実行時のメモリ構造(2)
Javaスタック内動作他

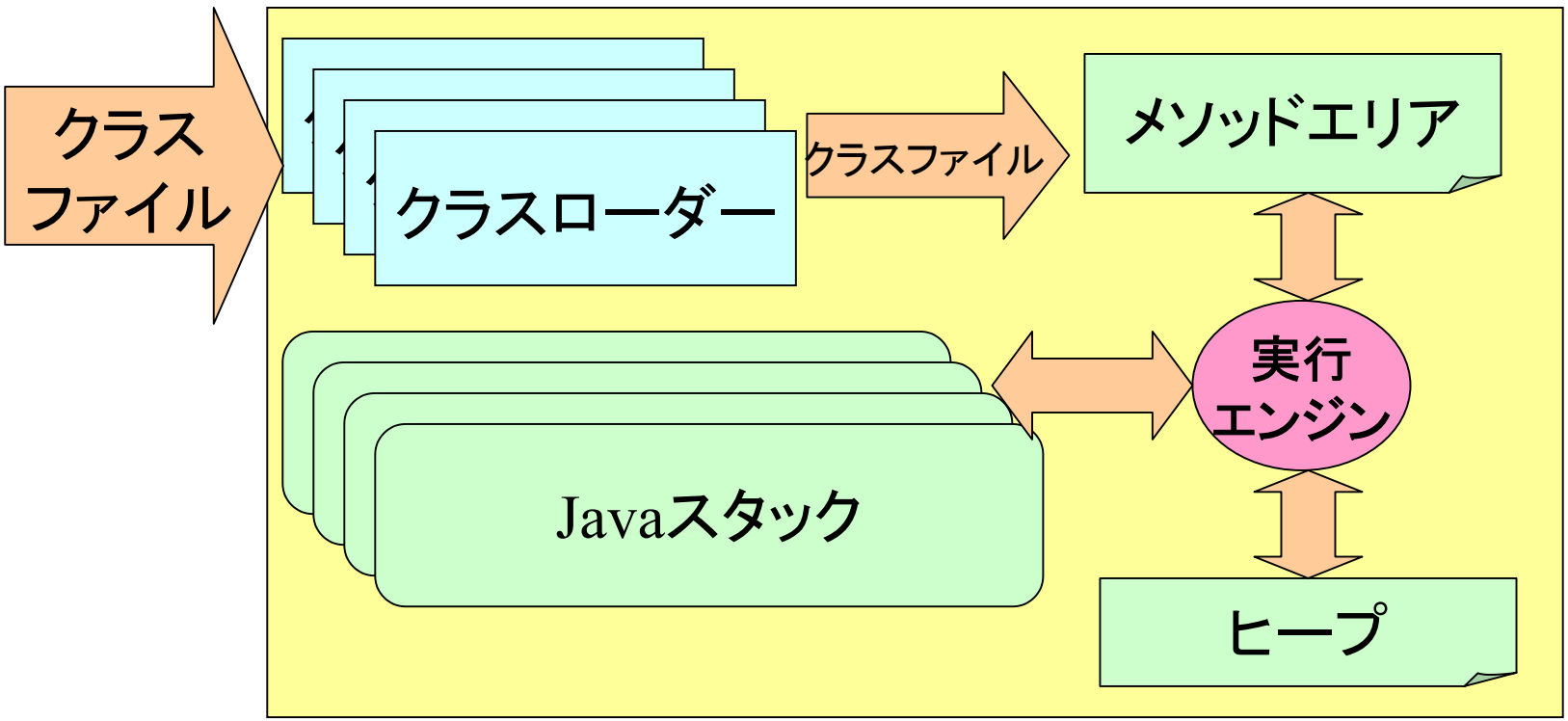
2002年5月27日

海谷 治彦

JVM内の基本構造(大雑把)

クラスファイルの
内容チェック

クラスデータを保存



各実行スレッドのローカルデータ
(実行経過)を保存

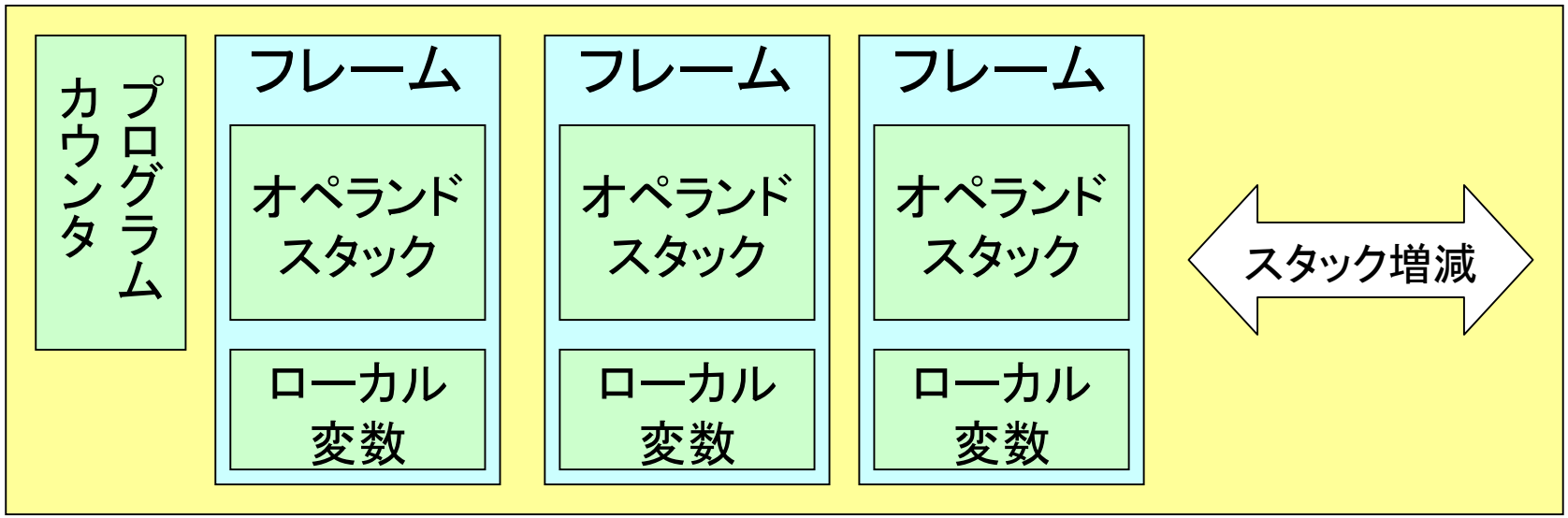
インスタンスデータ
を保存

教科書 p.15

* 原著および教科書p.15をベースに書いた.

リンク有

Javaスタック内の構造



- 「フレーム」という要素のスタック.
- フレームは，1回のメソッド呼び出しに対応.
- フレーム内の計算のためにも，スタック（オペランドスタック）が利用されている.
- 詳細は「実行時の構造」の回にて.

例えば教科書 p.20の図

本日のお題

- Javaスタック内のフレームの増減を理解
  - メソッド(関数)呼び出しの繰り返しによる計算機構の復習(or 理解).
- インスタンス変数の扱い
  - 情報隠蔽の実際
- 静的メソッド，変数について
- コンストラクタについて

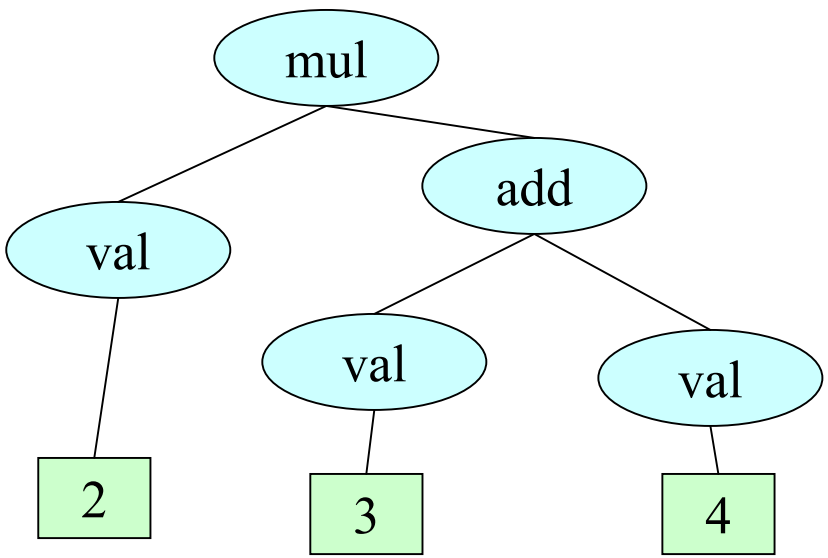
例題: 簡単な計算クラス

- calc/Calc.java

```
public class Calc{
    int val(int v){return v;}

    int add(int a, int b){return a+b; }
    int sub(int a, int b){ return a-b;}
    int mul(int a, int b){return a*b;}

    public static void main(String[] args){
        Calc c=new Calc();
            int a=c.mul(c.val(2), c.add(c.val(3), c.val(4))); // 2*(3+4)
            System.out.println(a);
    }
}
```

関数呼び出しの構造

- calc/Calc.java
- a=c.mul(c.val(2), c.add(c.val(3), c.val(4)));



特にvalはアホ見たいなメソッドだけど，インスタンス属性を使わないとこうなってしまう.....

逆ポーランドによる展開

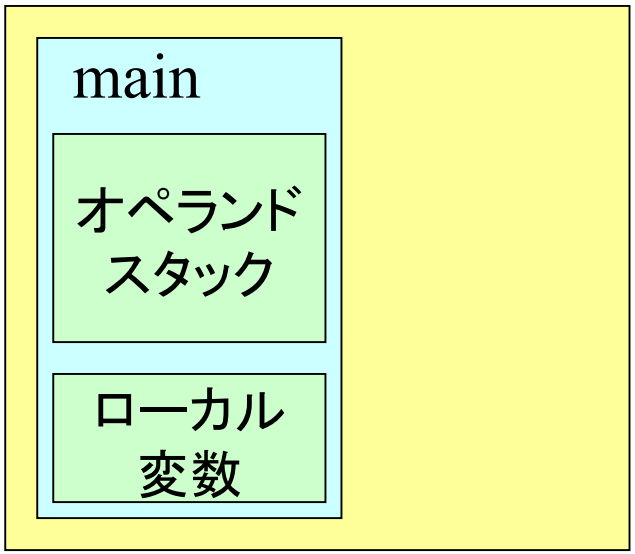
- calc/Calc.java
- a=c.mul(c.val(2), c.add(c.val(3), c.val(4)));
- メソッド呼び出し自体，逆ポーランドの展開される.
  - mainメソッド内のアセンブラ参照
- **2 val 3 val 4 val add mull**

```
.method .... main .....

aload_1
    aload_1
    iconst_2
    invokevirtual Calc/val(I)I
    aload_1
    aload_1
    iconst_3
    invokevirtual Calc/val(I)I
    aload_1
    iconst_4
    invokevirtual Calc/val(I)I
    invokevirtual Calc/add(II)I
    invokevirtual Calc/mul(II)I
    istore_2
```

invokevirtual
インスタンスメソッドの呼び出し

- 例 invokevirtual Calc/val(I)I
- これによって，新しいフレームが生成される.
- 返り値がV(void)でない限り，返り値はすぐ下のフレームのオペランドスタックに載せられる. (これがメソッドの返り値)

例題のトーレス 1/11



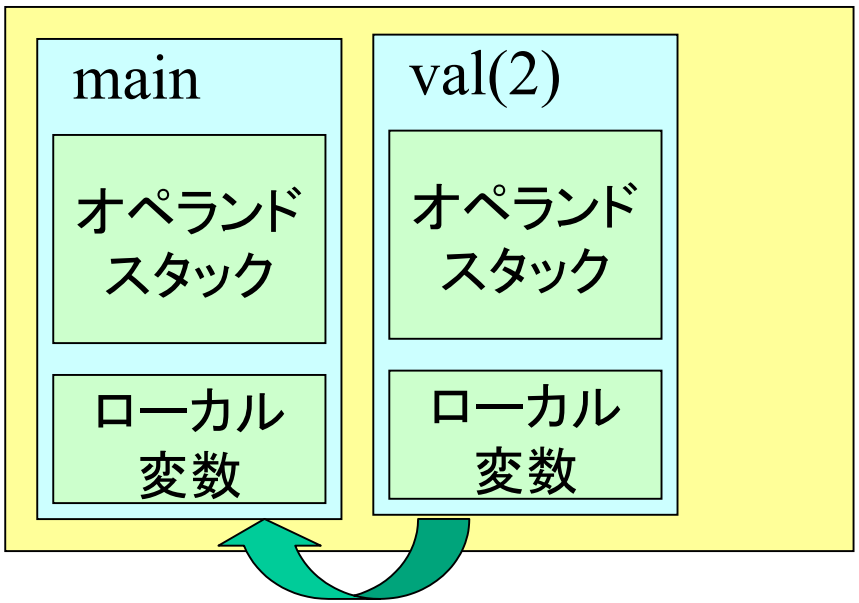

```
.method .... main .....

    aload_1
    aload_1
    iconst_2
    invokevirtual Calc/val(I)I
    aload_1
    aload_1
    iconst_3
    invokevirtual Calc/val(I)I
    aload_1
    iconst_4
    invokevirtual Calc/val(I)I
    invokevirtual Calc/add(II)I
    invokevirtual Calc/mul(II)I
    istore_2
```

local=[args, c,] stack=[c, c, 2]
invokevirtual Calc/val(I)I

スタック上からcと2をとって，valを呼び出す.

- インスタンスメソッドなのでターゲットインスタンスリファレンス(c)が必要.
- 2はval自体の引数.

例題のトーレス 2/11



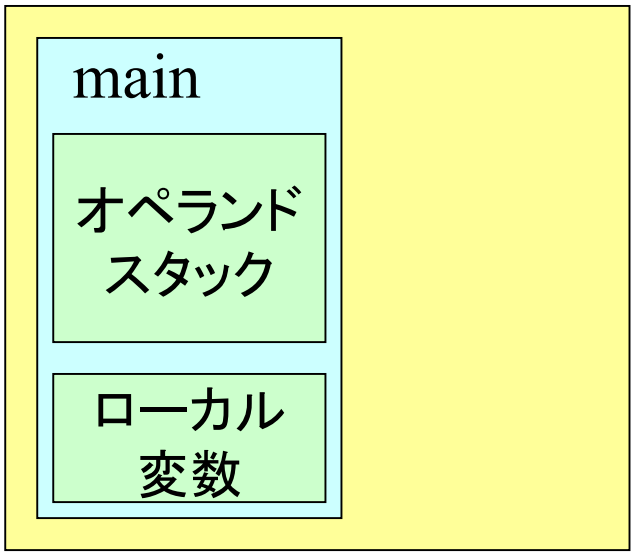
2が帰る

```
local=[this, 2] stack=[]
iload_1
local=[this, 2] stack=[2]
ireturn
```

```
.method .... main .....

    aload_1
    aload_1
    iconst_2
    invokevirtual Calc/val(I)I
    ...
    ...lc/val(I)I
    ...
    ...lc/val(I)I
    ...lc/add(II)I
    invokevirtual Calc/mul(II)I
    istore_2
```

```
.method val(I)I
    .limit stack 1
    .limit locals 2
    iload_1
    ireturn
.end method
```

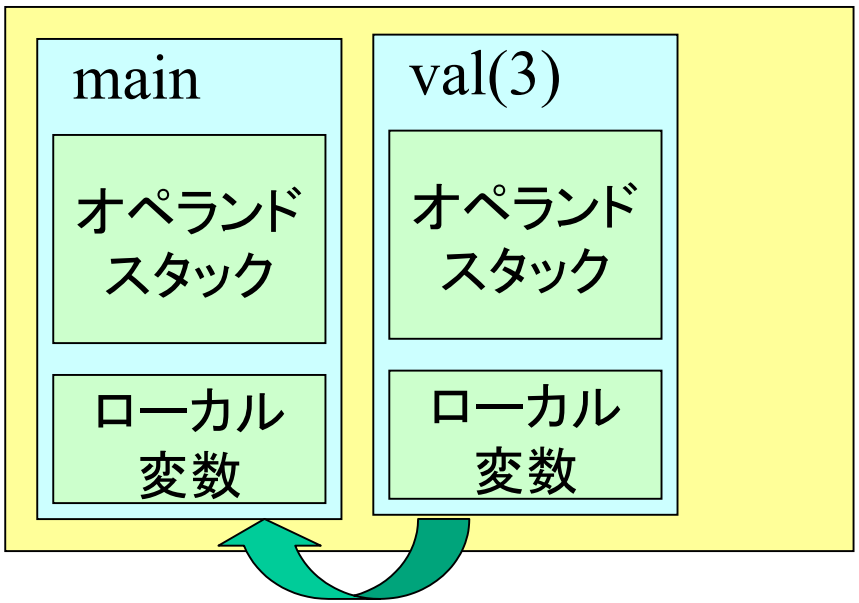
例題のトーレス 3/11



local=[args, c,] stack=[c, c, 2]
invokevirtual Calc/val(I)I
local=[args, c,] stack=[c, 2]
aload_1
aload_1
iconst_3
local=[args, c,] stack=[c,2,c,c,3]
invokevirtual Calc/val(I)I

```
.method .... main .....

    aload_1
    aload_1
    iconst_2
    invokevirtual Calc/val(I)I
    aload_1
    aload_1
    iconst_3
    invokevirtual Calc/val(I)I
    aload_1
    iconst_4
    invokevirtual Calc/val(I)I
    invokevirtual Calc/add(II)I
    invokevirtual Calc/mul(II)I
    istore_2
```

例題のトーレス 4/11

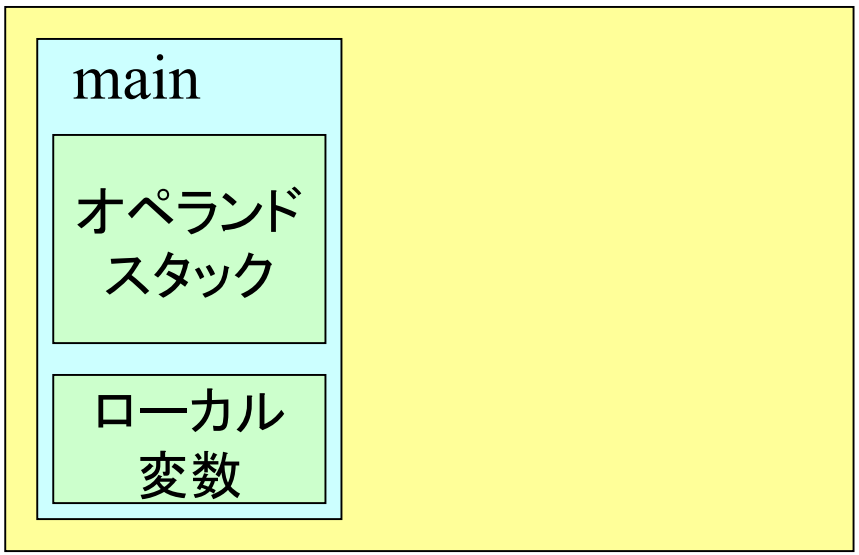


3が帰る

.....
local=[args, c,] stack=[c,2,c,c,3]
invokevirtual Calc/val(I)I
local=[args, c,] stack=[c,2,c,3]

.....
.method val(I)I
.limit stack 1
.limit locals 2
iload_1
ireturn
.end method
Calc/val(I)I
iconst_3
invokevirtual Calc/val(I)I
aload_1
iconst_4
invokevirtual Calc/val(I)I
invokevirtual Calc/add(II)I
invokevirtual Calc/mul(II)I
istore_2

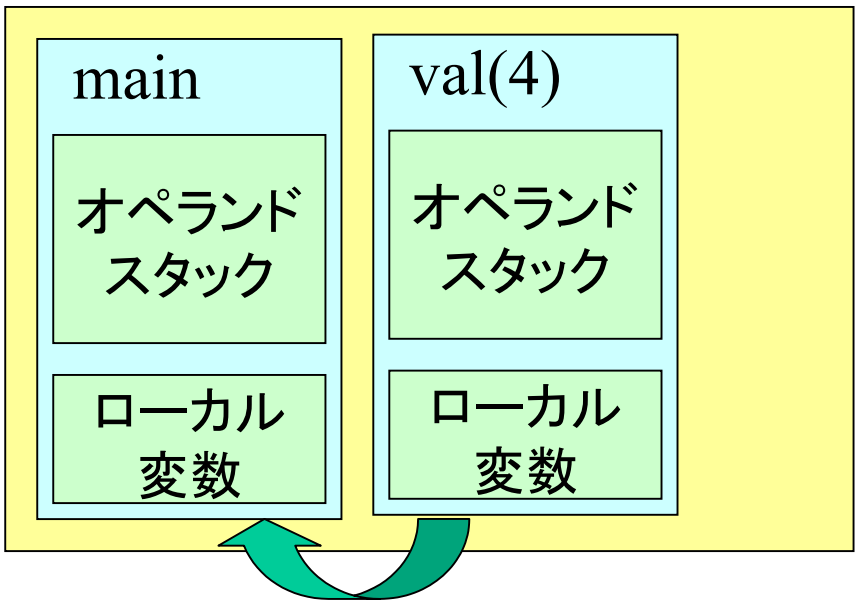
例題のトーレス 5/11




```
.....
local=[args, c, ] stack=[c,2,c,3]
aload_1
iconst_4
local=[args, c, ] stack=[c,2,c,3,c,4]
```

```
.method .... main .....

    aload_1
    aload_1
    iconst_2
    invokevirtual Calc/val(I)I
    aload_1
    aload_1
    iconst_3
    invokevirtual Calc/val(I)I
    aload_1
    iconst_4
    invokevirtual Calc/val(I)I
    invokevirtual Calc/add(II)I
    invokevirtual Calc/mul(II)I
    istore_2
```

例題のトーレス 6/11



4が帰る

.....

local=[args, c,] stack=[c,2,c,3,c,4]
invokevirtual Calc/val(I)I
local=[args, c,] stack=[c,2,c,3,4]

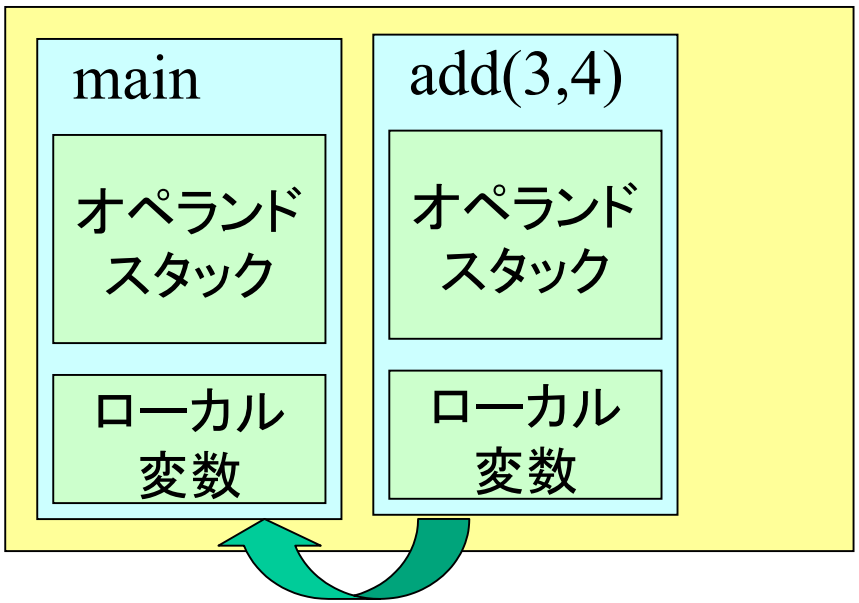
```
.method val(I)I
    .limit stack 1
    .limit locals 2
    iload_1
    ireturn
.end method
```

.....

alc/val(I)I

```
    iconst_3
    invokevirtual Calc/val(I)I
    aload_1
    iconst_4
    invokevirtual Calc/val(I)I
    invokevirtual Calc/add(II)I
    invokevirtual Calc/mul(II)I
    istore_2
```

例題のトーレス 7/11

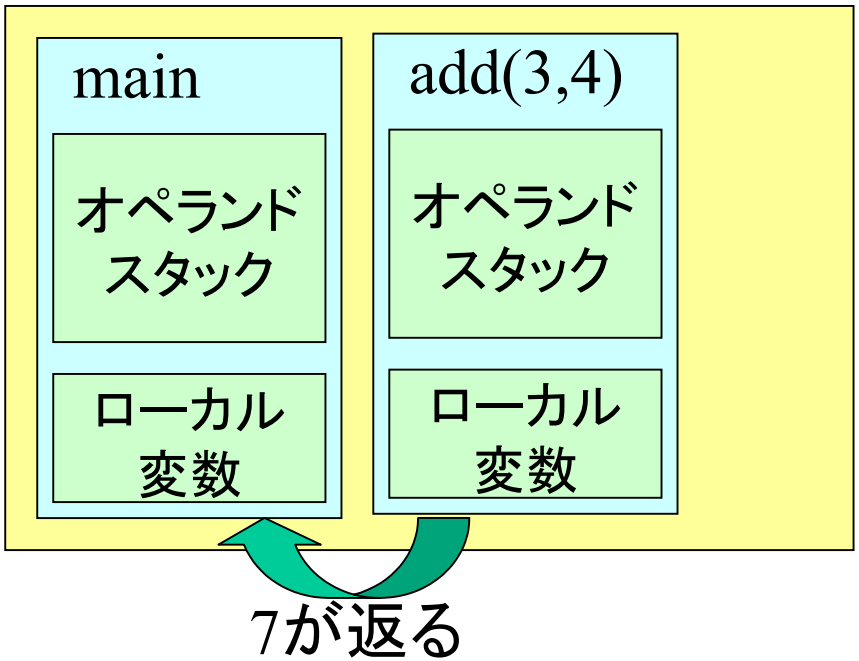


.....
local=[args, c,] stack=[c,2,c,3,4]
invokevirtual Clac/add(II)I

```
.method add(II)I
    .limit stack 2
    .limit locals 3
    iload_1
    iload_2
    iadd
    ireturn
.end method
```

```
method  main .....
                                  Calc/val(I)I
                                  Calc/val(I)I
    aload_1
    iconst_4
    invokevirtual Calc/val(I)I
    invokevirtual Calc/add(II)I
    invokevirtual Calc/mul(II)I
    istore_2
```

例題のトーレス 8/11

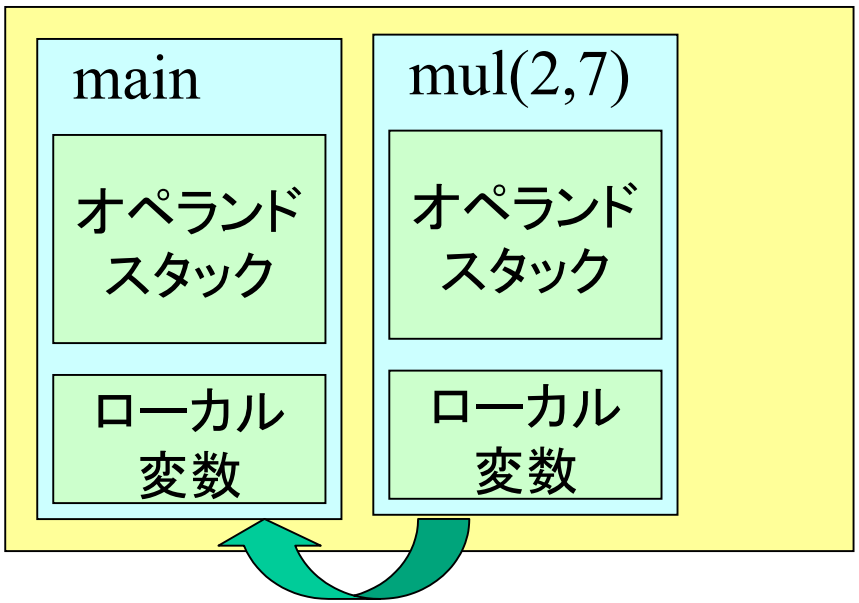


local=[this,3,4] stack=[]
iload_1
iload_2
local=[this,3,4] stack=[3,4]
iadd
local=[this,3,4] stack=[7]
ireturn

```
.method add(II)I
    .limit stack 2
    .limit locals 3
    iload_1
    iload_2
    iadd
    ireturn
.end method
```

```
method  main .....
                                  Calc/val(I)I
                                  Calc/val(I)I
    aload_1
    iconst_4
    invokevirtual Calc/val(I)I
    invokevirtual Calc/add(II)I
    invokevirtual Calc/mul(II)I
    istore_2
```

例題のトーレス 9/11

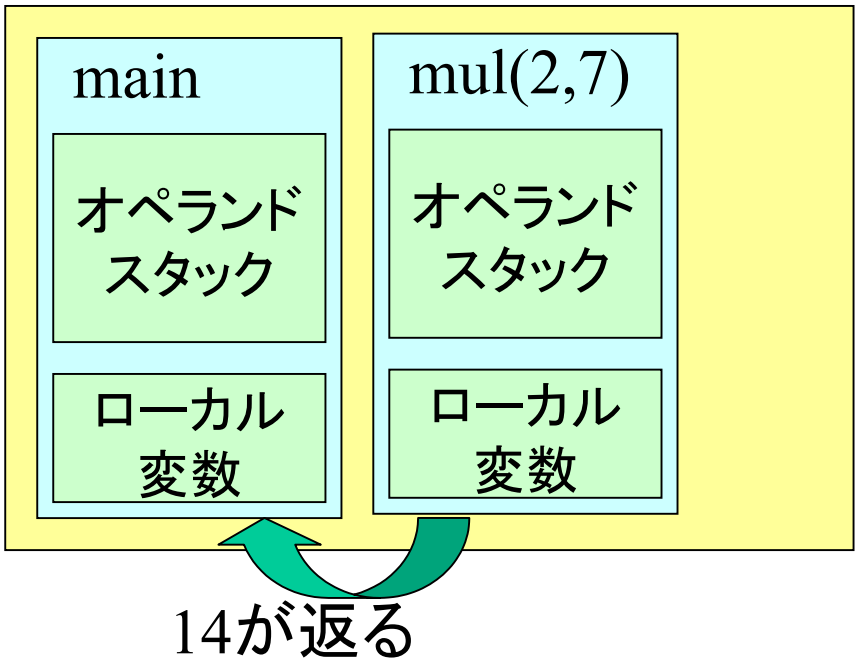


.....
local=[args, c,] stack=[c,2,c,3,4]
invokevirtual Clac/add(II)I
local=[args, c,] stack=[c,2,7]
invokevirtual Calc/mul(II)I

```
.method mul(II)I
    .limit stack 2
    .limit locals 3
    iload_1
    iload_2
    imul
    ireturn
.end method
```

```
method    main .....
                                    Calc/val(I)I
                                    Calc/val(I)I
    aload_1
    iconst_4
    invokevirtual Calc/val(I)I
    invokevirtual Calc/add(II)I
    invokevirtual Calc/mul(II)I
    istore_2
```

例題のトーレス 10/11

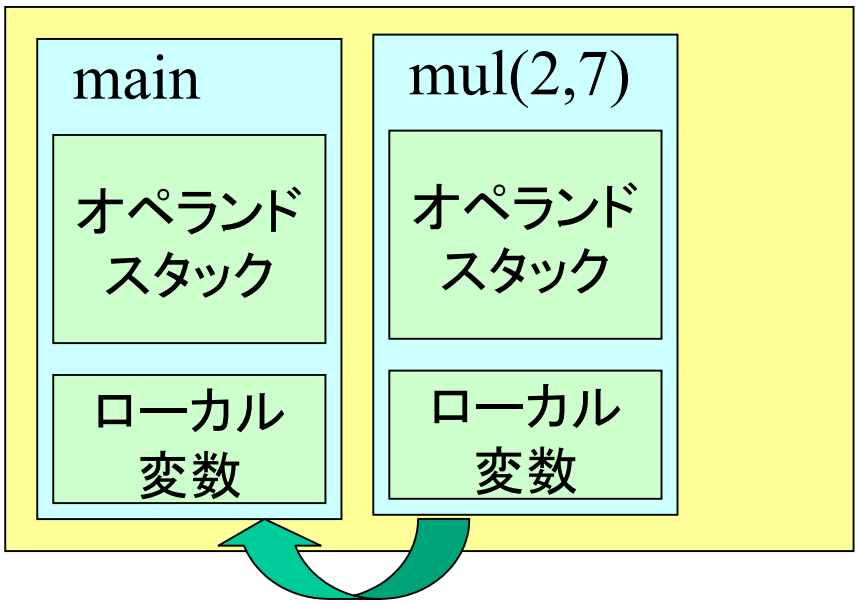


local=[this,2,7] stack=[]
iload_1
iload_2
local=[this,2,7] stack=[2,7]
imul
local=[this,2,7]stack=[14]
ireturn

```
.method mul(II)I
    .limit stack 2
    .limit locals 3
    iload_1
    iload_2
    imul
    ireturn
.end method
```

```
method    main .....
                                  Calc/val(I)I
                                  Calc/val(I)I
    aload_1
    iconst_4
    invokevirtual Calc/val(I)I
    invokevirtual Calc/add(II)I
    invokevirtual Calc/mul(II)I
    istore_2
```

例題のトーレス 11/11



.....
local=[args, c,] stack=[c,2,7]
invokevirtual Calc/mul(II)I
local=[args, c,] stack=[14]
istore_2
local=[args, c, 14]

```
.method mul(II)I
    .limit stack 2
    .limit locals 3
    iload_1
    iload_2
    imul
    ireturn
.end method
```

```
method  main .....
                                  Calc/val(I)I
                                  Calc/val(I)I
    aload_1
    iconst_4
    invokevirtual Calc/val(I)I
    invokevirtual Calc/add(II)I
    invokevirtual Calc/mul(II)I
    istore_2
```

トレースの感想

- つ，疲れる.
- 単純作業を正確にこなすコンピュータの奇跡に驚嘆. (人間に無理)
- インスタンスのリファレンスがほとんど無駄に見えたけど，これに関しては，methodの入れ子呼び出しを行っていないため.
  - Javaスタック内のフレームが2個以上になってない.
- mainメソッド上からの呼び出しなので，例題が悪かった....

再帰

- 詳細は rpower/ の下を参照.
- 解説はWebページを参照.

```
int rpower(int a, int b){
      if(b>0)
        return a * rpower(a, b-1);
      else
        return 1;
   }
```

```
.method rpower(II)I
    .limit stack 5
    .limit locals 3
    iload_2
    ifle Label1
    iload_1
    aload_0
    iload_1
    iload_2
    iconst_1
    isub
    invokevirtual Calc/rpower(II)I
    imul
    ireturn
Label1:
    iconst_1
    ireturn
.end method
```

インスタンス変数

```
class IncDec{
private int s;
    IncDec(int a){ s=a; }

    void inc(int a){ s+=a; }
    void dec(int a){ s-=a; }

    int value(){return s;}

    public static void main(String[] args){
        IncDec id=new IncDec(10);
        id.inc(3);
        id.dec(8);
        System.out.println(id.value());
    }
}
```

increment and
decrement methods.

初期値を10にして，
3増やして，inc(3)
8減らす dec(8)
という.... な計算.

静的メソッド，変数

- 詳細は webページ static/ を参照.
- invokestatic
- getstatic / putstatic
- ローカル変数0 にインスタンスが入らない.
- 静的変数は名前がそのまま残る. (vなど)

```
static int v;
static void add(int a){v += a;}
```

```
.class  public Static
.super  java/lang/Object
.field private static v I

.method static add(I)V
    .limit stack 2
    .limit locals 1
    getstatic Static/v I
    iload_0
    iadd
    putstatic Static/v I
    return
.end method
```

静的変数の初期化

```
class InitStatic{
static int uptoval=100;
int val;
    boolean add(int a){
        if(val+a>uptoval) return false;
        else val += a;
        return true;
    }
}
```

```
.class  InitStatic
.super  java/lang/Object
.field static uptoval I
.field val I

; 中略

.method static <clinit>()V
    .limit stack 1
    .limit locals 0
    bipush 100
    putstatic InitStatic/uptoval I
    return
.end method
```

変数操作のまとめ

- インスタンス変数
  - getfield p.302
  - putfield p.444
- 静的変数
  - getstatic p.306
  - putstatic p.447

コンストラクタ

- デフォルトのコンストラクタ
- 明示的に再定義した場合
- 明示的にスーパークラスを指定
- 実装するインタフェースを指定

デフォルトコンストラクタ

- 例えば，calc/ の例など.
- デフォではObjectのサブクラスなので，そのコンストラクタを呼んでいる.

```
.class  public Calc
.super  java/lang/Object

.method public <init>()V
    .limit stack 1
    .limit locals 1
    aload_0
    invokespecial java/lang/Object/<init>()V
    return
.end method
```

明示的に再定義

- 例えば，inc-dec/ の例など.
- デフォルトのコンストラクタは無くなる.
- それでも，superの初期化はする.
- superの初期化が先に行われている.(重要)

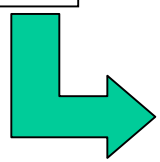
```
class IncDec{
private int s;
IncDec(int a){ s=a; }
```

```
.class  IncDec
.super  java/lang/Object
.field private s I

.method <init>(I)V
    .limit stack 2
    .limit locals 2
    aload_0
    invokespecial java/lang/Object/<init>()V
    aload_0
    iload_1
    putfield IncDec/s I
    return
.end method
```

明示的なスーパークラス

- 明示的再定義とそう変わらない.

```
class MyThread
   extends Thread{
private int s;
    MyThread(int s){
        this.s=s;
    }
}
```



```
.class  MyThread
.super  java/lang/Thread
.field private s I

.method <init>(I)V
    .limit stack 2
    .limit locals 2
    aload_0
    invokespecial java/lang/Thread/<init>()V
    aload_0
    iload_1
    putfield MyThread/s I
    return
.end method
```

明示的スーパークラス その2

```
public class Supers extends Super1{
private int ss;
   Supers(){
      super(3,2);  ss=4;
   }
}

class Super1 extends Super2{
private int s1;
   Super1(int s1, int s2){
      super(s2);  this.s1=s1;
   }
}

class Super2 {
private int s2;
   Super2(int s2){ this.s2=s2; }
}
```

```
.class  public Supers
.super  Super1
.field private ss I

.method <init>()V
    .limit stack 3
    .limit locals 1
    aload_0
    iconst_3
    iconst_2
    invokespecial Super1/<init>(II)V
    aload_0
    iconst_4
    putfield Supers/ss I
    return
.end method
```

続き

```
.class  Super1
.super  Super2
.field private s1 I

.method <init>(II)V
    .limit stack 2
    .limit locals 3
    aload_0
    iload_2
    invokespecial Super2/<init>(I)V
    aload_0
    iload_1
    putfield Super1/s1 I
    return
.end method
```

```
.class  Super2
.super  java/lang/Object
.field private s2 I

.method <init>(I)V
    .limit stack 2
    .limit locals 2
    aload_0
    invokespecial java/lang/Object/<init>()V
    aload_0
    iload_1
    putfield Super2/s2 I
    return
.end method
```

インタフェースの実装

```
import java.util.*;

public class MyVector
    extends Vector
    implements Runnable{

    public void run(){  }
}
```

実装してるインタフェース情報が追加されている.
importは展開されている.

```
.class  public MyVector
.super  java/util/Vector
.implements java/lang/Runnable

.method public <init>()V
    .limit stack 1
    .limit locals 1
    aload_0
    invokespecial java/util/Vector/<init>()V
    return
.end method

.method public run()V
    .limit stack 0
    .limit locals 1
    return
.end method
```

インスタンスの作成

- new クラス名
- 必要なヒープ確保がされるだけ.
- 初期設定は明示的に他で.

```
class MyThread extends Thread{
private int s;
   MyThread(int s){
      this.s=s;
   }

   public static void main(String[] args){
      MyThread m=new MyThread(3);
   }
}
```

```
.limit stack 3
.limit locals 2
new MyThread
dup
iconst_3
invokespecial MyThread/<init>(I)V
astore_1
return
```

- 通常は，後の操作のため，参照の複製を作る. (dup)
- その複製をローカル変数へ書き込む. (astore_?)

メソッド呼び出しのまとめ

- invokespecial 下記のメソッド呼び出し p.364
  - インスタンス初期化
  - privateなインスタンスメソッド
  - superのインスタンスメソッド
- invokevirtual 上記以外のインスタンスメソッド p.372
- invokestatic 静的メソッド p.369
- invokeinterface インタフェースのメソッド p.361