public WithdrawTask(BankAccount account, long amount, int times, int delay) {
        this.account = account ;
        this.amount = amount ;
        this.times = times ;
        this.delay = delay ;
    }

    // 引き落としを行うメソッド
    public void run( ) {
        try {
            for ( int i = 1; i <= times; i++ ) {
                account.withdraw(amount); // 1回引き落とす
                Thread.sleep(delay); // 連続する引き落としでは0.001秒間の間隔を置く
            }
        } catch (InterruptedException e) {Thread.currentThread().interrupt( ) ; }
    }
}

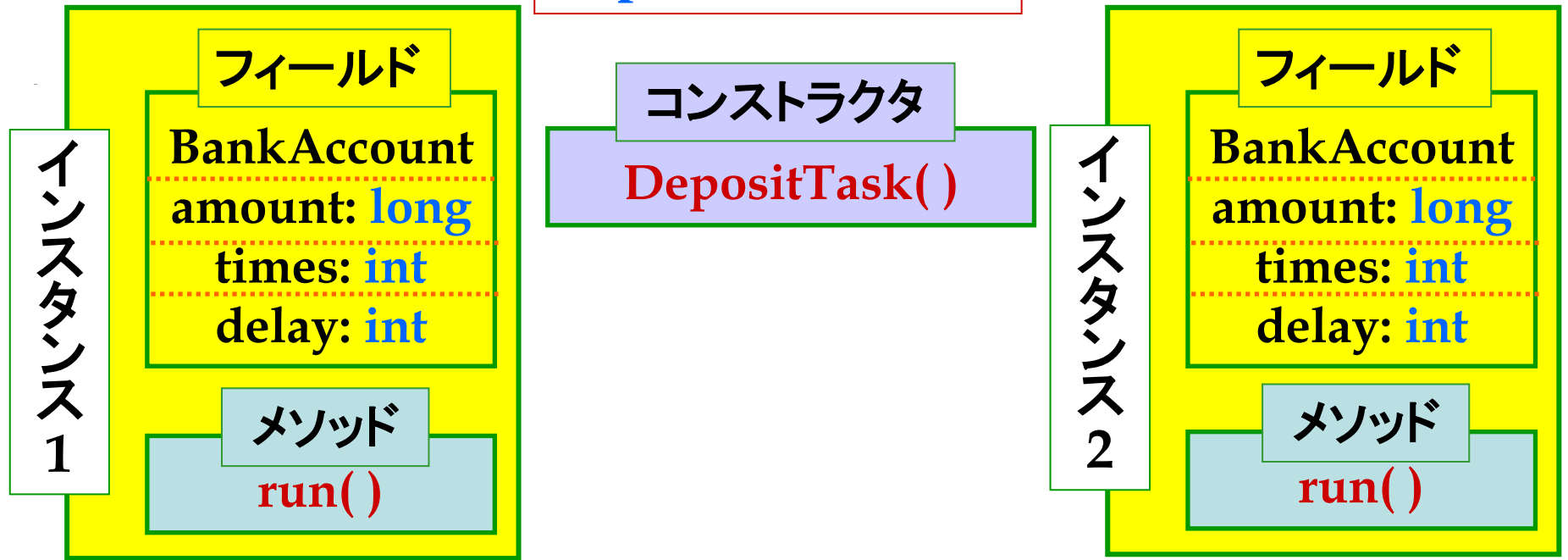
```

WithdrawTask クラス

WithdrawTask.java

預け入れを表す簡単なタスク・クラス

DepositTask クラス



BankAccount: 預け入れ先の口座番号を覚える.

amount: 預け入れ金額を覚える

times: 預け入れの回数を覚える

run: 預け入れを行うメソッド


```

public class DepositTask implements Runnable {
    private int delay; //預け入れ間隔(定数)を覚えるインスタンス・フィールド
    private BankAccount account; // 預け入れ先口座番号を覚えるインスタンス・フィールド
    private long amount ; // 預け入れ金額を覚えるインスタンス・フィールド
    private int times ; // 預け入れの回数を覚えるインスタンス・フィールド

    // コンストラクタ(預け入れ先の口座, 金額, 回数を初期化する)
    public DepositTask(BankAccount account, long amount, int times, int delay) {
        this.account = account ;
        this.amount = amount ;
        this.times = times ;
        this.delay = delay ;
    }

    // 預け入れを行うメソッド
    public void run( ) {
        try {
            for ( int i = 1; i <= times; i++ ) {
                account.deposit(amount); // 1回引き落とす
                Thread.sleep(delay); // 連続する預け入れでは0.001秒間の間隔を置く
            }
        } catch (InterruptedException e) {Thread.currentThread().interrupt( ) ; }
    }
}

```

DepositTask クラス

DepositTask.java

Agentクラス

```
public class Agent {  
    public 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1つ生成する  
        Runnable dTask = new DepositTask(account, amount, times, delay);  
        Thread dAgent = 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1つ生成する  
        Runnable wTask = new WithdrawTask(account, amount, times, delay);  
        Thread wAgent = new Thread(wTask);  
  
        // 先ほど生成した2つのスレッドを起動する  
        dAgent.start();  
        wAgent.start();  
    }  
  
    public static void main(String[] args) {  
        testAccount(new BankAccount(), 100, 1000, 1);  
    }  
}
```

Agent.java

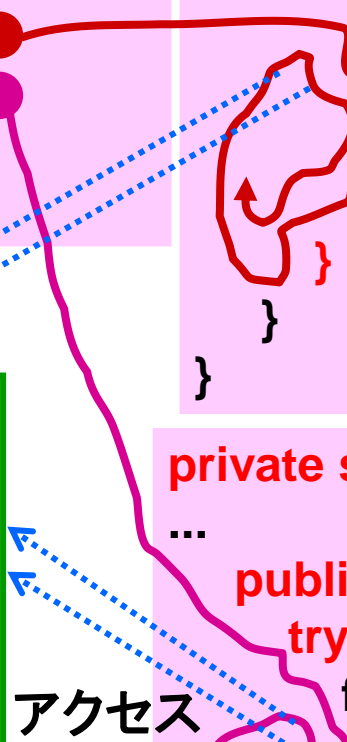
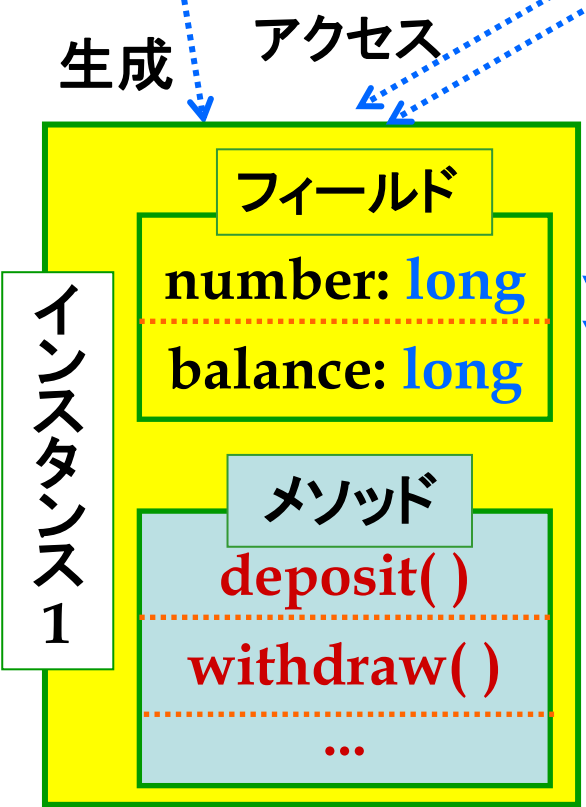
dAgentスレッドとwAgentスレッドは同じ口座にアクセスする. 
dAgentスレッドは1000回(毎回100円)の預け入れを0.001秒間隔で行う.
wAgentスレッドは1000回(毎回100円)の引き落としを0.001秒間隔で行う.

3つのスレッドの関係

```
public class Agent {  
    public static void main ... {  
        testAccount(  
            new BankAccount(), ...  
        )  
    }  
    public static void testAccount ... {  
        ...  
        dAgent.start( );  
        wAgent.start( );  
    }  
}
```

```
private static class DepositTask implements ... {  
    ...  
    public void run( ) { // 預け入れを行うメソッド  
        try {  
            for ( int i = 1; i <= times; i++ ) {  
                account.deposit(amount);  
                Thread.sleep(delay);  
            }  
        } catch (InterruptedException e) { ... }  
    }  
}
```

```
private static class WithdrawTask implements ... {  
    ...  
    public void run( ) { // 引き落としを行うメソッド  
        try {  
            for ( int i = 1; i <= times; i++ ) {  
                account.withdraw(amount);  
                Thread.sleep(delay);  
            }  
        } catch (InterruptedException e) { ... }  
    }  
}
```



インスタンス 1

フィールド

number: long
balance: long

メソッド

```
deposit( ... ) {  
    ...  
    long newBalance =  
        balance + amount;  
    ...  
    balance=newBalance;  
}
```

```
withdraw( ... ) {  
    if (amount >= balance)  
        System.out.println(...);  
    else{ ...  
        long newBalance =  
            balance - amount ;  
        ...  
        balance=newBalance;  
    }  
}
```

口座へのアクセスの詳細

```
private class DepositTask implements Runnable{  
    ...  
    public void run() { // 預け入れを行うメソッド  
        try {  
            for ( int i = 1; i <= times; i++ ) {  
                account.deposit(amount);  
                Thread.sleep(delay);  
            }  
        } catch (InterruptedException e) { ... }  
    }  
}
```

```
private class WithdrawTask implements Runnable {  
    ...  
    public void run() { // 引き落としを行うメソッド  
        try {  
            for ( int i = 1; i <= times; i++ ) {  
                account.withdraw(amount);  
                Thread.sleep(delay);  
            }  
        } catch (InterruptedException e) { ... }  
    }  
}
```

フィールド

number: long
balance: long

メソッド

```
deposit( ... ) {
    ...
    long newBalance =
        balance + amount;
    ...
    balance=newBalance;
}

withdraw( ... ) {
    if (amount >= balance)
        System.out.println(...);
    else{ ...
        long newBalance =
            balance - amount ;
        ...
        balance=newBalance;
    }
}
```

スレッド間の干渉

大事なポイント

JVM は各スレッドにすこしずつの実行時間を割り当て、スレッドの実行を頻繁に切り替える。

- ∴ 次の状況が生じる可能性あり:
- 「預け入れ」スレッドが **deposit** メソッドを呼び出して
文 `newBalance = balance + amount ;` を実行し終わった瞬間に、
 - 「引き落とし」スレッドが **割り込んで**きて最後まで実行した後、
 - 「預け入れ」スレッドが
文 `balance = newBalance ;` を実行する。

上記の状況で、銀行が損する。

実際の実行結果をみてみよう

Depositing 100, new balance is 100
Depositing 100, new balance is 200
Withdrawing 100, new balance is 100
...(省略)
Withdrawing 100, new balance is 0
insufficient balance.
Depositing 100, new balance is 100
...
Depositing 100, new balance is 300
Withdrawing 100Depositing 100, new balance is 400
, new balance is 200
Depositing 100, new balance is 300
...
Depositing 100, new balance is 800

両方のスレッド
からの表示内容
が混ざっている



∴ スレッドの実行
の切り替えが
分かる.

dAgentスレッドは1000回(毎回100円)の預け入れを0.001秒間隔で行う。
wAgentスレッドは1000回(毎回100円)の引き落としを0.001秒間隔で行う。

「悪い銀行口座」の問題点 と 解決策1

問題点: あるスレッドが **deposit**メソッド(「預け入れ」メソッド) または **withdraw**メソッド(「引き落とし」メソッド)を実行している途中で、別のスレッドが割り込んできて 同じ銀行口座の中身を変えてしまう.

そのため, **deposit**メソッドも **withdraw**メソッドも**スレッドセーフ**ではないと言われる. この理由で **BankAccount**クラスも**スレッドセーフ**ではないと言われる.

一般的に, マルチスレッドの環境でも正しく動作するメソッド や クラスのことを**スレッドセーフである**と言う.

解決策1: **java.util.concurrent.locks.ReentrantLock**クラスのインスタンス(**ロック・オブジェクト**という)を使う. (詳細は次頁から)

(**注意:** 次のメソッドを除いてSwingのメソッドはスレッド・セーフではない:

JTextComponent.setText

JTextArea.insert

JTextArea.append

JTextArea.replaceRange

JComponent.repaint

JComponent.revalidate)

java.util.concurrent.locks.ReentrantLock クラス

主なコンストラクタ

public ReentrantLock()

ReentrantLockクラスのインスタンスを作成する.

主なメソッド

public void lock()

ロックを取得する.

public void unlock()

このロックの解放を試みる.

public Condition newCondition()

このロックで使用する Condition型インスタンスを返す.

「悪い銀行口座」の 問題点 と 解決策1 (続き)

// **Lock**インターフェースと**ReentrantLock**クラスをつかうので次の1行が必要

import java.util.concurrent.locks.* ;



public class BankAccount2 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;



// コンストラクタ(新しい口座を開く)

public BankAccount2() {

// 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()** ;



}

次頁に続く...

BankAccount2.java

「悪い銀行口座」の 問題点 と 解決策1(続き)

// 貯金を預けるメソッド

@Override // コンパイラへの指示

public void deposit(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.deposit(amount);

} finally {

balanceChangeLock.unlock(); // 鍵を外す

}

}

同期
ブロッ
クとい
う

次頁に続く...

BankAccount2.java

「悪い銀行口座」の問題点と解決策1(続き)

// 貯金を引き落とすメソッド

@Override // コンパイラへの指示

public 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

}

}

次頁に続く...

BankAccount2.java

同期ブロックという

「悪い銀行口座」の問題点と解決策1(続き)

// 残高を更新するメソッド

@Override // コンパイラへの指示

public 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);

} finally {

balanceChangeLock.unlock(); // 鍵を外す

}

}

BankAccount2.java

同期ブロックという

getNumber()とgetBalance()は元々スレッドセーフであるので、何も対策を取らなくてよい。

フィールド

number: long
balance: long

メソッド

```
deposit( ... ) {
    ...
    balanceChangeLock.lock( );
    ...
    newBalance=balance+amount;
    ...
    balanceChangeLock.unlock( );
    ...
}

withdraw( ... ) {
    ...
    balanceChangeLock.lock( );
    ...
    newBalance=balance-amount;
    ...
    balanceChangeLock.unlock( );
}
```

スレッド間の干渉の解消

スレッドAが、ロックで括っている部分を実行している間、

スレッドCが、**lock**メソッドの実行で待たされる(ブロックされる)。

スレッドBが、**lock**メソッドの実行で待たされる(ブロックされる)。

フィールド

number: long

balance: long

メソッド

```
deposit( ... ) {
    ...
    balanceChangeLock.lock( );
    ...
    newBalance=balance+amount;
    ...
    balanceChangeLock.unlock( );
    ...
}
```

```
withdraw( ... ) {
    ...
    balanceChangeLock.lock( );
    ...
    newBalance=balance-amount;
    ...
    balanceChangeLock.unlock( );
}
```

スレッド間の干渉 の解消(続き)

スレッドAが、ロックで括っている部分を実行し終わった瞬間、

スレッドCが、(そのロックを取得しようとしている他のスレッドがある場合そのどれかに負けたら) **lock**メソッドの実行で待たされるまま。

スレッドBが、(そのロックを取得しようとしている他のスレッドがある場合それらに勝って) やっとロックで括っている部分に入れる。

フィールド

number: long
balance: long

インスタンス
1

メソッド

```
deposit( ... ) {  
    ...  
    balanceChangeLock.lock( );  
    ... // 排他的処理  
    balanceChangeLock.unlock( );  
    ...  
}  
-----  
withdraw( ... ) {  
    balanceChangeLock.lock( );  
    ... // 排他的処理  
    balanceChangeLock.unlock( );  
    ...  
}  
-----  
getNumber( ... )  
{ ... }  
-----  
getBalance( ... )  
{ ... }
```

鍵（ロック）のイメージ

別のスレッドが同期ブロックに含まれていない文を実行してよい。

鍵（ロック）

別のスレッドがこのインスタンス（銀行口座）のどの同期ブロックも実行できない。

あるスレッドがこのインスタンス（銀行口座）のある同期ブロックを実行しているとき、

別のスレッドがこのインスタンス（銀行口座）の同期ではないブロックを実行してよい。

フィールド

number: long

balance: long

メソッド

```
deposit( ... ) {
    ...
    balanceChangeLock.lock( );
    ...
    newBalance=balance+amount;
    ...
    balanceChangeLock.unlock( );
    ...
}
```

```
withdraw( ... ) {
    if (amount >= balance)
        System.out.println(...);
    else{ ...
        long newBalance =
            balance - amount ;
        ...
        balance=newBalance;
    } }
```

withdrawメソッドでロックを使用しなかったとすると,

スレッドAが、ロックで括っている部分を実行している間、

スレッドCが、**lock**メソッドの実行で待たされる(ブロックされる)。

スレッドBが、(ロックを取得しなくても)**withdraw**メソッドの文を実行して、残高を変えることができる。

∴ 排他的処理が必要な部分をすべて**同じロック**でくくらないといけない。

鍵(ロック)に関する注意

- スレッドはインスタンスメソッドに対して鍵をかけているのではなく、
(そのメソッドを持つ)インスタンスに対して鍵(ロック)をかけている。
- 1つのスレッドは、あるインスタンスのある同期ブロックを続けて実行できる(自分自身が鍵によって締め出されることはない)。また、その同期ブロックを実行し終わった瞬間、鍵を外す。
- あるスレッドは、あるインスタンスに対して鍵をかけようとしたとき、すでにそのインスタンスに鍵がかかっていたら、その鍵が(他のスレッドによって)外されるまで待たされる。
- あるスレッドが同期ブロックを実行している最中でも、そのインスタンスの同期ではないブロックならば、どのスレッドも実行できる。
- あるスレッドが同期ブロックを実行している最中でも、別のインスタンスのブロックは、(同期であってもなくても)実行できる。

「悪い銀行口座」の 問題点 と 解決策2

問題点: あるスレッドが **deposit**メソッド(「預け入れ」メソッド) または **withdraw**メソッド(「引き落とし」メソッド)を実行している途中で、別のスレッドが割り込んできて 同じ銀行口座の中身を変えてしまう.

そのため, **deposit**メソッドも **withdraw**メソッドも**スレッドセーフ**ではないと言われる. この理由で **BankAccount**クラスも**スレッドセーフ**ではないと言われる.

一般的に, マルチスレッドの環境でも正しく動作するメソッド や クラスのことを**スレッドセーフである**と言う.

↑
復習

解決策2: その実行中に割り込まれては困るようなメソッドを **同期メソッド**にする. そうすれば, その実行中に割り込まれなくてスレッドセーフになる.


あるメソッドを**同期メソッド**にするには, そのメソッドの定義時に **キーワード synchronized** を付ければよい(次頁にて詳細).

「悪い銀行口座」の 問題点 と 解決策2(続き)



```
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  
    public 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  
    public 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  
    public synchronized void setBalance(long amount) {  
        super.setBalance(amount);  
    }  
}
```





各インスタンスに**固有のロック**が1つあり, 前頁の3つのメソッドはそれぞれ, 下記の(擬似)メソッドと等価である.




```
public 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
    super.deposit(amount);
    this.固有ロック.unlock( ); // 擬似文
}
```




```
public 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
    super.withdraw(amount);
    this.固有ロック.unlock( ); // 擬似文
}
```





```
public void setBalance(long amount) { // 貯金を設定するメソッド(同期メソッド)
    this.固有ロック.lock( ); // 擬似文
    super.setBalance(amount);
    this.固有ロック.unlock( ); // 擬似文
}
```





各インスタンスに**固有のロック**が1つあり, 前頁の3つのメソッドはそれぞれ, 下記のメソッドと等価である.



```
public void deposit(long amount) { // 貯金を預けるメソッド(同期メソッド)
    synchronized ( this ) {
        super.deposit(amount) ;
    }
}
```



```
public void withdraw(long amount) { // 貯金を引き出すメソッド(同期メソッド)
    synchronized ( this ) {
        super.withdraw(amount) ;
    }
}
```



```
public void setBalance(long amount) { // 貯金を設定するメソッド(同期メソッド)
    synchronized ( this ) {
        super.setBalance(amount) ;
    }
}
```

固有鍵(ロック)のイメージ

フィールド

number: long

balance: long

メソッド

synchronized deposit(...) {

...
long newBalance =
balance + amount ;

...
balance=newBalance;
}

synchronized withdraw(...){

if (amount >= balance) ...
else { long newBalance
= balance - amount ;
balance=newBalance;
...
}

getNumber(...)

{ ... }

getBalance(...)

{ ... }

インスタンス固有の鍵(ロック)

インスタンス 1

あるスレッドがこのインスタンス(銀行口座)のある同期メソッドを実行しているとき,

別のスレッドがこのインスタンス(銀行口座)のどの同期メソッドも実行できないが,

別のスレッドがこのインスタンス(銀行口座)の同期ではないメソッドを実行してよい。

固有ロックと自作ロックの併用はだめ

// 貯金を預けるメソッド

```
public void deposit(long amount) {  
    balanceChangeLock.lock();  
    System.out.print("Depositing " + amount); // 預け金額を表示  
    try {  
        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  
    } finally { balanceChangeLock.unlock(); }  
}
```

これでは、この2つのメソッド間の干渉は解消されない。

// 貯金を引き出すメソッド(同期メソッド)

```
public 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  
    }  
}
```

固有鍵(ロック)に関する注意

- スレッドはインスタンスメソッドに対して鍵をかけているのではなく、
(そのメソッドを持つ)インスタンスに対して鍵(ロック)をかけている。
- 1つのスレッドは、あるインスタンスのある同期メソッドを続けて実行できる(自分自身が鍵によって締め出されることはない)。また、その同期メソッドを実行し終わった瞬間、鍵を外す。
- あるスレッドは、あるインスタンスに対して鍵をかけようとしたとき、すでにそのインスタンスに鍵がかかっていたら、その鍵が(他のスレッドによって)外されるまで待たされる。
- あるスレッドが同期メソッドを実行している最中でも、そのインスタンスの同期ではないメソッドならば、どのスレッドも実行できる。
- あるスレッドが同期メソッドを実行している最中でも、別のインスタンスのメソッドは、(同期であってもなくても)実行できる。

2つの解決法の比較

- 解決法2の方が簡単である. しかし, この方法では, インスタンスが現在どのスレッドによって鍵をかけられているか知ることができない.
- 解決法1の方がやや手間がかかる. しかし, この方法では,
 - インスタンスは現在どのスレッドによって鍵をかけられているか
 - 鍵の解除を待っている他のスレッドはあるかなどを(**ReentrantLock**クラスのメソッドを使えば)知ることができる.
また, メソッド単位ではなく, 文単位での同期を取ることができる.

スレッドセーフなデータ構造 と そうではないデータ構造

Javaには、**コレクション**（いわゆる**データ構造**）がいくつか用意されている：

- **LinkedList<E>**クラス・・・**連結リスト**を使いたいとき
- **ArrayList<E>**クラス・・・**サイズが自動的に大きくなる配列**を使いたいとき
- **Vector<E>**クラス・・・**サイズが自動的に大きくなる配列**を使いたいとき（スレッドセーフ）
- **Stack<E>**クラス・・・**スタック**を使いたいとき（スレッドセーフ）
- **ConcurrentLinkedQueue<E>**クラス・・・**スレッドセーフなキュー**を使いたいとき
- **PriorityQueue<E>**クラス・・・**優先度付きのキュー**を使いたいとき
- **HashMap<K,V>**クラス・・・**キーと値の対応関係を管理**したいとき
- **ConcurrentHashMap<K,V>**クラス・・・**スレッドセーフなHashMap**を使いたいとき
- **Hashtable<K,V>**クラス・・・**キーと値の対応関係をハッシュテーブル**で管理したいとき
- **TreeMap<K,V>**クラス・・・**キーと値の対応関係を2色木**で管理したいとき

- マークが付いているクラスは良く使われるもの。
しかし、そのどれもスレッドセーフではない。

注：スレッドセーフなクラスはそうでないクラスに比べて遅い。

∴ シングル・スレッドの応用では、あまり使わない方がよい。

演習課題

「ランダムに生成した文章を描画する」プログラム [SimplePainterS.java](#) をダウンロードせよ。

(注: [SimpleAnimeS.java](#) では, 短い文章を繰り返してランダムに生成するスレッドが生成され, 起動される. そのスレッドが文章(単語のリスト)にどんどん単語を追加する一方, **event dispatch thread** がそのリストの単語をどんどん描画する.)

ゆえに, 上記の2つのスレッドは文章リスト(追加した単語を覚える連結リスト)を共有しており, 互いに干渉する可能性がある.

∴ [SimpleAnimeS.java](#) にスレッドセーフではないメソッドがある.

課題: [SimpleAnimeS.java](#) にスレッドセーフではないメソッドを見出してさらに [ReentrantLock](#) クラスのインスタンス(ロックオブジェクト)を使って, スレッドセーフではないメソッドを**細かく**スレッドセーフにせよ

BankAccountを表す簡単なクラス

インスタンスフィールド:

- ① 口座番号
- ② 残高
- ③ 名前
- ④ 住所
- ⑤ 電話番号
- ⑥ 電子メール
- ⑦ 性別
- ⑧ 婚姻状況
- ⑨ ニックネーム

必須項目

その他の項目は必須ではないとする.

∴ 32通り以上の組み合わせがある.

∴ 1つの組み合わせにつき 1つの
コンストラクタを用意するのが
得策ではない.

...

BankAccountが複雑すぎるので、より簡単なAppointmentクラスで
このような場合にどうすればよいか次に説明する.

Appointmentを表す簡単なクラス

インスタンスフィールド:

- ① 開始時間
- ② 終了時間
- ③ 場所
- ④ 参加者(のリスト)
- ⑤ メモ

必須項目

その他の3項目は必須ではない.

∴ 8通りの組み合わせがある.

∴ 1つの組み合わせにつき 1つの
コンストラクタを用意するのが
得策ではない.

Contact クラス

```
public class Contact {  
    private String name ; // 名前  
    private String relation ; // 関係  
  
    public Contact(String name, String relation) // コンストラクタ  
    { this.name = name ; this.relation = relation ; }  
  
    public String toString( ) {  
        return "¥nContact:¥n" + " Name: " + name + "¥n" +  
            " Relationship: " + relation;  
    }  
    //accessorメソッド(略)  
}
```

Contact.java

Location クラス

```
public class Location {  
    private String street ; // 街  
    private String city ; // 市  
    private String region ; // 地域  
    private String postal ; // 郵便番号  
  
    public Location(String s, String c, String r, String p) // コンストラクタ  
    { this.street = s ; this.city = c ; this.region = r ; this.postal = p ; }  
  
    public String toString( ) {  
        return street + " , " + city + " , " + region + " , " + postal ;  
    }  
    //accessorメソッド(略)  
}
```

Location.java

Appointment クラス

```
public class Appointment { // コンストラクタが1つもないことに注意
    private Date startDate ; // 開始日時分
    private Date endDate ; // 終了日時分
    private Location location ; // 場所
    private ArrayList<Contact> attendees = new ArrayList<Contact>() ; //参加者リスト
    private String description ; // メモ
    public static final String EOL_STRING = System.getProperty("line.separator") ;

    public void addAttendee(Contact attendee)
    { if (! attendees.contains(attendee)) attendees.add(attendee) ; }

    public void removeAttendee(Contact attendee)
    { attendees.remove(attendee) ; }

    public String toString( ) {
        return " Description: " + description + EOL_STRING +
            " Start Date: " + startDate + EOL_STRING +
            " End Date: " + endDate + EOL_STRING +
            " Location: " + location + EOL_STRING +
            " Attendees: " + attendees;
    }
    //accessorメソッド(略)
}
```

Appointment.java

AppointmentBuilder2 クラス

```
public class AppointmentBuilder2 {  
    private Date startDate ; // 開始日時分  
    private Date endDate ; // 終了日時分  
    private Location location ; // 場所  
    private ArrayList<Contact> attendees = new ArrayList<Contact>() ; //参加者リスト  
    private String description ; // メモ  
  
    //必須フィールドを初期化するコンストラクタ  
    public AppointmentBuilder2(Date start, Location location)  
    {    this.startDate = start ;    this.startDate = start ; }  
  
    //必要に応じて呼ばれるメソッド(setterメソッドみたいなもの). その戻り値に注目.  
    public AppointmentBuilder2 endDate(Date end)  
    {    this.endDate = end ;    return this ; }  
  
    public AppointmentBuilder2 description(String description)  
    {    this.description = description ;    return this ; }  
  
    public AppointmentBuilder2 location(Location location)  
    {    this.location = location ;    return this ; }  
  
    public AppointmentBuilder2 attendees(ArrayList<Contact> attendees)  
    {    this.attendees = attendees ;    return this ; }  
}
```

次頁に続く...

AppointmentBuilder2.java

AppointmentBuilder2 クラス(続)

```
public AppointmentBuilder2 addAttendee(Contact attendee) {  
    if (!attendees.contains(attendee)) this.attendees.add( attendee );  
    return this ;  
}
```

```
public AppointmentBuilder2 removeAttendee(Contact attendee) {  
{ attendees.remove(attendee); return this ; }
```

//AppointmentBuilder2からAppointmentを作るためのメソッド
//作り上げた口座を返すメソッド

```
public Appointment build( ) {  
    Appointment app = new Appointment( ) ;  
    app.setStartDate( startDate ) ;  
    app.setEndDate( endDate ) ;  
    app.setLocation( location ) ;  
    app.setAttendees( attendees ) ;  
    app.setDescription( description ) ;  
    return app ;  
}  
}
```

AppointmentBuilder2.java

Scheduler クラス(メソッドを追加)

```
public class Scheduler {
```

```
//AppointBuilder2を使ってAppointmentのインスタンスを作る(例)
```

```
public Appointment createAppointment(Date start, Location location) {  
    return new AppointmentBuilder2( ) //ビルダーを生成  
        .description("To see my old friends at " + location.toString() )  
        .addAttendee(new Contact("Taro Tokyo", "close friend")) )  
        .addAttendee(new Contact("Hanako Tokyo", "ex-girlfriend"))  
        .build( ) ;
```

```
}
```

```
}
```

Scheduler.java

Builderデザインパターン

ポイント: 多くのインスタンスフィールドを持ち、その中に必須なものと必須ではないものがあるクラスの場合、すべての組み合わせを考慮してコンストラクタを用意すると、コンストラクタの数が膨大になる。そのようなときに **Builderデザインパターン** が役に立つ。

Builder役 (建築者役): インスタンスの各部分を作るためのメソッド、そして1つのインスタンスを作り上げるためのメソッドも用意する役。例では、**AppointmentBuilder2**クラスがこの役をつとめている。

Director役 (監督者役): **Builder役** の用意したメソッドを使ってインスタンスを生成する役。この例では、**Scheduler**クラスがこの役をつとめている。

Builderデザインパターン(続)

Builderデザインパターンと**Template Methodデザインパターン**の違い:
productの生成手順を決める役の違いにある. **Builderデザインパターン**では生成手順は**Director**クラス(他のクラス)で決められるのに対し,
Template Methodデザインパターンでは, 生成手順は親クラスで決められる.

Builderデザインパターンの長所:

productの生成手順が変わったり, どんなに複雑な処理をしても, userからしてみれば, 変更する必要がない.

Builderデザインパターンの短所:

Builderの提供する生成手順は, 増やしたり減らしたりするのが困難.

ConcreteBuilderの拡張は容易にできるが, 親クラスのBuilderの機能を増やすことは子クラス全体に修正が及ぶ可能性がある.

Builderデザインパターンを使うべき場合:

インスタンス生成に手順がある場合や,

外部リソースを使ってインスタンスを作成しなければならない場合や,
コンストラクタで引数の違うものがたくさんできてしまった場合など.