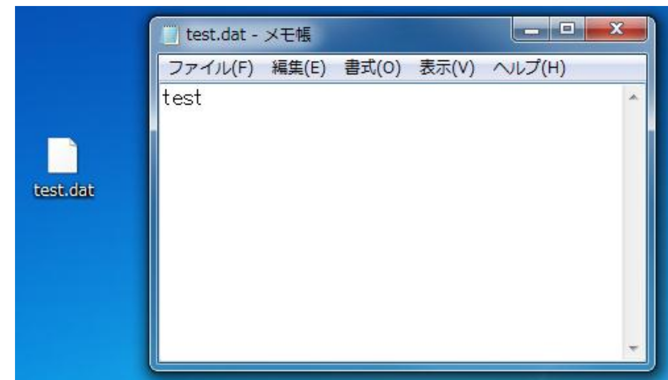
プログラミング基礎

第12回

ファイル入出力／キーボード入力と
画面出力／静的メンバ

ファイルについて

- ファイル名とファイル内容は独立している（ファイル名を変えてもファイル内容に影響はない，逆もしかり）



（ファイル名（拡張子）を変更しても同じアプリケーションで開ける）

ファイルについて

- 拡張子はZIPだったが，ファイル内容は実はLZHファイルだった，ということもありうる

→拡張子（ファイル名）によってファイル内容を判断するプログラムを書いてはいけない

→通常，ファイル先頭にファイル内容を表す情報が書かれているので，それでファイル内容を判断するのが常套手段

ファイルについて

ファイル入出力

プログラムの話の前に：
ファイルは内容の形式によって
大きく2種類に分かれる．

テキスト形式：
文字通り，文字列が書かれたファイル
（各種ソースコード，ログファイル，etc.）

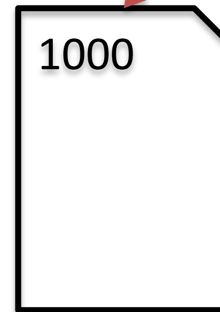
バイナリ形式：
メモリ上のデータを
そのまま書き出したファイル
（実行ファイル，ダンプファイル，
ゲームのセーブデータ，etc.）
＊バイナリ＝2進数

本講義ではこちらを扱う

例：

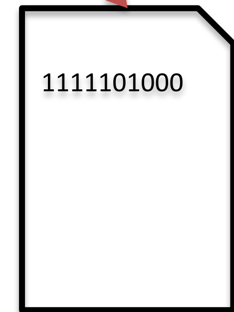
```
int x;
x = 1000;
```

xをテキスト形式
として出力



'1''0''0''0'
という文字の羅列
（テキスト）になる

xをバイナリ形式
として出力



1000を表す
2進数が出力される
（2進数では長いので
通常は16進数で扱う）

演習

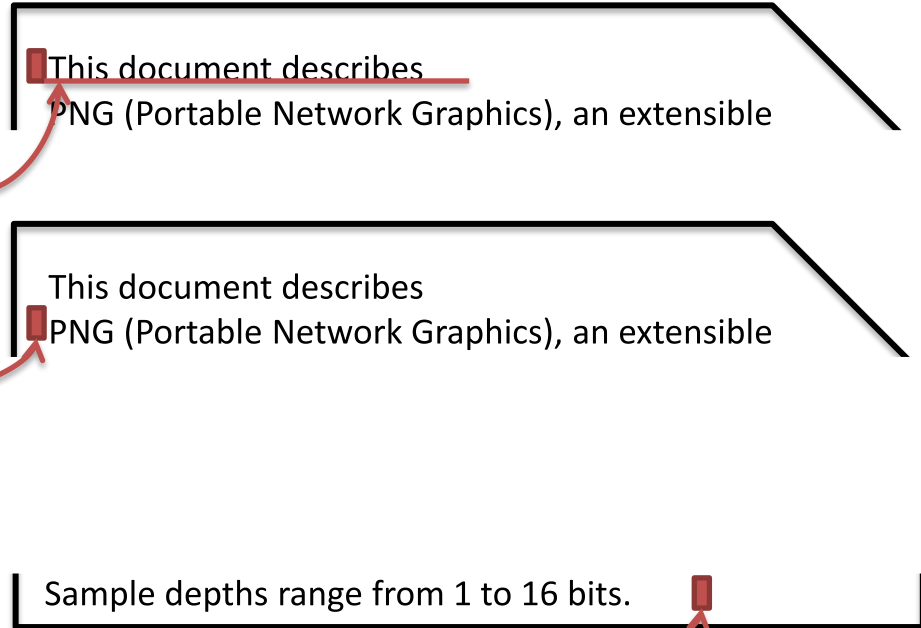
- Web資料の「ファイルからの入力の例」を動かしなさい

→読み込むファイルは，ファイル名だけ指定した場合は，classファイルと同じフォルダ（ディレクトリ）に置く（test.txt）
→もしくは絶対パスで指定すること（c:¥hoge¥test.txt）
→表示するファイルは各自適当に用意する
http://tools.ietf.org/rfc/

BufferedReader

str = in.readLine();

⇒ファイルから1行読み込み，
  str配列へ代入する
⇒その後次の行の先頭に進む



...
str = in.readLine();
if (str == null) {...}

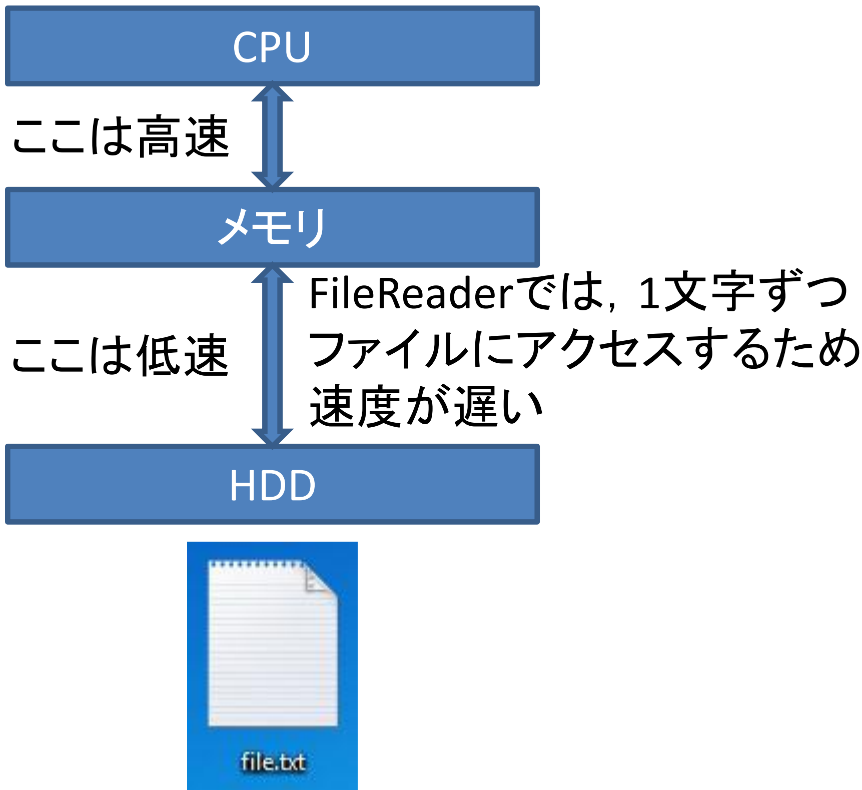
⇒ファイルの終端の場合，
  （以降に文字が存在しない）
  readLineメソッドはnullを戻り値として返す

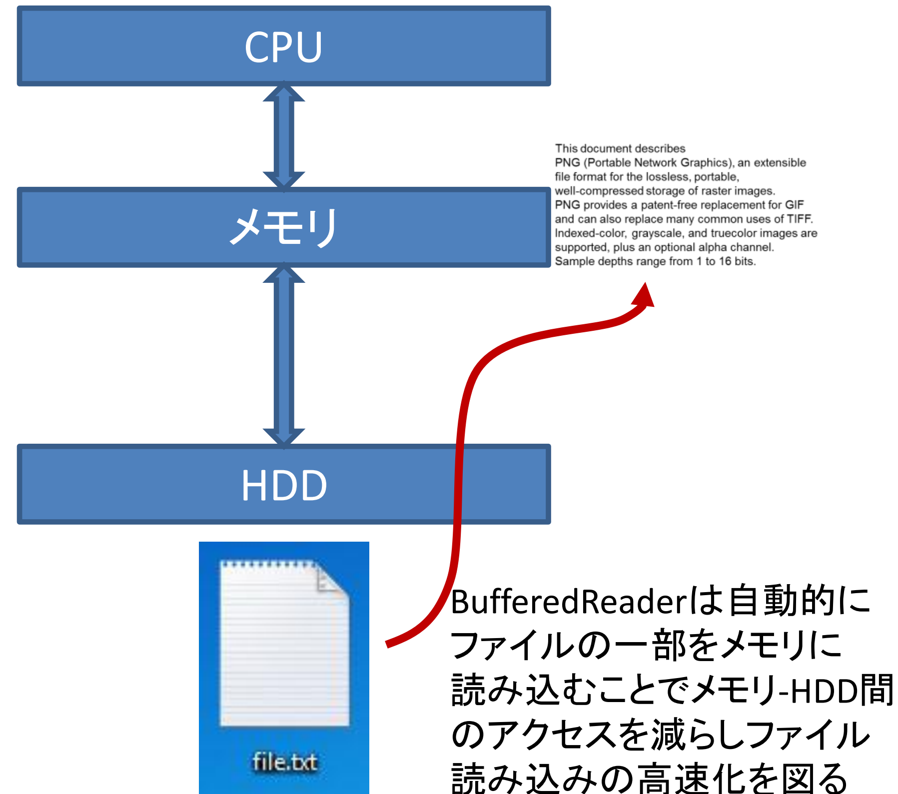
BufferedReaderの役割

```
innerReader = new FileReader("sample.txt");
in = new BufferdReader(innerReader);
```

FileReaderだけでもファイルから
文字を読み込むことができる

ではなぜBufferedRederを使うのか？





ファイルへの出力の例

- PrinteWriterクラスを使用すると普段使用しているSystem.out.printlnと同様の操作でファイルへの出力が可能

- BufferedWriterでも同等のことが可能

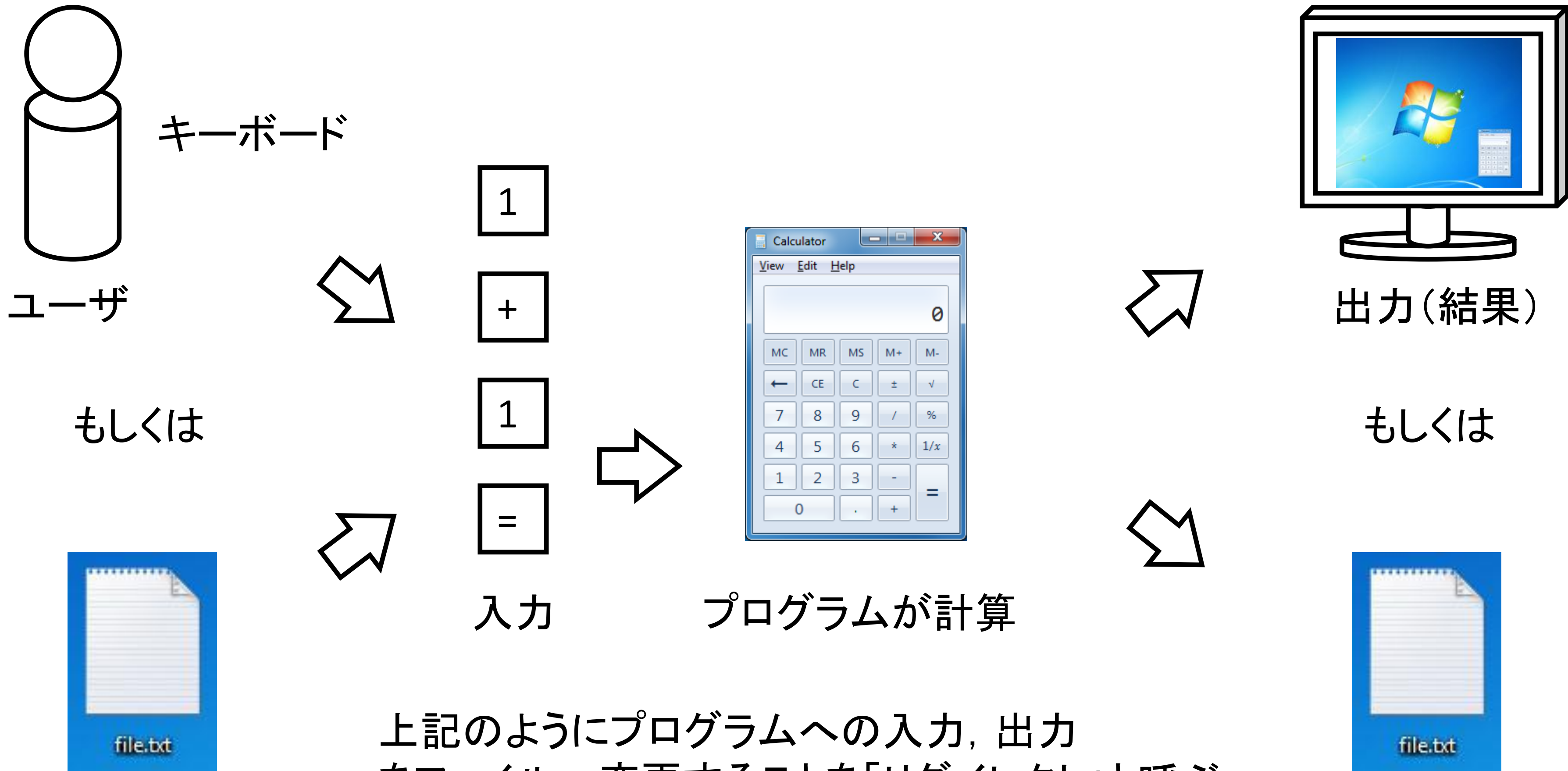
```
BufferedWriter out = null;
out = new BufferedWriter( new FileWriter("sample.txt"));

out.write(("Hello, Java.");
out.newLine();
```

標準入出力

- いままで使用してきた，System.out，
  と今回キーボード入力に使用したSystem.in
  は標準入出力を扱うためのクラス

- デフォルトでは標準入力：キーボード
  標準出力：画面
  となるが，プログラム起動時の指定によって
  変更が可能

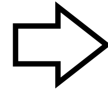
標準入出力



上記のようにプログラムへの入力，出力をファイルへ変更することを「リダイレクト」と呼ぶ

ストリーム

外部から送られてくる（流れてくる＝ストリーム）
データを統一して扱う仕組みのこと



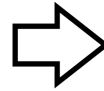
1

+

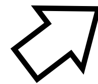
1

=

入力



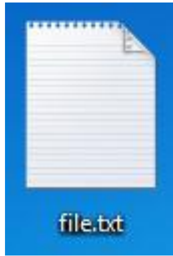
クラスが計算

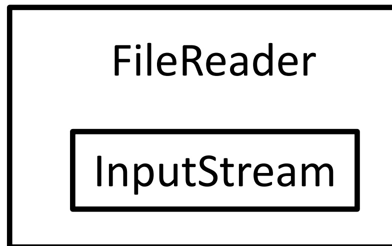


ファイルから

クラスはどこから文字を入力されるかは知らないが
それを気にせずに文字を読み込んで処理する
ことが可能になる

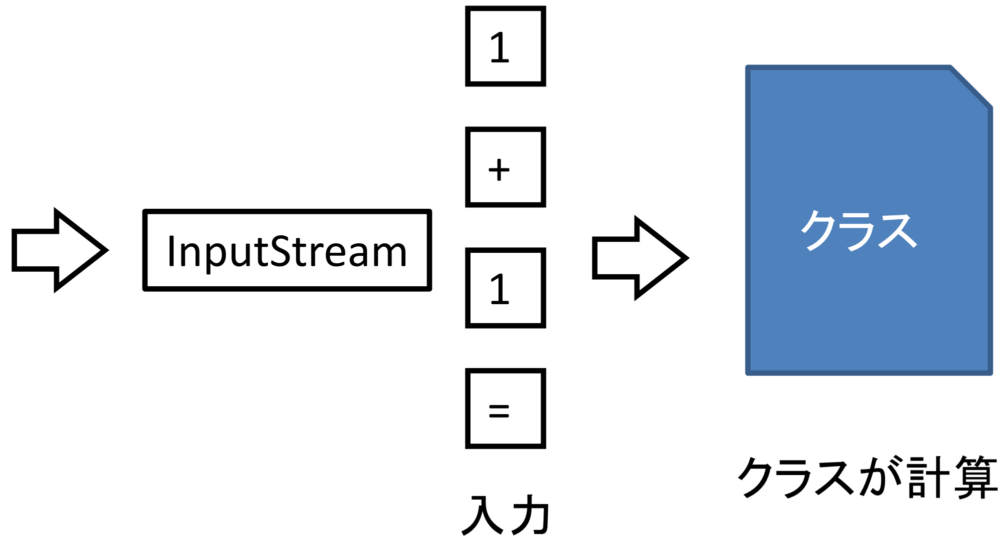
ストリーム





あるクラスAにファイルから文字を読み込ませたいのであれば，

InputStreamクラスを継承した，FileReaderクラスのインスタンスを渡す



入力

クラスが計算

あるクラスAはInputStreamクラスのインスタンスから文字を読み取る

⇒クラスAは文字をどこから読み込むのか意識しなくてよい

try catch文（例外）

• 基本的な書き方

プログラム実行中になんらかの
異常が発生すること⇒例外

これからプログラムをやってみる⇒try
例外を捕まえた⇒catch

```
try
{
        //例外が発生する可能性のある
        //処理をここに書く
}catch(Exception error)
{
        //例外が発生した場合の処理をここに書く
}
```

try catch文

- 基本的な書き方

```
try
{
        //例外が発生する可能性のある
        //処理をここに書く
}catch(Exception e)
{
        //例外が発生した場合の処理をここに書く
}
```

⇒例外の基本クラス

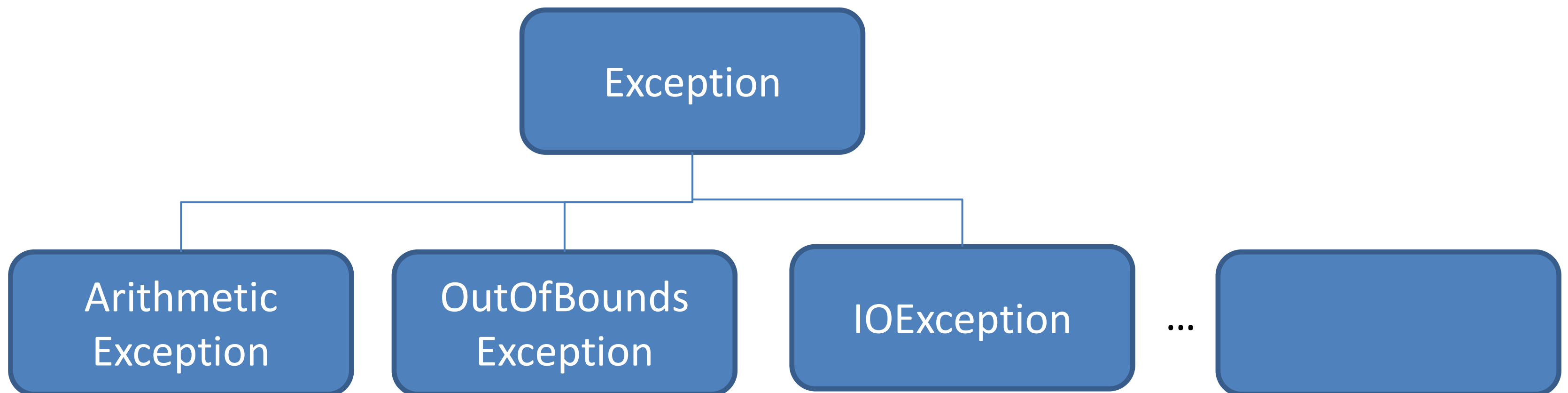
Exceptionクラス

```
public class Exceptions
{
    public static void main(String[] args)
    {
        try
        {
            int x;
            x = 1 / 0;
            int a = new int[3];
            a[3] = 100;
        }
        catch(Exception e)
        {
            System.out.println("例外が発生しました");
        }
        System.out.println("プログラムを終了します");
    }
}
```

⇒さまざまな種類の例外が発生する可能性があるが・・・

⇒どんな例外が発生したか分からない

Exceptionクラス



＊プログラマが独自のExceptionを定義することも可能

Exceptionクラス

```
public class Exceptions
{
   public static void main(String[] args)
   {
      try
      {
         int x;
         x = 1 / 0;
         int a = new int[3];
         a[3] = 100;
      }
      catch(ArithmeticException e)
      {
         System.out.println("算術計算に失敗しました");
      }
      catch(OutOfBoundsException e)
      {
         System.out.println("配列の範囲外にアクセスしようとしました");
      }
      System.out.println("プログラムを終了します");
   }
}
```

⇒発生した例外の種類によって
処理を書き分けることが出来る

演習

- 以下のプログラムにtrycatch文を追記し，途中で止まらないようにせよ（プログラム名：ZeroException）

```
public class ZeroException
{
    public static void main(String[] args)
    {
        int x;
        x = 1 / 0;

        System.out.println("プログラムを終了します");
    }
}
```

なぜ例外を使うのか

- 例外を使用しない場合

```
結果1=処理1;
If (結果1が〇〇エラーの場合)
{
    ....
    エラー処理;
}
else if (結果1が〇〇エラーの場合)
{
    ....
    エラー処理;
}
```

なぜ例外を使うのか

- 例外を使用しない場合

```
結果1=処理1;
If (結果1が〇〇エラーの場合)
{
    ....
    エラー処理;
}
else if (結果1が〇〇エラーの場合)
{
    ....
    エラー処理;
}


結果2=処理2;
If (結果2が〇〇エラーの場合)
{
    ....
    エラー処理;
}
else if (結果2が〇〇エラーの場合)
{
    ....
    エラー処理;
}
```

エラー処理の記述が長くなり，
プログラムが見づらくなる

なぜ例外を使うのか

- 例外を使用した場合

```
try
{
   処理1;
   処理2:
}catch(〇〇エラーの場合)
{
    ....
    エラー処理;
}catch(〇〇エラーの場合)
{
    ....
    エラー処理;
}
```

エラー処理の記述が短くなり，
プログラムが見やすい

import文

- Javaには標準で約3800個のクラスがある
  http://java.sun.com/javase/ja/6/docs/ja/api/

- そのままでは，どれがどのような役割を持つクラスなのか分かりづらい

→同じような役割を持つクラスをまとめたもの

⇒**パッケージ**と呼ばれる

＊ユーザ自身がパッケージを作ることも可能

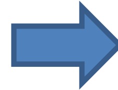
import文

- 通常は「パッケージ名.クラス名」と記述してクラスを使用する
  例：java.util.Random

→しかしクラスを使用する度，毎回パッケージ名を書くのは面倒くさいのでimport文を使用して省略できる

import文

```
public class RandomTest
{
    static public void main(String[] args)
    {
        Random rand = new java.util.Random();
        int x;
        x = rand.nextInt();
    }
}
```



```
import java.util.Random;
//もしくは import java.util.*;
public class RandomTest
{
    static public void main(String[] args)
    {
        Random rand = new Random();
        int x;
        x = rand.nextInt();
    }
}
```

インスタンスメンバ

- いままでクラスで書いていたフィールド，メソッドのこと

⇒newを使ってインスタンスを生成しないと使うことが出来ない

⇒ただし，メモリが許す限り何個でもインスタンスを作れる

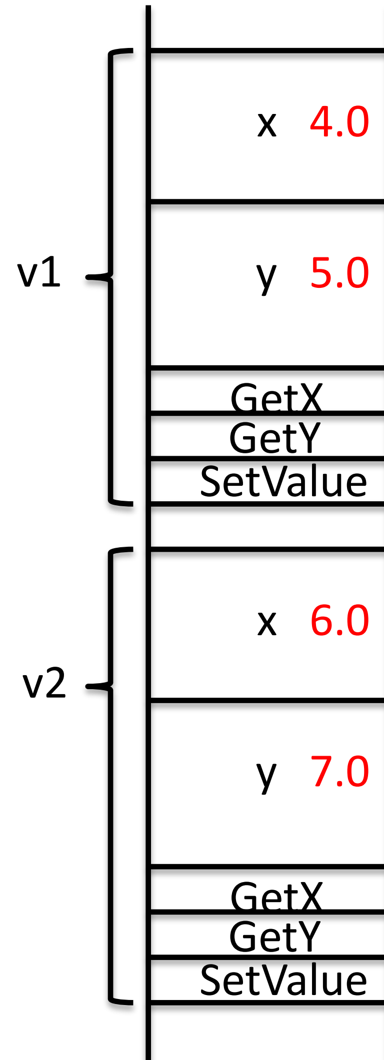
インスタンスメンバ

インスタンスフィールド

インスタンスメソッド

```
class Vector2D
{
   double x;
   double y;

   double GetX()
   {
      return x;
   }

  double GetY()
   {
      return y;
   }

   void SetValue(double newx, double newy)
   {
      x = newx;
      y = newy;
   }
}
```

```
Vector2D v1, v2;
v1 = new Vector2D();
v2 = new Vector2D();

double vx = 4.0;
double vy = 5.0;
v1.SetValue(vx, vy);

vx = 6.0;
vy = 7.0;
v2.SetValue(vx, vy);
```



静的メンバ

- フィールド，もしくはメソッドにstaticを付けると，静的メンバとなる

⇒newによってインスタンスを作らないで使用できる

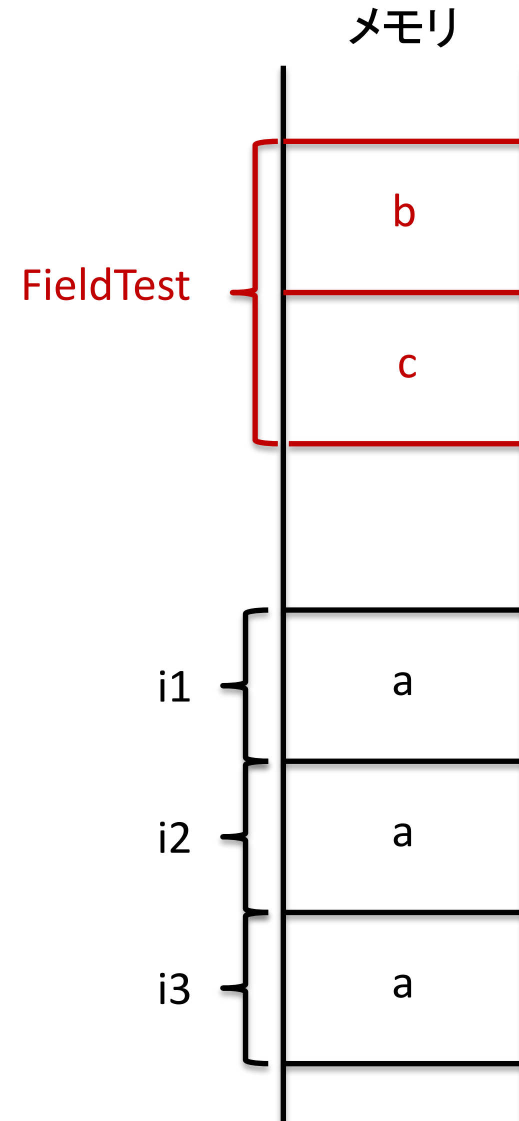
⇒ただし，プログラム実行中，1つしか存在しない

静的フィールド

静的フィールドは
初めてクラスが使用された際*に
メモリ上に1回だけ生成され，
プログラムが終了するまで残る

```
class FieldTest
{
    int a;
    static int b;
    static int c;
}
```

```
static public void main()
{
    FieldTest i1 = new FieldTest();
    FieldTest i2 = new FieldTest();
    FieldTest i3= new FieldTest();
}
```



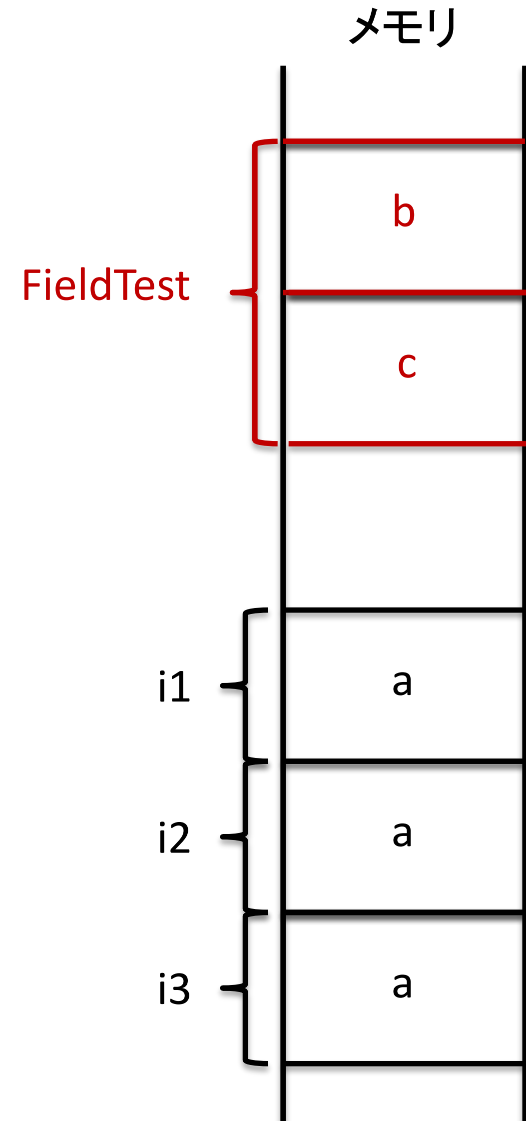
* http://www.y-adagio.com/public/standards/tr_javalang/8.doc.htm#37544

静的フィールド

静的フィールドの使用：
⇒クラス名.静的フィールド名

```
class FieldTest
{
    int a;
    static int b;
    static int c;
}
```

```
static public void main()
{
    FieldTest i1 = new FieldTest();
    FieldTest i2 = new FieldTest();
    FieldTest i3= new FieldTest();

    FieldTest.b = 3;
    FieldTest.c = 4;
    int x;
    x = FieldTest.b;
}
```

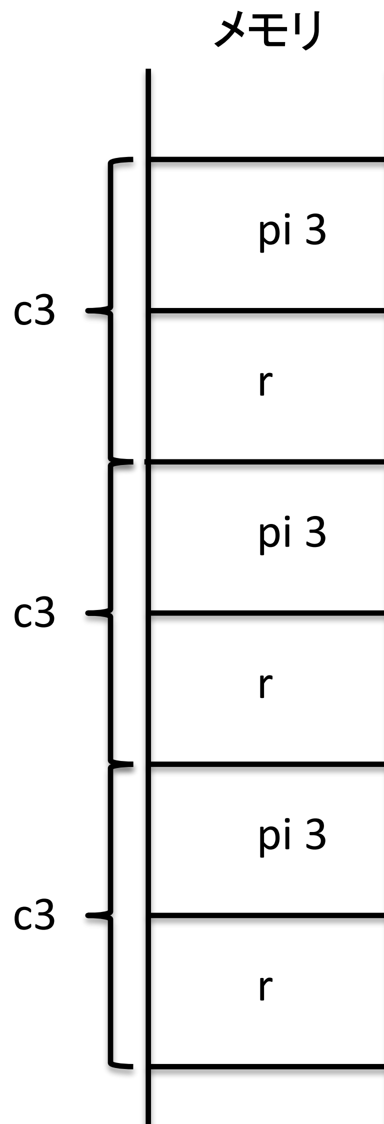


静的フィールド

静的フィールドの使い道：
クラス内で共有する値を静的フィールドにする

```
class Circle
{
   int pi = 3;
   int r;
}
```

```
static public void main()
{
   Circle c1 = new Circle();
   Circle c2 = new Circle();
   Circle c3= new Circle();
}
```



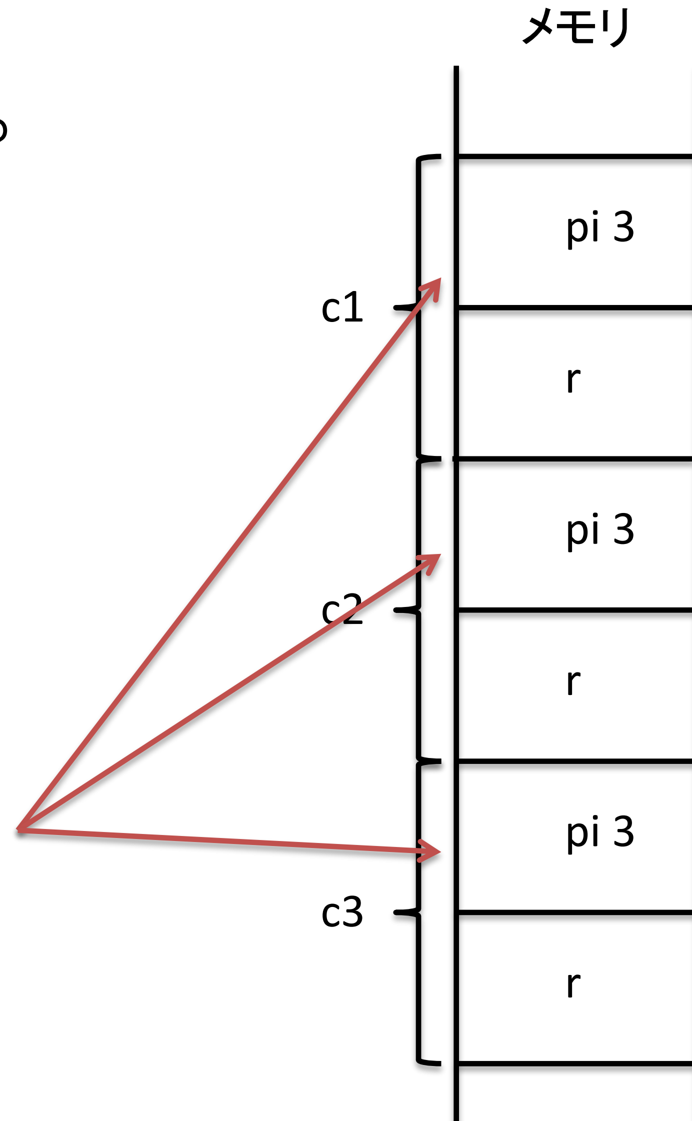
*

静的フィールド

静的フィールドの使い道：
クラス内で共有する値を静的フィールドにする

```
class Circle
{
   int pi = 3;
   int r;
}
```

```
static public void main()
{
   Circle c1 = new Circle();
   Circle c2 = new Circle();
   Circle c3= new Circle();
}
```



同じ値であることが分かっているのに
インスタンスフィールドとして
何個も作るのはメモリの無駄使い

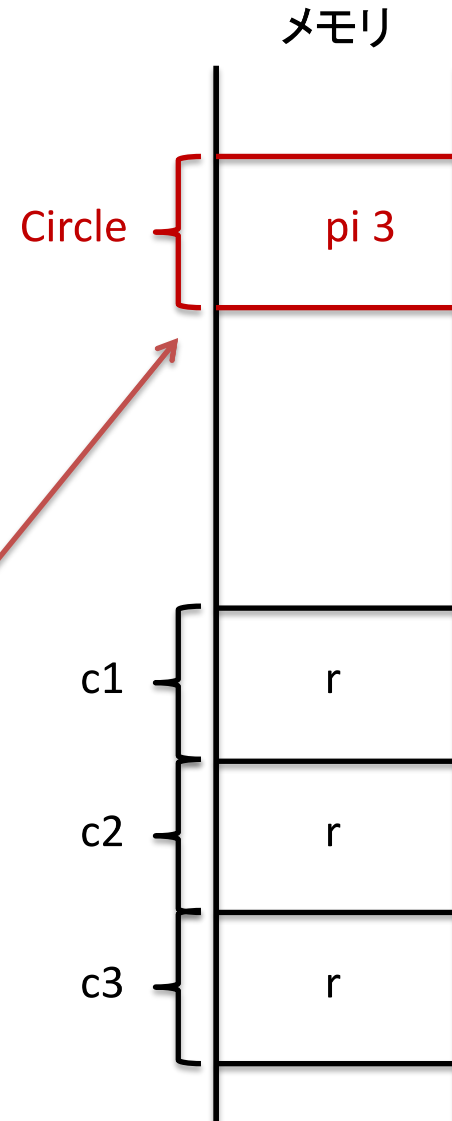
静的フィールド

静的フィールドの使い道：
クラス内で共有する値を静的フィールドにする

```
class Circle
{
   static int pi = 3;
   int r;
}
```

```
static public void main()
{
   Circle c1 = new Circle();
   Circle c2 = new Circle();
   Circle c3= new Circle();
}
```

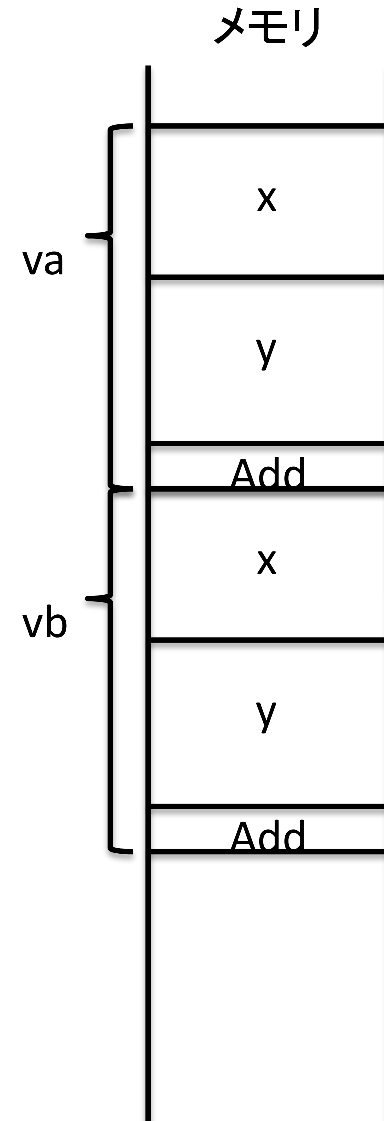
静的フィールドとすることで，1つのみ作られ
メモリ使用量を抑えることが出来る



静的メソッド

```
class Vector2D
{
   double x;
   double y;

   Vector2D Add(Vector2D v1, vector2D v2)
   {
      Vector2D v = new Vector2D();
      v.x = v1.x + v2.x;
      v.y = v1.y + v2.y;
      return v;
   }
}
```

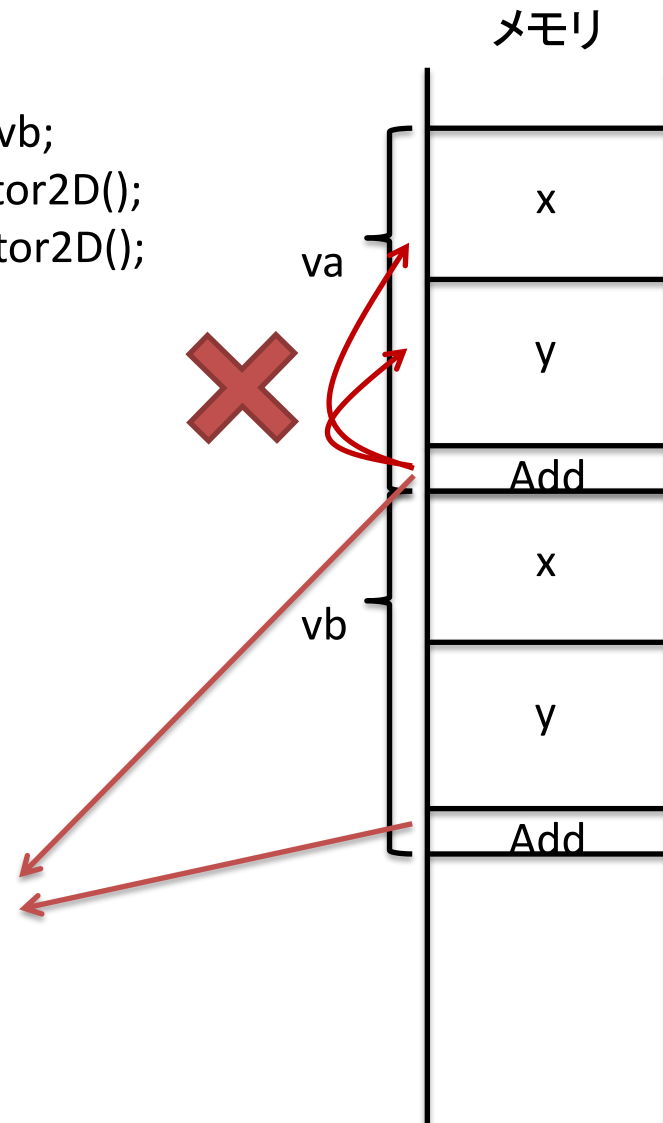
```
Vector2D va, vb;
va = new Vector2D();
vb = new Vector2D();
```



静的メソッド

```
class Vector2D
{
    double x;
    double y;

    Vector2D Add(Vector2D v1, vector2D v2)
    {
        Vector2D v = new Vector2D();
        v.x = v1.x + v2.x;
        v.y = v1.y + v2.y;
        return v;
    }
}
```

```
Vector2D va, vb;
va = new Vector2D();
vb = new Vector2D();
```



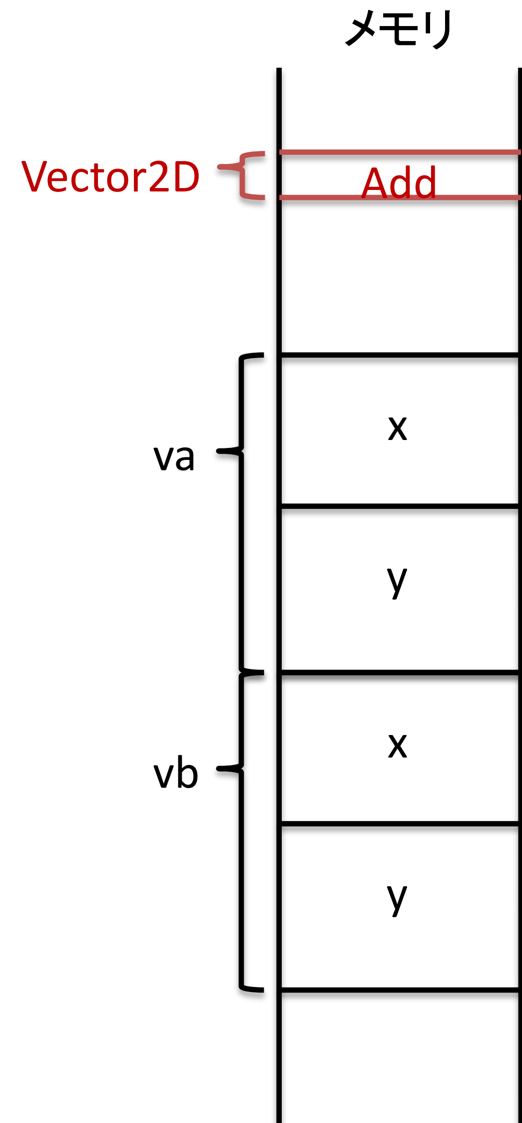
⇒よく見ると，
上記のAddメソッドは，同じインスタンス内のフィールドを全く使用していない

静的メソッドの使い道：よって，このメソッドは，
インスタンスメソッドとして
何個も作る必要がない
⇒静的メソッドとする

静的メソッド

```
class Vector2D
{
    double x;
    double y;

    static Vector2D Add(Vector2D v1, vector2D v2)
    {
        Vector2D v = new Vector2D();
        v.x = v1.x + v2.x;
        v.y = v1.y + v2.y;
        return v;
    }
}
```

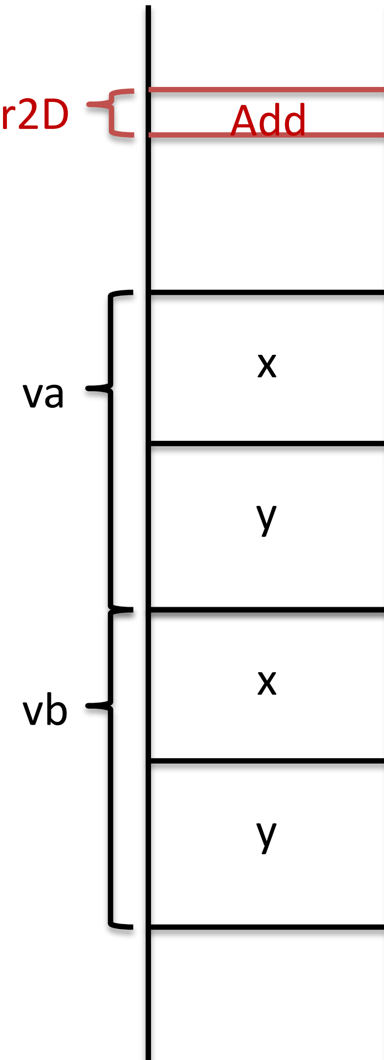
```
Vector2D va, vb;
va = new Vector2D();
vb = new Vector2D();
```



静的メソッド

```
class Vector2D
{
    double x;
    double y;

    static Vector2D Add(Vector2D v1, vector2D v2)
    {
        Vector2D v = new Vector2D();
        v.x = v1.x + v2.x;
        v.y = v1.y + v2.y;
        return v;
    }
}
```

```
Vector2D va, vb;
va = new Vector2D();
vb = new Vector2D();

Vector2D vc;
vc = Vector2D.Add(va, vb);
```



静的メソッドの使い方：
クラス名.メソッド名(引数1, 引数2, ...);

静的メソッド

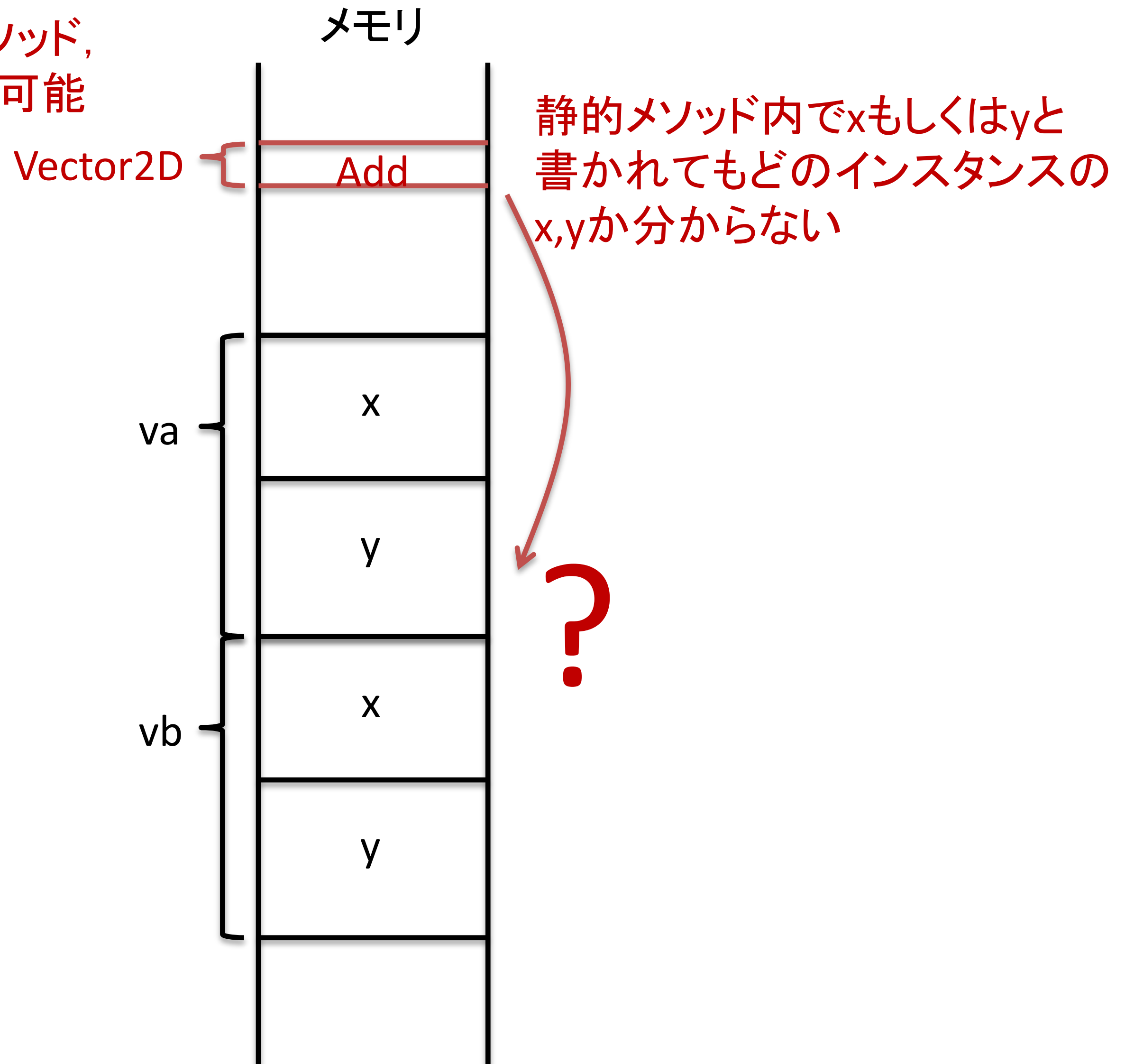
```
class Vector2D
{
    double x;
    double y;

    static double getLength(Vector2D v)
    {
        double l = Math.sqrt(v.x * v.x + v.y *v.y);
        return l;
    }

    void Normalize()
    {
        double l = getLength(this);
        this.x /= l;
        this.y /= l;
    }
}
```

インスタンスメソッド内で静的メソッド，
また静的フィールドを使うことは可能
⇒逆は不可能

静的メソッド

インスタンスメソッド内で静的メソッド,
また静的フィールドを使うことは可能
⇒逆は不可能

⇒静的メソッドないで,
インスタンスフィールド,
インスタンスメソッドを
使うことはできない



静的メンバ

- なにか新しいデータのクラスを作ったとき（例：3次元ベクトル）フィールド（x, y, z）はインスタンスフィールド，

  データを計算するメソッドは（例：足し算，引き算，内積，外積，…）静的メソッドとする

  という使い方が多い