2年目プログラマのための Java再入門講座

基本文法からWeb-DBアプリ開発まで基礎知識を総復習！

WINGSプロジェクト代表

講師：山田 祥寛

（やまだ よしひろ）

主催：日経BP社 日経ソフトウエア　日時：2016年3月4日（金）10:00～16:30　場所：富士ソフト アキバプラザ 6Fセミナールーム

第1部　座学でウォーミングアップ

Javaの基本文法のポイントを復習

10:00～11:00
＜60分＞

10個のポイントを押さえて Javaの基礎を復習しよう

ポイント1　プリミティブ型と参照型の違いを理解する

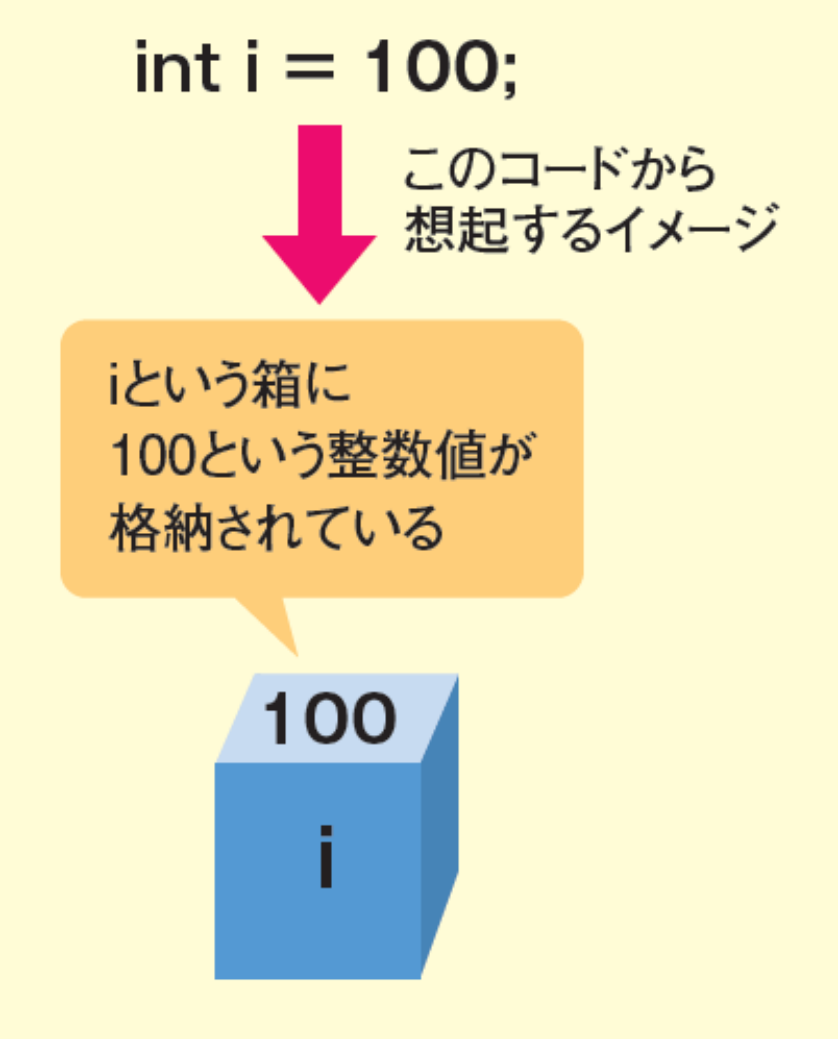
- プリミティブ型（基本型）→ 次の8種類だけ

byte、short、int、long、float、double、char、boolean

- 参照型（クラス／インターフェイス型、配列型）
  → String、ArrayList など無数にある

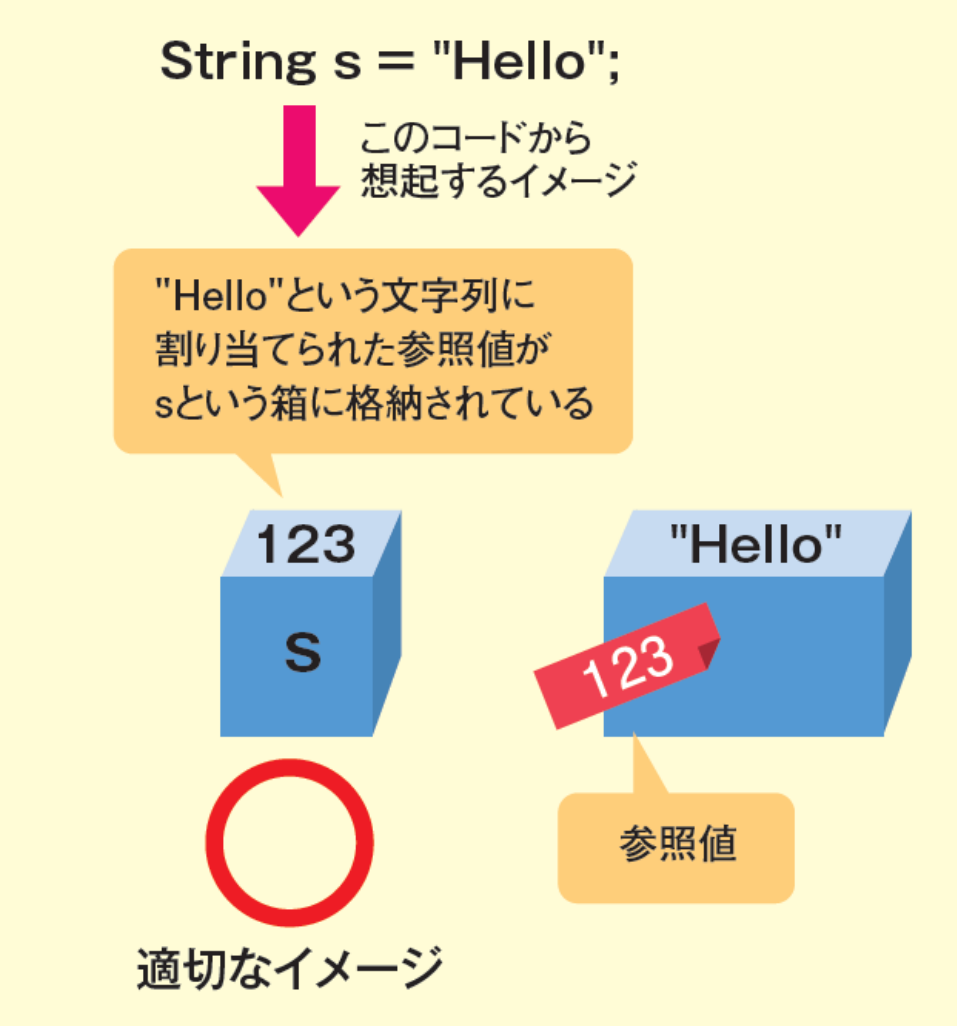
両者の違いは、値を格納する方法

プリミティブ型変数のイメージ



参照型変数の不適切なイメージ

参照型変数の適切なイメージ



基本型／参照型で注意すべき状況

- 値を代入／コピーする時
- 値同士を比較する時

ポイント2　Stringの扱いをマスターしよう

- **クラス型。でもよく使うので、ちょっと別格**

- **"文字列リテラル"でインスタンスを生成できる（因みに、シングルクォートはcharリテラル）**

例　String s = "Hello";

- **文字列連結演算子「＋」がある**

例　String s = "Hello" + "World";

※ただし、ループの中などで繰り返し行うのは厳禁

エラーではないが不適切なコード

String s = new String("Hello");

↑　2つのインスタンスが作られる。メモリーの無駄！

片方が数値なら自動的に文字列に変換される

```
String s1 = "Hello" + 7;
String s2 = 7 + "Hello";
```

↑ s1には“Hello7”の参照値が、s2には“7Hello”の参照値が格納される

※オマケ：では、「3 + "2"」は？

1. 5
2. 3
3. エラー
4. 32

文字列はequalsメソッドで比較

```
String s1 = "Hello";
String s2 = "Hello";

if（s1.equals（s2））{
  System.out.println（"等しい"）;
}
```

同一性と同値性

- 比較には2種類ある

- 同一性：オブジェクトの参照値が等しいこと
  →「==」演算子。
  　参照型の値比較には役に立たない

- 同値性：オブジェクトの値が等しいこと
  →equalsメソッド。参照型の比較はこちらで

確認クイズ
このプログラムの実行結果は？

```
public class JavaTest {
  public static void main(String[] args) {
    String s1 = "Hello";
    String s2 = new String("Hello");

    if(s1 == s2) {
      System.out.println("==で等しい");
    }
    if(s1.equals(s2)) {
      System.out.println("equalsで等しい");
    }
  }
}
```

1. 「==で等しい」を表示
2. 「equalsで等しい」を表示
3. 「==で等しい」「equalsで等しい」を両方表示
4. 何も表示されない

ポイント3　文字列を扱うクラスを使い分け

- String
- StringBuilder
- StringBuffer

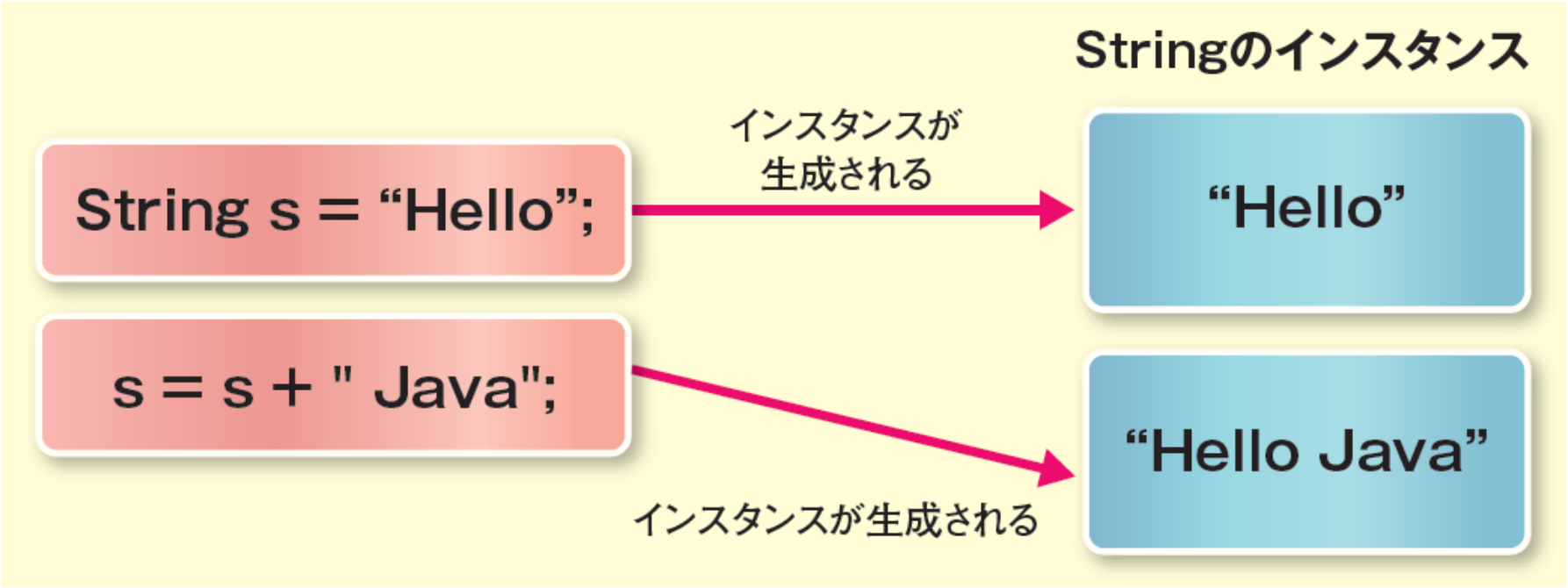
Stringは文字列を変更できない

- Stringは「不変オブジェクト」

- Stringの文字列は、一度生成したら変更できない

次のコードでは、インスタンスが書き換わっているのではなく、新しいインスタンスが作られている

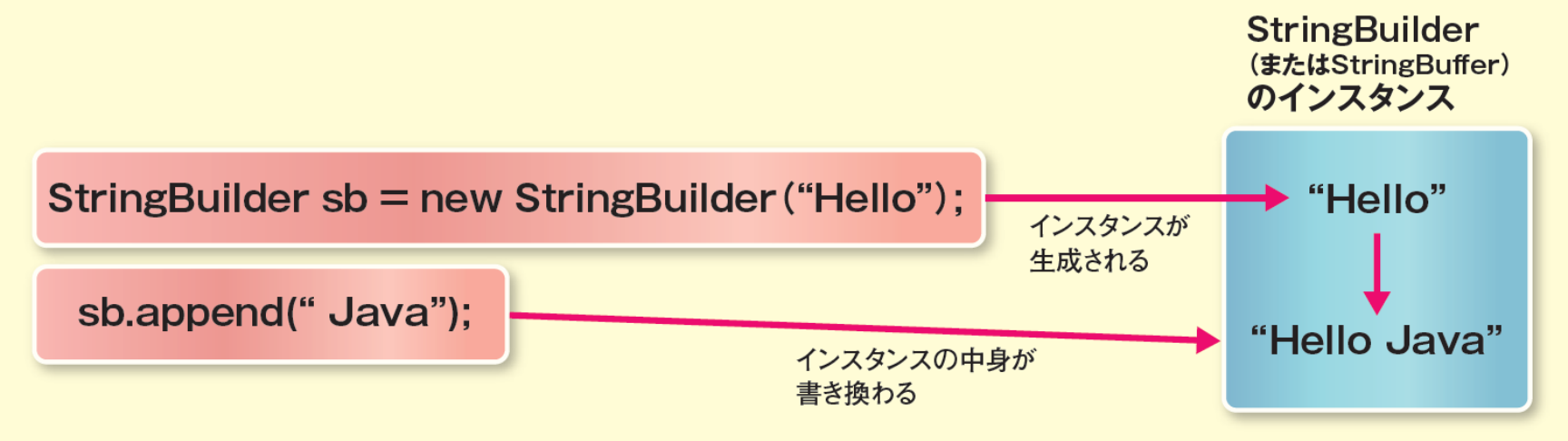
```
String s = "Hello";
s = s + " Java"; // sは"Hello Java"になる
```



不変オブジェクトになっている理由

- 文字列が不用意に変更されてプログラムの挙動がおかしくなることを防ぐため
  - 文字列が書き換え可能だと外部からの攻撃で不正な処理が行われる危険性が
- マルチスレッド環境で有利。排他制御が不要なので処理の負荷が軽い

文字列を変更できる StringBuilderとStringBuffer



文字列の連結にStringを使う例とStringBuilderを使う例

```
public class JavaTest {
  public static void main(String[] args) {
    String s = "";
    for (int i = 0; i < 10; i++) {
      s += "A";
    }
    System.out.println(s); // AAAAAAAAAA
    StringBuilder sb = new StringBuilder();
    for (int i = 0; i < 10; i++) {
      sb.append("A");
    }
    System.out.println(sb.toString()); // AAAAAAAAAA
  }
}
```

負荷が高い

実はループの中で10個のインスタンスを生成している

ループの中で新たなインスタンスは生成されない

StringBuilderとStringBufferの違い

- スレッドセーフであるかどうか

	StringBuilder	StringBuffer
排他制御	×	○
高速	○	×
文字列が壊れる	可能性あり	なし

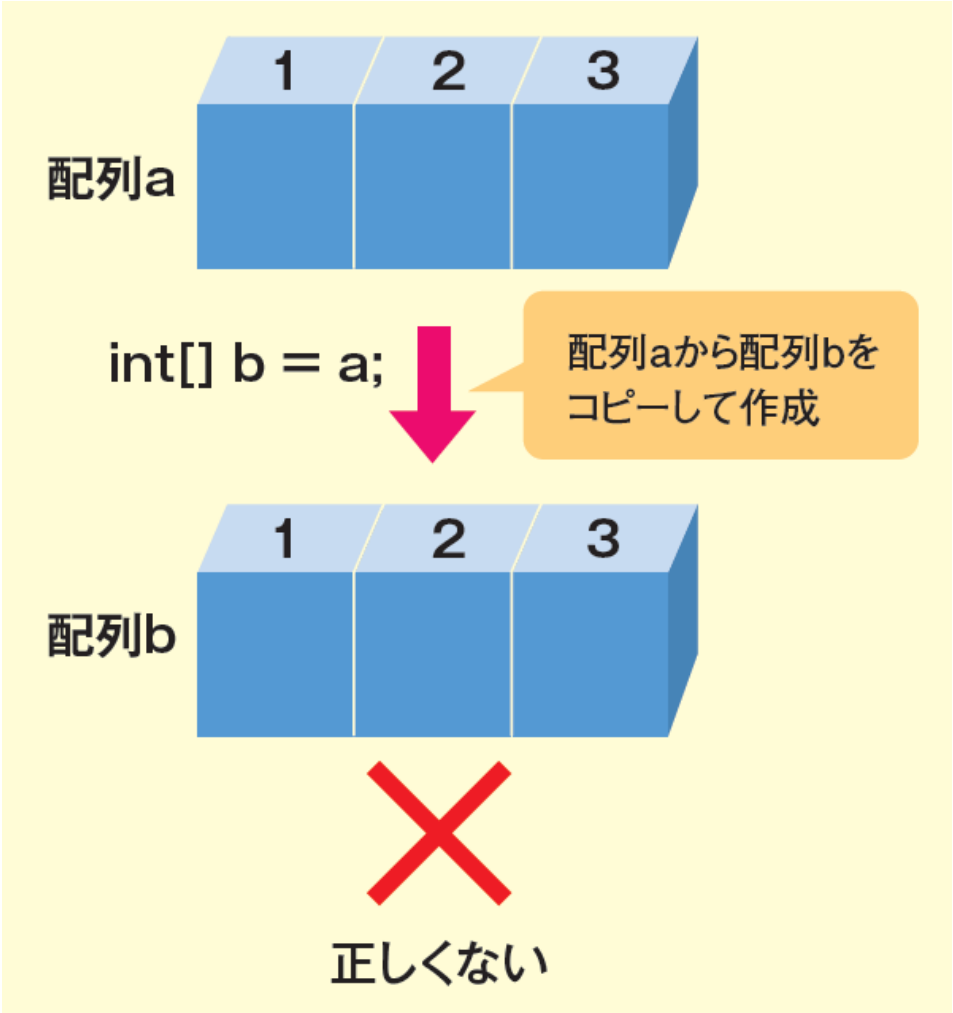
ポイント4　参照値のコピーとインスタンスのコピーの違いを理解する

int[] a = { 1, 2, 3 };

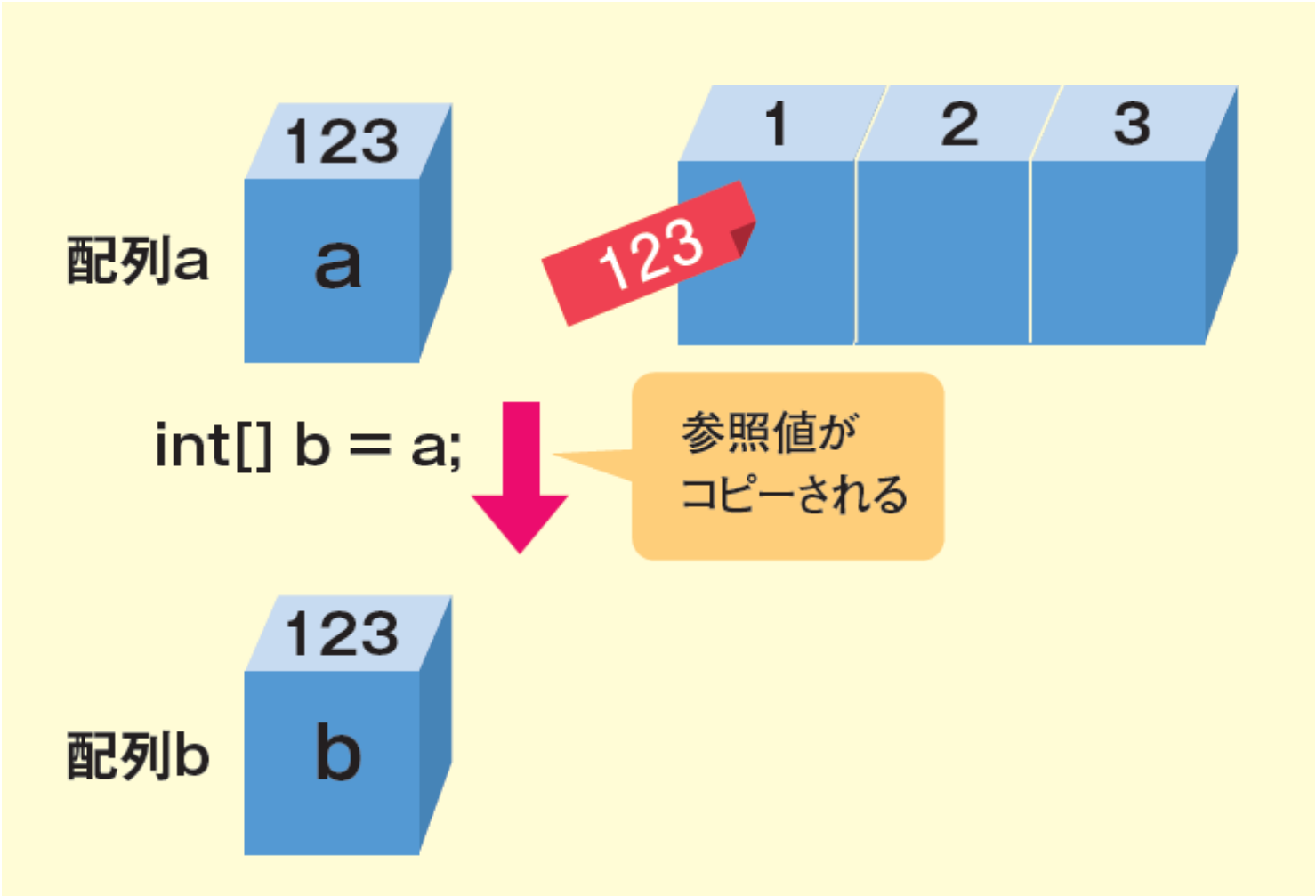
int[] b = a;

のイメージは？

int[] b = a の誤ったイメージ



int[] b = a の適切なイメージ



確認クイズ
このプログラムの実行結果は？

```
int[] a = { 1, 2, 3 };
int[] b = a;
b[0] = 7;
System.out.println( a[0] );
```

①1 ②7 ③0

補足：配列の実体をコピーする例

```
import java.util.Arrays;

public class JavaTest {
  public static void main(String[] args) {
  int[] a = { 1, 2, 3 };
  int[] b = new int[3];
  for(int i = 0; i < a.length; i++) {
    b[i] = a[i];
  }
  b[0] = 7;
  System.out.println(a[0]);   // 1を出力

  int[] c = Arrays.copyOf(a, a.length);
    c[0] = 7;
    System.out.println(a[0]);   // 1を出力
  }
}
```

forループの中で個々の要素をコピー

専用のcopyOfメソッドを利用

ポイント5　クラス設計の定石を学ぼう

クラスの設計には様々な「定石」がある

Javaはオブジェクト指向言語だが・・・

- Javaは「オブジェクト指向プログラミングを容易に実現できるように作られた言語」と考えた方がよい

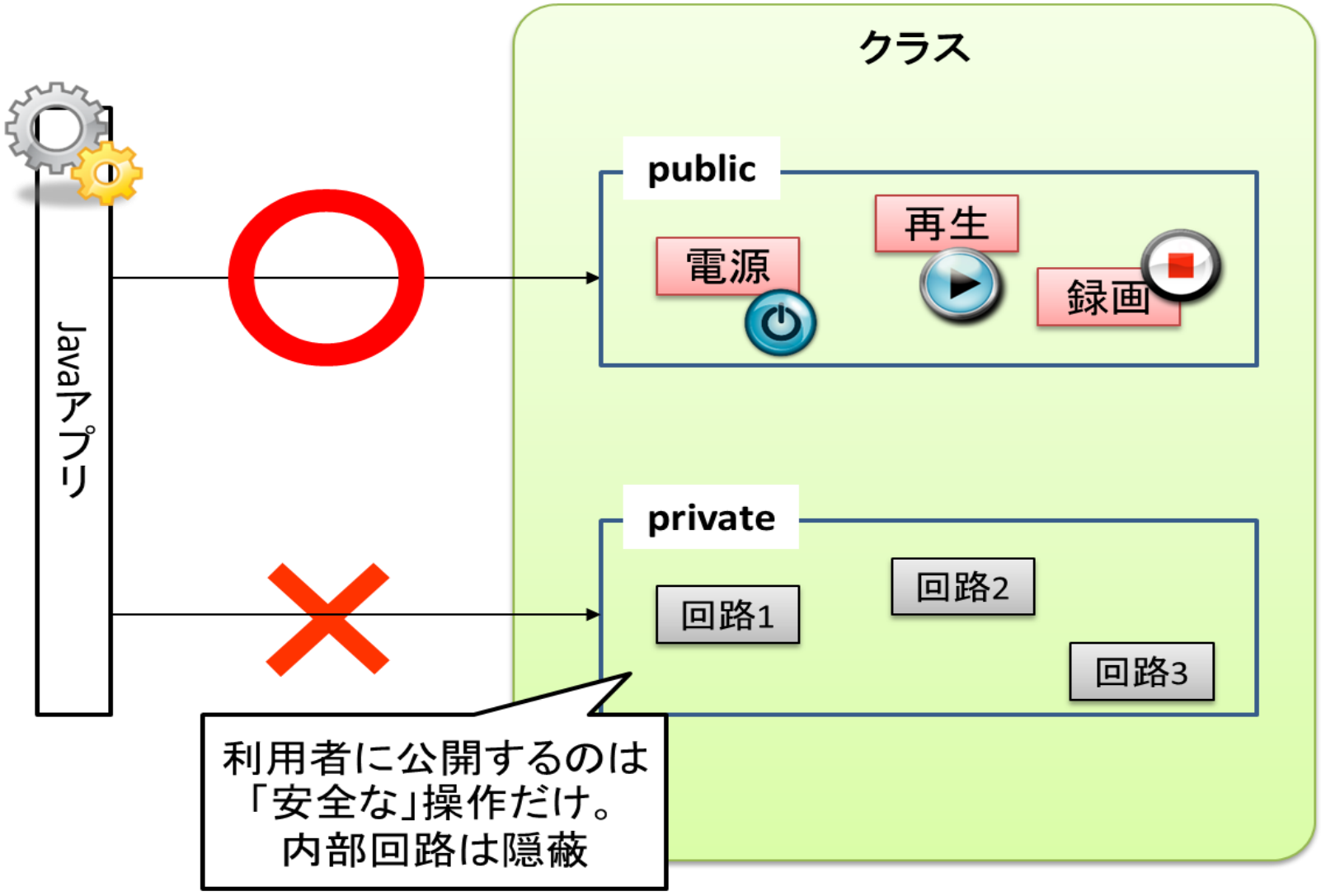
- 設計思想や定石を知らなければ、Javaを使ってもオブジェクト指向っぽくならない
  - たとえばmainメソッドの中にすべてのコードを詰め込んでも、正しいJavaのコード

そもそもオブジェクト指向とは？

- 大規模なソフトウエアを開発する際にプログラムが複雑になるのを防ぐための設計思想

- 厳密な定義はないが、一般には…
  - 継承：あるクラスの機能を引き継いで新たなクラスを定義
  - 多態性：同じ名前で異なる機能を持つメソッド
  - カプセル化：余計なものは見せないという思想

カプセル化


クラス
public
電源
再生
録画
private
回路1
回路2
回路3
Javaアプリ
利用者に公開するのは
「安全な」操作だけ。
内部回路は隠蔽

定石の例:カプセル化を実現するための「getter/setterメソッド」

```
1  public class Triangle {
2      private double _base;   // 底辺
3      private double _height; // 高さ
4
5      public Triangle(double base, double height) {
6          _base = base;
7          _height = height;
8      }
9
10     public double getArea() {
11         return _base * _height / 2.0;
12     }
13
14     public double get_base() {
15         return _base;
16     }
17
18     public void set_base(double _base) {
19         this._base = _base;
20     }
21
22     public double get_height() {
23         return _height;
24     }
25 }
```

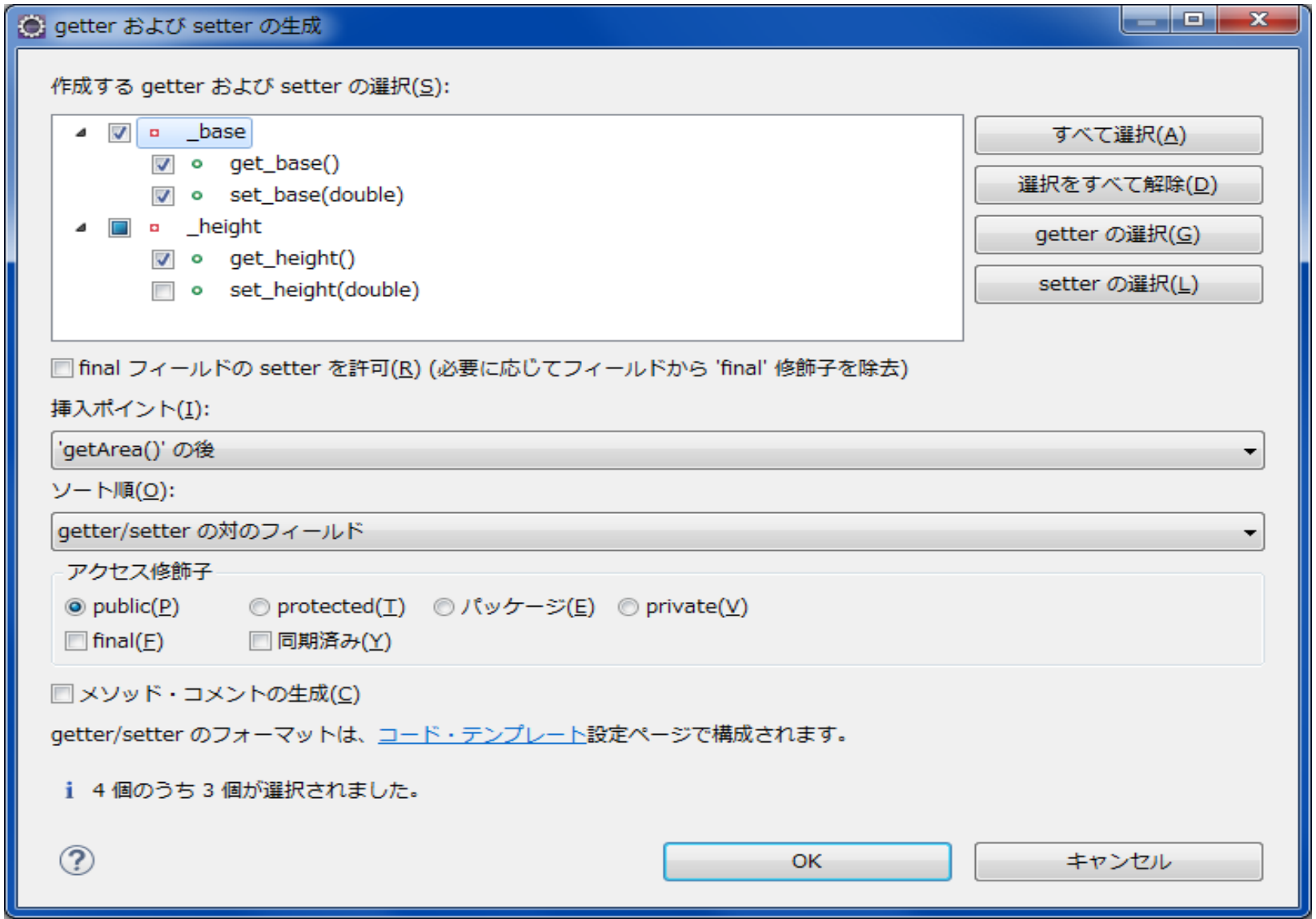
フィールドそのものは
privateで保護

フィールドにアクセスする
ための専用メソッドを用意

getter/setterメソッドのメリット

- フィールドだけでは、値を保証できないし、読み書きも自由
- setterメソッドには「フィールドに設定されようとしている値が適切なものかどうかをチェックする機能」「値が設定されたことをログに記録する機能」などを実装できる
- getterメソッドだけを記述してsetterメソッドを用意しなければ、そのフィールドを読み取り専用にできる

IDEがgetter/setterを自動生成してくれる



ポイント6　継承と多態性の意義を理解しよう

- 多態性とは
  - 同じ名前のメソッドの機能が、型に応じて自動的に切り替わること
- 多態性は何のための機能？
  - プログラミングの複雑化を抑制するための機能

例：TriangleクラスとRectangleクラスを定義

- プログラムの複雑化を防ぐため、面積を求めるメソッドの名前を「getArea」に統一したい

- 口頭でルールを決めても構わないが…
- 単なるルールでは破られる
  - プログラムで“強制”したい！
- このような場合は、「抽象クラスShape」と「抽象メソッドgetArea 」を導入する

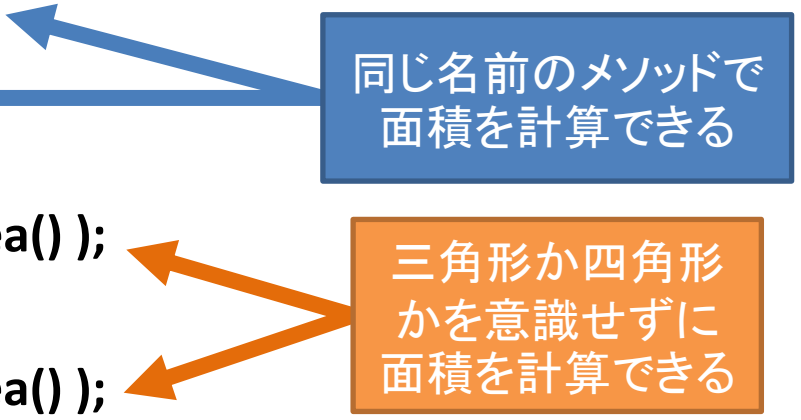
抽象クラスを使う多態性の例（1）

```
// 抽象クラス
abstract public class Shape {
  abstract public double getArea();  // 抽象メソッド
}

// Shapeクラスを継承してTriangleクラスを定義
public class Triangle extends Shape {
  private double _base, _height;

  public Triangle(double base, double height) {
    _base = base; _height = height;
  }

  // getAreaメソッドをオーバーライドする
  @Override
  public double getArea() {
    return _base * _height / 2.0;
  }
}
```

```
// Shapeクラスを継承してRectangleクラスを定義
public class Rectangle extends Shape {
  private double _length, _width;

  public Rectangle(double length, double width) {
    _length = length; _width = width;
  }

  // getAreaメソッドをオーバーライドする
  @Override
  public double getArea() {
    return _length * _width;
  }
}
```

抽象クラスを使う多態性の例（2）

```
public class JavaTest {
  public static void main(String[] args) {
    Rectangle rect = new Rectangle(20.0, 10.0);
    Triangle tri = new Triangle(20.0, 10.0);
    System.out.println( rect.getArea() );
    System.out.println( tri.getArea() );
    Shape rect_or_tri = rect;
    System.out.println( rect_or_tri.getArea() );
    rect_or_tri = tri;
    System.out.println( rect_or_tri.getArea() );
  }
}
```



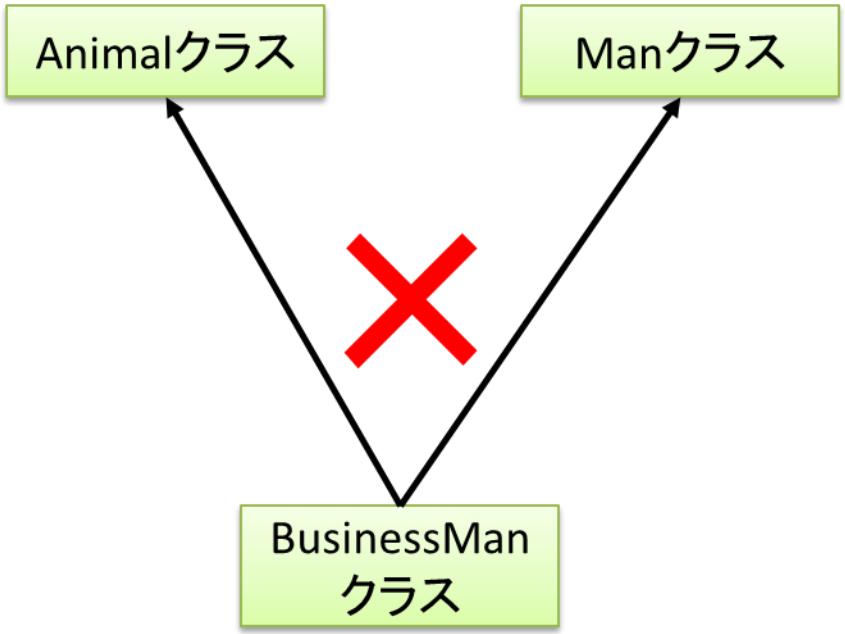
例：TriangleクラスとRectangleクラスを定義（2）

- Shape クラスを継承して、TriangleクラスとRectangleクラスを定義する
- すると、TriangleクラスとRectangleクラスでgetAreaメソッドをオーバーライドで定義しないと、コンパイルエラーになる
- よって、getAreaメソッドの実装を強制できる
- その後、CircleクラスやPentagonクラスなどを定義するときも、Shapeクラスを継承するクラスにすれば、getAreaメソッドの実装を強制できる

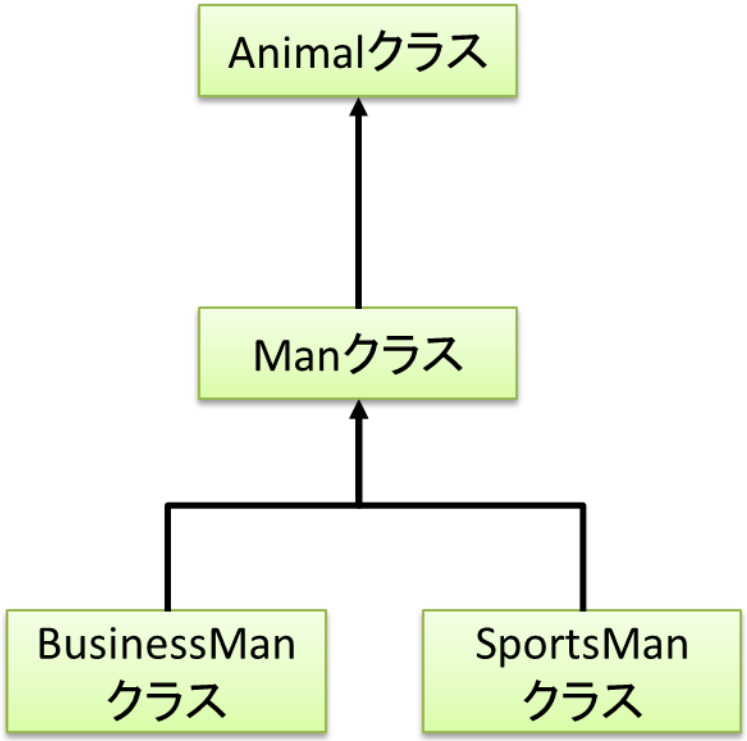
ポイント7 「インタフェース」の役割を理解

・まず、継承の制限を理解しよう

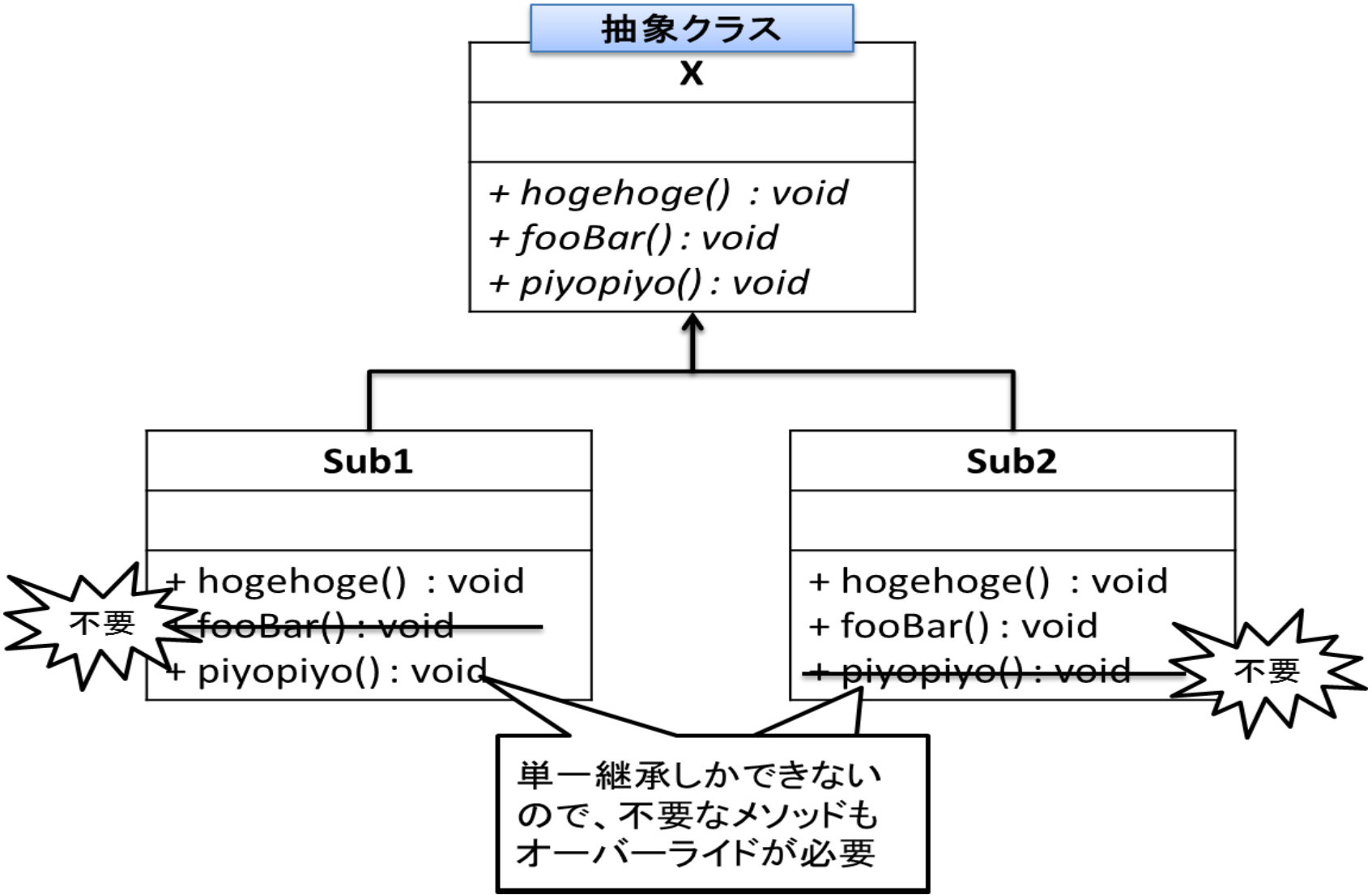
多重継承 複数クラスを同時には継承できない



単一継承 あるクラスのサブクラスを更に継承するのは可



こんな時はどうする？

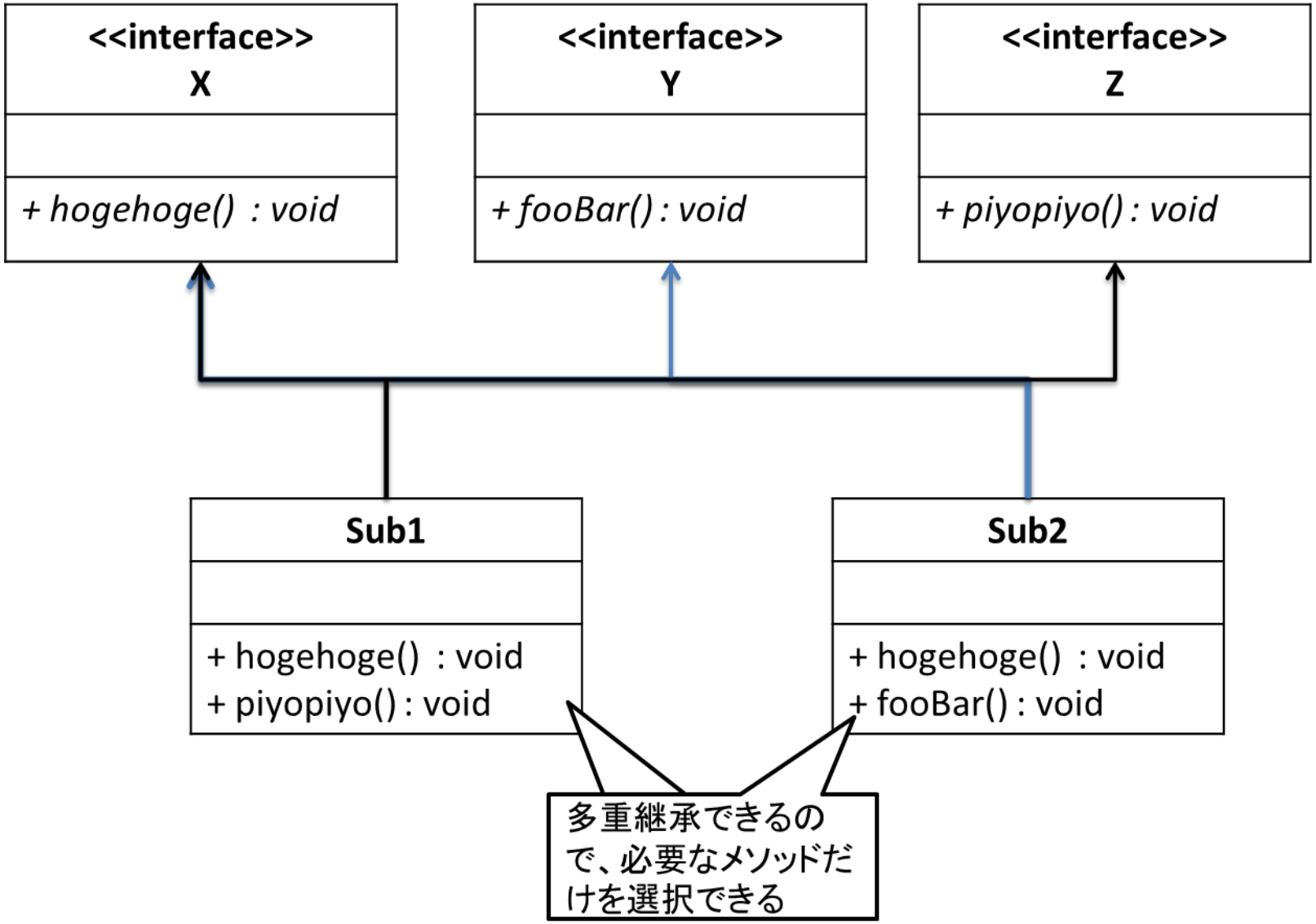

抽象クラス
X
+ hogehoge() : void
+ fooBar() : void
+ piyopiyo() : void
Sub1
+ hogehoge() : void
+ fooBar() : void
+ piyopiyo() : void
不要
Sub2
+ hogehoge() : void
+ fooBar() : void
+ piyopiyo() : void
不要
単一継承しかできないので、不要なメソッドもオーバーライドが必要

そこでインタフェース

- インタフェースは「インタフェース型」という抽象的な型を定義する機能
- すべてが抽象メソッドなクラス？
- いわゆる多重継承も可能
- 定義例

```
public interface IShape {
  double getArea();
}
```

インタフェースを利用すれば

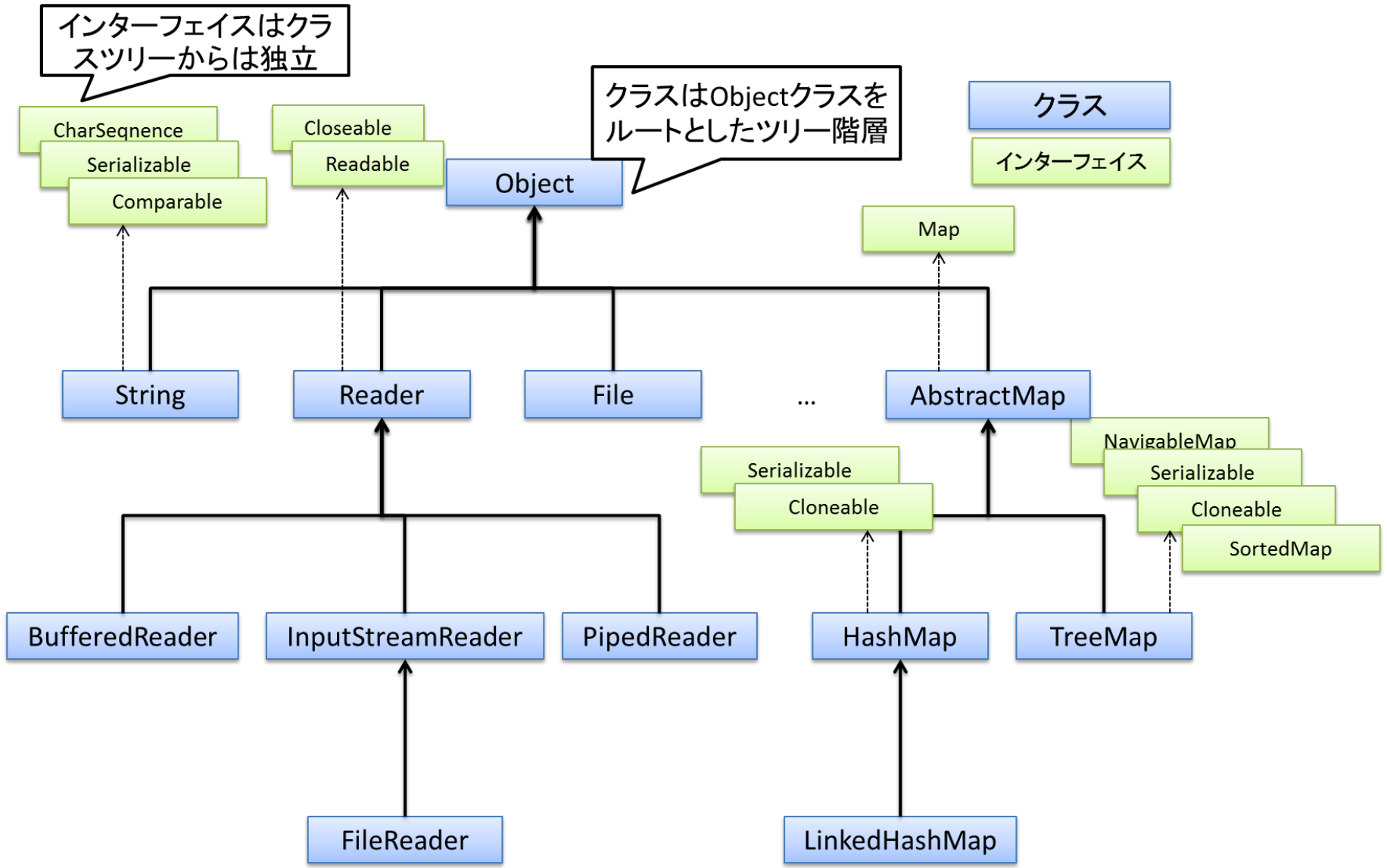


インタフェースによる多態性の例

```
// インタフェース
interface IShape {
double getArea();
}

// IShapeインタフェースを実装してTriangleクラスを定義
public class Triangle implements IShape {
  private double _base, _height;

  public Triangle(double base, double height) {
    _base = base; _height = height;
  }

  // getAreaメソッドをオーバーライドする
  @Override
  public double getArea() {
    return _base * _height / 2.0;
  }
}
```

```
// IShapeクラスを実装してRectangleクラスを定義
public class Rectangle implements IShape {
  private double _length, _width;

  public Rectangle(double length, double width) {
    _length = length; _width = width;
  }

  // getAreaメソッドをオーバーライドする
  @Override
  public double getArea() {
    return _length * _width;
  }
}
```

継承 or インタフェース



継承 or インタフェース

- 継承の問題
  - 「飛ぶ」メソッドを持った飛行機クラスと雀クラス
  - おそらく乗り物クラス、動物クラスを継承している
  - Flyingのようなクラスが割り込む余地はない
  - Flyableインターフェイスで型階層から独立した機能を実装できる

継承 or インタフェース

- 既存のクラスに機能を挿入する場合も
  - 関係する機能を洗い出して、すべての上位クラスに対して機能を挿入しなければならない
  - 階層の途中に、機能が不要なクラスが混じっていても除外できない
  - インタフェースならば、自由に機能を挿入できる

継承よりもインタフェースを使った方が良いの？

↓

その通りです

- 振る舞いよりも実装に関心がある場合
- デフォルト実装があるので今後は少なく？
- 通常のプログラムで継承を利用するのは「フレームワークなどを使う際に継承することが決まっているクラス」から自分のクラスを定義する場合

ポイント8　Stream APIとラムダ式を使う

- Java 8で加わった新機能

- Stream API：主に配列やコレクションをもとに、値の集計や絞り込みなどをするAPI
- ラムダ式（LAMBDA）：匿名メソッドを記述する構文

- これらを使うと、プログラムを簡潔に記述でき、記述量の削減を期待できる

Stream APIの利用イメージ

Stream APIの利用例

```
public class JavaTest {
  public static void main(String[] args) {
    List<String> list =
      new ArrayList<>(Arrays.asList("AAAA", "AAAAA", "AAAAAA", "AAAAAAA"));
    long count = list.stream().filter(s -> s.length() >= 5).count();
    System.out.println(count); // 3
    String[] array = {"AAAA", "AAAAA", "AAAAAA", "AAAAAAA"};
    count = Arrays.stream(array).filter(s -> s.length() >= 5).count();
    System.out.println(count); // 3
  }
}
```

ラムダ式

- ラムダ式

```
s -> s.length() >= 5
```

- 通常のメソッド

```
boolean test(String s) {
  if(s.length() >= 5) {
    return true;
  } else {
    return false;
  }
}
```

- **引数 -> 処理**
- ラムダ式では「処理の内容」だけを記述する
- メソッドに名前は付けない
- 処理の結果がそのまま戻り値になるのでreturn文は書かない
- 引数の型はコンパイラが周囲のコードから推論するので省略

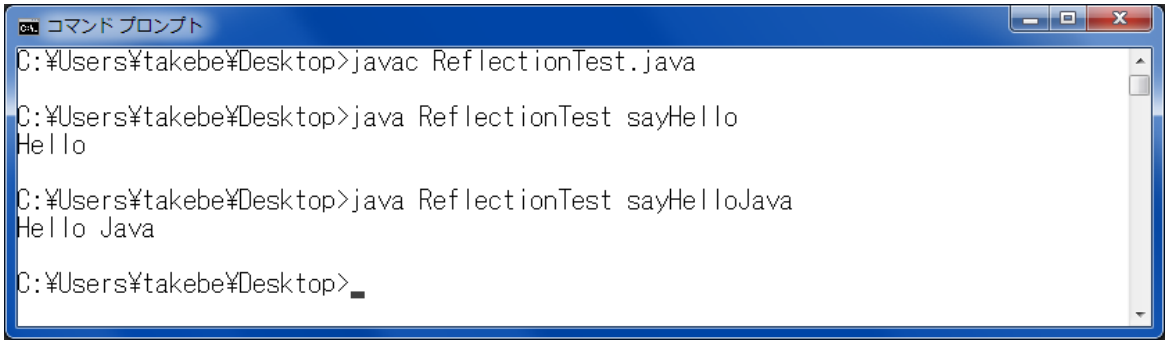
ポイント9　リフレクションを使おう

- コンパイル時ではなく、プログラムの実行時に、利用するクラスやメソッドを決定できる機能。柔軟なプログラムを作成できる
- **リフレクトAPIを使う**
- 便利だが、性能の低下やプログラムが読みにくくなるなどデメリットもあるので多用すべきではない

リフレクションの利用例

プログラムの実行時に、実行するメソッドを文字列で指定する

```
import java.lang.reflect.Method;

public class ReflectionTest {
  public void sayHello() { System.out.println("Hello"); }
  public void sayHelloJava() { System.out.println("Hello Java"); }

  public static void main(String[] args) throws Exception {
    Class<?> clazz = Class.forName("ReflectionTest");
    Object instance = clazz.newInstance();
    Method method = clazz.getMethod(args[0]);
    method.invoke(instance);
  }
}
```



ポイント10　nullが入っている可能性を意識する

- 参照型変数にはnullが格納されている可能性がある
- nullが入った変数からメソッドを呼び出したり、フィールドにアクセスしたりすると、有名な「NullPointerException」例外が発生する
- nullチェックを行おう

例：NullPointerExceptionを引き起こす

```
Map<Integer, String> map = new HashMap<>();
map.put(0, "camera");
String product = map.get(1);
System.out.println(product.toUpperCase());
```

nullチェックでNullPointerExceptionを避ける

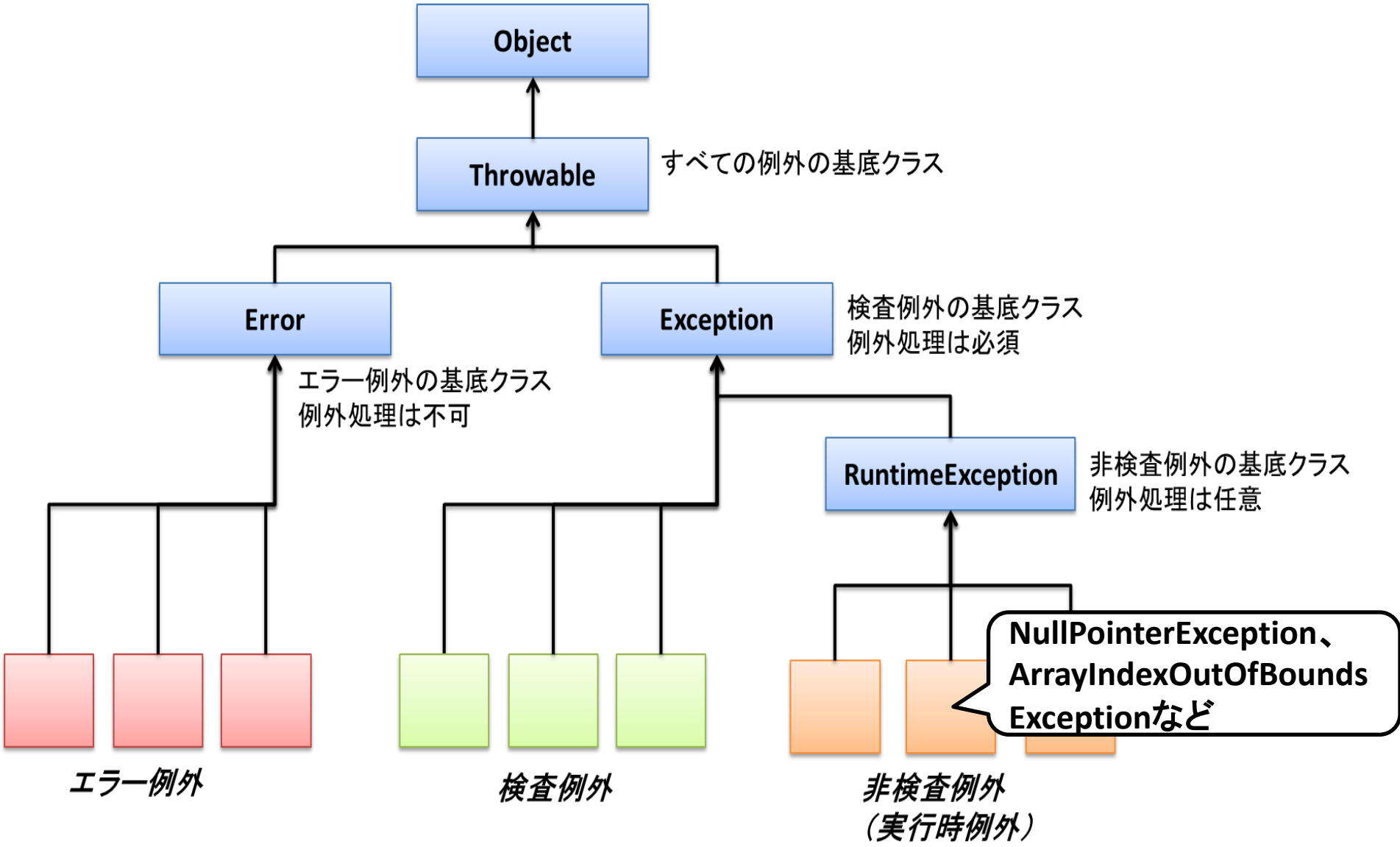
```
Map<Integer, String> map = new HashMap<>();
map.put(0, "camera");
String product = map.get(1);
if(product != null) {
  System.out.println(product.toUpperCase());
} else {
  System.out.println("そのデータはありません");
}
```

nullチェックの良くないやり方

```
try {
  System.out.println(product.toUpperCase());
} catch (NullPointerException e) {
  System.out.println("そのデータはありません");
}
```

try～catch句はオーバーヘッドが大きいだけではなく、
コードのどこに問題があるのかが曖昧になりがち

例外の種類


Object
Throwable
すべての例外の基底クラス
Error
Exception
検査例外の基底クラス
例外処理は必須
エラー例外の基底クラス
例外処理は不可
RuntimeException
非検査例外の基底クラス
例外処理は任意
NullPointerException、
ArrayIndexOutOfBounds
Exceptionなど
エラー例外
検査例外
非検査例外
（実行時例外）

質疑応答

第2部　手を動かして学ぶ

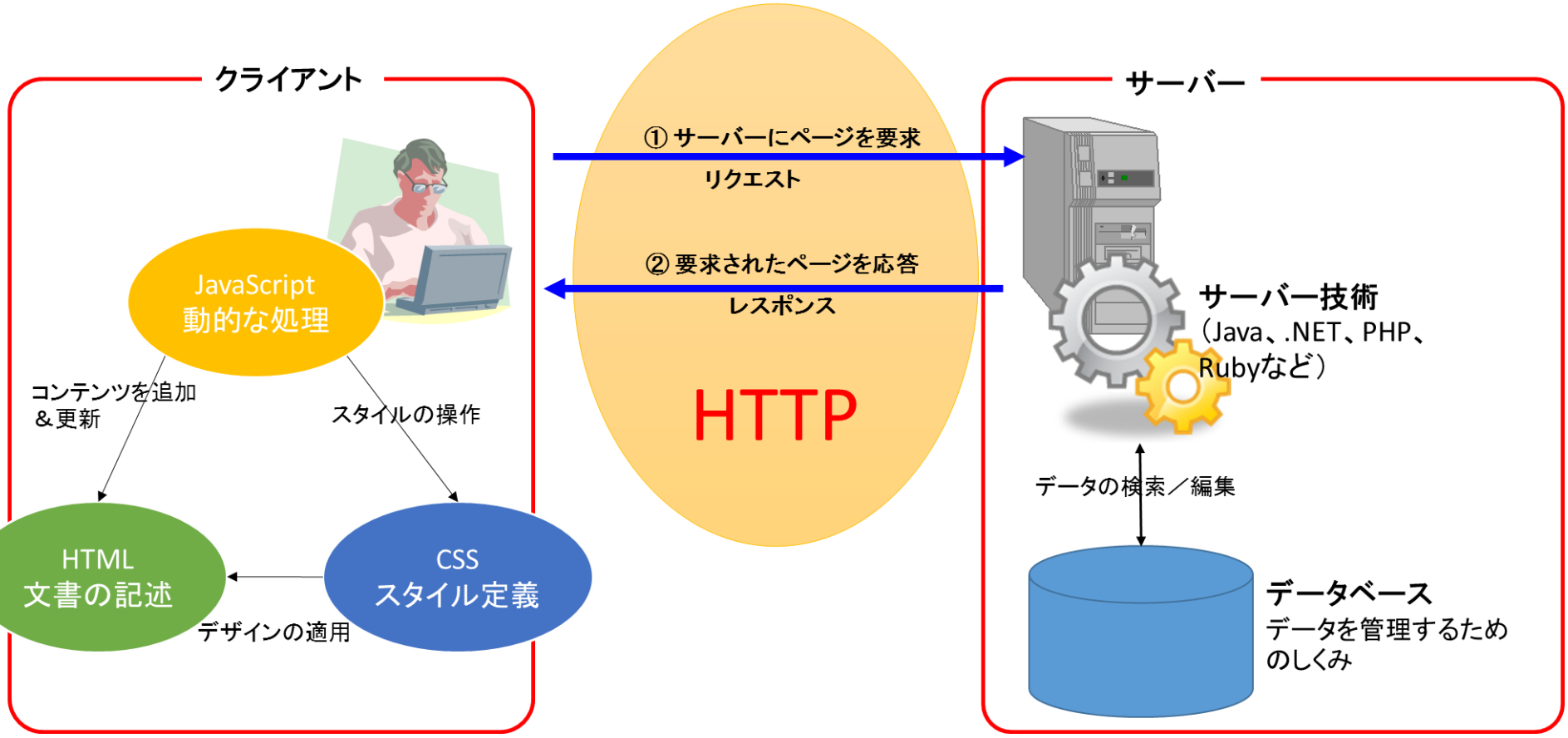
Web-DBアプリ開発の基本を復習 JSP/JSTL編

11:15～12:30
＜75分＞

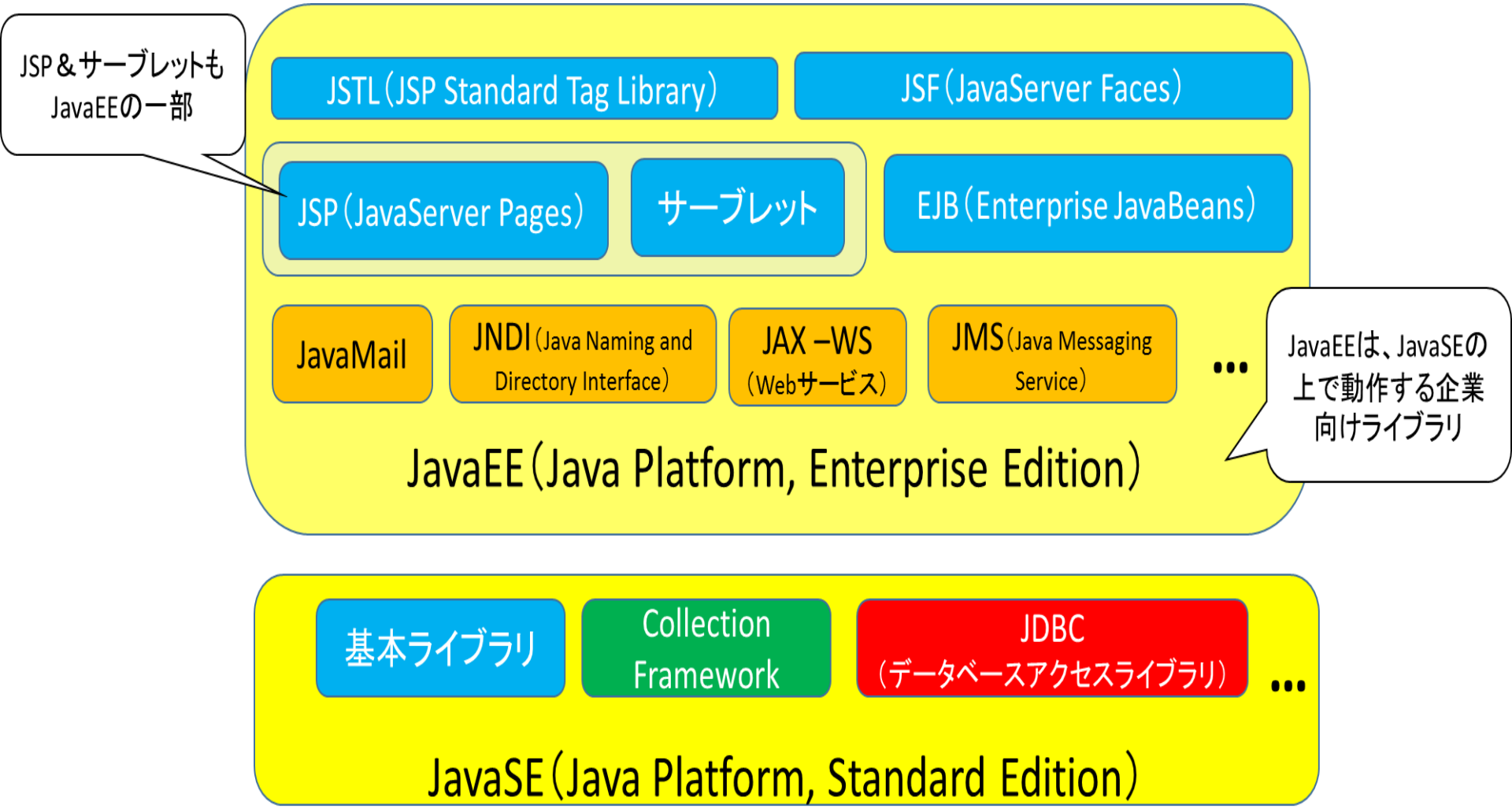
Webアプリとは？

- ブラウザー上で動作するアプリ
  - ブログ、SNS、ショッピングサイト etc...
- 実際の開発ではさまざまな技術が使われる
- JSP＆サーブレットはWebアプリを開発するための技術（の一部）

Webアプリを構成する技術



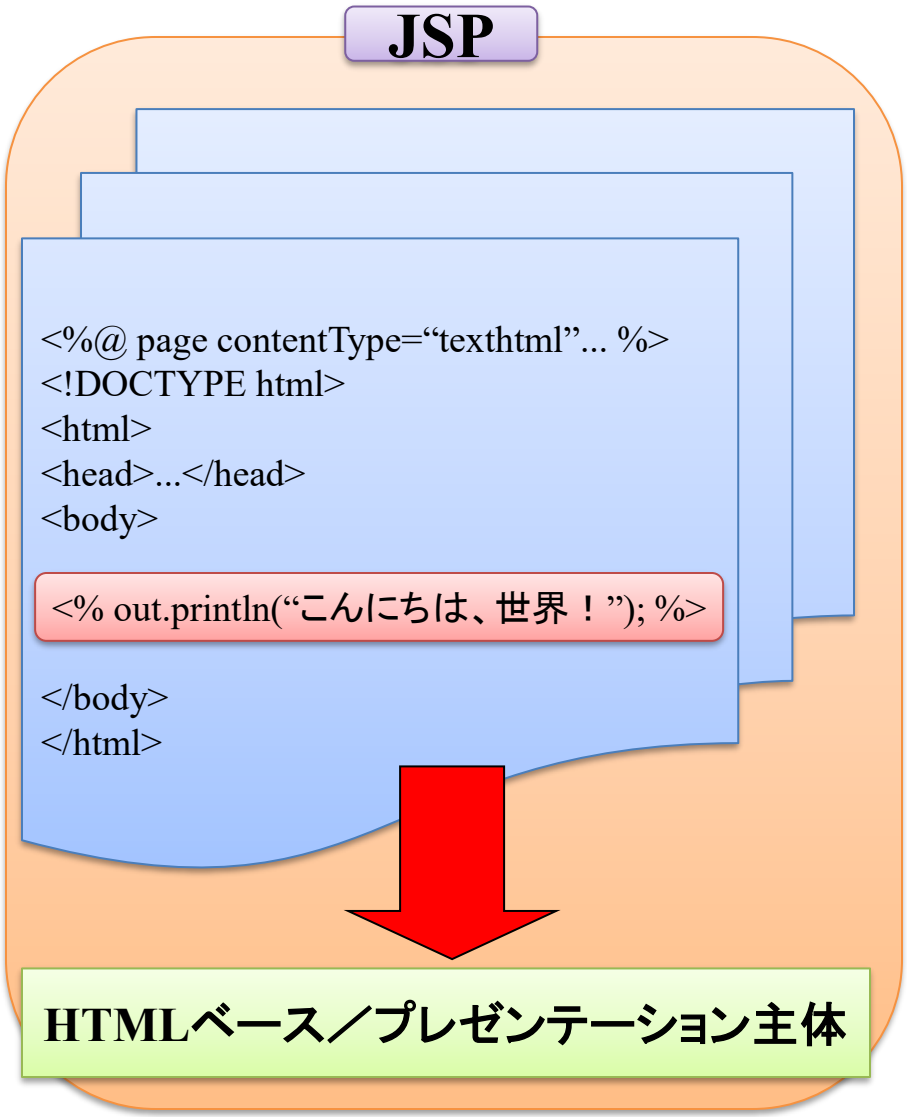
JSP＆サーブレットの位置づけ

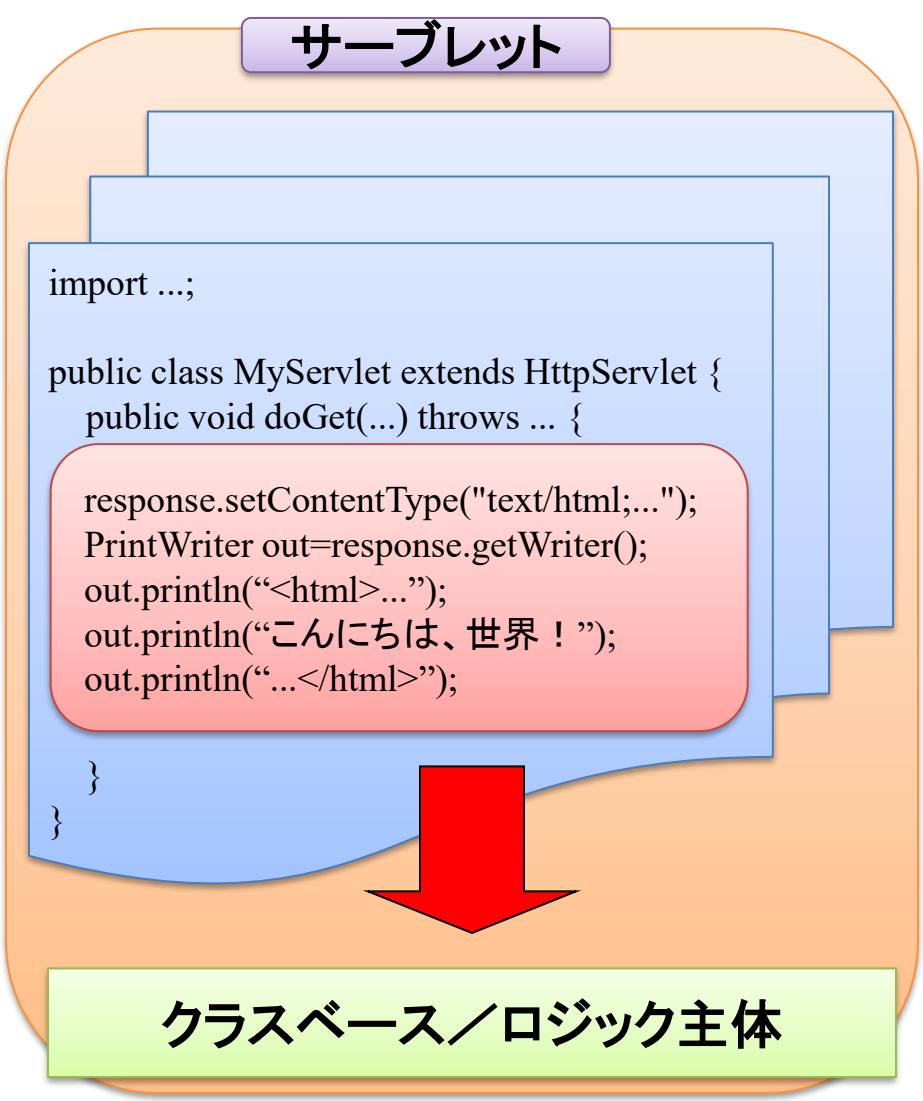


JSPとサーブレット（1）

- 非常に良く似た技術
- JSPでできることの“すべて”は、サーブレットでもできる。サーブレットでできることの“ほとんど”は、JSPでもできる
- 違いは、利用場面

JSPとサーブレット（2）

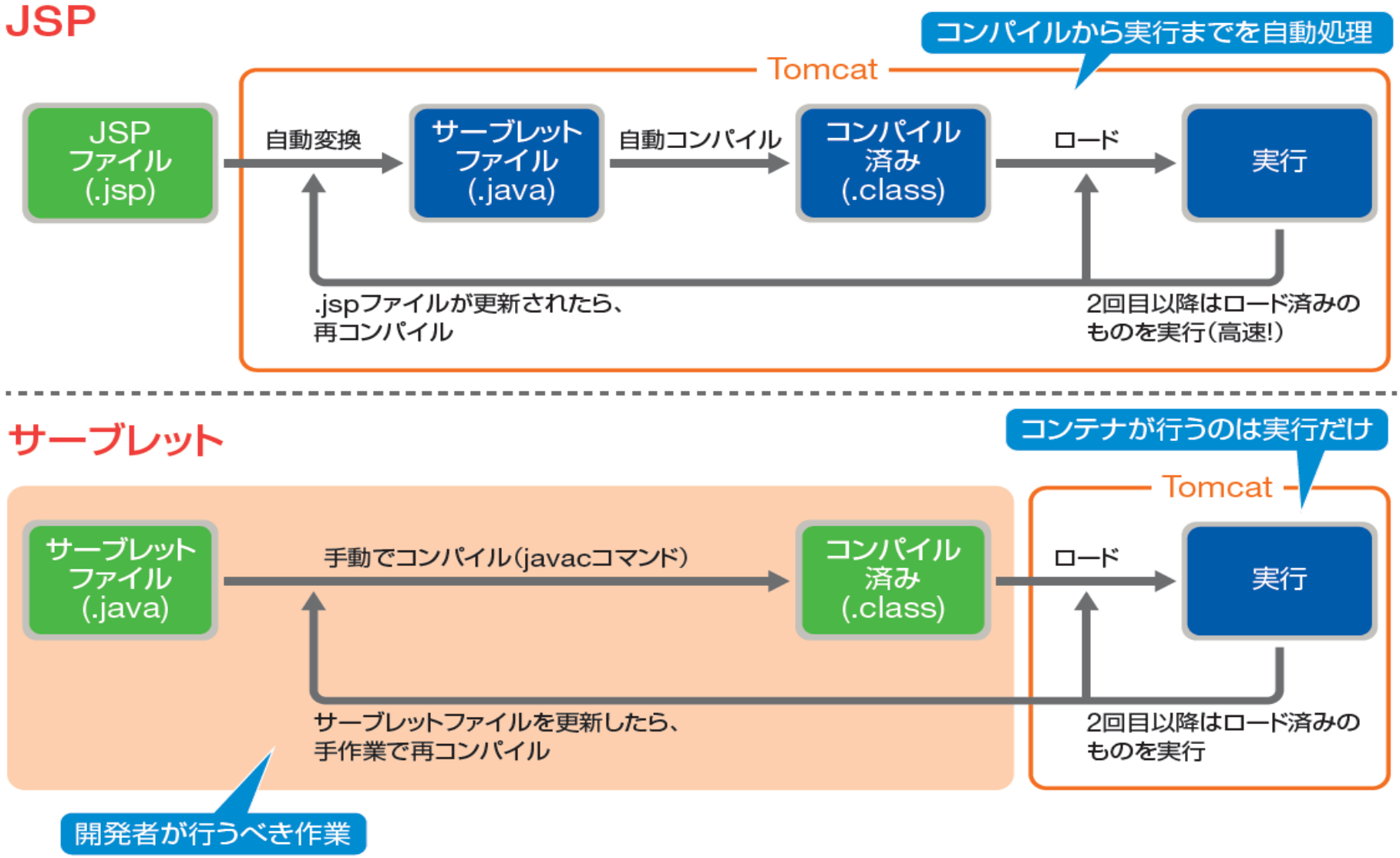




JSPとサーブレット（3）

- JSPはHTMLの中にJavaのコードを記述
- JSPはWebページのデザインとロジックのコードを混在させることができる。難易度が低いので、専門のJavaプログラマではないWebデザイナーでも使いこなせる
- ただし、デザインとロジックの混在は、Webアプリが大規模になったとき、プログラムが複雑になりがち
- 大規模なWebアプリの開発は、デザインの部分はJSPで、ロジックの部分はサーブレットで行われる

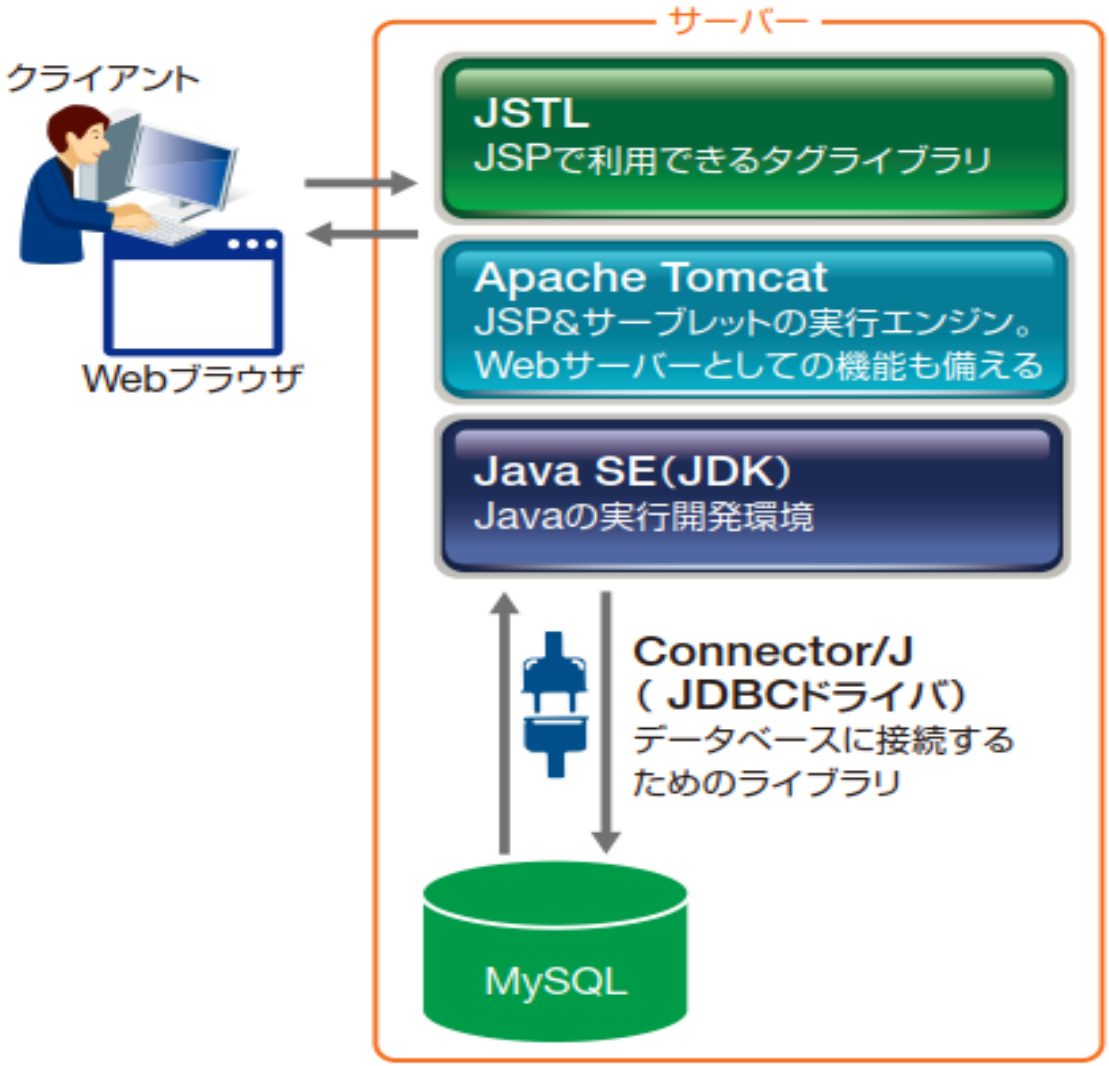
JSPとサーブレット（4）



JSPとサーブレット(5)

- JSPは 開発者自身によるコンパイルが**不要**
- サーブレットは開発者自身によるコンパイルが**必要**
- **JSPは初回実行時にサーブレットに変換される。よって、初回の実行は若干遅くなる**

JSP＆サーブレット開発に必要な環境



実習の準備

- JDK8／Tomcat8／MySQL 5.7／JSTL
- 環境変数の設定は大丈夫ですか？
- 配布されたサンプルファイルの/beforeフォルダ配下の/nikkeiフォルダーを丸ごと、%CATALINA_HOME%/webappsフォルダーにコピー
- JSTLのjavax.servlet.jsp.jstl-api-1.2.1.jar／javax.servlet.jsp.jstl-1.2.1.jarは、/Nikkei/WEB-INF/libフォルダーにコピー

アプリにアクセスしよう

- Tomcatはタスクバーから起動

- トップページURL
  http://localhost:8080/

- アプリURL
  http://localhost:8080/nikkei

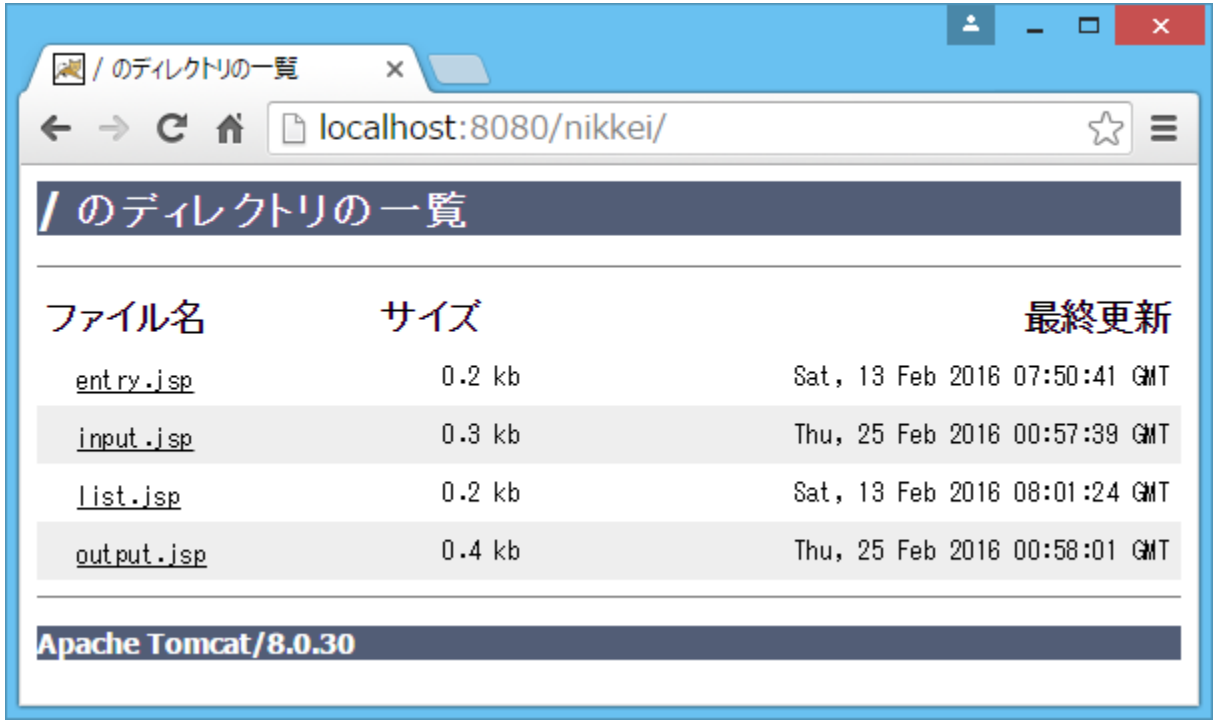
準備：ファイルリストを表示

- %CATALINA_HOME%/conf配下のweb.xml
- 編集後はTomcatを再起動

```
<servlet>
  <servlet-name>default</servlet-name>
  <servlet-class>org.apache.catalina.servlets.DefaultServlet</servlet-class>
  ...中略...
  <init-param>
    <param-name>listings</param-name>
    <param-value>true</param-value>
  </init-param>
  <load-on-startup>1</load-on-startup>
</servlet>
```

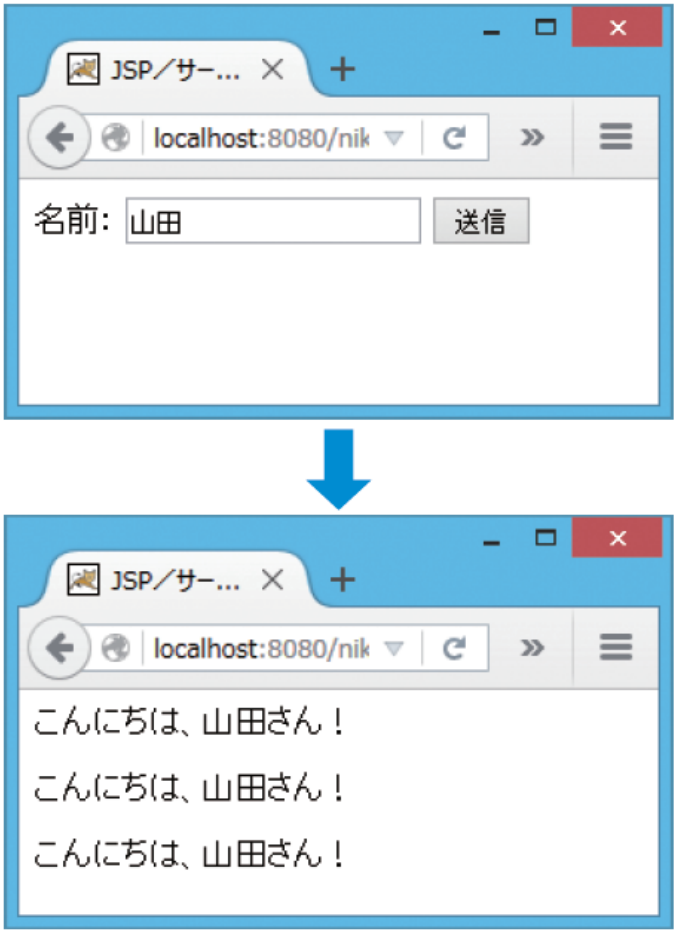
ファイルリストURL

http://localhost:8080/nikkei/



実習：こんにちは、●○さん！

input.jspとoutput.jspの実行例



実習：こんにちは、●○さん！

input.jsp。UTF-8の文字コードで記述する

```
<%@ page contentType="text/html;charset=UTF-8" pageEncoding="UTF-8" %>
<!DOCTYPE html>
<html>
<head>
<meta charset="UTF-8" />
<title>JSP／サーブレット</title>
</head>
<body>
<form method="POST" action="output.jsp">
  <label for="name">名前：</label>
  <input id="name" name="name" type="text" />
  <input type="submit" value="送信" />
</form>
</body>
</html>
```

実習：こんにちは、●○さん！

output.jsp。UTF-8の文字コードで記述する

```
<%@ page contentType="text/html;charset=UTF-8" %>
<% request.setCharacterEncoding("UTF-8"); %>
<!DOCTYPE html>
<html>
<head>
<meta charset="UTF-8" />
<title>JSP／サーブレット</title>
</head>
<body>
<p><% out.print("こんにちは、" + request.getParameter("name") + "さん！
"); %></p>
<p>こんにちは、<%= request.getParameter("name") %>さん！</p>
<p>こんにちは、${param['name']}さん！</p>
</body>
</html>
```

実習：こんにちは、●○さん！

実行URL

http://localhost:8080/nikkei/input.jsp

JSPの要素：ディレクティブ

<@ ディレクティブ名 属性="値" … %>

例

<%@ page contentType="text/html;charset=UTF-8" pageEncoding="UTF-8" %>

- **コンテナー（Tomcat）に対する指令を表す**
- ディレクティブの中でも、@pageはページの基本情報を表すことから、最もよく使われる

@pageディレクティブは同じページの中で複数記述してもOK

- **これはOK**

```
<%@page contentType="text/html;charset=UTF-8" %>
<%@page pageEncoding="UTF-8" %>
```

同じ属性を複数指定するのは不可

- **これは不可。エラーになる**

```
<%@page contentType="text/html;charset=UTF-8" %>
<%@page contentType="text/html;charset=Windows-31J" %>
```

JSPの要素：スクリプトレット

<% Javaのコード %>

例

```
<% out.print("こんにちは、" +
request.getParameter("name") + "さん！"); %>
```

- プログラムを実行する最も原始的で手っ取り早い方法
- requestやoutは、「暗黙オブジェクト」と呼ばれる

<%=...%>は、<% out.print(...); %>の省略構文

<%= 式 %> ＝ <% out.print(...); %>

簡単な式を出力するにはこちら

JSPの要素:**式言語**

${ 式言語 }

例

${param['num'] * 10}

- データ型の制約が緩い独自のスクリプト言語
- シンプルな構文。習得は簡単

式言語とJavaの比較

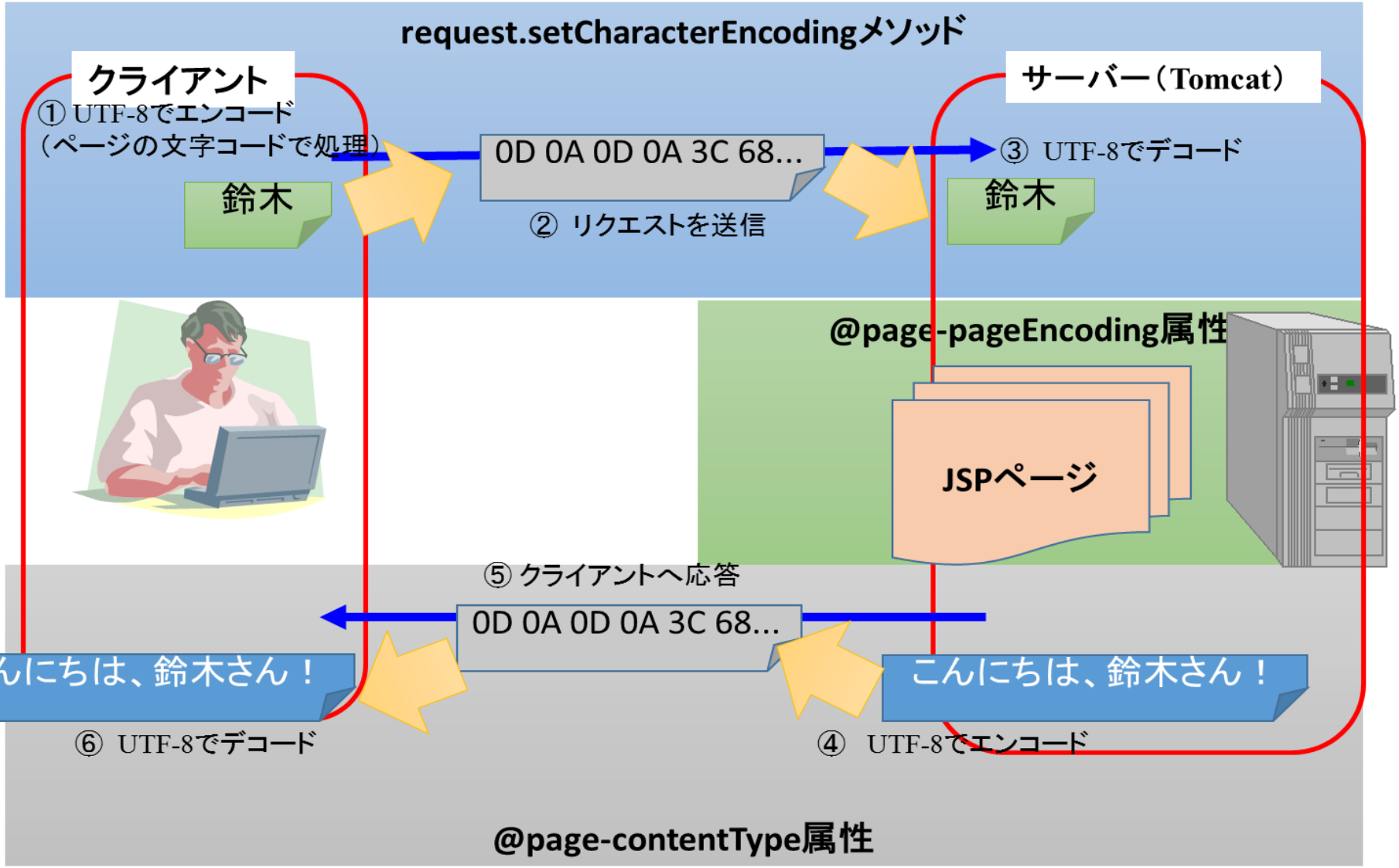
- 式言語

${param['num'] * 10}

- Java

<%=Integer.parseInt(request.getParameter("num") * 10 %>

入力文字コードの宣言


request.setCharacterEncodingメソッド
クライアント
① UTF-8でエンコード
（ページの文字コードで処理）
鈴木
0D 0A 0D 0A 3C 68…
② リクエストを送信
サーバー（Tomcat）
③ UTF-8でデコード
鈴木
@page-pageEncoding属性
JSPページ
⑤ クライアントへ応答
0D 0A 0D 0A 3C 68…
こんにちは、鈴木さん！
⑥ UTF-8でデコード
④ UTF-8でエンコード
こんにちは、鈴木さん！
@page-contentType属性

実習：HTTP GETとHTTP POST

- <form method="POST">
  - HTTP POSTという命令でデータを送信しなさい

- <form method="GET">と書き換えてみよう！（input.jsp）

```
<form method="GET" action="output.jsp">
  …中略…
</form>
```

HTTP GETでは入力値が見える(1)

HTTP GETでは入力値が見える(2)

- GETでは入力値が「～?name=山田」の形式で送信される。？以降の情報は「クエリー情報」と言う
- GETでは、性質上、たくさんのデータを送信できない
- 結果ページは「http://search.yahoo.co.jp/search?p=JSP」のようなURLで表示されるので、そのままブックマークできるメリットがある

HTTP POSTでは…

- アドレス欄に入力値が反映されることはない
- 「リクエストボディ」と呼ばれるデータ専用の領域で、データが送信される
  - 大量のデータを送信するのに向いている
- 結果ページをブックマークすることはできない

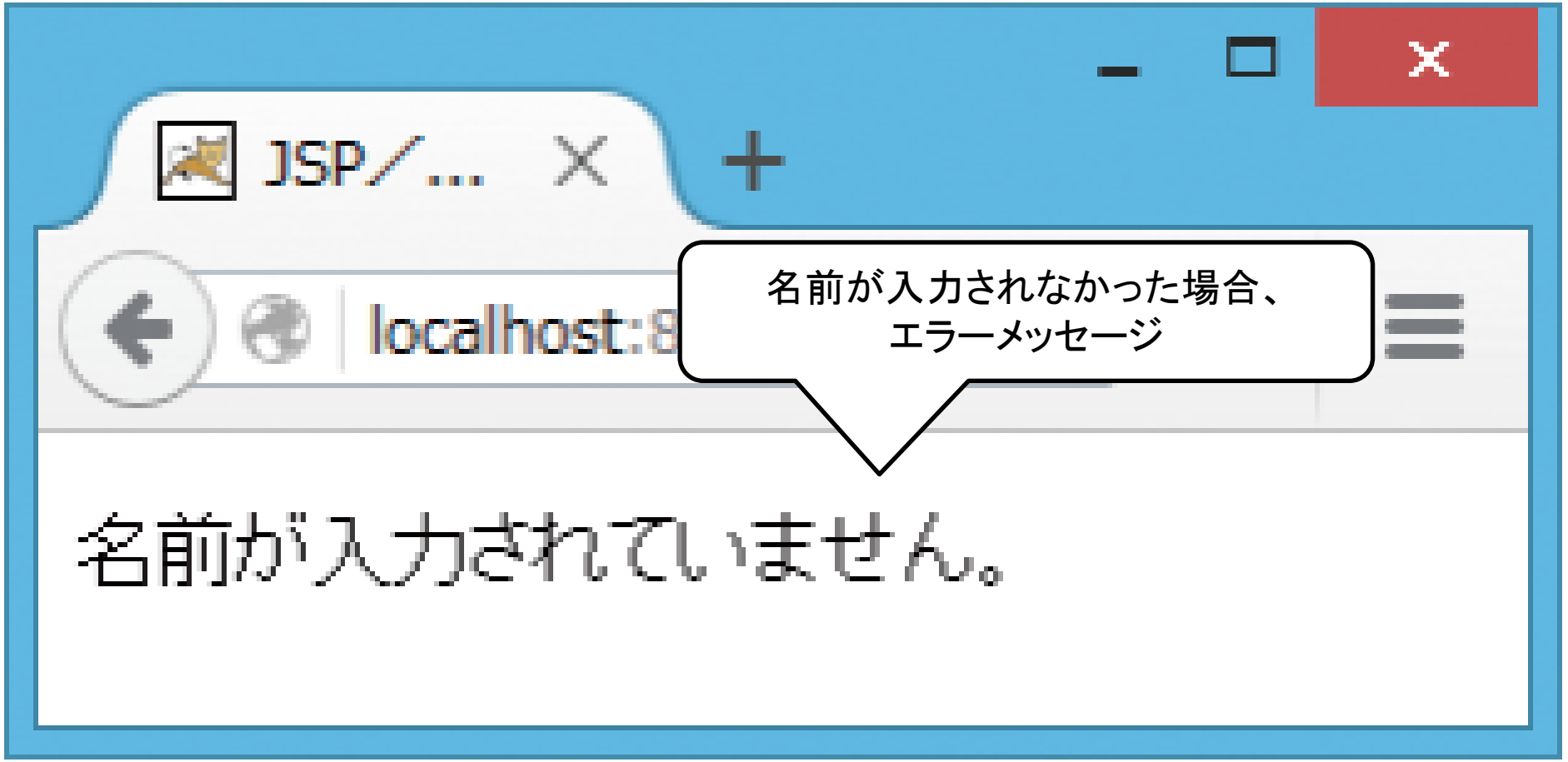
JSTLでタグプログラミング

- JSPには「アクションタグ」がある
  - <jsp:include page="include.jsp" />
- アクションタグは、条件分岐や繰り返しのような制御処理を、HTMLによく似たタグの形式で実現するための仕組み
- 標準タグの機能は貧相
- JSTLは、アクションタグのライブラリ
  - 式言語との組み合わせでの開発が一般的
  - できることは限られる（自ずとビューをすっきりと）

JSTLに含まれる各種ライブラリ

ライブラリ	概要
Core	基本的な制御構文(変数の設定／参照、条件分岐、繰り返し処理など)
i18n	国際化機能(数値／日付の整形、メッセージの出力など)
Database	データベースの操作(登録／検索、トランザクションの制御など)
Xml	XML文書の操作(ノードの抽出、XSLT変換など)
Functions	関数機能(文字列加工や配列のサイズ取得など)

実習：if文に相当する<c:if>要素



実習：if文に相当する<c:if>要素

```
<%@ page contentType="text/html;charset=UTF-8" %>
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
<% request.setCharacterEncoding("UTF-8"); %>
<!DOCTYPE html>
<html>
<head>
<meta charset="UTF-8" />
<title>JSP／サーブレット</title>
</head>
<body>
<p>
<c:if test="${empty param['name']}">
 名前が入力されていません。
</c:if>
<c:if test="${!empty param['name']}">
 こんにちは、${param['name']}さん！
</c:if>
</p>
</body>
</html>
```

<c:if>要素を使って書き換えた output.jsp

実習：if文に相当する<c:if>要素

実行URL

http://localhost:8080/nikkei/input.jsp

<c:if>要素（1）

- @taglibディレクティブで「どのライブラリを利用するのか」を宣言
- uri属性にはタグライブラリを特定するための“キー情報”を記述。見かけはURLだが、単にユニークな文字列として使われているだけ。参照すべきコンテンツがある必要はない
- prefix属性にはタグを呼び出すときの接頭辞を記述

<c:if>要素（2）

- test属性の式がtrueである場合に、配下のコンテンツを出力
- if文のelse文に相当するタグはない。反対の条件の<c:if>を列挙することで、else文に相当する処理を記述する

実習：<c:choose>要素で書き換え

```
<%@ page contentType="text/html;charset=UTF-8" %>
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
<% request.setCharacterEncoding("UTF-8"); %>
<!DOCTYPE html>
<html>
<head>
<meta charset="UTF-8" />
<title>JSP／サーブレット</title>
</head>
<body>
<p>
<c:choose>
 <c:when test="${empty param['name']}">
  名前が入力されていません。
 </c:when>
 <c:otherwise>
  こんにちは、${param['name']}さん！
 </c:otherwise>
</c:choose>
</p>
</body>
</html>
```

<c:choose>要素を使って書き換えた output.jsp

実習：<c:choose>要素で書き換え

実行URL

http://localhost:8080/nikkei/input.jsp

多岐分岐を表す<c:choose>要素

```
<c:choose>
<c:when test="条件式">
...</c:when>
<c:when test="条件式">
...</c:when>
...
<c:otherwise>
...</c:otherwise>
</c:choose>
```

<c:choose>要素

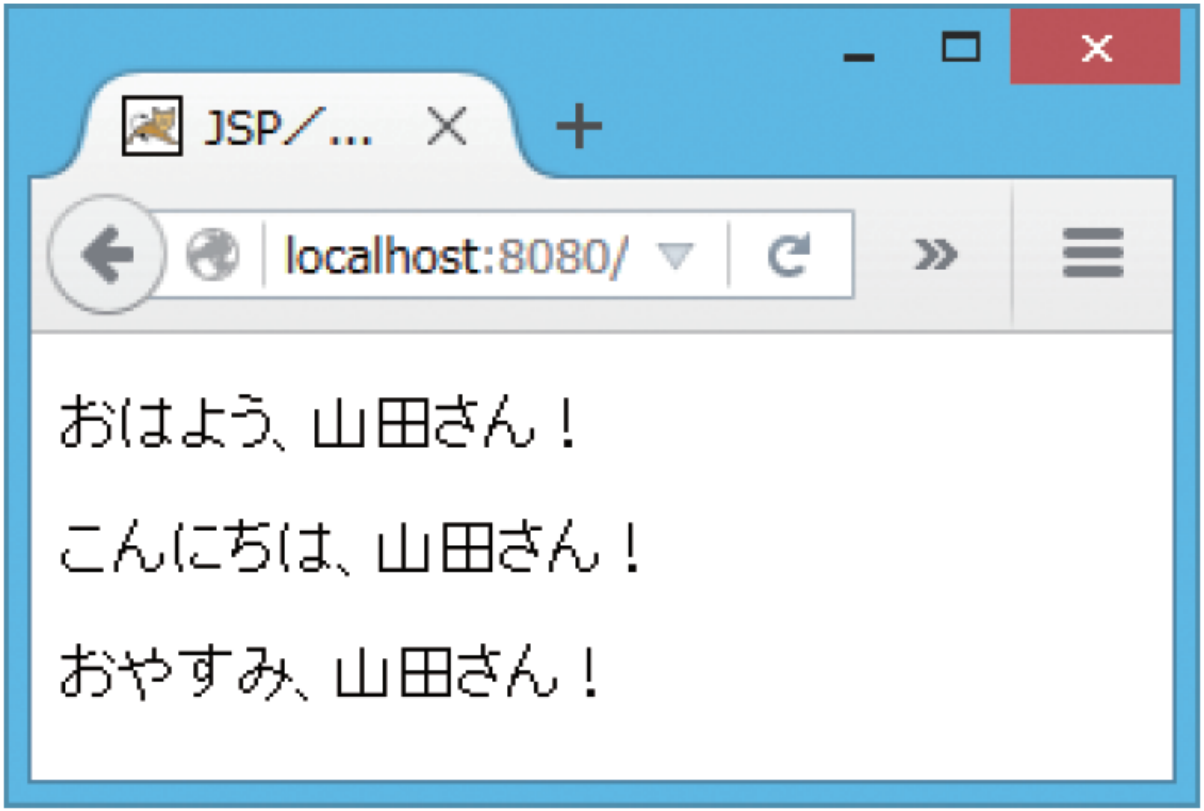
- switch命令に似ている
- switchに当たるのが<c:choose>
- caseに当たるのが<c:when>
  - ただし、指定するのは値ではなく条件式
  - switch命令よりもif...else ifに近い
  - 複数の条件式がtrueになった場合に出力されるのは最初の一つだけ
- defaultに当たるのが<c:otherwise>
- <c:otherwise>は必須ではない

配列/コレクションを順に処理する<c:forEach>要素

- 条件分岐の次によく使う繰り返し構文
- Javaの拡張for文に相当

実習：挨拶を順番に表示

- あらかじめ用意しておいた「おはよう」「こんにちは」「おやすみ」の挨拶を順番に表示



実習：挨拶を順番に表示

```
<%@ page contentType="text/html;charset=UTF-8" %>
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
<% request.setCharacterEncoding("UTF-8"); %>
<!DOCTYPE html>
<html>
<head>
<meta charset="UTF-8" />
<title>JSP／サーブレット</title>
</head>
<body>
<c:set var="msgs" value="おはよう,こんにちは,おやすみ" />
<c:forEach var="msg" items="${msgs}">
  <p>${msg}、${param['name']}さん！</p>
</c:forEach>
</body>
</html>
```

output.jsp

実習：挨拶を順番に表示

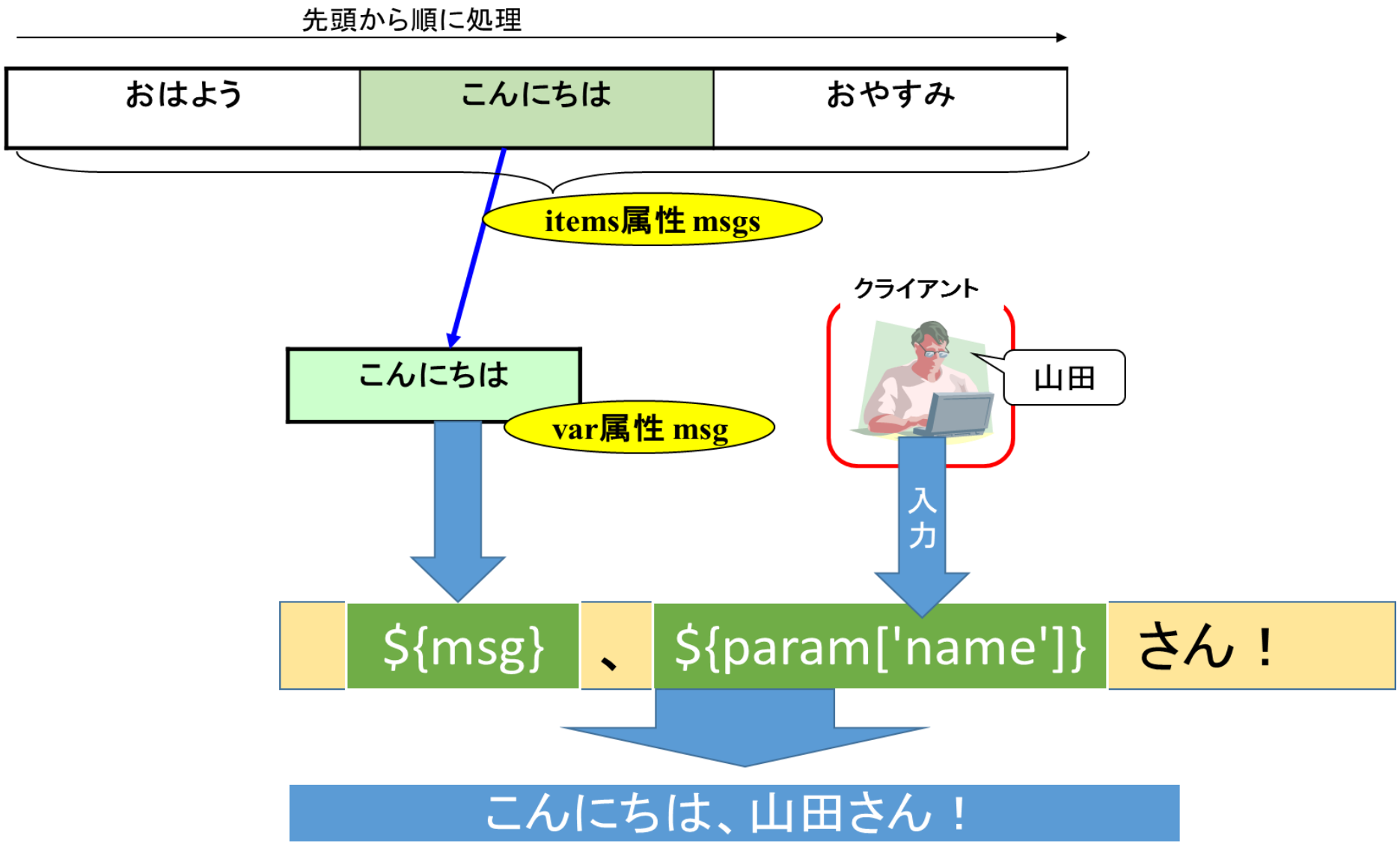
実行URL

http://localhost:8080/nikkei/input.jsp

<c:forEach>要素の使い方

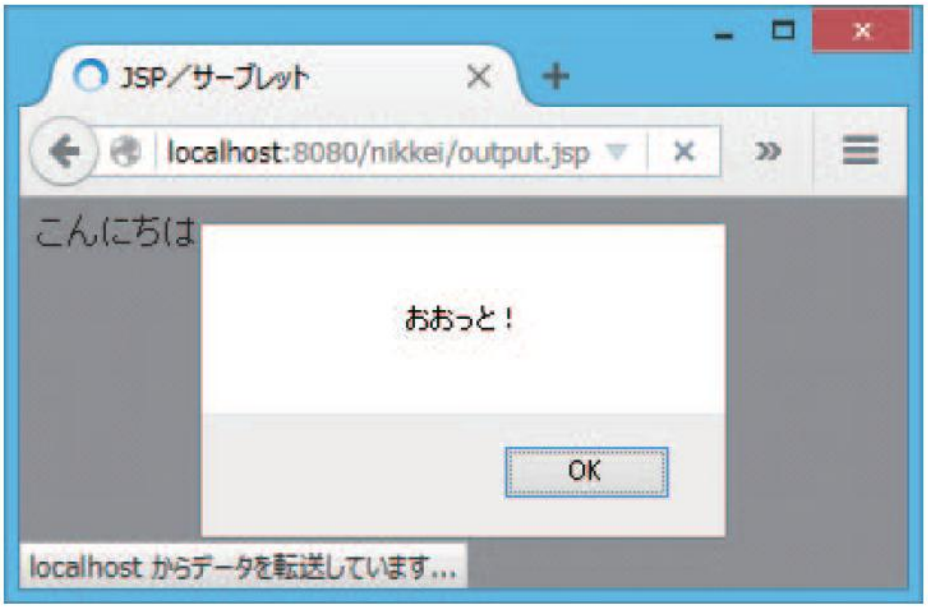
```
<c:forEach var="仮変数名"
  items="配列、文字列など">
  …
</c:forEach>
```

<c:forEach>要素のイメージ



実習：文字列をHTMLエスケープ

- 名前欄に「<script>alert('おおっと！');</script>」を入力すると･･･実行されてしまう！



この問題は「クロスサイトスクリプティング（XSS）脆弱性」と呼ばれる

- クッキー情報が盗み出される可能性や、ページそのものを書き換えられてしまう可能性がある
- XSS脆弱性はHTMLエスケープ処理で防ぐ

HTMLエスケープ処理

- プログラムによって生成されるすべての文字列に対して「HTMLエスケープ処理」を施す
- 「<」や「>」「&」のようなHTMLの予約文字を「<」や「>」「&」のような無害な文字列に置き換えること
- Functionsライブラリの「fn:escapeXml関数」で簡単に行える

実習：output.jspをHTMLエスケープするように修正

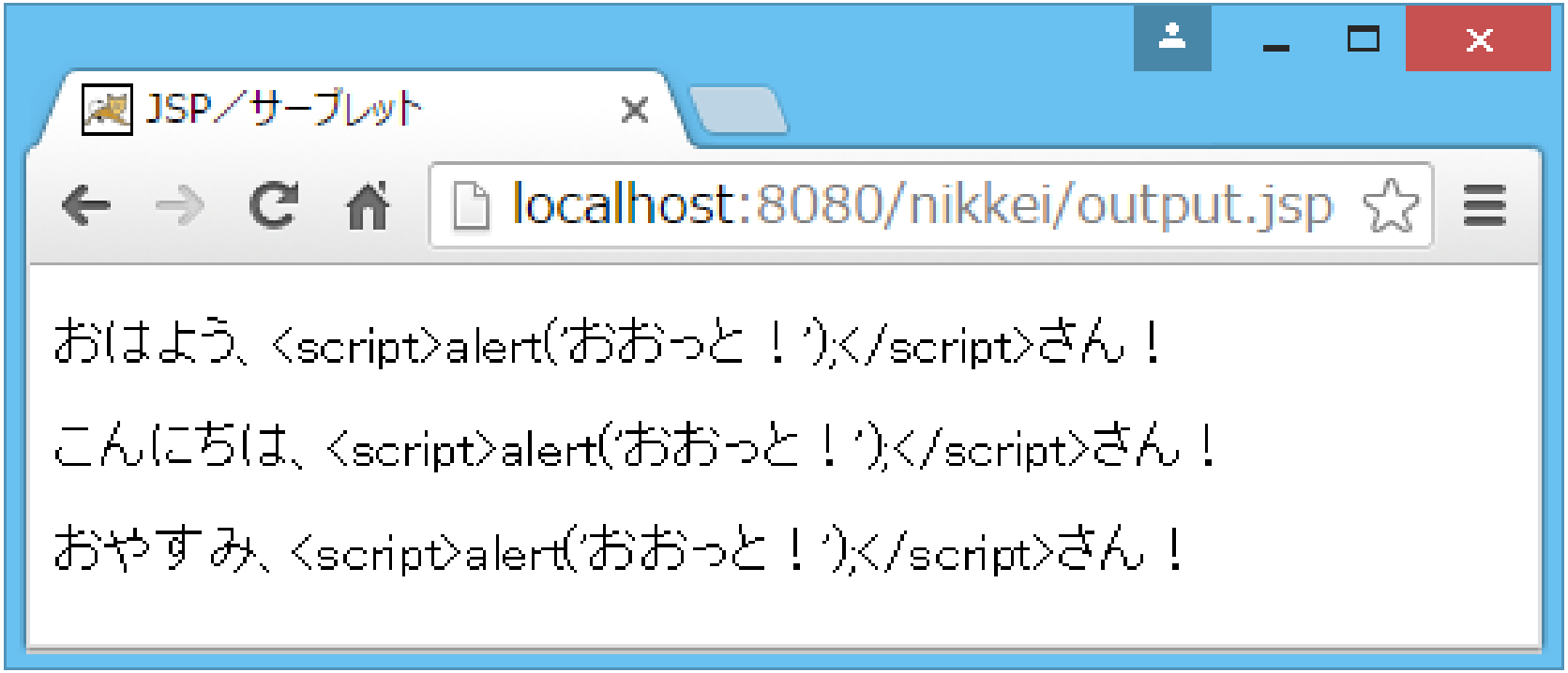
```
<%@ page contentType="text/html;charset=UTF-8" %>
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
<%@ taglib prefix="fn" uri="http://java.sun.com/jsp/jstl/functions" %>
<% request.setCharacterEncoding("UTF-8"); %>
<!DOCTYPE html>
<html>
<head>
<meta charset="UTF-8" />
<title>JSP／サーブレット</title>
</head>
<body>
<c:set var="msgs" value="おはよう,こんにちは,おやすみ" />
<c:forEach var="msg" items="${msgs}">
  <p>${msg}、${fn:escapeXml(param['name'])}さん！</p>
</c:forEach>
</body>
</html>
```

実習：output.jspをHTMLエスケープするように修正

実行URL

http://localhost:8080/nikkei/input.jsp

JavaScriptのコードの実行を防ぐことができた



質疑応答

第3部　手を動かして学ぶ

Web-DBアプリ開発の基本を復習 サーブレット編

13:30～14:50
＜80分＞

実習：サーブレットの基本

- JSPでできることは必ずサーブレットでもできる！
- output.jspをサーブレットOutputServlet.javaとして書き直す
- UTF-8でソースコードを記述する
- .javaファイルは以下に配置

%CATALINA_HOME%¥webapps¥nikkei¥WEB-INF¥src¥to¥msn¥wings¥nikkei

OutputServlet.java（1）

```
package to.msn.wings.nikkei;

import java.io.IOException;
import java.io.PrintWriter;

import javax.servlet.ServletException;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
```

OutputServlet.java（2）

```
@WebServlet("/OutputServlet")
public class OutputServlet extends HttpServlet {
  @Override
  protected void doPost(HttpServletRequest request,
    HttpServletResponse response) throws ServletException, IOException {

    request.setCharacterEncoding("UTF-8");
    response.setContentType("text/html;charset=UTF-8");
    PrintWriter out = response.getWriter();
    out.print("<!DOCTYPE html><html><head><meta charset='UTF-8' />");
    out.print("<title>JSP／サーブレット</title>");
    out.print("</head><body>");
    out.print("<p>こんにちは、"+request.getParameter("name")+"さん！</p>");
    out.print("</body></html>");
  }
}
```

input.jsp

```
<%@ page contentType="text/html;charset=UTF-8"
pageEncoding="UTF-8" %>
<!DOCTYPE html>
<html>
<head>
<meta charset="UTF-8" />
<title>JSP／サーブレット</title>
</head>
<body>
<form method="POST" action="OutputServlet">
  <label for="name">名前：</label>
  <input id="name" name="name" type="text" />
  <input type="submit" value="送信" />
</form>
</body>
</html>
```

OutputServlet.java をコンパイルする

```
> cd %CATALINA_HOME%¥webapps¥nikkei¥
WEB-INF¥src¥to¥msn¥wings¥nikkei
> javac -encoding UTF-8 OutputServlet.java
```

- 環境変数CLASSPATHで必要なクラスを認識
  - .;%CATALINA_HOME%¥lib¥servlet-api.jar;
  - %CATALINA_HOME%¥lib¥jsp-api.jar;
  - %CATALINA_HOME%¥webapps¥nikkei¥WEB-INF¥classes
- javacのデフォルトの文字コードはプラットフォームのデフォルト（WindowsであればShift-JIS）。
  - -encodingオプションを使って、ソースコードと同じUTF-8に変更しておく！

コンパイルに失敗したら

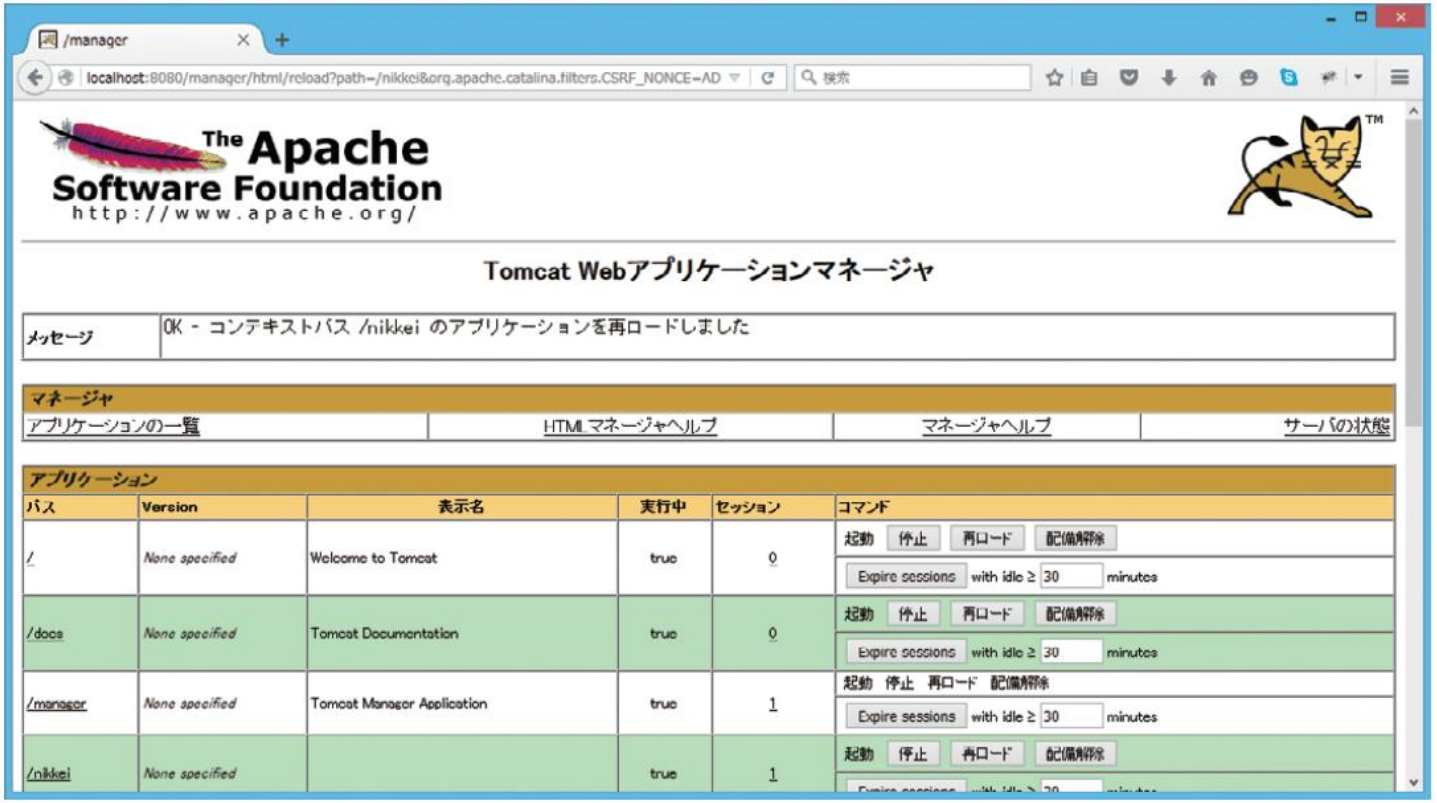
- コマンドが見つかりません
  - 環境変数PATHは？
- Hoge.javaを読み込めません
  - カレントフォルダーは大丈夫？
- クラスHogeはpublicであり、ファイルHoge.javaで～
  - クラス名とファイル名は一致している？
- パッケージHogeが存在しません
  - 環境変数CLASSPATHは？
- シンボルを解決できません
  - 環境変数CLASSPATHは？
  - 変数名、メソッド名が間違っていない？

OutputServlet.classを配置する

- ちょっと面倒
- 「package to.msn.wings.nikkei;」と宣言しているので、これに合わせる
- 例えば、「C:¥Apache Software Foundation¥Tomcat 8.0¥webapps¥nikkei¥WEB-INF¥classes¥to¥msn¥wings¥nikkei」フォルダーに配置

Tomcat Webアプリケーションマネージャでアプリを再起動する

- 「http://localhost:8080/manager/」にアクセス。
  「/nikkei」の行にある「再ロード」ボタンをクリック

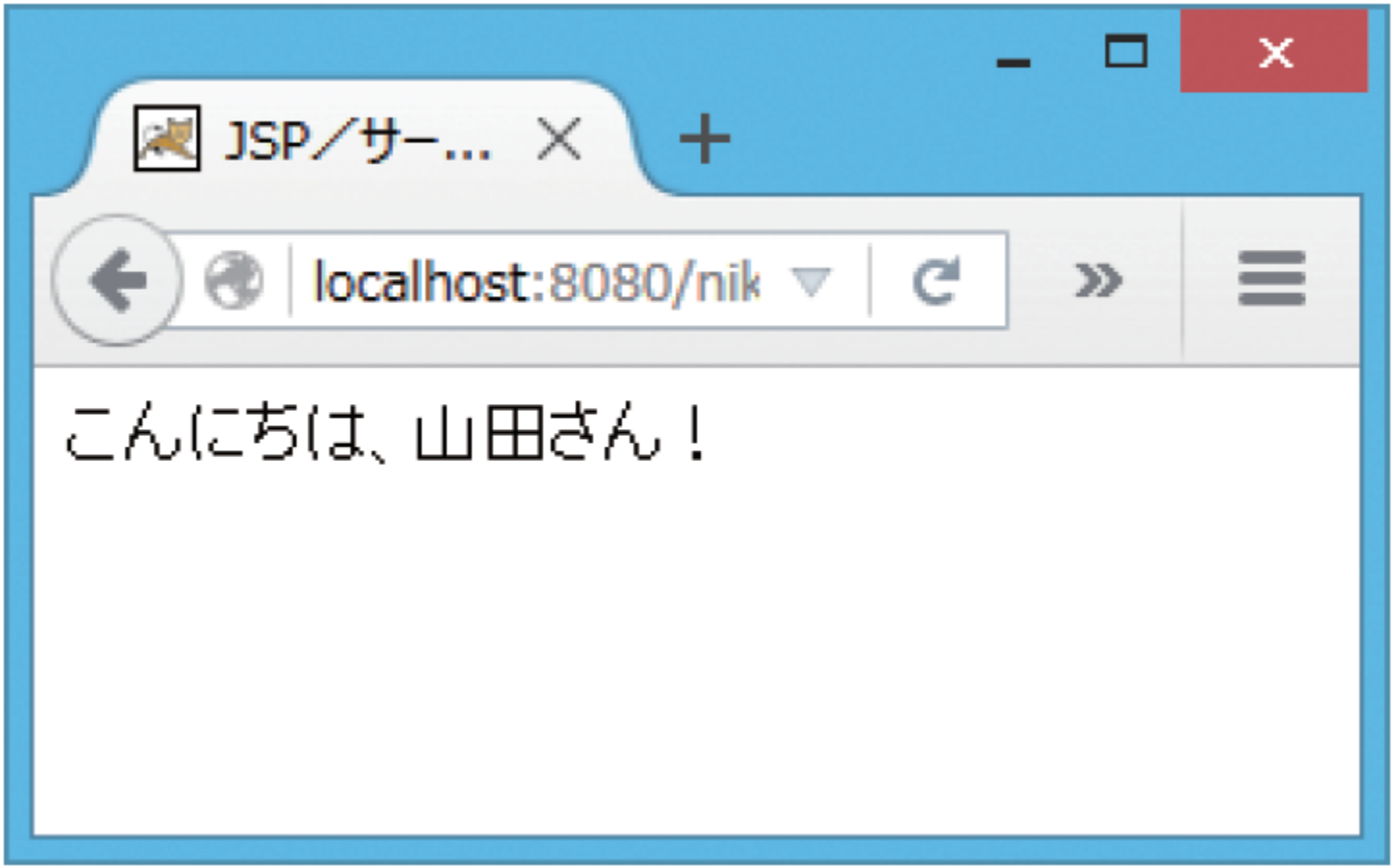


実習：サーブレットの基本

実行URL

http://localhost:8080/nikkei/input.jsp

OutputServlet.javaとinput.jspの実行例



OutputServlet.javaのポイント（1）

- サーブレットクラスはHttpServletクラスを継承しなければならない
- HttpServletを継承したクラスは、1つ以上の「do〇〇メソッド」を持っていなければならない
  - 呼び出しのHTTPメソッドによって決定
- 「doGetメソッド」か「doPostメソッド」。これらのメソッドの中で、受け取った値に対する処理を行う

OutputServlet.javaのポイント（2）

- do〇〇メソッドは、リクエスト情報を表すHttpServletRequestと、レスポンスを操作するためのHttpServletResponseオブジェクトを引数として受け取る
- サーブレットでは、これらのオブジェクトを利用して、データを受け取り、出力を生成する
- 出力には、PrintWriterオブジェクトを利用する
  - 暗黙オブジェクトoutに相当

@WebServletアノテーション

- サーブレットを呼び出すためのパスの設定
- 「@WebServlet(“/OutputServlet”)」と記述すると、「http://localhost:8080/nikkei/OutputServlet」のパスで、サーブレットを呼び出せるようになる

@Overrideアノテーション

- スーパークラスのメソッドをオーバーライドしていることを宣言
  - 付けなくてもエラーにはならない
- メソッドの名前や引数の型に誤りがある場合に、コンパイラによる警告が発生

実習:データベースを準備する

- 本格的なWebアプリの開発で欠かせないのが、データを管理するソフトウエアであるデータベース（DB）
- 今回は「MySQL」を使う
- MySQLの操作には、「mysqlクライアント」を利用する

mysqlクライアントを起動

> mysql -u root -p

Enter password:

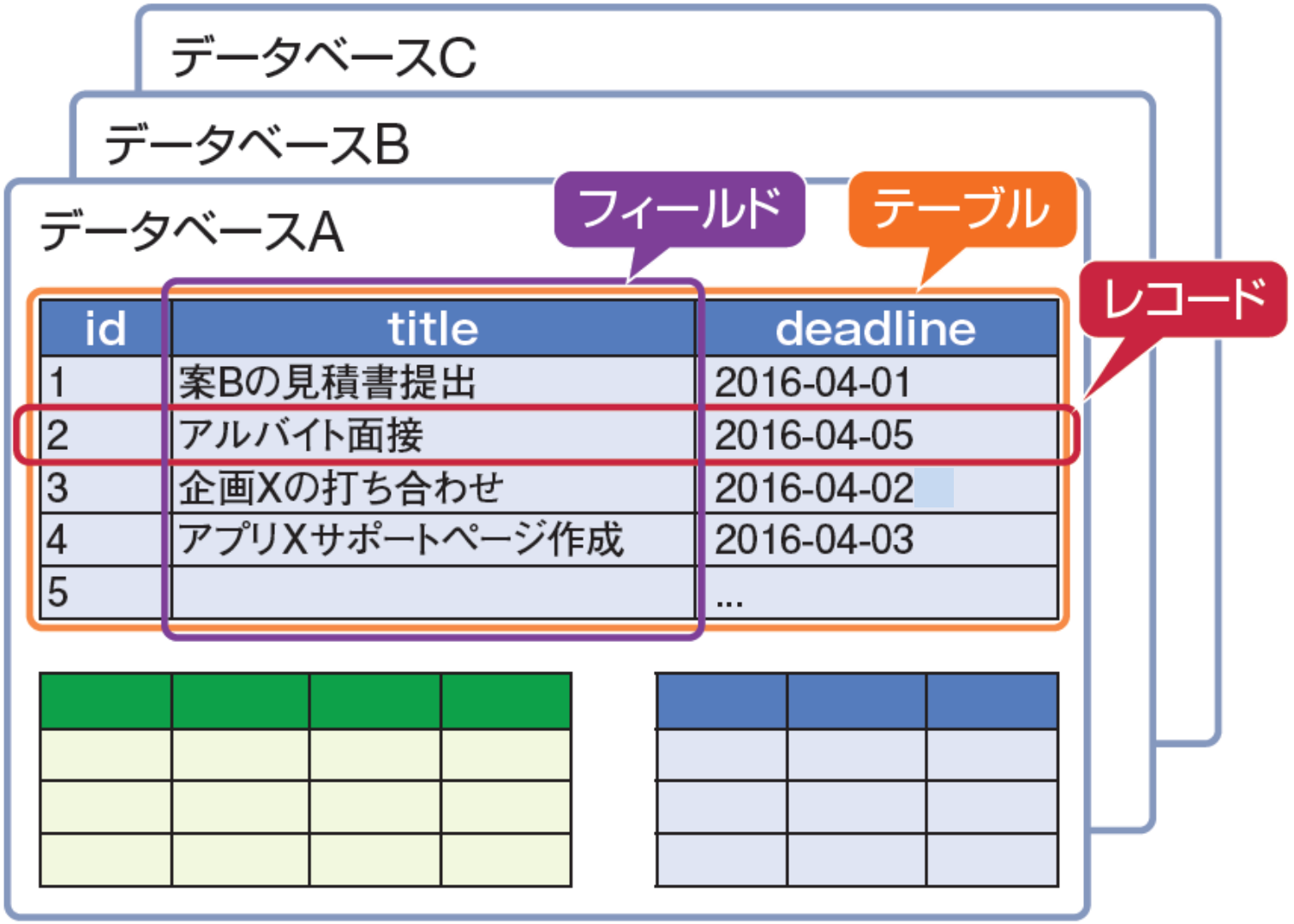
Welcome to the MySQL monitor. Commands end with ; or ¥g.

...中略...

Type 'help;' or '¥h' for help. Type '¥c' to clear the current input statement.

mysql>

データベースの構成要素(1)



データベースの構成要素(2)

- 1つのデータベースには1つ以上の「テーブル」がある
- テーブルは2次元の表で、データの項目である「フィールド」(列)と、データそのものの「レコード」で構成される
- SQL言語で操作する

実習：nikkeiデータベースを作成

```
mysql> CREATE DATABASE nikkei;
mysql> USE nikkei;
```

実習：todoテーブルを作成

```
mysql> CREATE TABLE todo(
id INT PRIMARY KEY AUTO_INCREMENT,
title VARCHAR(50),
deadline VARCHAR(10));
```

todoテーブル

id (整数型)	title (文字列型)	deadline (文字列型)

テーブルを作る構文(1)

```
CREATE TABLE テーブル名 (
    フィールド名 データ型 列フラグ,
    フィールド名 データ型 列フラグ,
     …
)
```

テーブルを作る構文(2)

- データ型には整数の「INT」や文字列の「VARCHAR(最大文字数)」などを指定
- 列フラグは、フィールドの性質を表す付加情報。例えば、列フラグで「PRIMARY KEY」を指定すると、そのフィールドが各レコードを一意に表すための列になる
- PRIMARY KEYはレコードのIDのようなもので、必ず用意する

AUTO_INCREMENT

- 「PRIMARY KEY AUTO_INCREMENT」の指定によって、idフィールドの値はレコードを追加するたびに、1、2、3…のように＋1される

todoテーブルを削除する

- 作成済みのテーブルを作り直したいときは、「DROP TABLE」命令でテーブルを削除する

mysql> DROP TABLE todo;

レコードの挿入／検索／更新／削除

- レコードの挿入には「INSERT」命令を使う
- レコードの検索には「SELECT」命令を使う
- レコードの更新には「UPDATE」命令を使う
- レコードの削除には「DELETE」命令を使う

レコードの挿入には「INSERT命令」を使う

- **INSERT命令の構文**

  INSERT テーブル名 (フィールド名,...)

  VALUES (値, ...)

実習 : 3つのレコードを挿入する例

mysql> INSERT INTO todo (id, title, deadline) VALUES (1, '案Aの見積書提出', '2016-03-31');

mysql> INSERT INTO todo (title, deadline) VALUES ('アルバイト面接', '2016-04-05');

mysql> INSERT INTO todo VALUES (3, '企画Xの打ち合わせ', '2016-04-20');

SELECT命令でテーブルを表示する

- **SELECT命令の構文**

  SELECT フィールド名, …

  FROM テーブル名

  [条件句]

実習:テーブルの全レコードを表示

mysql> SELECT id, title, deadline FROM todo ORDER BY id;

```
mysql> SELECT id, title, deadline FROM todo ORDER BY id;
+----+--------------------+------------+
| id | title              | deadline   |
+----+--------------------+------------+
|  1 | 案Aの見積書提出    | 2016-03-31 |
|  2 | アルバイト面接     | 2016-04-05 |
|  3 | 企画xの打ち合わせ  | 2016-04-20 |
+----+--------------------+------------+
3 rows in set (0.00 sec)
```

実習：指定したレコードを表示

mysql> SELECT id, title, deadline FROM todo WHERE id = 2;

```
mysql> SELECT id, title, deadline FROM todo WHERE id = 2;
+----+----------------+------------+
| id | title          | deadline   |
+----+----------------+------------+
|  2 | アルバイト面接     | 2016-04-05 |
+----+----------------+------------+
1 row in set (0.02 sec)
```

WHERE

- 条件句で「WHERE」を使うと、条件に合致したレコードだけを取り出せる。例えば、「WHERE id = 2」とすれば、idが2のレコードだけが表示される。
- つまり、WHEREを使うとテーブル内の検索が可能になる

レコードの更新には「UPDATE命令」を使う

- **UPDATE命令の構文**

  UPDATE テーブル名

  SET フィールド名 = 値, ...

  WHERE 条件式

指定のレコードの内容を更新

```
mysql> UPDATE todo
SET title = '案Bの請求書提出',
deadline = '2016-04-01'
WHERE id = 1;
```

レコードの削除には「DELETE命令」を使う

- **DELETE命令の構文**

  DELETE FROM テーブル名

  WHERE 条件式

指定のレコードを削除

mysql> DELETE FROM todo WHERE id = 3;

- DELETE命令でWHEREの条件句を省略すると、テーブル内のすべてのデータが削除されてしまうので、要注意

質疑応答

第4部　手を動かして学ぶ

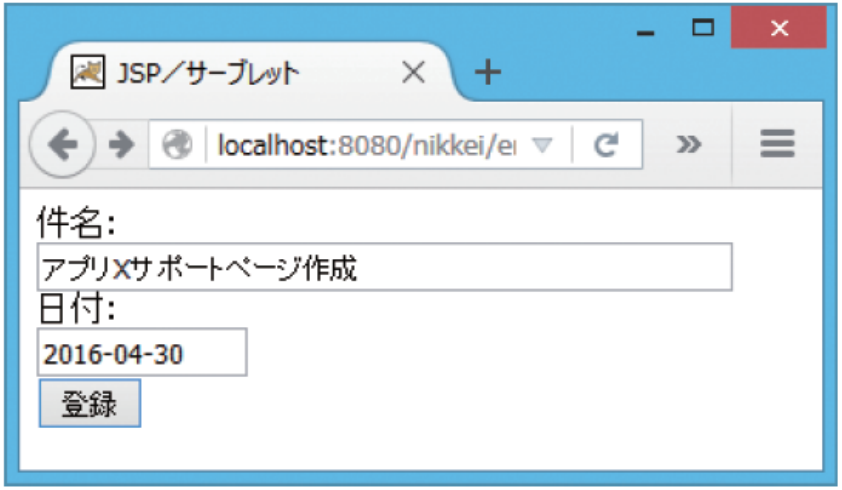
Tomcat＋MySQLでWeb-DBアプリを実際に開発してみよう

15:10～16:30
＜80分＞

実習：「TODOアプリ」を作ってみよう

- Webベースの「TODOアプリ」（予定表アプリ）を作る
- データベースは第３部で作成したnikkeiデータベースのtodoテーブルをそのまま利用

TODOアプリの完成イメージ



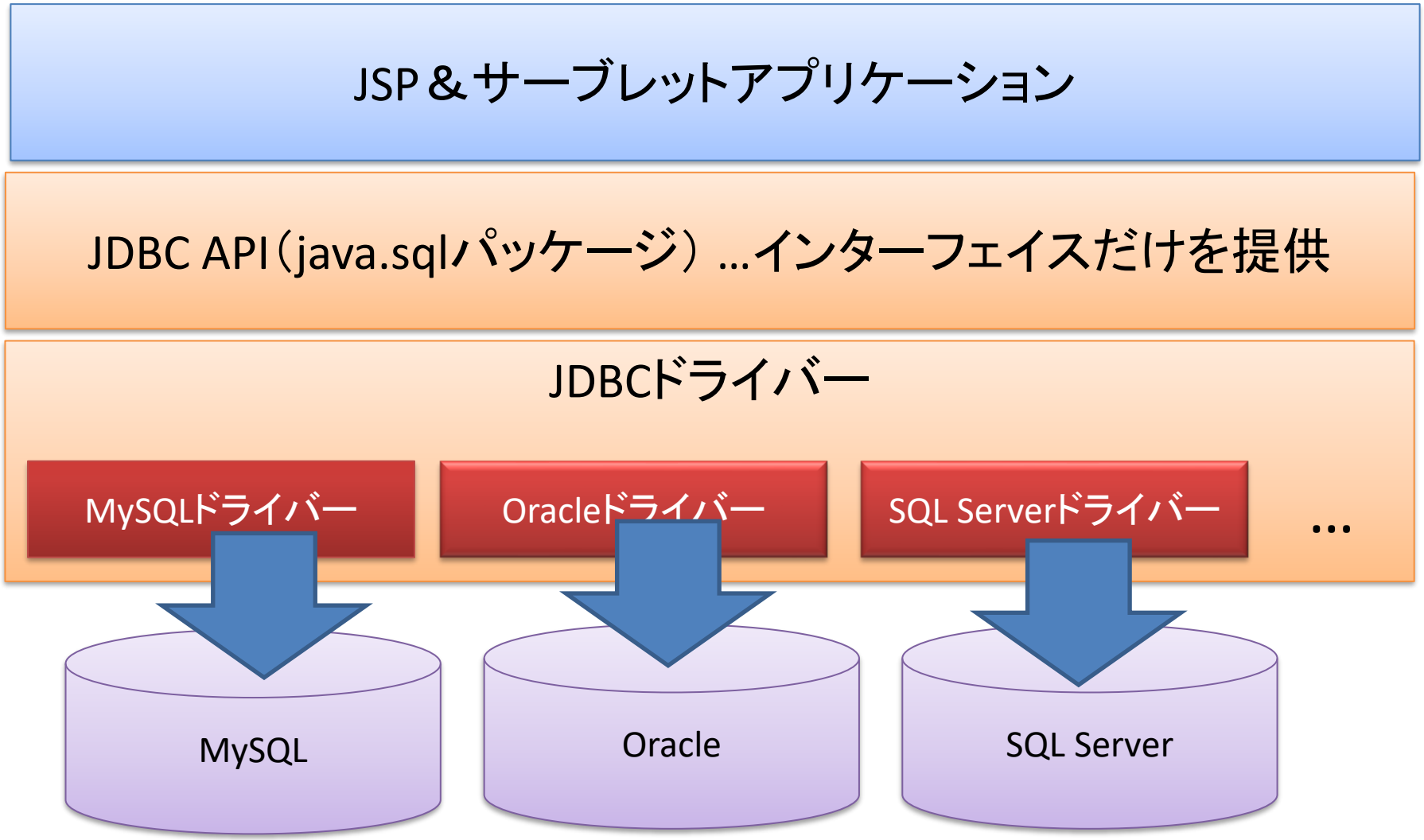
TODOの入力画面

TODOのリスト表示

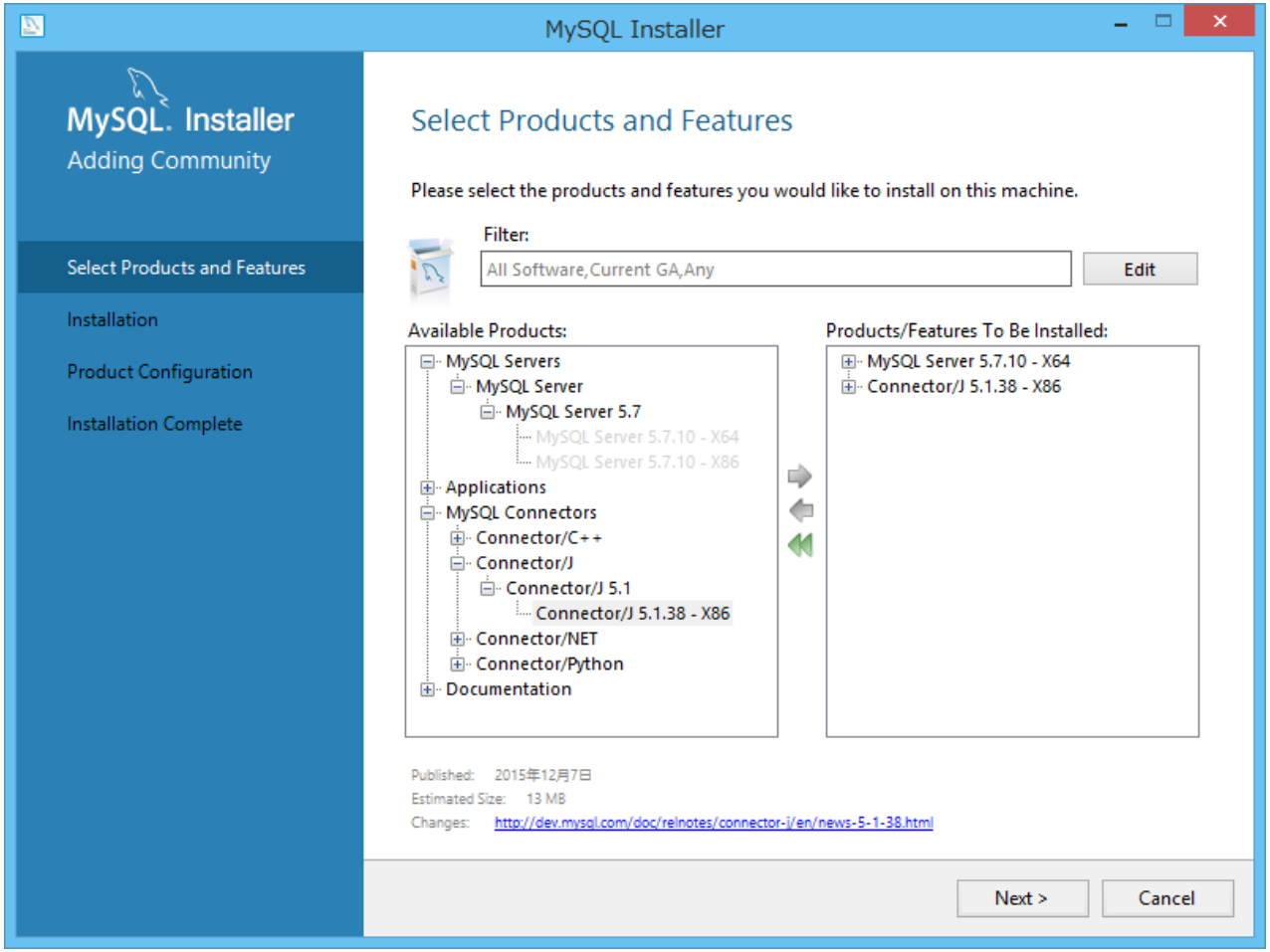
JavaのプログラムからMySQLに接続するためには？

- 標準のJavaSEだけではダメ
- MySQL専用の「JDBCドライバ」を導入する

JDBCドライバとは？


JSP＆サーブレットアプリケーション
JDBC API（java.sqlパッケージ）…インターフェイスだけを提供
JDBCドライバー
MySQLドライバー
Oracleドライバー
SQL Serverドライバー
…
MySQL
Oracle
SQL Server

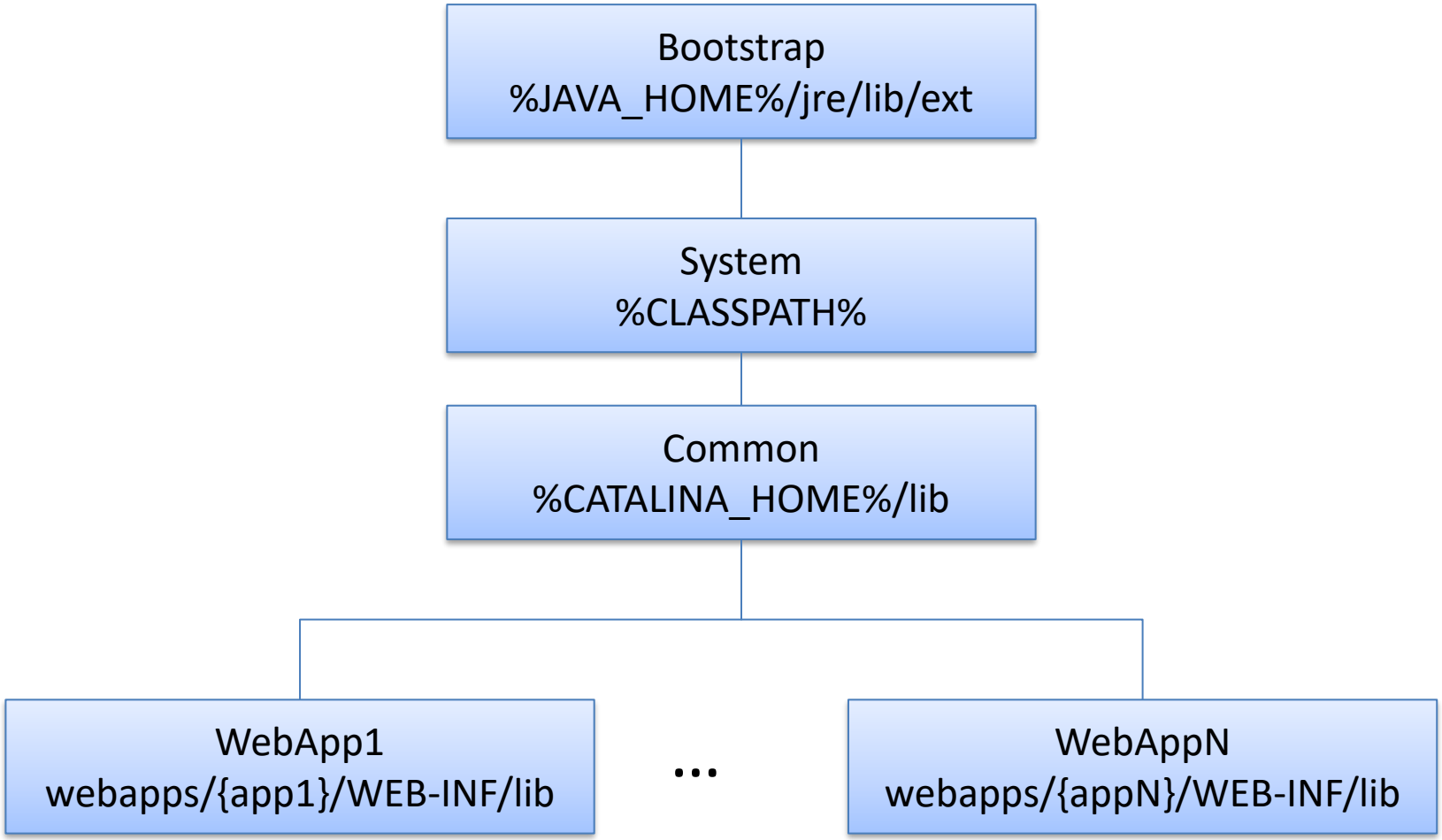
MySQLと同時にインストールするConnector/J



TomcatからConnector/Jを利用できるようにする

- 「C:¥Program Files (x86)¥MySQL¥Connector.J 5.1」フォルダーにある「mysql-connectorjava-5.1.38-bin.jar」を、「C:¥Apache Software Foundation¥Tomcat 8.0¥lib」フォルダーにコピー

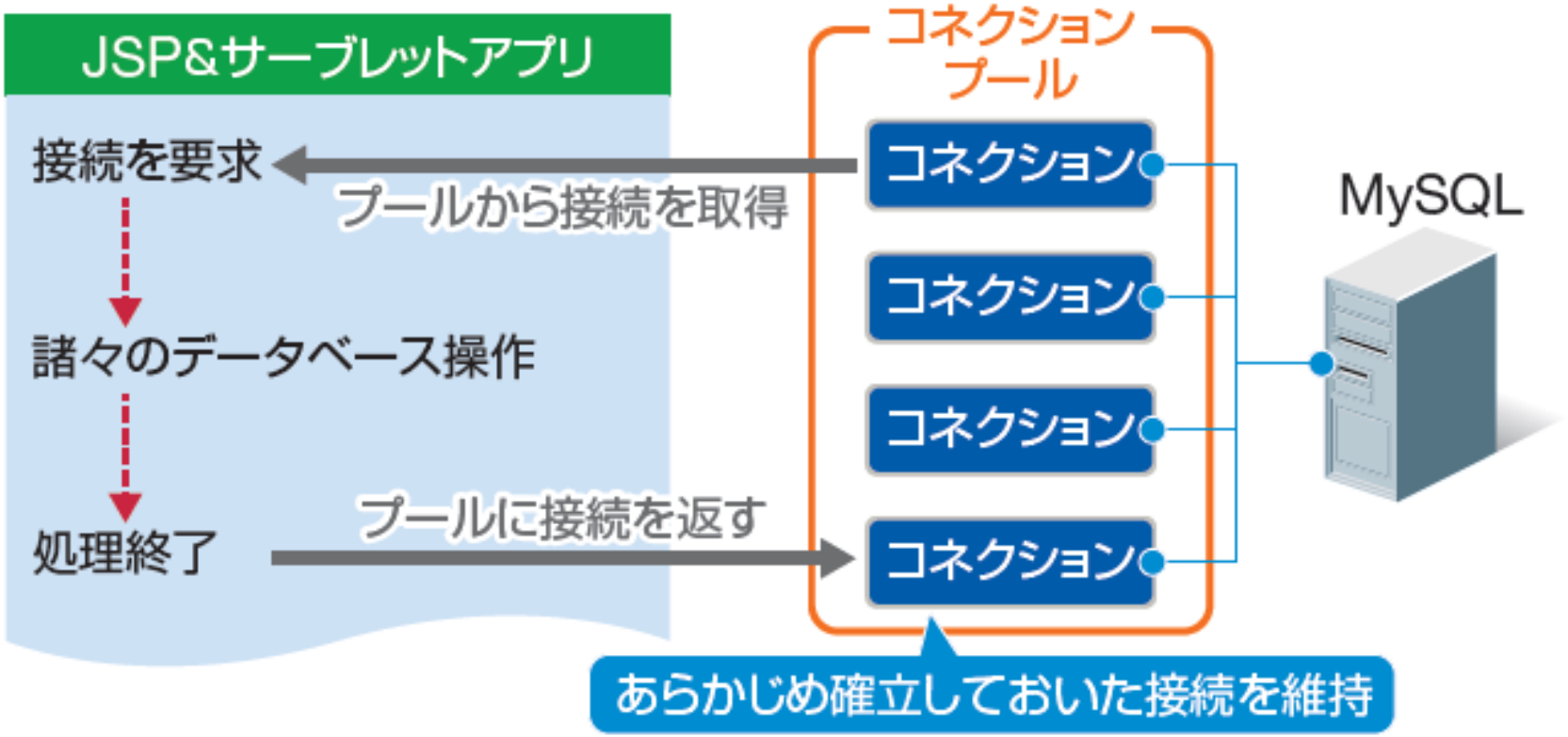
補足：クラスローダー


Bootstrap
%JAVA_HOME%/jre/lib/ext
System
%CLASSPATH%
Common
%CATALINA_HOME%/lib
WebApp1
webapps/{app1}/WEB-INF/lib
…
WebAppN
webapps/{appN}/WEB-INF/lib

負荷軽減のために「コネクションプール」を使う

- DBを利用するには、まずDBへの“接続”を確立する処理が必要
- Webアプリでは、不特定多数のユーザーが同時に、絶え間なく、DBへの接続を要求することになる
- 要求の都度、接続処理を繰り返すのは、限られたコンピュータ資源の浪費
- そこで、Tomcatが用意する「コネクションプール」の機能を使う

コネクションプールの仕組み（1）



コネクションプールの仕組み(2)

- DBへの接続をあらかじめ“プール”(保管)しておき、アプリで接続が必要になったら、そこから取り出す
- 使用済みの接続は再びプールに戻して、再利用する
- コネクションプールによって、接続処理のオーバーヘッドを大幅に軽減できる

実習：コネクションプールに対応する Tomcatの設定ファイル「context.xml」を作成

```
<?xml version="1.0" encoding="UTF-8" ?>
<Context displayName="Nikkei" docBase="nikkei" path="/nikkei"
 reloadable="true">
 <Resource name="jdbc/nikkei" auth="Container"
  type="javax.sql.DataSource"
  username="root" password="root"
  driverClassName="org.gjt.mm.mysql.Driver"
  url="jdbc:mysql://localhost/nikkei?useUnicode=true&amp;characterEncoding=UTF-8"
  maxActive="4" maxWait="5000" maxIdle="2"
  validationQuery="SELECT count(*) FROM todo" />
</Context>
```

UTF-8で記述

context.xmlを配置

No.	配置先	影響範囲
1	%CATALINA_HOME%/conf/server.xml	すべてのアプリ
2	%CATALINA_HOME%/conf/context.xml	すべてのアプリ
3	%CATALINA_HOME%/conf/＜エンジン名＞/＜ホスト名＞/context.xml.default	指定されたホストのすべてのアプリ
4	%CATALINA_HOME%/conf/＜エンジン名＞/＜ホスト名＞/＜アプリケーション名＞.xml	指定されたアプリ
5	**%CATALINA_HOME%/webapps/＜アプリケーション名＞/META-INF/context.xml**	指定されたアプリ

<Resource>要素の各属性の意味

属性	概要
name	接続（データソース）の名前
auth	リソースの制御方法（この場合は「Container」で固定）
type	リソースのデータ型（この場合は「javax.sql.DataSource」で固定）
driverClassName	JDBCドライバの完全修飾名（使用するドライバで固定）
url	接続文字列
username	接続に使用するユーザー名
password	接続に使用するパスワード
maxActive	プールする最大接続数
maxIdle	アイドル時に維持する最小の接続数
maxWait	接続に対する最大待ち時間（ミリ秒）
validationQuery	接続検証用のSQL命令

context.xmlの補足

- 接続文字列は「jdbc:mysql://ホスト名/データベース名?オプション」
- validationQuery属性は、プールしている接続が切れていないかどうかを確認するときに使うSQL命令の設定
- reloadable属性をtrueに。これで第３部で行ったアプリの再起動の作業が不要になる。ただし、この設定はTomcatへの負荷が大きいので、テスト環境でのみ使う

実習：Todoの登録画面を作成

- テキストボックスを表示するためのJSPのプログラム「entry.jsp」
- entry.jspからHTTP POSTでテキストボックスの値を受け取って、todoテーブルに挿入するサーブレットのプログラム「InsertServlet.java」

entry.jsp

```
<%@ page contentType="text/html;charset=UTF-8" %>
<!DOCTYPE html>
<html>
<head>
<meta charset="UTF-8" />
<title>JSP／サーブレット</title>
</head>
<body>
<form method="POST" action="InsertServlet">
 <div>
  <label for="title">件名：</label><br />
  <input id="title" name="title" type="text" size="50" maxlength="100" />
 </div>
 <div>
  <label for="deadline">日付：</label><br />
  <input id="deadline" name="deadline" type="text" size="12" maxlength="10" />
 </div>
 <div>
  <input type="submit" value="登録" />
 </div>
</form>
</body>
</html>
```

InsertServlet.java（1）

```
package to.msn.wings.nikkei;

import java.io.IOException;
import java.io.PrintWriter;

import java.sql.Connection;
import java.sql.PreparedStatement;
import java.sql.SQLException;

import javax.naming.Context;
import javax.naming.InitialContext;
import javax.naming.NamingException;

import javax.sql.DataSource;

import javax.servlet.ServletException;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
```

InsertServlet.java（2）

```
@WebServlet("/InsertServlet")
public class InsertServlet extends HttpServlet {
  @Override
  protected void doPost(HttpServletRequest request, HttpServletResponse response)
    throws ServletException, IOException {
    request.setCharacterEncoding("UTF-8");
    try{
      // データソースを取得
      Context context = new InitialContext();
      DataSource ds = (DataSource)context.lookup("java:comp/env/jdbc/nikkei");
```

InsertServlet.java（3）

```
    // データベースに接続＆INSERT命令を実行
    try(
      Connection db = ds.getConnection();
      PreparedStatement ps =
        db.prepareStatement("INSERT INTO todo(title, deadline) VALUES (?, ?)")
    ) {
      ps.setString(1, request.getParameter("title"));
      ps.setString(2, request.getParameter("deadline"));
      ps.executeUpdate();
    } catch(SQLException e) {
      e.printStackTrace();
    }
   } catch(NamingException e) {
    e.printStackTrace();
   }
  response.sendRedirect("entry.jsp");
 }
}
```

InsertServlet.javaをコンパイル

javac -encoding UTF-8 InsertServlet.java

- 作成されたInsertServlet.classを「C:¥Apache Software Foundation¥Tomcat 8.0¥webapps¥nikkei¥WEB-INF¥classes¥to¥msn¥wings¥nikkei」フォルダーに移動する

実行

「http://localhost:8080/nikkei/entry.jsp」にアクセスして実行

結果はmysqlクライアントで確認してみよう！

データベースへの接続処理（1）

- 「接続」（データソース）をlookupメソッドで取得
- lookupメソッドの戻り値はObject型なので、利用の際はDataSourceオブジェクトに型キャスト
- DataSourceオブジェクトのgetConnectionメソッドでConnectionオブジェクトを取得。これでMySQLとの接続を確立できる

データベースへの接続処理(2)

- Connectionオブジェクトは不要になったら、確実に解放する。そうしないと、処理を終えた後もしばらくは接続を占有し、別のリクエストを処理できなくなる → 接続リーク
- 確実に解放したいオブジェクトは、try - with - resources構文で扱う。tryブロックを出たところでリソース(AutoCloseable実装クラス)を解放
- JavaSE 7より前であれば、finally句を利用

データベースにSQL命令を発行する

- PreparedStatementオブジェクトを利用

- ConnectionオブジェクトのprepareStatementメソッドで、発行すべきSQL命令を指定することで生成する

“プレイスホルダー”を使う

- SQL命令の中にある「？」に注目
- これは、後から具体的な入力値をセットするための“プレイスホルダー”
- 後からset〇〇メソッドで値をセット
- セットする値に応じて、setBooleanやsetInt、setString、setTimeなどを使い分ける
- 第1引数にはプレイスホルダーの順番を、第2引数には入力値を指定
- 用意した命令はexecuteUpdateメソッドで実行

サーブレットは処理するだけ

- 画面に表示すべきものがない場合、サーブレットが適している

- 処理が終わった後は、HttpServlet#sendRedirectメソッドでJSPページに移動

文字列連結でSQL命令を作成してはいけない（1）

- プレイスホルダーを使わずにSQL命令を組み立てることもできるが・・・
- 例えば、

```
PreparedStatement ps = db.prepareStatement
("INSERT INTO todo(title, deadline) VALUES
('" + request.getParameter("title") +"', '"
+ request.getParameter("deadline") +"')");
```

文字列連結でSQL命令を作成してはいけない（2）

- 読みにくいし、SQLインジェクションの原因となる
- 例えば、件名の欄に、「**‘,’‘);DELETE FROM todo; --**」の文字列をエンドユーザーに入力されたら・・・
- 次のようなSQL命令が生成される

INSERT INTO todo (title, deadline) VALUES

('','');DELETE FROM todo; --','');

文字列連結でSQL命令を作成してはいけない(3)

- 「データを1 件挿入した後、todo テーブルの内容をすべて削除する」命令になる

SQLエスケープ（1）

- SQLインジェクション対策
- 文字列の開始と終了を表す「‘」を「’‘」に置き換える処理。連続した「’‘」は文字列の開始や終了ではなく、単一の「’」と見なされる
- 先ほどのSQL命令にSQLエスケープを施すと・・・

INSERT INTO todo (title, deadline) VALUES

('',''');DELETE FROM todo; --','');

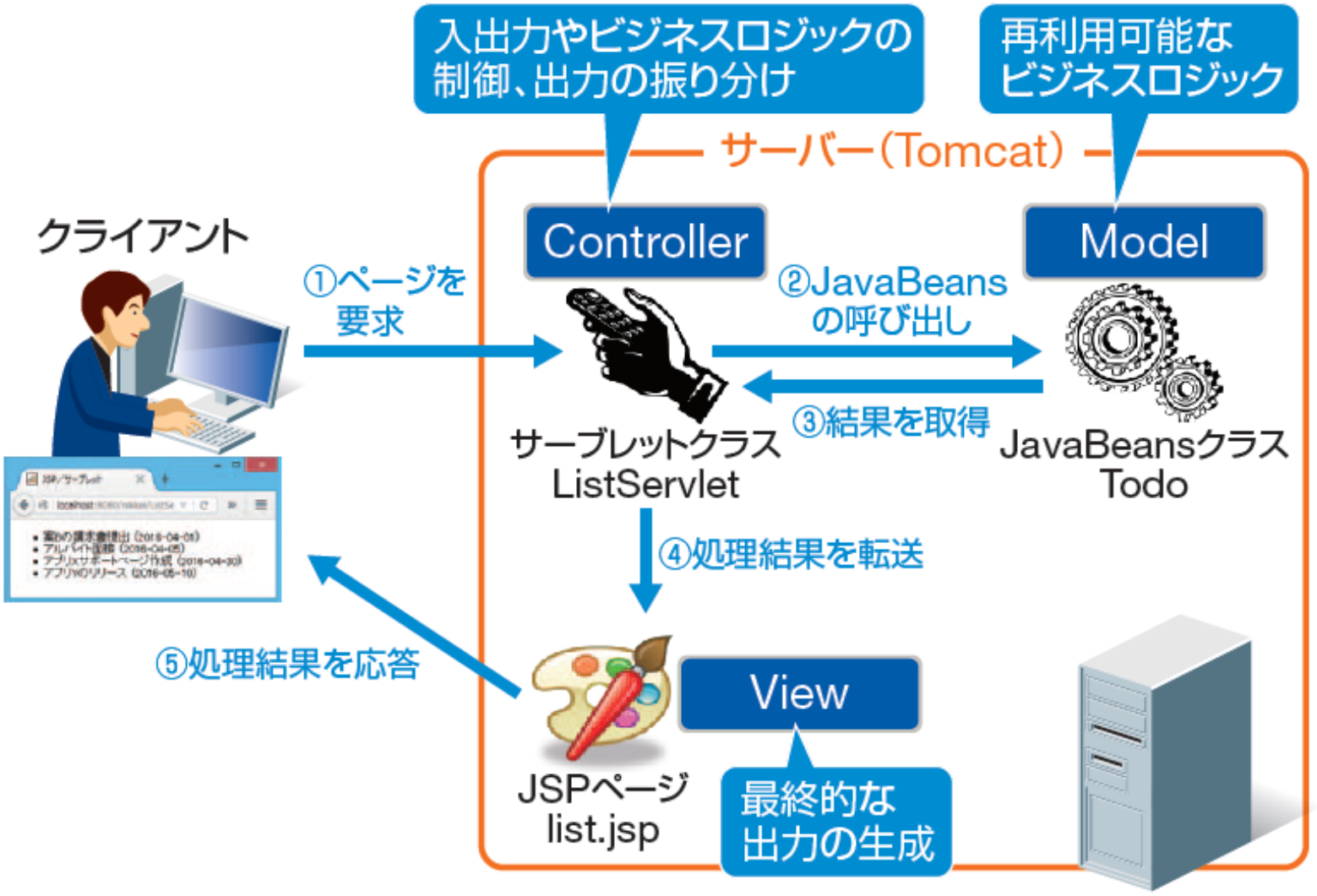
SQLエスケープ（2）

- 「VALUES (‘」の次から「’,‘’);」の前までは単なる文字列になるので、「DELETE FROM todo」は実行されない
- プレイスホルダーを利用すれば、SQLエスケープをPreparedStatementオブジェクトが内部的に行ってくれる
- 動的なSQL命令の組み立て処理には、必ずプレイスホルダーを利用すべき

実習：TODOリストを表示しよう

- ListServlet.java
  - ー Todoクラスを呼び出すサーブレット
- Todo.java
  - – Todoリストを取得するクラス
- list.jsp
  - – TodoリストをJSPページ

サンプルの構造を理解しよう(1)



サンプルの構造を理解しよう(2)

- JavaBeansでビジネスロジック(Model)を記述
- JSPでユーザーインタフェース(View)を実装
- 両者の間をサーブレットが取り持つ(Controller)
- このようなアプリの設計モデルのことを「Model-View-Controller」(MVC)と呼ぶ

コードの再利用性を高めるJavaBeans（1）

- プログラムには再利用できる処理とできない処理がある
- DBからデータを取得する処理や、DBにデータを登録する処理は、再利用しやすい
- テキストボックスからデータを取得する処理や、結果を表示する処理は、そのプログラム固有のもので、他のプログラムで再利用することはほぼない

コードの再利用性を高めるJavaBeans（2）

- 再利用できる処理はできるだけ独立させせた方が、後々のため
- 再利用できる処理と再利用できない処理を1つにまとめると、再利用しにくくなるので、望ましくない

コードの再利用性を高めるJavaBeans（3）

- 再利用できない「入力データの処理」と「処理結果の引き渡し」をサーブレットに残し、再利用できる処理を抜き出す
- 抜き出した処理は「JavaBeans」と呼ばれるクラスにまとめる

Todo.java（1）

```
package to.msn.wings.nikkei;

import java.io.IOException;
import java.io.Serializable;

import java.util.ArrayList;

import java.sql.Connection;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.SQLException;

import javax.naming.Context;
import javax.naming.InitialContext;
import javax.naming.NamingException;

import javax.sql.DataSource;
```

Todo.java（2）

```
public class Todo implements Serializable {
  public Todo(){ /* コンストラクタ */ }

  // アクセサメソッドを準備
  private int id;
  public void setId(int id) { this.id = id; }
  public int getId() { return this.id; }

  private String title;
  public void setTitle(String title) { this.title = title; }
  public String getTitle() { return this.title; }

  private String deadline;
  public void setDeadline(String deadline) { this.deadline = deadline; }
  public String getDeadline() { return this.deadline; }
```

Todo.java（3）

```
public static ArrayList<Todo> getInfos() {
  ArrayList<Todo> list = new ArrayList<>();

  try{
    // データソースを取得
    Context context = new InitialContext();
    DataSource ds = (DataSource)context.lookup("java:comp/env/jdbc/nikkei");

    // データベースに接続＆データを取得
    try(
      Connection db = ds.getConnection();
      PreparedStatement ps =
      db.prepareStatement("SELECT id, title, deadline FROM todo ORDER BY id");
      ResultSet rs = ps.executeQuery()
    ) {
```

Todo.java（4）

```
      // 結果セットの内容を順にTodoオブジェクトに詰め替え
      while(rs.next()) {
        Todo td = new Todo();
        td.setId(rs.getInt("id"));
        td.setTitle(rs.getString("title"));
        td.setDeadline(rs.getString("deadline"));
        list.add(td);
      }
    } catch(SQLException e) {
      e.printStackTrace();
    }
  } catch(NamingException e) {
    e.printStackTrace();
  }
  return list;
 }
}
```

コンパイル

- **javac -encoding UTF-8 Todo.java**

- Todo.javaをコンパイル。作成されたTodo.classは、「C:¥Apache Software Foundation¥Tomcat 8.0¥webapps¥nikkei¥WEB-INF¥classes¥to¥msn¥wings¥nikkei」フォルダーに格納

JavaBeansであるための条件

- 引数のないコンストラクタを持つこと
- プロパティを取得・設定するためのアクセサメソッドを持つこと
- JavaBeansをシリアライズするためのSerializableインタフェースを実装すること

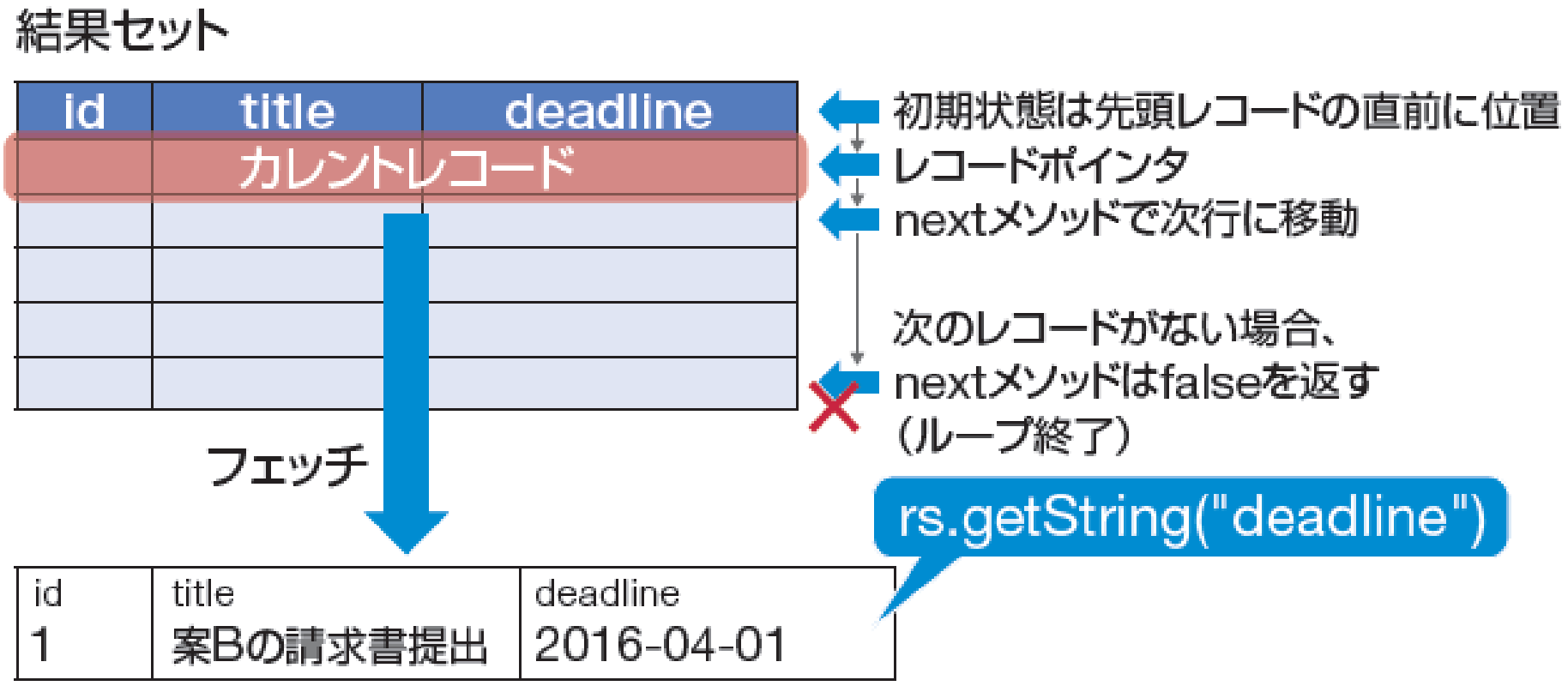
Todo.javaのポイント（1）

- TodoクラスがJavaBeansのクラス
- idプロパティ、titleプロパティ、deadlineプロパティはprivate
  - アクセサメソッドを用意
  - カプセル化
- データアクセスのためのメソッドを準備
  - SELECT命令はexecuteQueryメソッドで実行
  - executeQueryメソッドの戻り値は、「結果セット」（ResultSetオブジェクト）

Todo.javaのポイント（2）

- 結果セットとは、テーブルから取り出したデータを保持するために、メモリー上に用意された仮想テーブル
- 結果セットでは、1行単位にレコードを読んでいく
- 現在読み取り可能なレコード位置を表す「レコードポインタ」がある
- レコードポインタが示す現在行のことを「カレントレコード」と言う

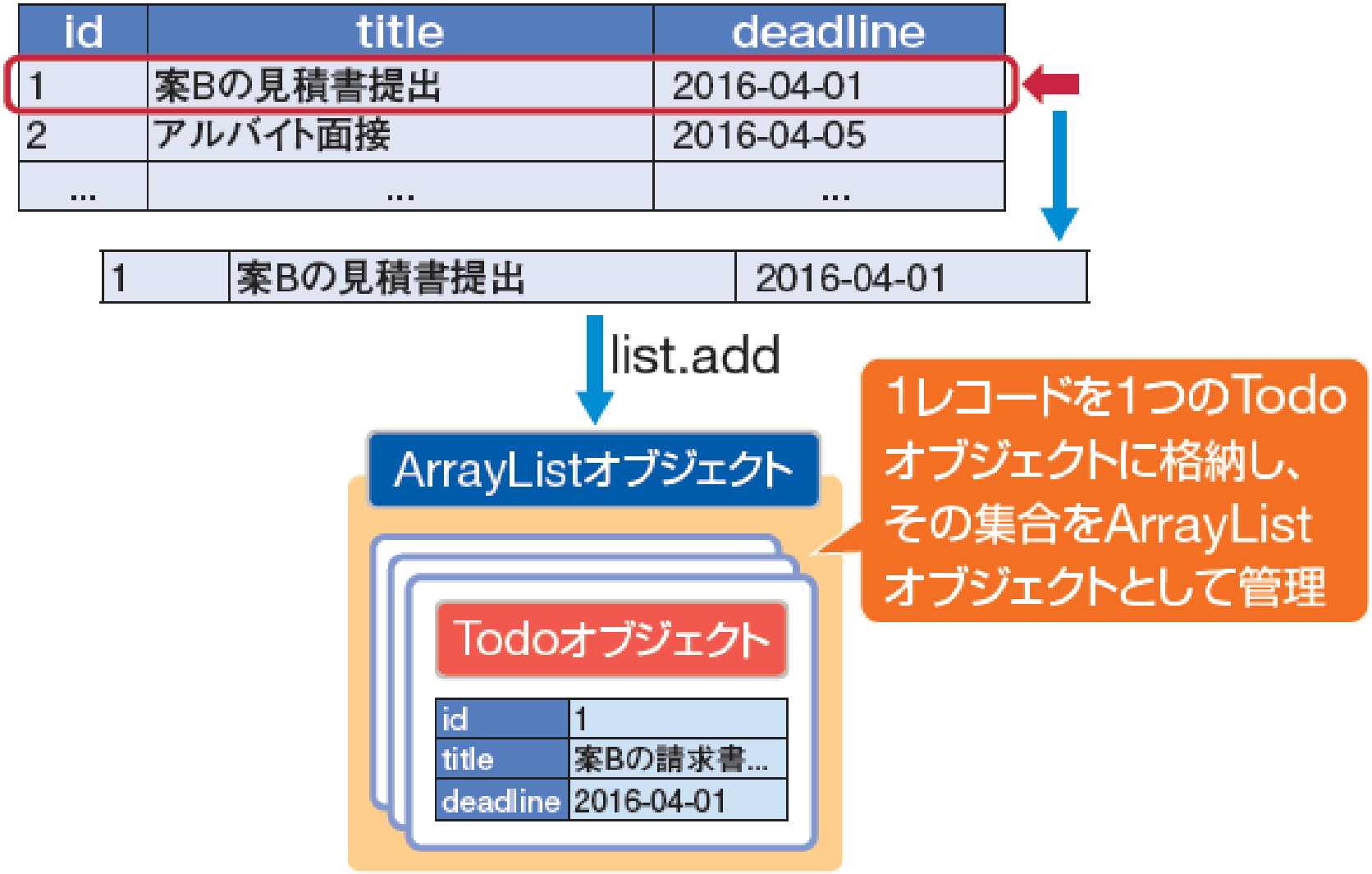
レコードポインタは結果セットの中のカレントレコードを指す



Todo.javaのポイント（3）

- 取得したデータは、Todoオブジェクトの対応するプロパティid、title、deadlineに設定した上で、ArrayListオブジェクトのlistに追加
- これを繰り返すと、ArrayListオブジェクトには、1つで1レコードの情報を表すTodoオブジェクトが、レコード数だけ格納される

MySQLのtodoテーブルのデータを、JavaのArrayListオブジェクトに詰め替える



ListServlet.java

```
package to.msn.wings.nikkei;

import java.io.IOException;
import javax.servlet.ServletException;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

@WebServlet("/ListServlet")
public class ListServlet extends HttpServlet {
  @Override
  protected void doGet(HttpServletRequest request, HttpServletResponse response)
   throws ServletException, IOException {
   // 処理結果をリクエスト属性にセット
   request.setAttribute("infos", Todo.getInfos());
   // JSPページに処理を転送
   getServletContext().getRequestDispatcher("/list.jsp").forward(request, response);
  }
}
```

```
<%@ page contentType="text/html;charset=UTF-8" %>
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
<%@ taglib prefix="fn" uri="http://java.sun.com/jsp/jstl/functions" %>
<% request.setCharacterEncoding("UTF-8"); %>
<!DOCTYPE html>
<html>
<head>
<meta charset="UTF-8" />
<title>JSP／サーブレット</title>
</head>
<body>
<ul>
<c:forEach var="info" items="${infos}">
 <li>
  ${fn:escapeXml(info.title)}
  （${fn:escapeXml(info.deadline)}）
 </li>
</c:forEach>
</ul>
</body>
</html>
```

list.jsp

コンパイル

- **javac -encoding UTF-8 ListServlet.java**

- ListServlet.javaをコンパイルし、作成されたListServlet.classを「C:¥Apache Software Foundation¥Tomcat 8.0¥webapps¥nikkei¥WEB-INF¥classes¥to¥msn¥wings¥nikkei」フォルダーに格納
- Todo.classが配置できていないとエラーになるので、要注意！

実行

「http://localhost:8080/nikkei/ListServlet」
にアクセスして実行

サーブレットでの処理結果をJSPに引き渡す

- JavaBeansのTodoクラスを使ってDBからデータを得られたら、これを表示するために、JSPのページにデータを引き渡す
- サーブレットを使うより、JSPの方がコンテンツの出力は手軽
- 「リクエスト属性」という仕組みを利用する

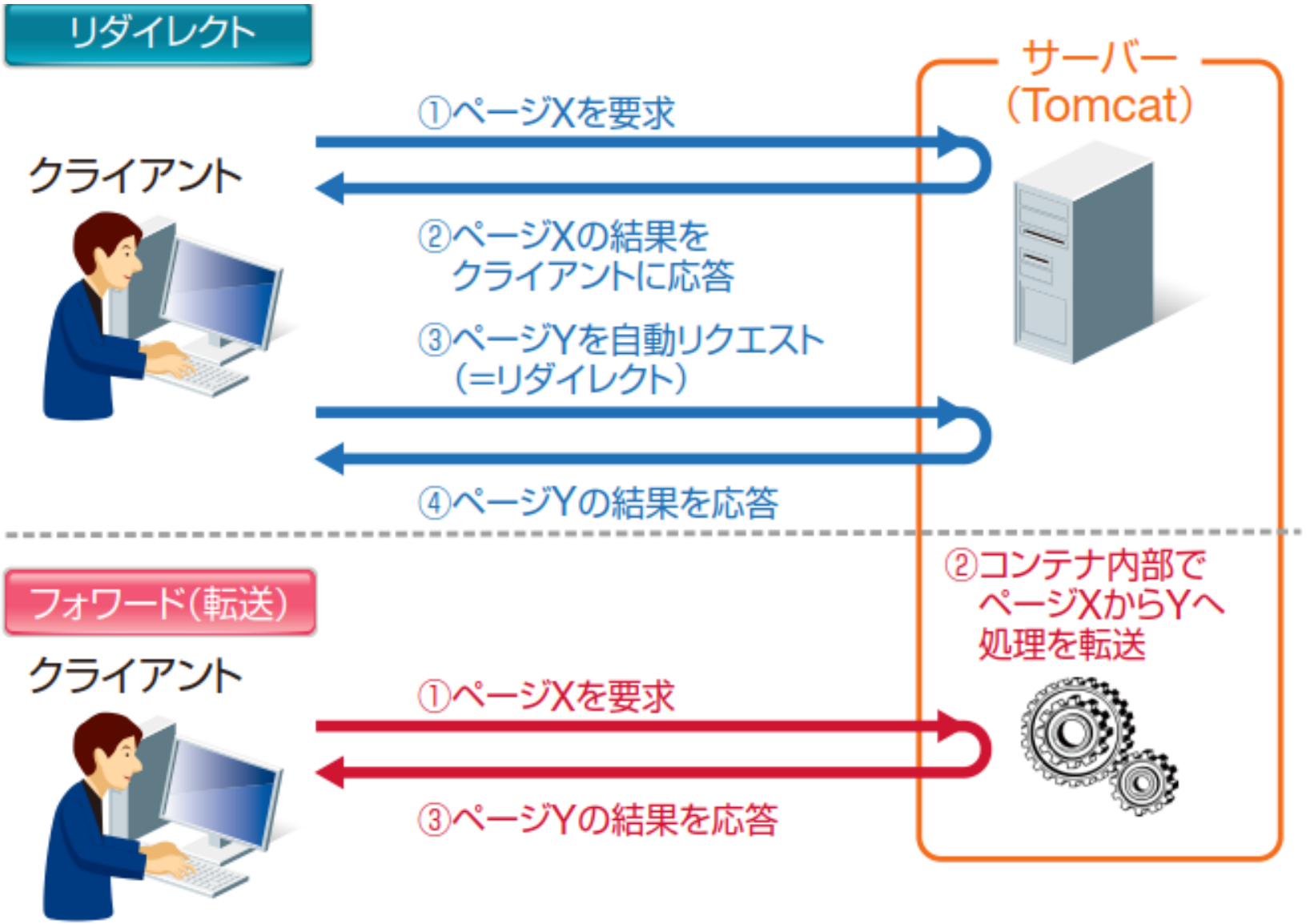
リクエスト属性

- 「サーブレットでリクエストを受けて、JSPページで結果を返すまで」の間、維持できる情報のこと
- サーブレットとJSP間で共有できる変数、と考えても良い
- HttpServletRequestオブジェクトのsetAttributeメソッドで設定する
- 後は、処理をJSPに転送（forward）する

転送されたリクエスト属性を取得

- list.jspの${infos}で受け取る
- infosの中身は、ArrayList<Todo>オブジェクト
  - 式言語は型を意識しなくても良い
- <c:forEach>要素でこれを順番に出力

リダイレクトとフォワード



リダイレクトとフォワード

- リダイレクトは内部的には2往復
- リダイレクトではリクエスト属性は引き継げない
- フォワードでは外のサイトに移動することはできない

質疑応答