Java2セキュリティ，クラスローダー，ベリファイア

2002年7月14日

2003年7月7日改定

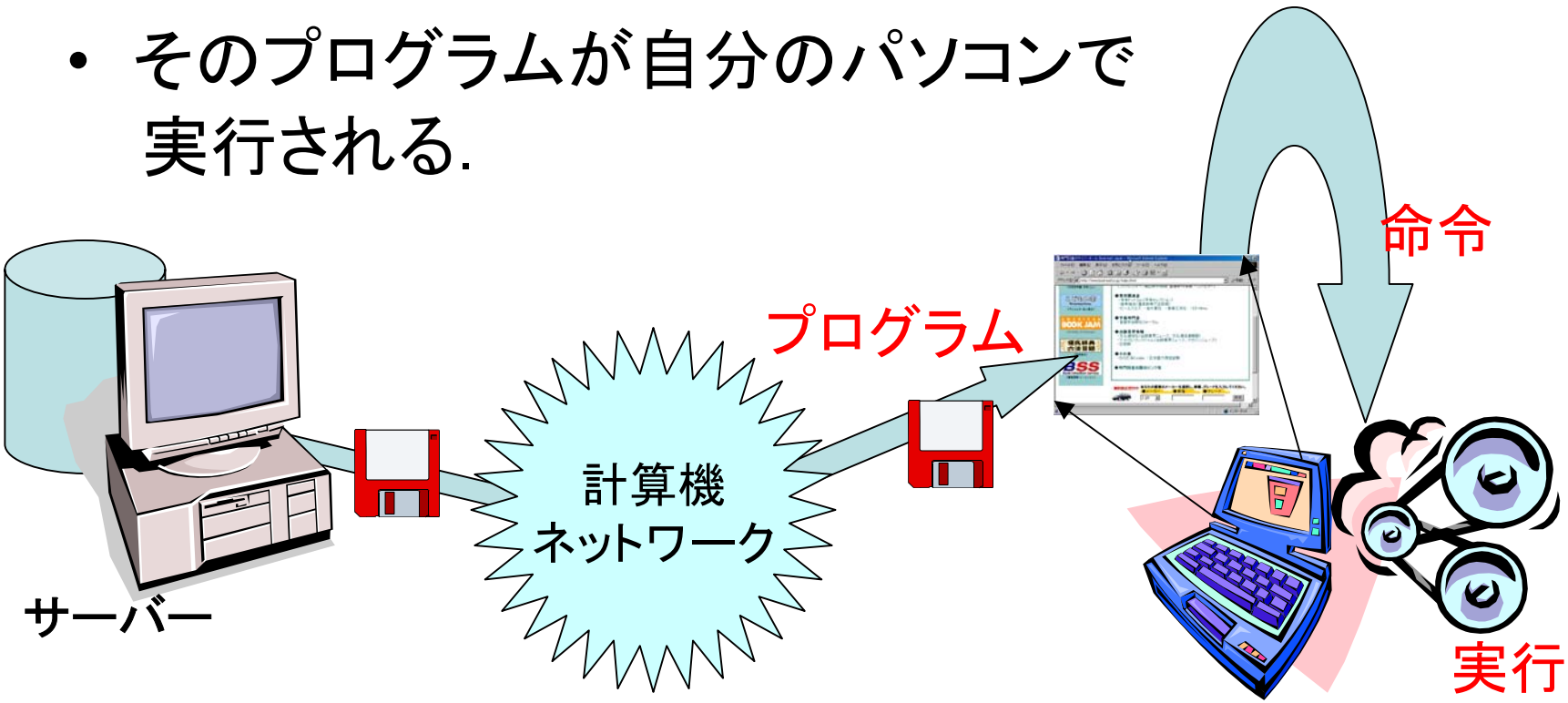
海谷 治彦

目次

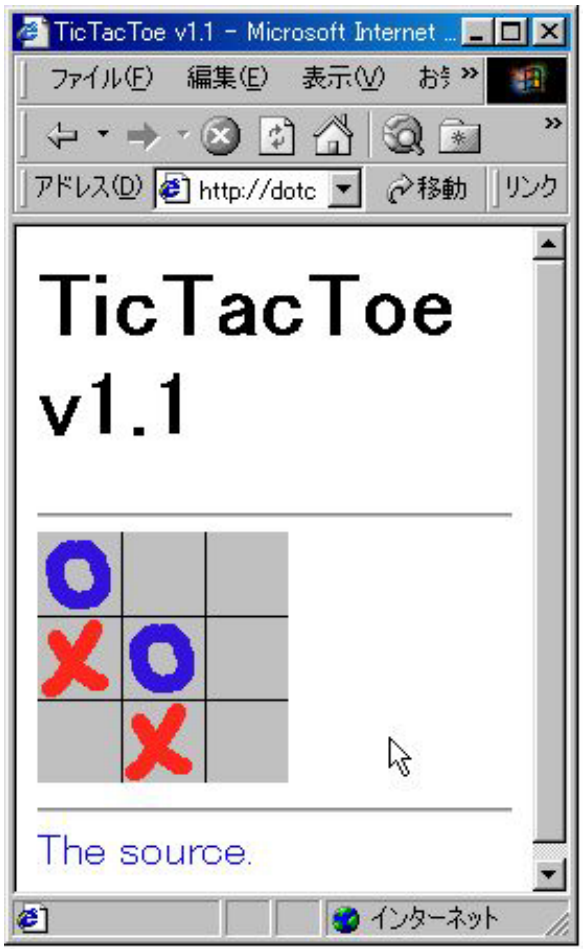
- モバイルコード技術について
- Java2セキュリティの概要
- クラスローダーの役割
- オンデマンドなロード再び
- バイトコード検証系 (バイトコードベリファイア)

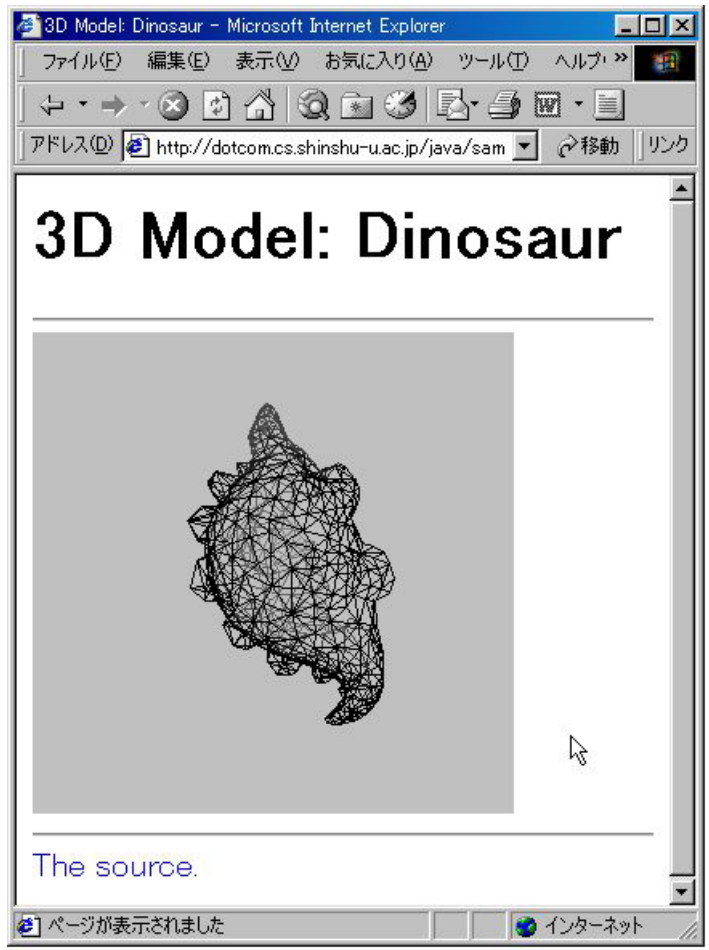
モバイル・コード技術

- ホームページ(データ)と同様に，
  プログラムがダウンロードされる．
- そのプログラムが自分のパソコンで
  実行される．



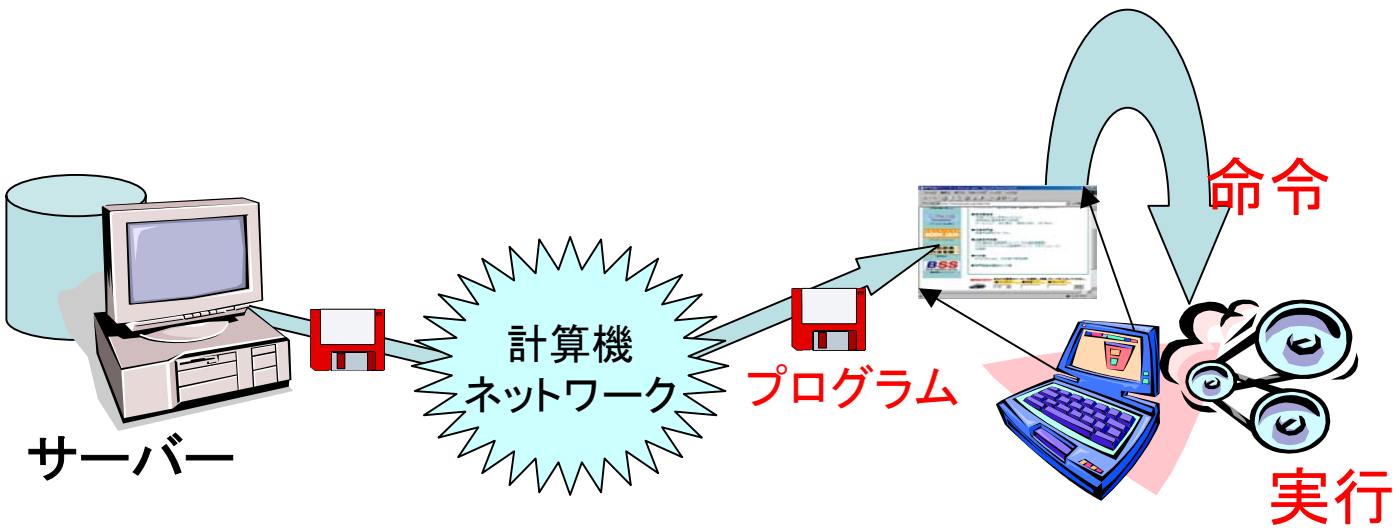
モバイルコードの例: Applet





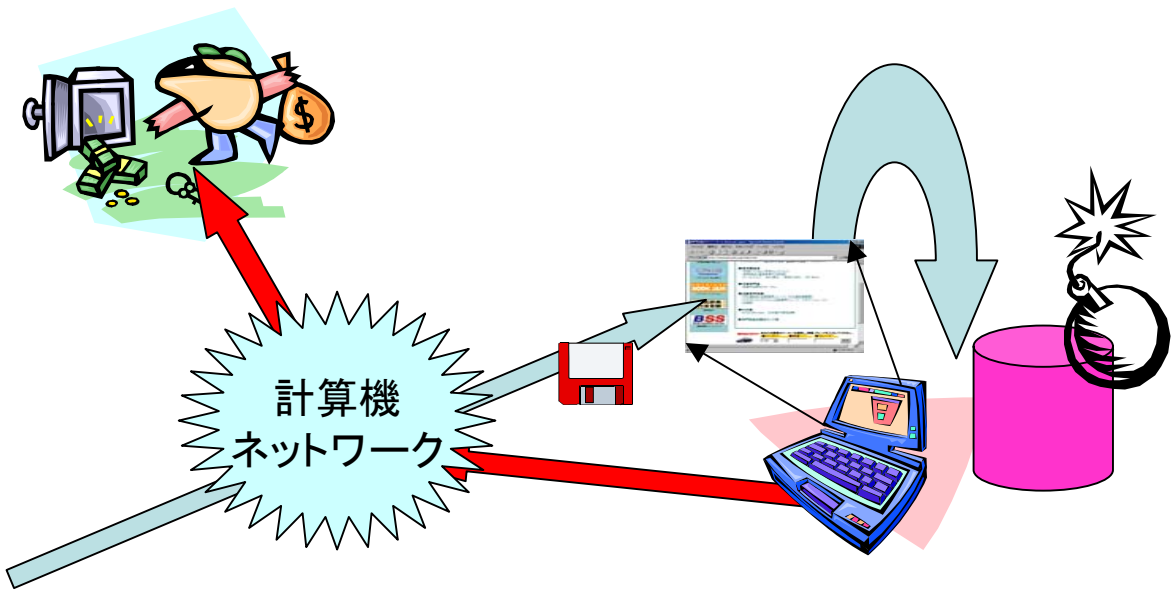
モバイルコードの利点

- ネットワーク上のデータの交通量の削減
- レスポンスの向上
- 一々，ソフトをインストールしないでも，他人の作ったソフトを利用できる.



モバイルコードの欠点

- 他人のプログラムに悪意があった場合，
  - 自分のデータが破壊されるかも？
  - 自分のデータが盗まれるかも？
  - 自分のパソコンが勝手に悪戯するかも？



コードを送り込むvsデータを送り込む

- コード(プログラム)を送り込む場合，
  - 通信量が少なくて済む
  - 送り込まれたマシンは事実上，コードに支配される.
  - 処理は送り込まれたマシンで行われる.
- データを送り込む場合，
  - 通信量は多い場合が多い
  - 支配は送り込まれたマシンにある.
  - 処理は送り込む側.

データを送り込む: 悪戯FAX

悪戯FAXでの通信

処理

データ

効果的？な悪戯FAXもどき

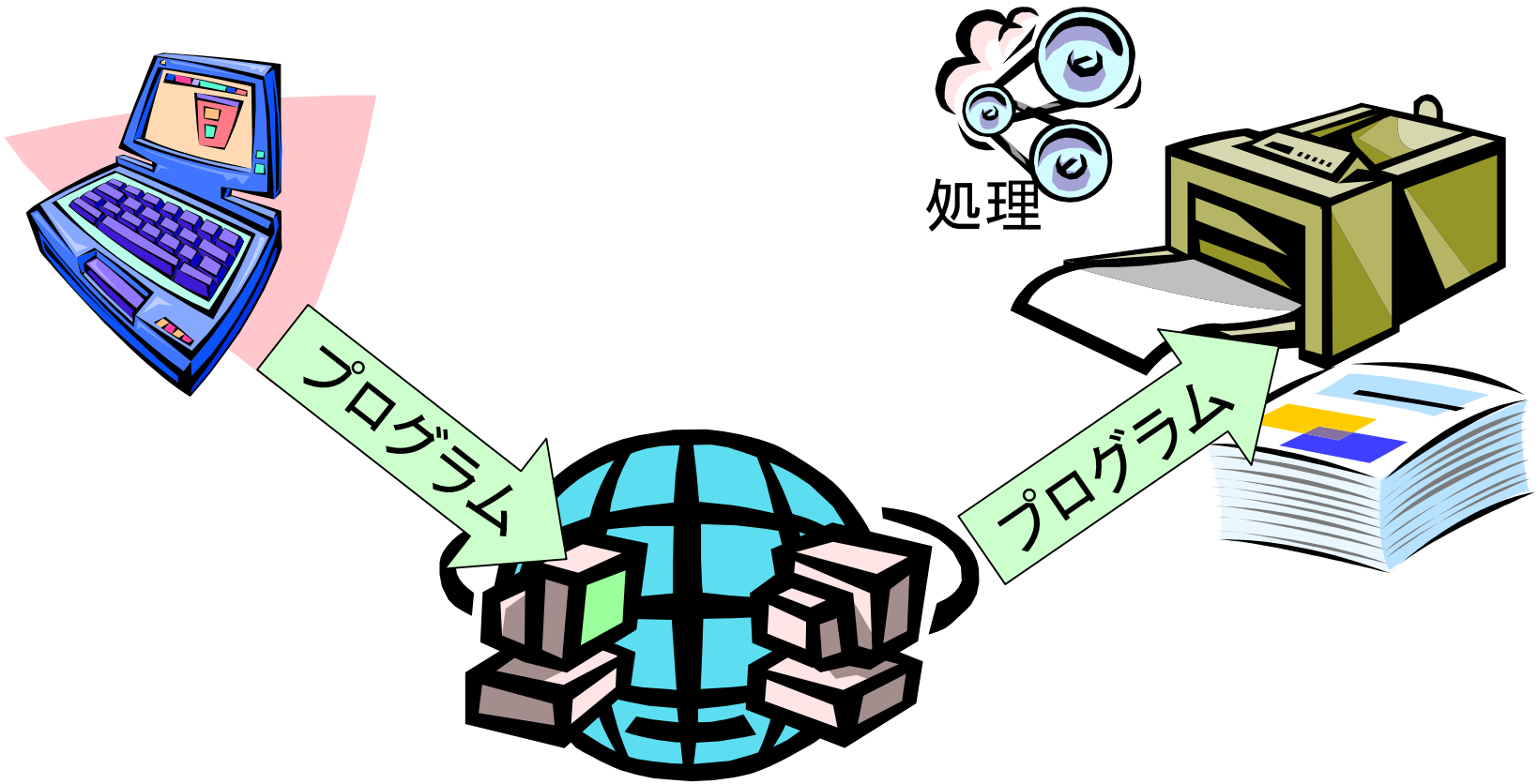
/Times-Roman findfont 100 scalefont setfont
{100 100 moveto (afo) show showpage} loop

100バイト以下

- ページ記述言語ポストスクリプトによるプリンタ攻撃.
- 要は「afo」の文字をプリンタに出しつづける.
- もし，PSプリンタが外部ネットワークに直結されていた場合，このコードに攻撃される可能性あり.

- 試したわけじゃぁないから，上手くいかないかも(笑)

PS悪戯での通信と処理



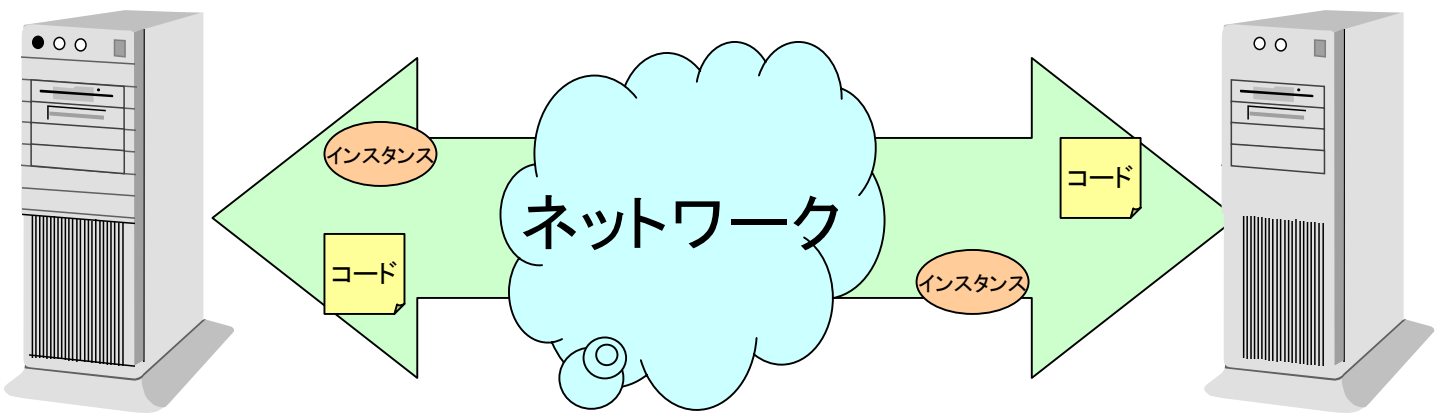
モバイルコード利用の留意点

- 他所からダウンロードしたコードに，ある程度の制限をつけないと危ない．
- ダウンロードしたコードの実行は，コード製作者に自分のPCを勝手にいじらせるのと本質的には同じ．

- マイコンといえど，ネットワークに接続され，外部からコードをロードする例が少なくなくなった．

Javaとネットワーク指向

- モバイルコード: ネットワーク越しにクラスを飛ばせる.
- シリアライズ: ネットワーク越しに飛ばせるインスタンスもある.

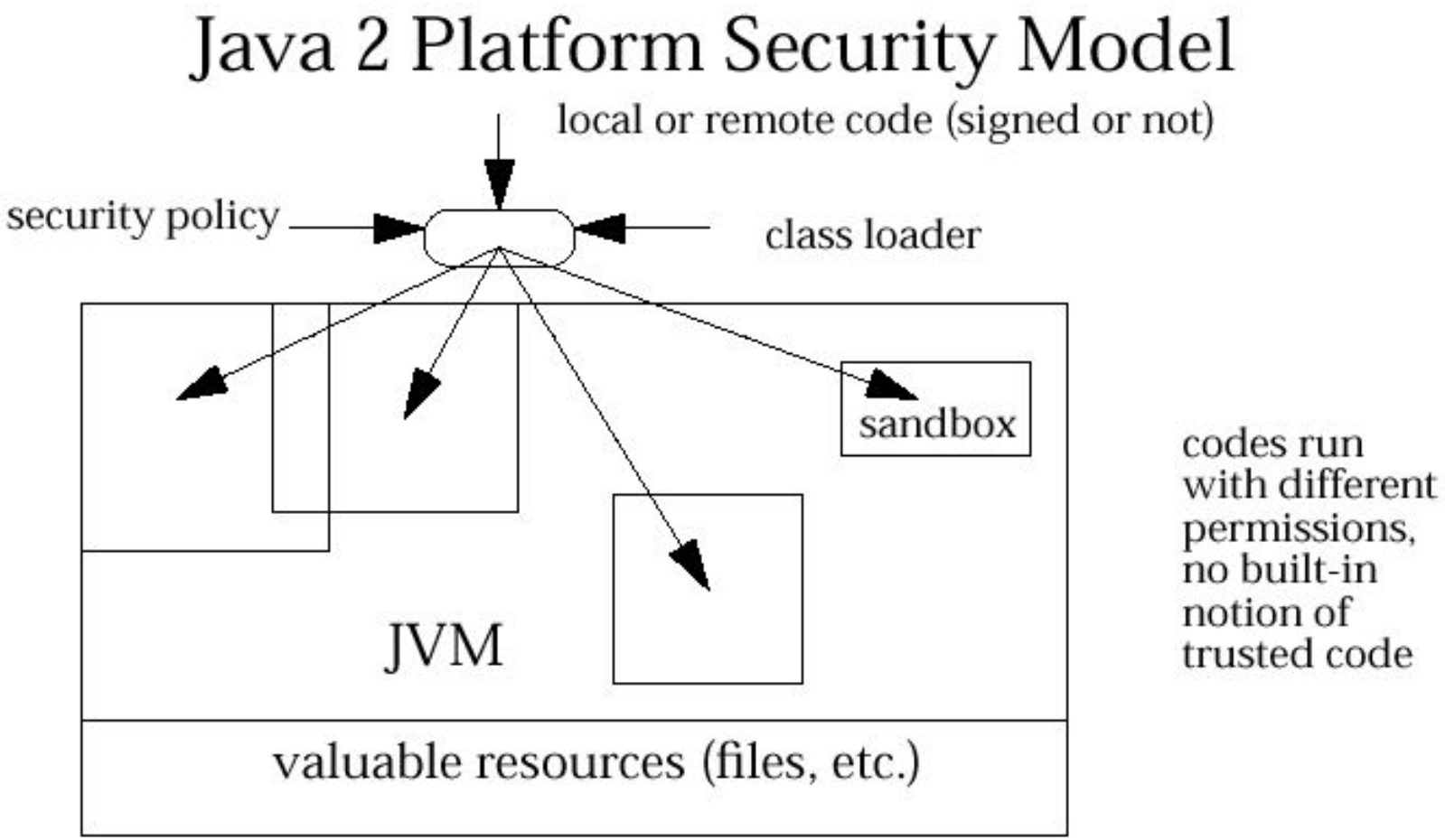


Javaセキュリティの方針

- JVMにロードされたコードの，
  - ロード元
  - 著名(サイン)の有無

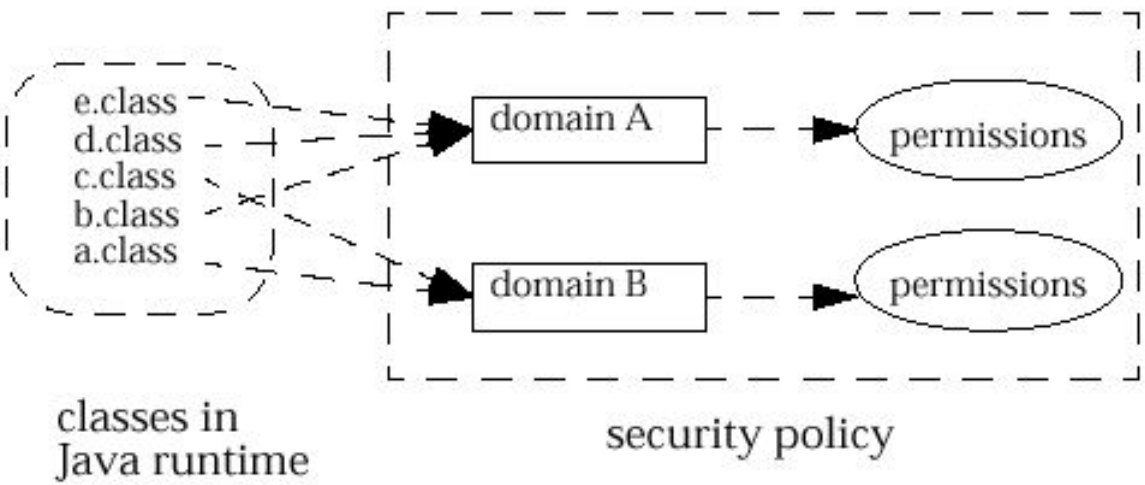
  によって，コードの実行に制限を加える．
- 例
  - 「www.cs.shinshu-u.ac.jp から落としたコードは信用して，なんでも実行させる．」
  - 「www.computer.org から落としたコードはファイル参照のみ許可する．」

Java2 砂箱モデル


Java 2 Platform Security Model
local or remote code (signed or not)
security policy
class loader
sandbox
JVM
valuable resources (files, etc.)
codes run with different permissions, no built-in notion of trusted code

Class Domain Permissions

•Permission: ファイルを読んで良い，JVMを停止して良いなどの許可.
•Domain: クラスの集まり
•policy: domainとpermissionの対応を書いたもの



デフォルトポリシィは $JRE/jre/lib/security/java.policy にあり

Policy記述の例

ココから拾ってきて，かつ，コイツがサインしたクラスは，

```
grant
      codeBase "http://java.sun.com/*",
      signedBy "Li" {
            permission java.io.FilePermission
                  "/tmp/*", "read";
            permission java.io.SocketPermission
                  "*", "connect";
};
```

/tmp/の下は読んでよし.

ネットワークソケットは自由に接続可.

それ以外はダメ.

Code-centric VS Use-centric

- ここまでの話は，「どのコードを実行したか？」という観点から，アクセス制限の議論をした.
  - Code-centric と呼ばれている見方.
- 加えて，「誰がコードを実行したか？」という観点から，アクセス制限を議論する場合もある.
  - これを，User-centric という.
- Javaセキュリティは後者のUser-centricもサポートしている.
  - Java Authentication and Authorization Service (JAAS)

User-Centricの必要性

- コードではなく，ユーザーが移動する場合の考慮.
  - 公共端末(ネットカフェ？)でプログラムを実行する等.
- JVMをマルチユーザーOSのように使うことができる.
  - 従来は，「誰が動かしているの？」という情報を利用しずらかった.
- 詳細は，ページからリンクされた情報参照.

オンデマンドロード＆動的リンク

- クラスは実際に使われる直前まで**JVM**に読み込まれない.
- よって，プログラムの一部(クラス)を付けたしながら実行できる – 増殖！
- 極端な話，プログラム自身が自分の一部を生成しながら実行を進められる – 自己増殖！

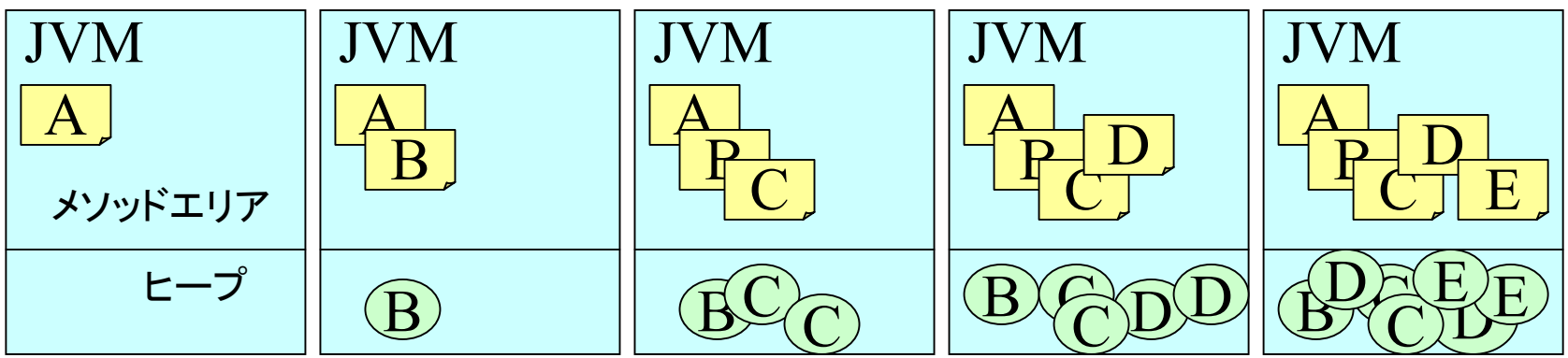
普通の増殖 (ロード)

```
class A{
static ... main(...){
  .... new B()....
  }
}
```

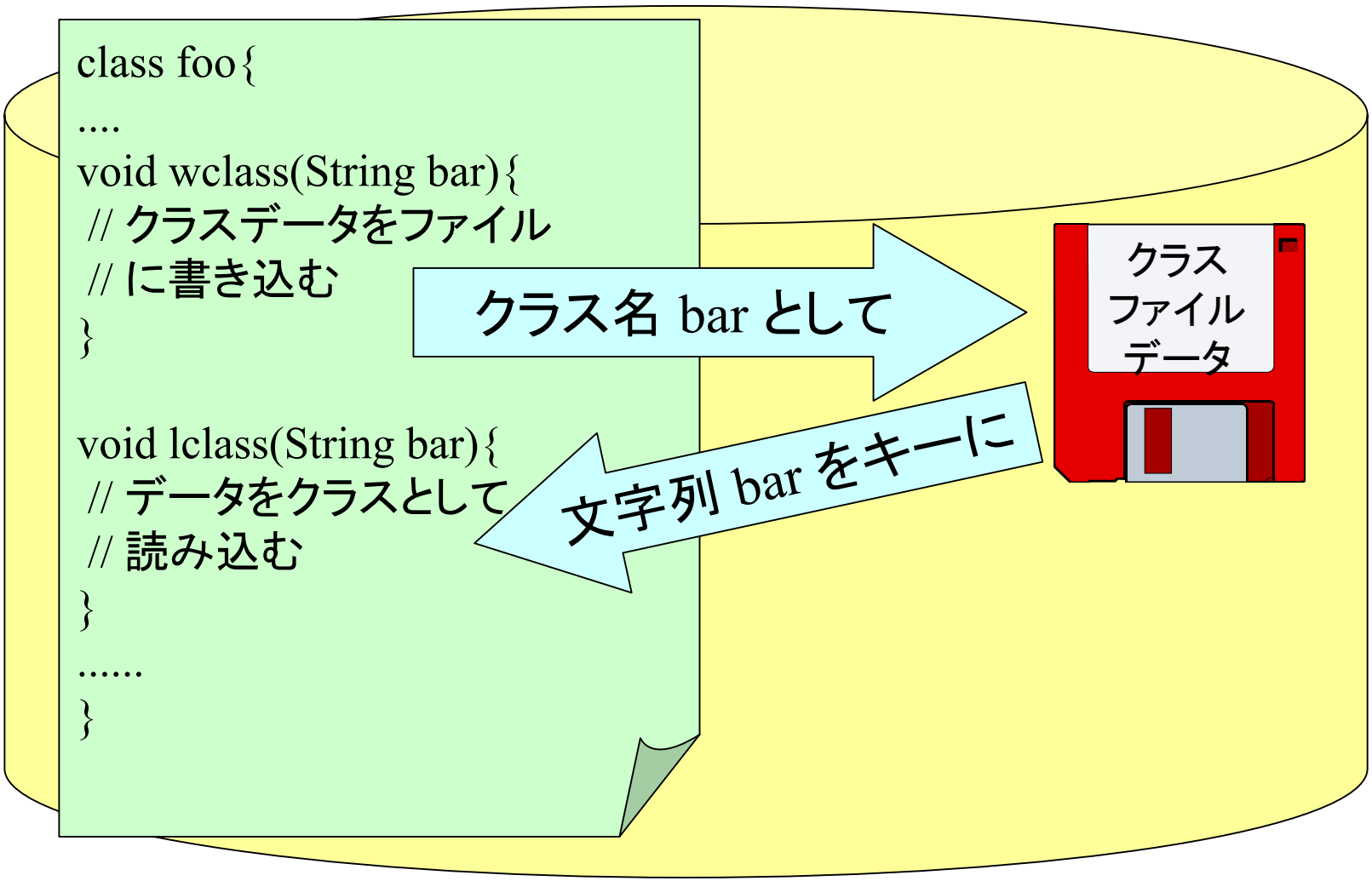
```
class B{
  .... new C()....
  .... new C()....
}
```

```
class C{
  .... new D()....
}
```

```
class D{
  .... new E()....
}
```



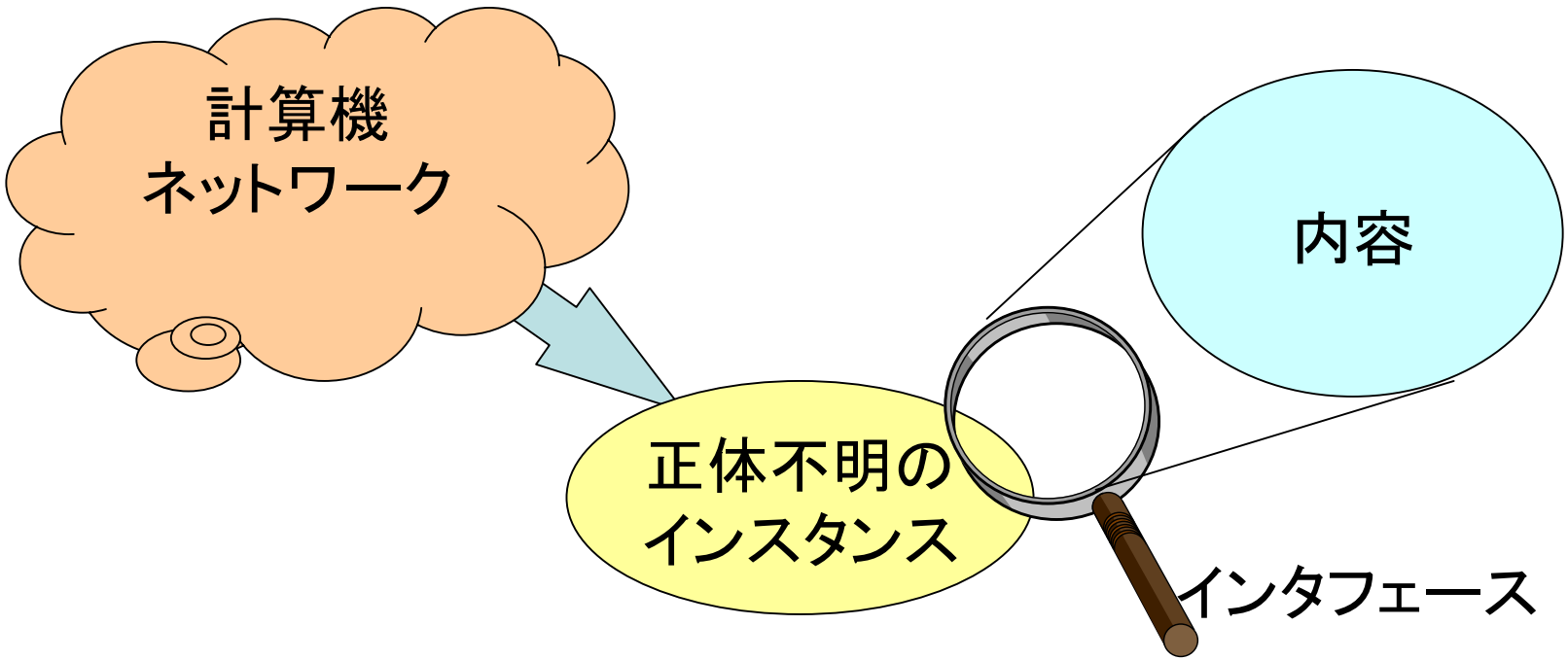
自己増殖の例


class foo{
....
void wclass(String bar){
// クラスデータをファイル
// に書き込む
}
void lclass(String bar){
// データをクラスとして
// 読み込む
}
.....
}
クラス名 bar として
文字列 bar をキーに
クラス
ファイル
データ

自己増殖に使える技術

- インタフェースの実装
  - Runnable などよく知られたインタフェースが実装されていれば，正体不明のクラスのインスタンスにも，メソッドを適用できる.
- リフレクション(自己反映計算)
  - 後述

Javaとインタフェース



正体不明のインスタンスも実装しているインタフェースが公開されていれば，アクセス可能となる.

Reflection

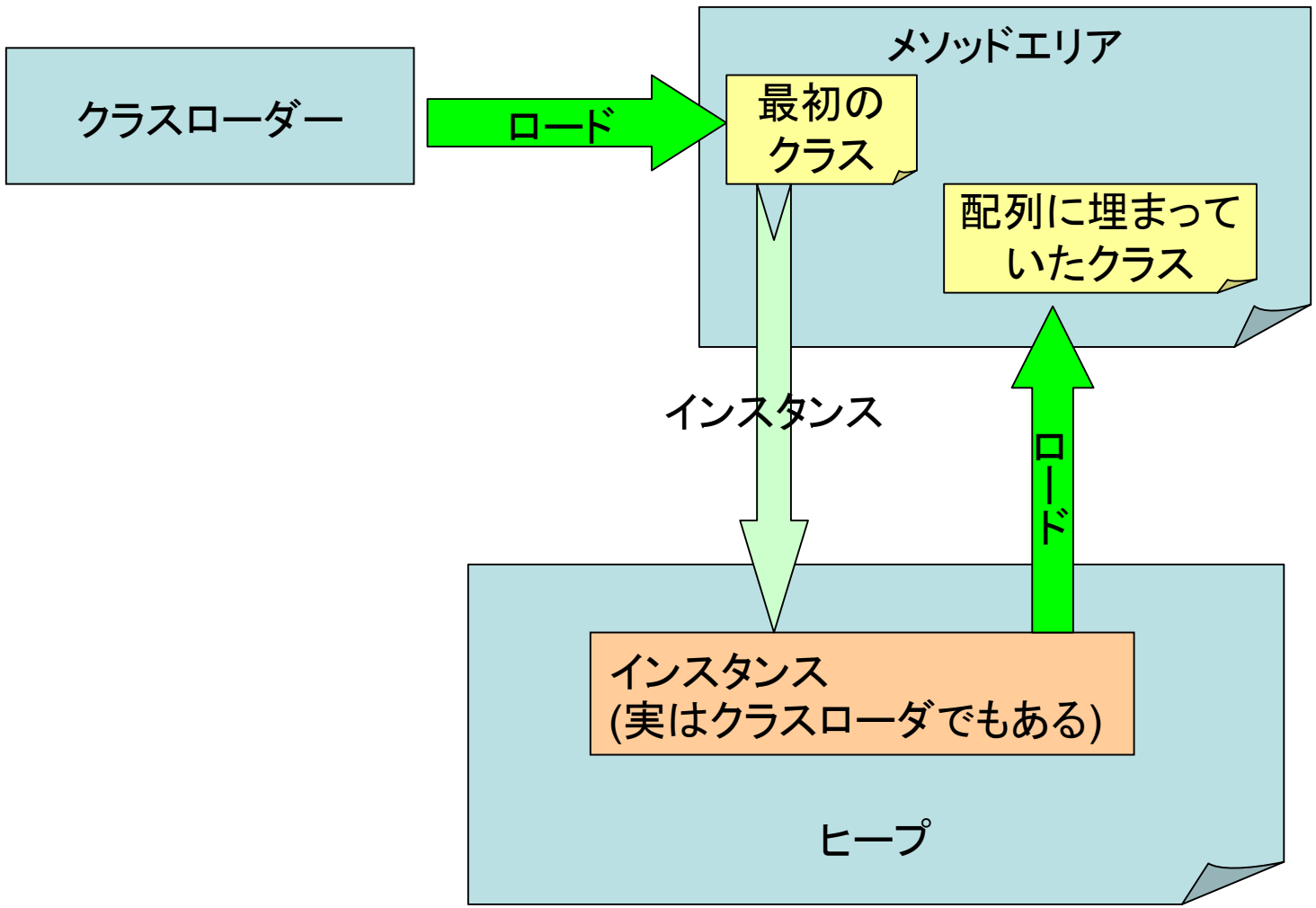
- もともと存在しないクラスは，言語要素として呼び出すことはできない．
  - 例: aclass.amethod() など．
- しかし，Javaでは，文字列を使って，クラスやメソッドを操作できる．
  - 例: c=loadclass(“aclass”);
  - c.getMethod(“amethod”).invoke();
- これによって未知のクラス，メソッドの操作が可能となる．

6つの例題

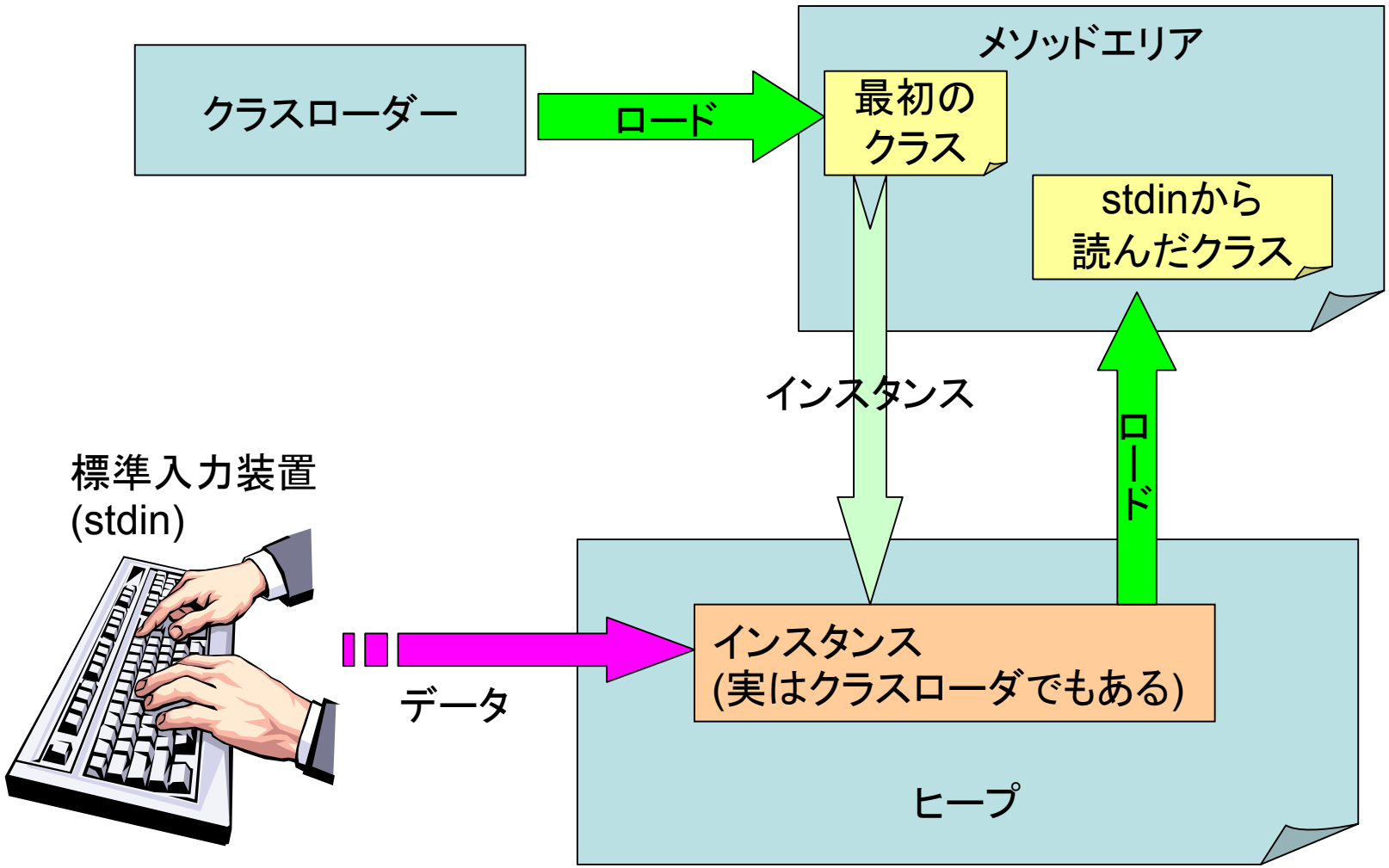
- 配列データをプログラムに組み込む
- 標準入力から読んだデータをプログラムに組み込む.
- ネットワークからダウンロードしたデータをプログラムに組み込む.

- それぞれ，インタフェースとリフレクションを使った例を提示.

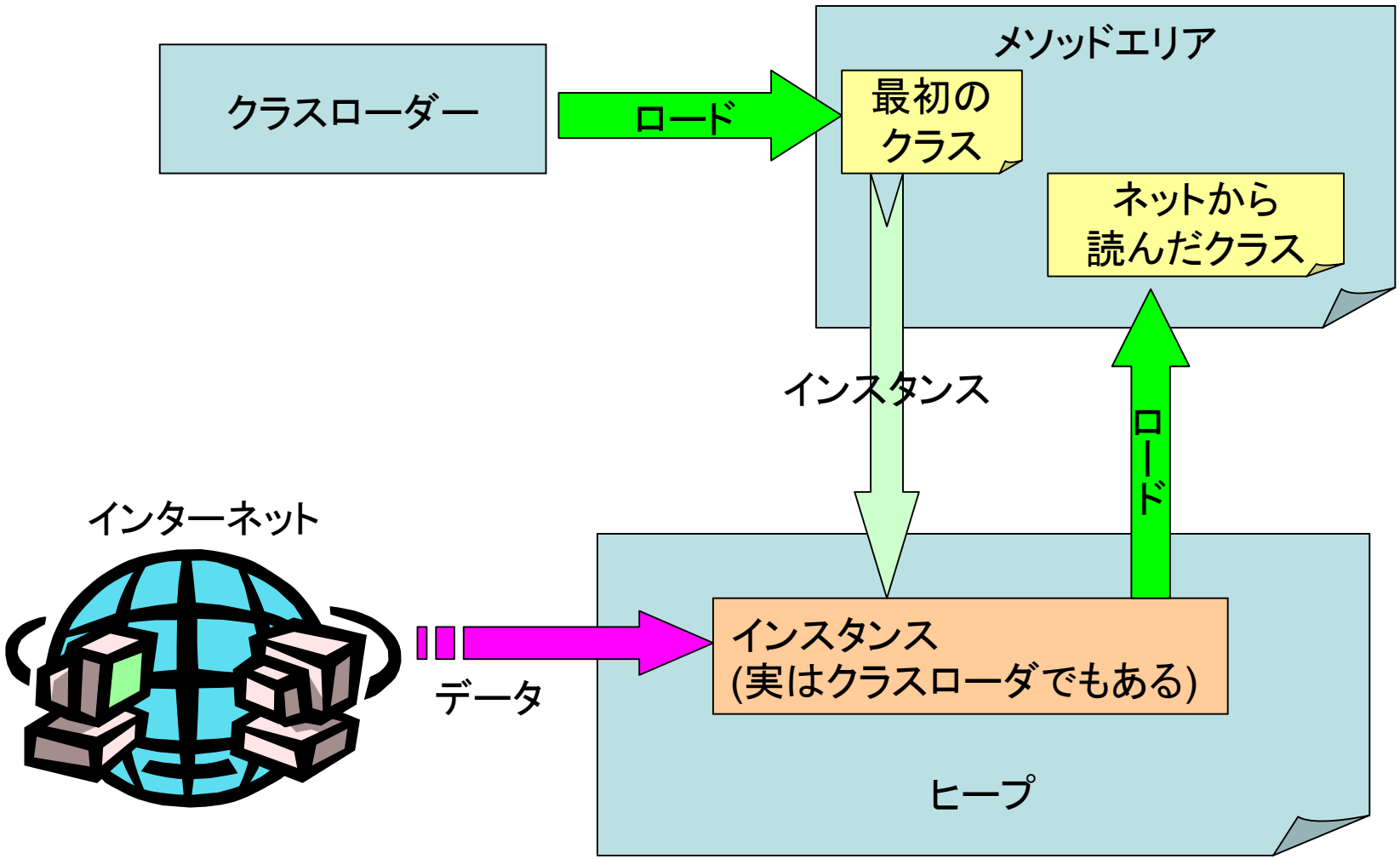
配列を使った場合


クラスローダー
ロード
メソッドエリア
最初の
クラス
配列に埋まって
いたクラス
インスタンス
ロード
インスタンス
(実はクラスローダでもある)
ヒープ

標準入力を使った場合


クラスローダー
ロード
メソッドエリア
最初の
クラス
stdinから
読んだクラス
インスタンス
ロード
標準入力装置
(stdin)
データ
インスタンス
(実はクラスローダでもある)
ヒープ

ネットワークを使った場合



検証・ベリファイ・Verify

- Verifyとは「正しいことを確かめる」こと.
- ソフトウェアの世界では，検証と訳される.
- プログラム(クラスファイル等)が正しいとは，なんなのか？それをどうやって確かめるのかを以下で議論する.

クラスファイルベリファイア

- クラスファイルが Java Class File Format に準拠しているかどうかをチェック
- クラスファイル内の一貫性のチェック
  - たとえば，存在しないCP参照が無いことを確認.
- 他のクラスファイルとの一貫性
- バイトコード検証

教科書 p.122～124

正しいバイトコードとは？

- 実行時に，オペランドスタックのオーバー/アンダーフローを起こしたりしない．
- スタックの値の出し入れの際に型の不一致が起こっていない．
  - 例: 以下の命令列は明らかに型が不一致．

```
iconst_0
astore_1
```

バイトコード検証系

- インストラクションコード列のすべての分岐を調べ，どのような条件分岐のパスでプログラムが実行されたとしても，決して致命的なエラーを引き起こさないことを検証するシステム.

(教科書 p.118より)

- Javaは，この検証を実行前(ロード時)に行っている.

前述のことができるのか？！

- 分岐の組み合わせなど，無数にあり，それを実行前に事前にチェックするなど，到底無理っぽい．
- JVMでは，以下の2つのポイントでこれを乗り切っている．
  - 値ではなく，型の一致をチェックしている．
  - 型の組み合わせが無数にあるようなプログラムの実行を許していない．(文法的に正しくても)

型の一致とは？1/2

例題のプログラム

```
int s=0;
      for(int i=0; i<4; i++) s+=i;
```

```
 .limit stack 2
    .limit locals 3
    iconst_0
    istore_1
    iconst_0
    istore_2
    goto Label2
Label1:
    iload_1
    iload_2
    iadd
    istore_1
    iinc 2 1
Label2:
    iload_2
    iconst_3
    if_icmplt Label1
    return
```

型の一致とは？ 2/2

```
iconst_0
istore_1
iconst_0
istore_2
goto Label2

local=[, 0, 0] stack=[ ,  ]
Label2:
iload_2
iconst_3
local=[, 0, 0] stack=[0, 3]
if_icmplt Label1
```

```
local=[, 0, 0] stack=[ ,  ]
Label1:
iload_1
iload_2
iadd
istore_1
iinc 2 1
local=[, 0, 1] stack=[ ,  ]
Label2:
iload_2
iconst_3
local=[, 0, 1] stack=[1, 3]
if_icmplt Label1
```

```
local=[, 0, 1] stack=[ ,  ]
Label1:
iload_1
iload_2
iadd
istore_1
iinc 2 1
local=[, 1, 2] stack=[ ,  ]
Label2:
iload_2
iconst_3
local=[, 1, 2] stack=[2, 3]
if_icmplt Label1
```

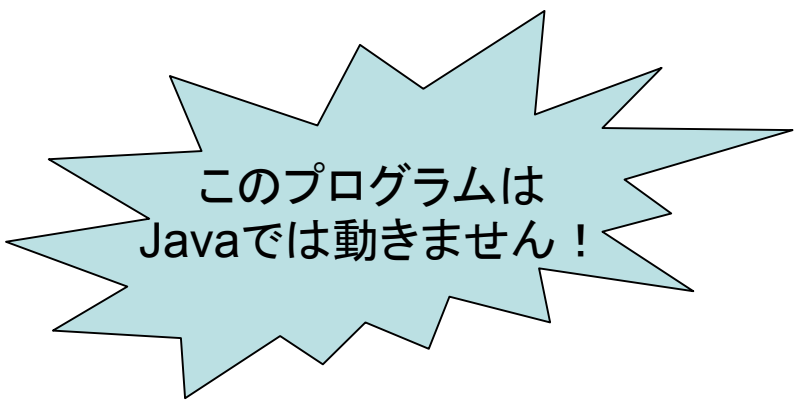
```
local=[, 1, 2] stack=[ ,  ]
Label1:
iload_1
iload_2
iadd
istore_1
iinc 2 1
local=[, 3, 3] stack=[ ,  ]
Label2:
iload_2
iconst_3
local=[, 3, 3] stack=[3, 3]
if_icmplt Label1

local=[, 3, 3] stack=[ ,  ]
return
```

- Label1, Label2 それぞれの直前のスタック，ローカルの値は違えども，つんである値の型は一致している．(この場合，Iのみだけど)
- こういった場合のことを型の一致と呼ぶ．

型の組み合わせが通る毎に異なる例

```
 iconst_3
    istore_1
Label1:
    aconst_null
    iinc 1 -1
    iload_1
    ifne Label1
```

- 教科書 p.119より
- 短にnullを3つ積むプログラム
- 左記のように，ラベル前で，毎回スタック列の型が異なる.




```
iconst_3
local=[this,  ] stack=[3, , , ]
istore_1
local=[this, 3] stack=[, , , ]
aconst_null
local=[this, 3] stack=[null, , , ]
iinc 1 -1
local=[this, 2] stack=[null, , , ]
iload_1
local=[this, 2] stack=[null, 2, , ] 参照, int

ifne Label1
local=[this, 2] stack=[null, , , ]
aconst_null
local=[this, 2] stack=[null, null, , ]
iinc 1 -1
local=[this, 1] stack=[null, null, , ]
iload_1
local=[this, 1] stack=[null, null, 1, ] 参照, 参照, int

ifne Label1
local=[this, 1] stack=[null, null, , ]
aconst_null
local=[this, 1] stack=[null, null, null, ]
iinc 1 -1
local=[this, 0] stack=[null, null, null, ]
iload_1
local=[this, 0] stack=[null, null, null, 0] 参照, 参照, 参照, int

ifne Label1
local=[this, 0] stack=[null, null, null, ]
```

おしまい