例外，並行・同期処理，ゴミ集め

2002年7月7日

海谷 治彦

目次

- 例外処理とは何かの復習とJVMでの扱い
  - try catch と throw
- スレッドと同期処理のJVMでの扱い
  - Threadクラスとモニタ，synchronized フラグ
- ゴミ集め．いわゆる Garbage Collection
  - 不用なメモリ領域の回収．

失敗と例外(一般論)

- あるメソッドが想定した利用条件下において実行を終了した場合，そのメソッド呼び出しは**成功(success)**したといい，そうでない場合，**失敗(failure)**したという.
- **例外(exception)**とは，メソッド呼び出しの失敗原因である実行時の出来事(event)である.

Bertrand Meyer.Object-oriented software construction. Prentice Hall, second edition, 1997. ISBN 0-13-629155-4 の p.412 より.

例外処理 (一般論)

- 例外処理は，例外が起きてしまったメソッドの呼び出し側が行う.
- 例外処理の方針は大きく分けて以下の2種類となる.
  - Retrying: 例外が起きた条件を変更して，メソッドの再実行を行う.
  - Failure: 呼び出し側の処理も停止し，そのまた呼び出し側に失敗を通知する. (要は始末を呼び出し元，呼び出し元へと押し付ける)

例外の具体例

- 配列添え時の範囲を超えて，アクセスを行った.
- ゼロで割り算してしまった.
- ロードしたいクラスが見つからなかった.
- 権限を越えた操作を行おうとして，セキュリティ違反が起こってしまった.
- 入出力装置がおかしくなった.

一般にメソッドの利用規定外のことが起こったことと考えて良い.

例外発生の例

メソッドの仕様:「引数に0から6の数字が与えられた場合，Sun, Mon ... の順番に曜日を表す3文字を返す．」

0から6の数字以外が引数に与えられても，このメソッド利用の想定外である．⇒ 例外である．

想定外の値に返り値を返す義理はない．

```
class WDay {
   String month(int s) throws Exception{
      String[] m={ "Sun", "Mon", "Tue", "Wed", "Thu", "Fri", "Sat"};
      if(s>=0 && s< m.length) return m[s];
      throw new Exception("Not Weekday");
}
// 以下，省略
}
```

例外発生の簡単な例

メソッドの仕様:「引数に0もしくは1の数字が与えられた場合，真もしくは偽を返す．」

0, 1の数字以外が引数に与えられても，このメソッド利用の想定外である． ⇒ 例外である．

想定外の値に返り値を返す義理はない．

```
class ZeroOne {
    boolean zeroOne(int s) throws Exception{
        if(s==0) return false;
        if(s==1) return true;
        throw new Exception("Neither 0 nor 1");
    }
// 以下，省略．
}
```

アホな例だが，前述の例ではアセンブラがデカいので，扱いきれないため，これを用意した．

```
.method zeroOne(I)Z
    .throws java/lang/Exception
    .limit stack 3
    .limit locals 2
    iload_1
    ifne Label1
    iconst_0
    ireturn
Label1:
    iload_1
    iconst_1
    if_icmpne Label2
    iconst_1
    ireturn
Label2:
    new java/lang/Exception
    dup
    ldc "Neither 0 nor 1"
    invokespecial java/lang/Exception/<init>(Ljava/lang/String;)V
    athrow
.end method
```

簡単な例に対応するアセンブラ

JVMでの例外発生のポイント

- 要は例外インスタンスを作成して，athrow 命令に渡す.
- 教科書にあるとおり，.throws 節はクラスファイルには無くてもOK
- Javaスタック上で，該当するハンドラ(後述)をもつ最も浅いフレームでcatchされる．(最後までcatchできるフレームが無いと，無視されるようだ．)
- 詳細は教科書p.168～，p.237 参照.

例外通過の例 1/2

```
class A{
    void meth1(){
        B b=new B();
        try{
            b.meth();
        }catch(Exception e){
            System.out.println("OK I caught "+e.toString());
        }
    }

public static void main(String[] args){
        new A().meth1();
    }
}
```

```
class B{
    void meth(){
        C c=new C();
            c.meth();
    }
}
```

```
class C{
    void meth()
    //{}
    throws Exception { throw new Exception("is C"); }
}
```

例外通過の例 2/2

- C.methの例外は，直接の呼び出し側 B.meth ではなく，1つおいた A.meth でキャッチされる.
- 実はこのコードのコンパイルにはトリックが必要.
  - まずは，Cは例外を投げないようにコーディングして，A.java B.java C.java をコンパイル.
  - C.java を編集しなおして，例外を投げるようにして，C.java を再コンパイル.
  - そして，実行.

  JVMとは関係なく，コンパイラが例外キャッチのチェックを行うので，それを騙す必要がある.

例外のキャッチ(捕獲)

- Javaコード上で，catch節にあったコードは，アセンブラ上では，goto文で，「飛ばし読み」しているように翻訳される.
- 別途，割り込みハンドラという部分が追加され，
  - 割り込みを監視範囲
  - 監視する割り込みの種類
  - 起こった場合の対処コード(catch内)の位置

  を示す.
- finallyの解説は省略. (話が厄介)

教科書p.170～

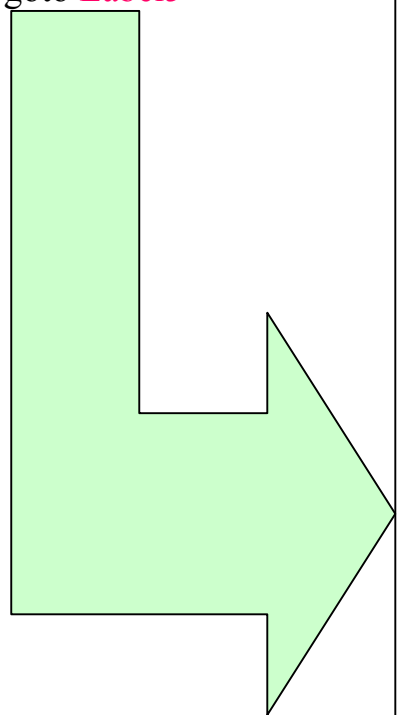
単なるtry-catch: javaソース

```
import java.io.*;

class ArrayFile1{
    public static void main(String[] args){
        try{
            new FileInputStream(args[0]);
        }catch(FileNotFoundException e){
            System.out.println("1: FileNotFoundException");
        }catch(Exception e){
            System.out.println("3: Exception");
        }
    }
}
```

飛んでくる例外を見張る範囲が共通.

単なる try-catch: アセンブラ

```
.method public static main([Ljava/lang/String;)V
    .limit stack 4
    .limit locals 3
Label1:
    new java/io/FileInputStream
    dup
    aload_0
    iconst_0
    aaload
    invokespecial java/io/FileInputStream/<init>(Ljava/lang/String;)V
    pop
Label2:
    goto Label5
```



```
Label3:
    astore_1
    getstatic java/lang/System/out Ljava/io/PrintStream;
    ldc "1: FileNotFoundException"
    invokevirtual java/io/PrintStream/println(Ljava/lang/String;)V
    goto Label5
Label4:
    astore_2
    getstatic java/lang/System/out Ljava/io/PrintStream;
    ldc "3: Exception"
    invokevirtual java/io/PrintStream/println(Ljava/lang/String;)V
Label5:
    return
.catch java/io/FileNotFoundException from Label1 to Label2 using Label3
.catch java/lang/Exception from Label1 to Label2 using Label4
.end method
```

単なる try-catch: ポイント

- catch節に相当する部分はgoto文で，すっ飛ばしている．(gotoからLabel5まで)
- 例外ハンドラ(.catch節)として，アセンブラに展開される．ここの例では，
  - 見張る範囲は，Label1から2
  - FNFExp.がおきると，Label3の処理をする．
  - Exp.がおきると，Labal4の処理をする．

  と書いてある．

```
.catch java/io/FileNotFoundException from Label1 to Label2 using Label3
.catch java/lang/Exception from Label1 to Label2 using Label4
```

入れ子のtry-catch: ソース

```
import java.io.*;

class ArrayFile2{
    public static void main(String[] args){
        try{
            try{
                new FileInputStream(args[0]);
            }catch(FileNotFoundException e){
                System.out.println("1: FileNotFoundException");
            }
        }catch(Exception e){
            System.out.println("3: Exception");
        }
    }
}
```

入れ子のtry-catch: アセンブラ

```
Label1:
    new java/io/FileInputStream
    dup
    aload_0
    iconst_0
    aaload
    invokespecial java/io/FileInputStream/<init>(Ljava/lang/String;)V
    pop
Label2:
    goto Label4
Label3:
    astore_1
    getstatic java/lang/System/out Ljava/io/PrintStream;
    ldc "1: FileNotFoundException"
    invokevirtual java/io/PrintStream/println(Ljava/lang/String;)V
Label4:
    goto Label6
Label5:
    astore_1
    getstatic java/lang/System/out Ljava/io/PrintStream;
    ldc "3: Exception"
    invokevirtual java/io/PrintStream/println(Ljava/lang/String;)V
Label6:
    return
.catch java/io/FileNotFoundException from Label1 to Label2 using Label3
.catch java/lang/Exception from Label1 to Label4 using Label5
```

見張る範囲が異なる以外，そんなに大きく前の例と変わらない.

並行処理と同期

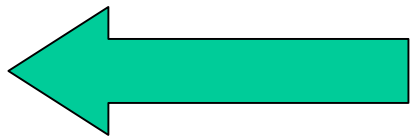
- スレッド自体は，アセンブラでもバイトコードでも，通常のコードを大きな違いは無い.
- 問題は，同期処理.
  - ブロックによる同期
    - モニタを利用
  - メソッドによる同期
    - メソッドのフラグが立つ (p.73)
- 教科書 p.180, p.73等を参照.

synchronizedブロックの場合

- 詳細は，./mutiIncB/ 以下の例を参照.
- monitorenter 教科書p.427
  - スタックトップから参照されているオブジェクトをロックする.
  - ロックするとは，他のスレッドから，そのオブジェクトへのアクセスをさせないこと.
- monitorexit 教科書p.431
  - 同，ロックを解除する.
- ロック区間で例外がおきたら，必ずロック解除をするように，自動的に仕込まれる.

例題の解説

```
.method inc()V
   .limit stack 3
   .limit locals 4
   aload_0
   getfield MultiIncB/c LCounter;
   astore_1
   aload_1
   monitorenter
Label1:
   aload_0
   getfield MultiIncB/c LCounter;
   invokevirtual Counter/val()I
   istore_2
   aload_0
   invokevirtual MultiIncB/waiting()V
   aload_0
   getfield MultiIncB/c LCounter;
   iload_2
   iconst_1
   iadd
   invokevirtual Counter/set(I)V
   aload_1
   monitorexit
   goto Label3
```



```
void inc(){
   synchronized(c)
   {
      int v=c.val();
      waiting();
      c.set(v+1);
   }
}
```

```
Label2:
   astore_3
   aload_1
   monitorexit
   aload_3
   athrow
Label3:
   return
.catch all from Label1 to Label2 using Label2
.end method
```

synchronizedメソッドの場合

- 詳細は，./mutiIncM/ 下を参照.
  - 3つのincrement スレッドが共有変数を1つづつ増加させる例題.
- アセンブラ，バイトコード共に変わったところはない.
- ちなみに，この例題では，synchronized をとると，排他制御に失敗する.

shynchronizeメソッドの実例

```
.method synchronized inc()V
    .limit stack 3
    .limit locals 3
    aload_0
    getfield Counter/c I
    istore_1
Label1:
    aload_0
    getfield Counter/r Ljava/util/Random;
    bipush 100
    invokevirtual java/util/Random/nextInt(I)I
    i2l
    invokestatic java/lang/Thread/sleep(J)V
Label2:
    goto Label4
Label3:
    astore_2
Label4:
    aload_0
    iload_1
    iconst_1
    iadd
    putfield Counter/c I
    return
.catch java/lang/Exception from Label1 to Label2 using Label3
.end method
```

```
import java.util.*;

class Counter{
private int c=0;
private Random r;

    Counter(){ r=new Random(); }

    synchronized void inc(){
        int i=c;
        try{ Thread.sleep(r.nextInt(100)); }
        catch(Exception e){}
        c=i+1;
    }

    int val(){return c;}
}
```

ゴミ集め Garbage Collection

- 不用になったメモリ領域を，別の目的に利用できるようにすること.
- 一応，「ゴミ集め」というGarbage Collectionの訳語は認知はされているが，単にGCと呼ぶ人が多い.
- C言語，C++等では，プログラマが明示的にゴミ集めしないといけない. (cfee関数など)

JavaでのGC

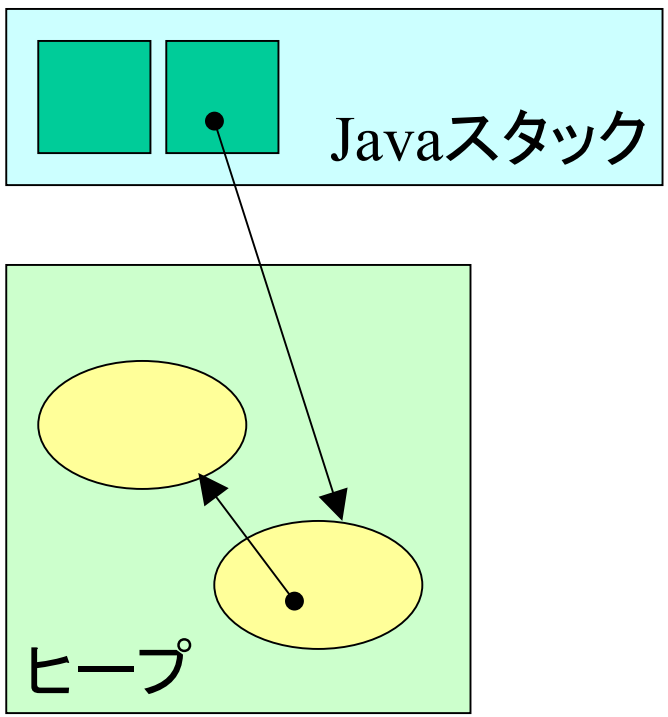
- 動的メモリ割り当てを基本とするJava計算システムでは必須の技術.
- Javaでは自動的にGCするので，プログラマがほとんど意識しない．(Auto GC)
- この自動GCのおかげで，実時間動作(リアルタイム)を阻害しているということもある.
  - GCが始まると，負荷があがり，計算の本筋に関係なく遅くなるため.
- 残念ながら教科書にはGCの記述は無い.

どんなインスタンスがGCされる？

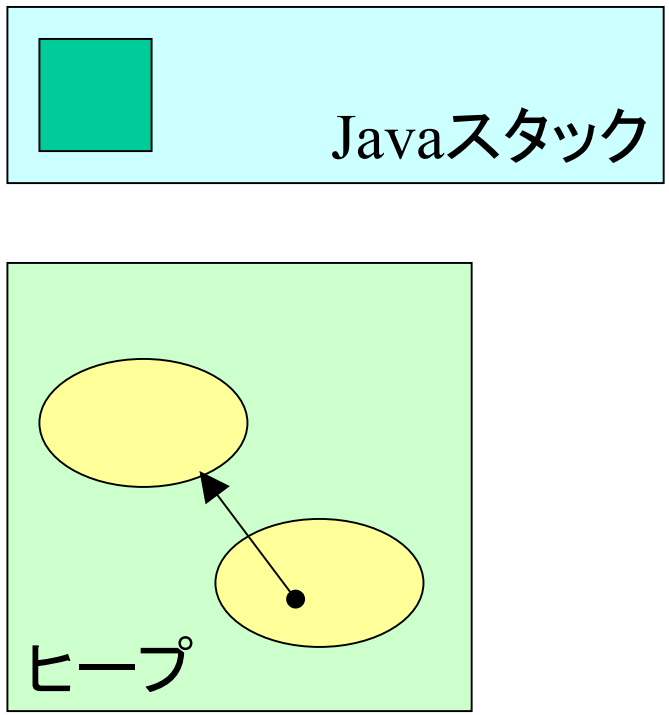
- aliveでなくなったインスタンスがGCされる.
- aliveの定義.
  - オペランドスタック内から参照されているインスタンスはalive.
  - フレーム内のローカル変数から参照されてるインスタンスはalive.
  - aliveなインスタンスのフィールドから参照されているインスタンスはalive.
  - nativeメソッドを持つインスタンスはalive.

注意しないといけない非alive例

- 非alive同士でリンクをもっていても，GCされる.



両方 alive



両方GC対象.

GCされる間際の処理を書くには？

- Object.finalize メソッドを再定義(オーバーライド)すればよい.
- Objectクラスでの定義は空.
- 要は「死に際」のユーザー処理を定義したい場合に使う.

finalizeの例

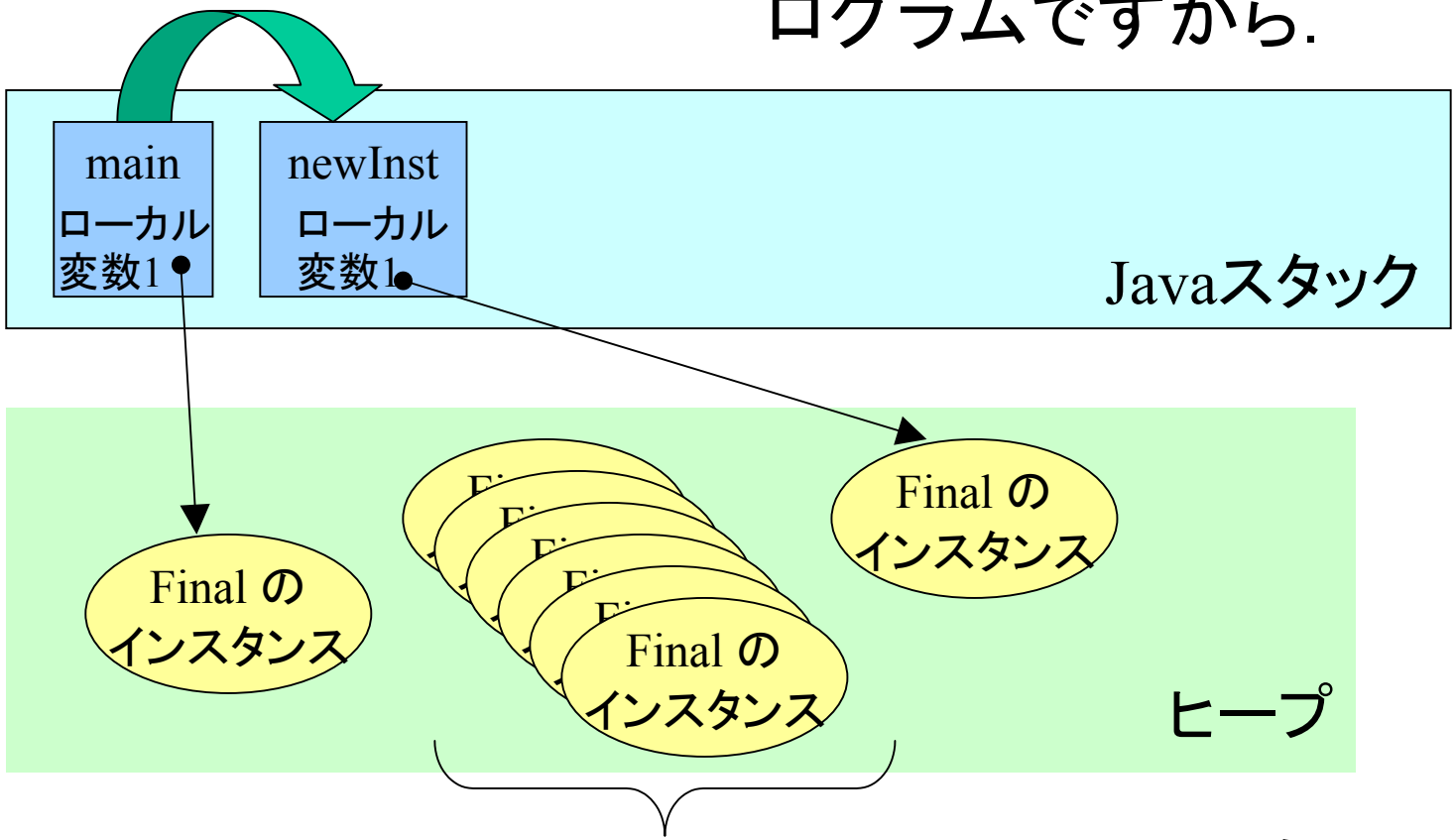
- 初めてfinalizeがよばれたら，VMを停止する.
- この例題から，使わなくなったからといって，すぐにGCされるわけでないことがわかる.

```
class Final{
static int c;
  void newInst(){
    Final f=new Final();
  }

  public void finalize(){
    System.out.println(this.toString()+" at "+Final.c);
    System.exit(0);
  }

  public static void main(String[] args){
    Final o=new Final();
    System.out.println(o.toString());
    for(c=0 ;true; c++) o.newInst();
  }
}
```

前述のプログラムの実行時構造

この呼び出しを繰り返す

下記の通り，かなりバカげてるが，まぁ説明用のプログラムですから．



この辺のインスタンスはリンク切れ ⇒ GCの対象

今日はこれまで