

Webアプリケーション開発基礎

株式会社ナレッジエックス
<http://www.knowledge-ex.jp/>

Version 0.9.007

本ドキュメントについて



- この作品は、クリエイティブ・コモンズの表示-改変禁止 2.1 日本ライセンスの下でライセンスされています。この使用許諾条件を見るには、<http://creativecommons.org/licenses/by-nd/2.1/jp/> をチェックするか、クリエイティブ・コモンズに郵便にてお問い合わせください。住所は: 559 Nathan Abbott Way, Stanford, California 94305, USA です。
- 本ドキュメントの最新版は、<http://www.knowledge-ex.jp/opendoc/javawebappdevelopment.html> より入手することができます。

あなたは以下の条件に従う場合に限り、自由に



本作品を複製、頒布、展示、実演することができます。

あなたの従うべき条件は以下の通りです。



表示. あなたは原作者のクレジットを表示しなければなりません。



改変禁止. あなたはこの作品を改変、変形または加工してはなりません。

- 再利用や頒布にあたっては、この作品の使用許諾条件を他の人々に明らかにしなければなりません。
- 著作[権]者から許可を得ると、これらの条件は適用されません。
- Nothing in this license impairs or restricts the author's moral rights.

Agenda

- Webアプリケーションとは 4
- Servletの基礎 14
- JSPの基礎 160
- ServletとJSPの連携 255

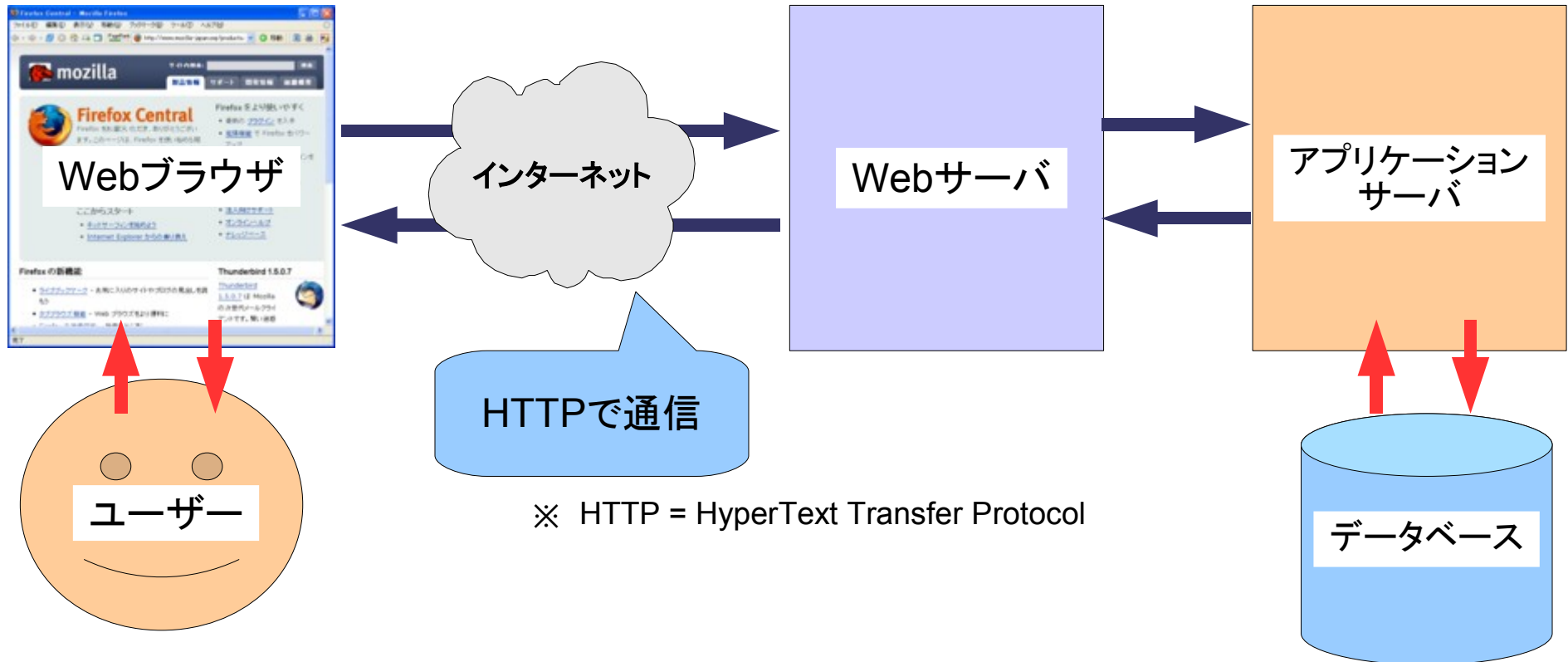
Webアプリケーションとは

株式会社ナレッジエックス
<http://www.knowledge-ex.jp/>

Webアプリケーションの仕組み

- それぞれの役割

- 「Webブラウザ」「Webサーバ」「アプリケーションサーバ」の連携によって動作するのが基本



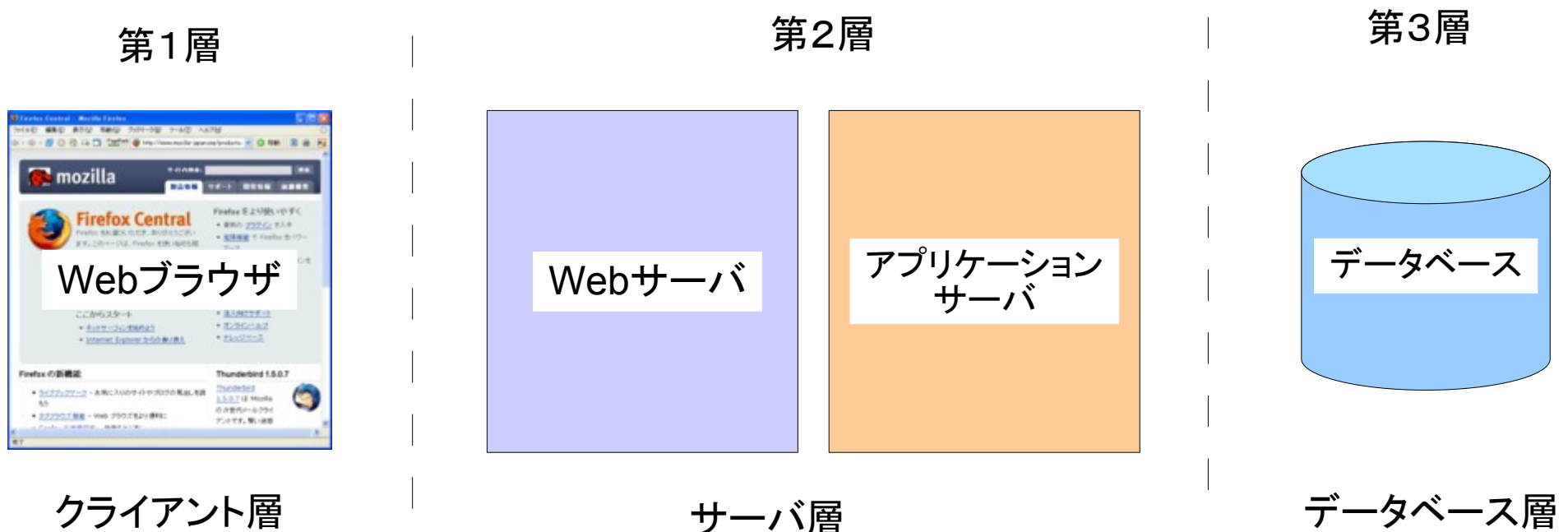
それぞれの役割

- Webブラウザ
 - ユーザーインターフェースの提供
- Webサーバ
 - Webブラウザとの通信
 - 静的コンテンツ(HTMLファイル、画像など)の提供
- アプリケーションサーバ
 - 動的コンテンツ(アプリケーションの処理によって作成された画面)の提供
- データベース
 - アプリケーションに必要な情報の提供

Webアプリケーションの階層構造

• 多層モデル

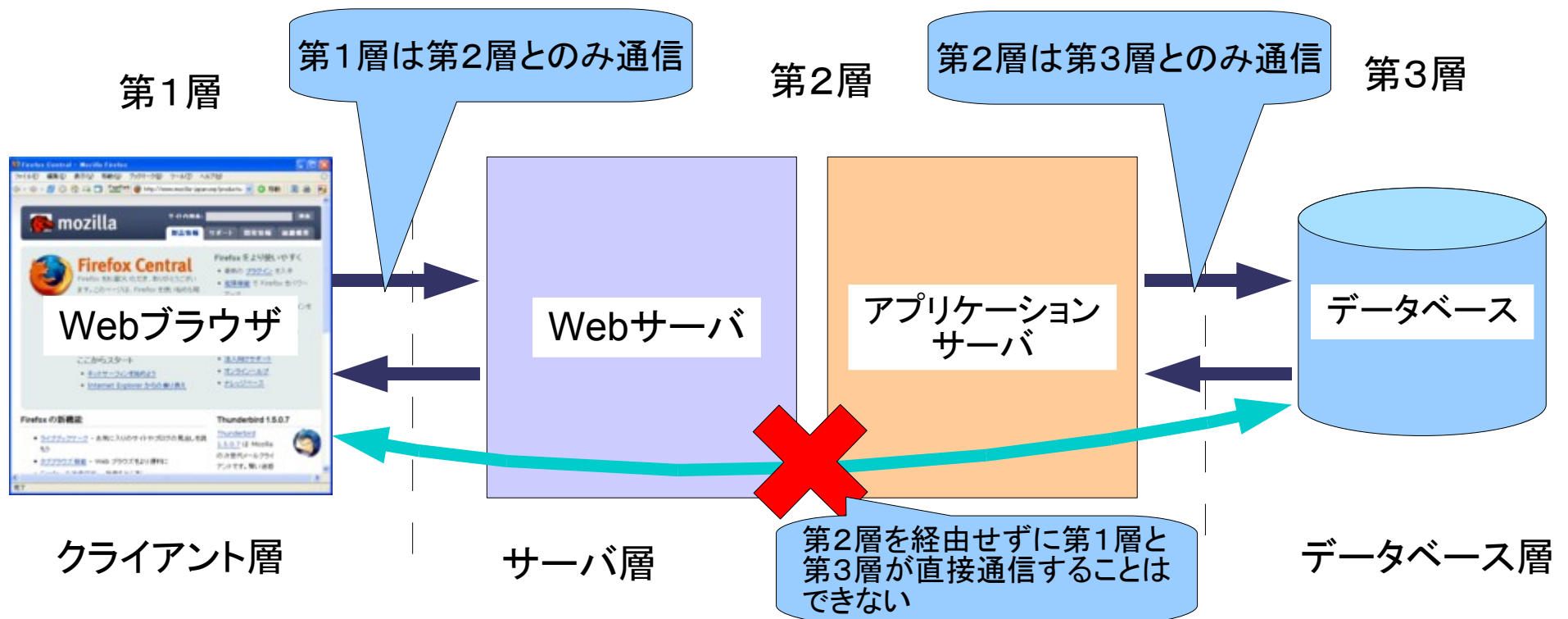
- Webアプリケーションは、1つのマシン、1つのソフトウェアだけで構成されているわけではなく、複数のマシン(ソフトウェア)の組み合わせによって提供されている
= 多層モデル



多層モデルの特徴

• 多層モデルの特徴

- 各層のソフトウェアは、隣り合う層としか通信をしない
 - Webアプリケーションでは、Webブラウザがデータベースサーバと直接通信することはない



ソフトウェアとテクノロジー

- 各層で用いられる製品と実装技術

	第1層	第2層		第3層
ソフトウェア	Webブラウザ	Webサーバ	アプリケーションサーバ	データベースサーバ
製品例	Internet Explorer Firefox Opera	Apache IIS	Tomcat WebSphere WebLogic	Oracle MySQL SQLServer PostgreSQL DB2
実装技術	HTML CSS JavaScript	Servlet JSP JDBC		SQL ストアド プロシージャ

Webアプリケーションの実装技術

- HTML(HyperText Markup Language)
 - Webブラウザ上に文書を表示するために用いられる言語
 - Webアプリケーションでは、アプリケーションの画面(ユーザーインターフェース)を提供するために利用
- Servlet(Java Servlet)
 - Webサーバを介して、Webブラウザとの通信を行うためのJava API
- JSP(JavaServer Pages)
 - Java言語を用いて、動的にHTML文書を構築するために用いられるJava API
- JDBC(Java DataBase Connectivity)
 - JavaアプリケーションからデータベースサーバにアクセスするためのAPI

Webアプリケーションのメリット・デメリット

- メリット

- クライアント層が軽量

- アプリケーションを利用するユーザーのマシンにWebブラウザしか必要としない
 - ユーザーの環境に新たなソフトをインストールする必要がない
 - ユーザーのマシンに負荷をかけない
 - アプリケーションの不具合は少数のサーバのみをメンテナンスすればよく、コストが削減できる

- クライアントのアーキテクチャを選ばない

- 異なるOS (Windows・MacOS・Linuxなど)、異なる機械 (PCと携帯電話など) であっても、ほぼ共通の内容を提供することができる
 - 同じ投資で、より幅広いユーザーに利用してもらえる
 - ビジネスチャンスがより広がる

Webアプリケーションのメリット・デメリット

- デメリット

- サーバに負荷の集中する構造

- 多数のユーザーに対して、少数のサーバが対応するため、特定の層に負荷が集中しやすい
 - 公開系アプリケーションでは、事前に想定しない量のアクセスが集中するケースがあり、サーバのダウンにつながる

- ユーザーインターフェースの使いにくさ

- もともとアプリケーションのユーザーインターフェースとして想定されていない技術（HTML）を使っているので、表現力や操作性に限界がある
 - Webアプリケーション以前からあったアプリケーションよりも操作性が劣ることが多くある
 - 近年は、Ajaxの登場（後述）を契機に、JavaScriptを積極的に使って表現力・操作性を向上する傾向が顕著に

Webアプリケーションの進化

- リッチクライアント化

- Ajax(Asynchronous Javascript And XML)

- HTMLとJavaScriptを組み合わせることでブラウザ上での表現力や操作性を向上させる技術
 - Googleが火付け役
 - Google Maps、Googleサジェスト、GMailなど

- Flexフレームワーク

- FlashアプリケーションとWebサーバを連携させ、表現力や操作性の高いユーザーインターフェースを実現する技術
 - 大多数のブラウザに標準でインストールされるFlashを使うことが強み

- Java Web Start

- Webを経由してSwing(JavaによるGUI)ベースのアプリケーションをダウンロードして実行し、通常のアプリケーションと同様の表現力や操作性を提供する技術
 - ダウンロードが伴うためユーザーへの負荷が避けられない

Servletの基礎

株式会社ナレッジエックス
<http://www.knowledge-ex.jp/>

Agenda

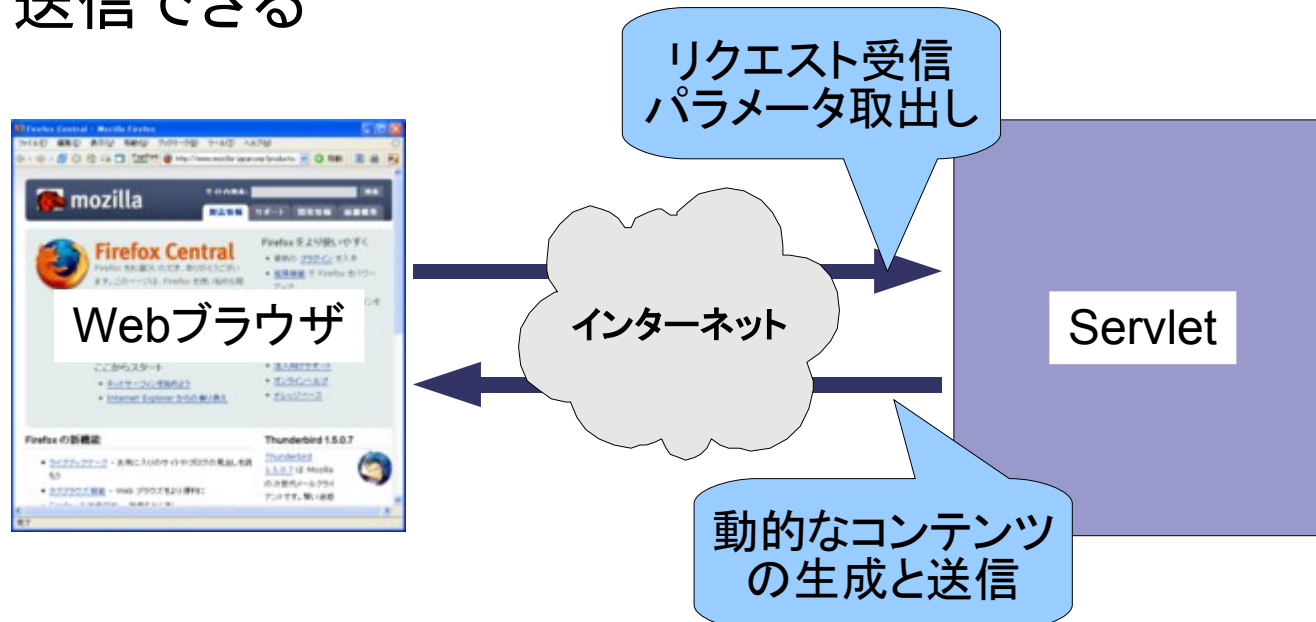
- Servletとは
 - Servletのしくみ
 - Servletのライフサイクル
- Servletをつくる
 - API概要
 - 開発手順
 - 各API詳細
 - web.xmlファイルの概要
 - パラメータの送受信
- コンテナへのデプロイ
- セッション管理
- サーブレットフィルタ

Servletとは

株式会社ナレッジエックス
<http://www.knowledge-ex.jp/>

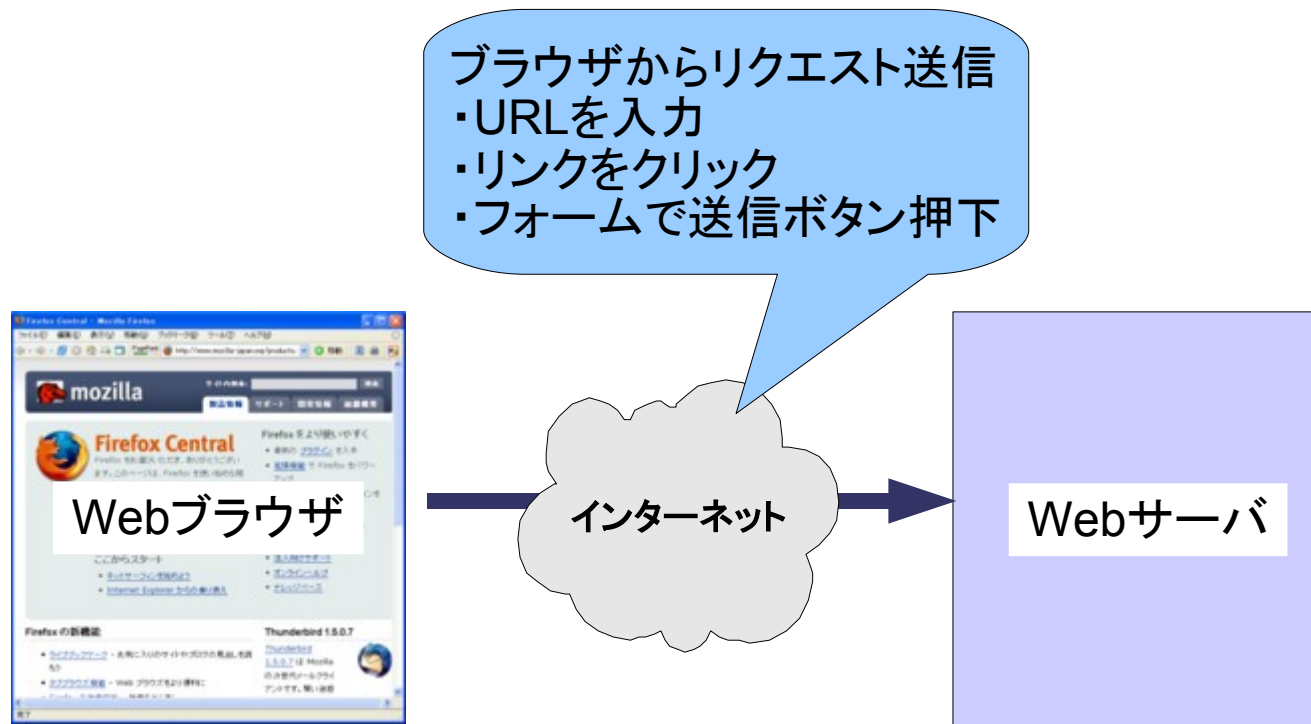
Servletとは

- Servlet(Java Servlet・サーブレット)
 - Webブラウザと通信を行うJavaアプリケーションを作成するためのAPI(フレームワーク)
 - ブラウザから送信されるリクエストを受信しパラメータを取り出すことができる
 - アプリケーションで処理を行いブラウザへ動的なコンテンツを送信できる





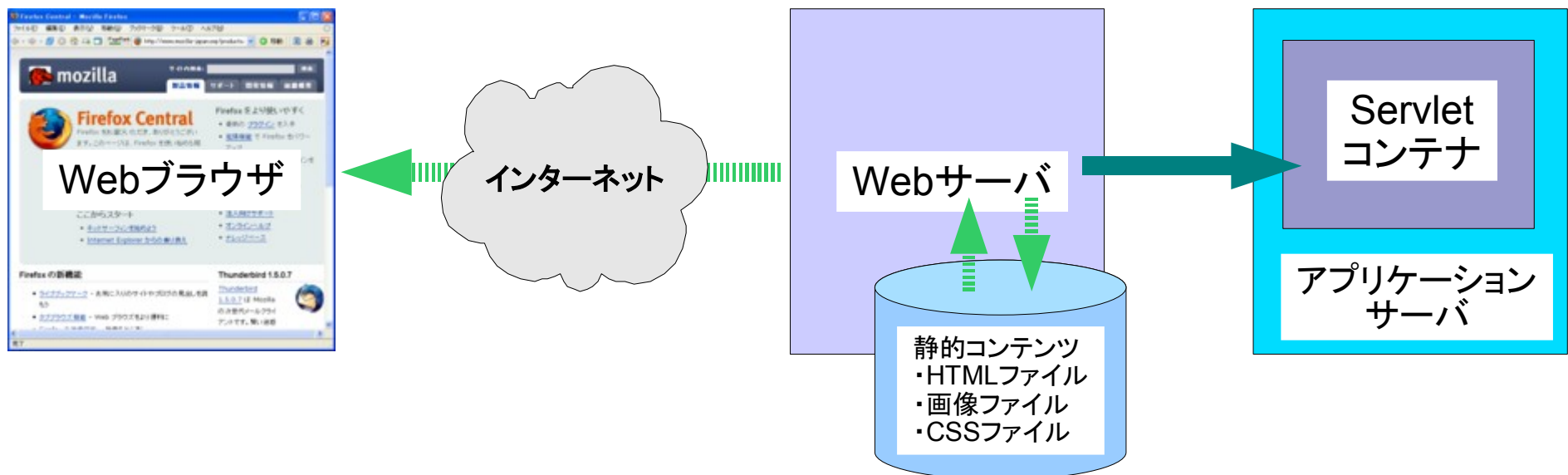
Servletのしくみ(1)

- WebブラウザとServletの通信手順
 - ①Webブラウザは基本的にWebサーバとしか通信できないため、ブラウザからのリクエストはいったんWebサーバに送られる



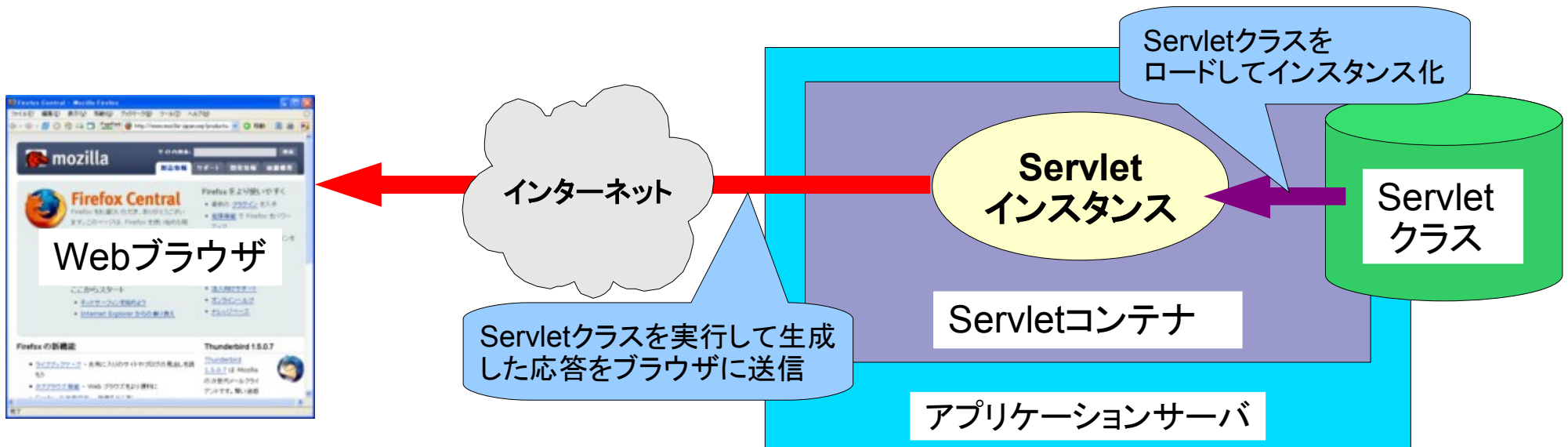
Servletのしくみ(2)

- WebブラウザとServletの通信手順
 - ②Webサーバではリクエスト内容を解析し
 - A)静的コンテンツ(HTMLファイルや、画像など)に対するリクエストであれば、自身で応答を送信()
 - B)Servletに対するリクエストであれば、「Servletコンテナ」にそのリクエストを転送()



Servletのしくみ(3)

- WebブラウザとServletの通信手順
 - ③Servletコンテナでは、リクエストで指定されたサーブレットクラスを実行
 - A)指定されたサーブレットクラスのインスタンスがまだ生成されていなければ、インスタンスを生成して実行、応答を送信
 - B)インスタンスがすでに生成されていれば、そのインスタンスを実行、応答を送信

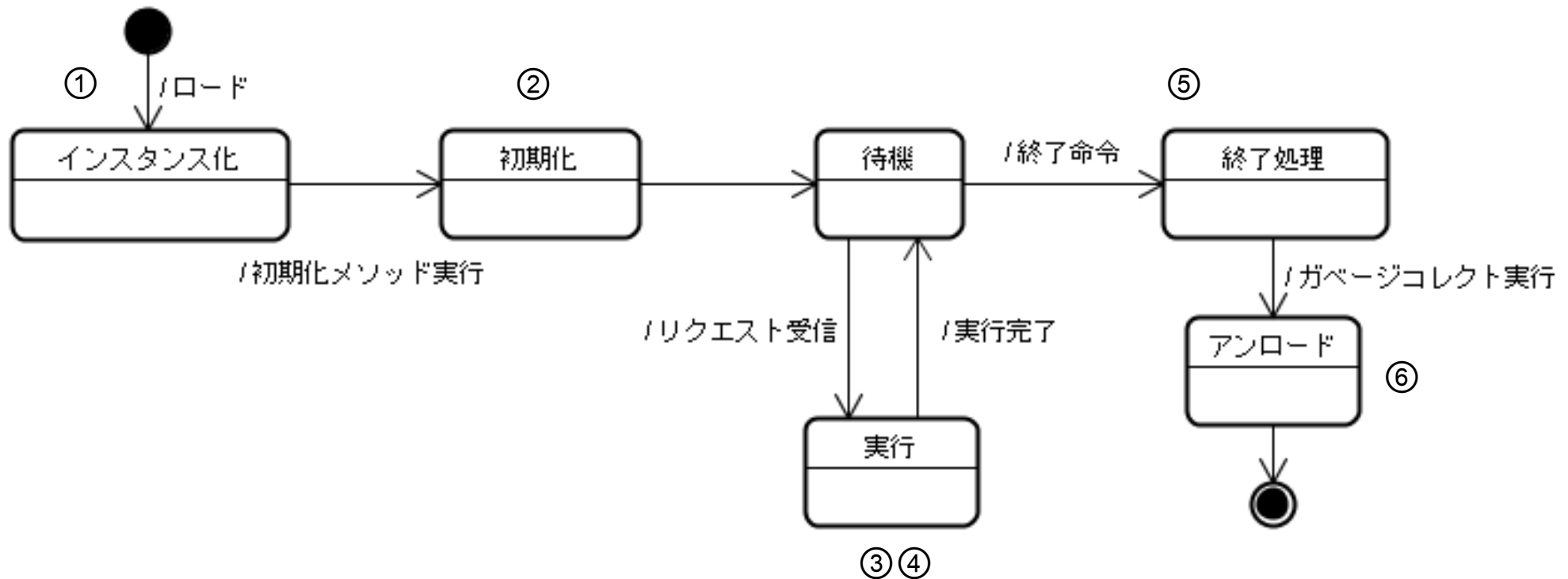


Servletコンテナとは

- Servletを管理するミドルウェア
 - 管理の内容
 - サーブレットのライフサイクルを管理
 - インスタンス生成
 - 各メソッドの呼び出し
 - リクエスト・レスポンス情報の提供
 - 生成済みインスタンスの保持
 - WebサーバやWebブラウザとの通信
 - Webサーバからリクエスト情報の受信
 - Webブラウザに対してレスポンス情報の送信
 - Java用のWebアプリケーションサーバに含まれている
 - TomcatやWebSphere、WebLogicなどに含まれる

※「コンテナ」=「オブジェクトの容れ物」という意味でよく使われます

Servletのライフサイクル



- ①Servletがブラウザからのリクエストによって初めて指定されると、インスタンス化を行う
- ②インスタンス化した後に初期化処理を実行
- ③リクエストを受信すると所定の処理を実行
- ④実行が完了すると待機状態となり、再びリクエストがあると処理を実行→待機状態となる
- ⑤外部 (Servletコンテナ) からの終了命令があると、終了処理を実行
- ⑥ガベージコレクションが実行されるとインスタンスはアンロードされ消滅

Servletをつくる

株式会社ナレッジエックス
<http://www.knowledge-ex.jp/>

Servlet API(1)

- javax.servlet パッケージ
 - Servletの基本的なクラス・インターフェースが含まれる
- javax.servlet.http パッケージ
 - Http通信に特化したクラス・インターフェースが含まれる

Servlet API(2)

- javax.servlet パッケージの主要インターフェース
 - Servlet
 - 全てのServletクラスが実装すべきインターフェース
 - ServletRequest
 - クライアントからのリクエストを扱うためのインターフェース
 - ServletResponse
 - クライアントへのレスポンスを扱うためのインターフェース
 - Filter
 - フィルタ機能を実装するためのインターフェース
 - RequestDispatcher
 - リクエスト・レスポンスを転送するためのインターフェース

Servlet API(3)

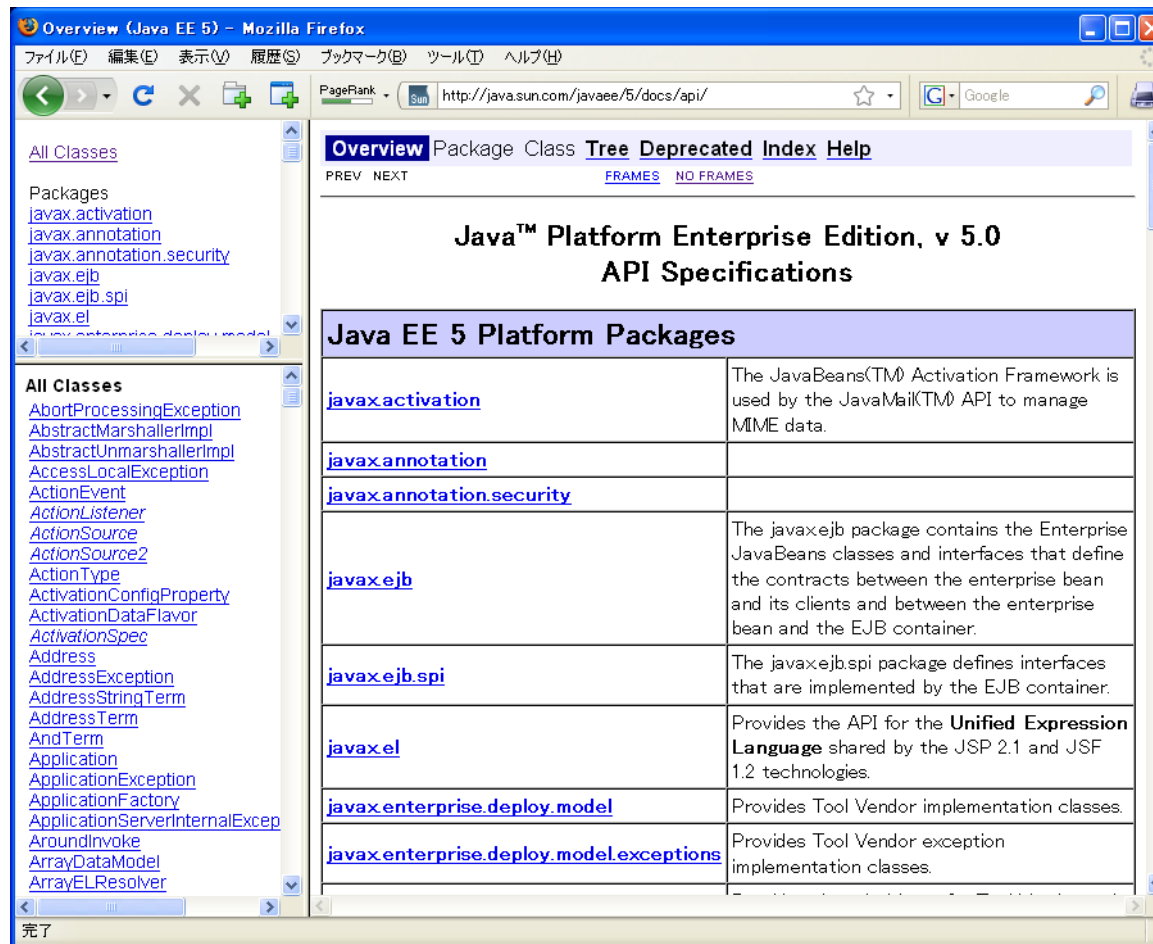
- javax.servlet パッケージの主要クラス
 - GenericServlet
 - Servletとしての基本機能を実装した基底クラス

Servlet API(4)

- javax.servlet.httpパッケージの主要クラス・インターフェース
 - HttpServletRequest
 - Webブラウザからのリクエストを扱うためのインターフェース
 - HttpServletResponse
 - Webブラウザへのレスポンスを扱うためのインターフェース
 - HttpSession
 - セッション管理機能を実現するためのインターフェース
 - HttpServlet
 - Http通信のServletの基本機能を実装した基底クラス

Servlet APIのJavadoc

- <http://java.sun.com/javaee/5/docs/api/>



Servletの実装(1)

- Servletの実装手順
 - HttpServletを継承したクラスを作成
 - 必要なメソッドをオーバーライド、もしくはメソッドを追加

Servletの実装(2)

- HttpServletの主なメソッド
 - init
 - doGet
 - doPost
 - service
 - destroy
 - getInitParameter

Servletの実装(3)

- HttpServletの主なメソッド
 - init
 - 初期化処理を行うためのメソッド
 - オーバーライドしておく、インスタンスが生成された後にServletコンテナによって自動で実行される

Servletの実装(4)

- HttpServletの主なメソッド
 - doGet
 - ブラウザに対する応答処理を行うためのメソッド
 - オーバーライドしておくと、Webブラウザから「HTTP GET」形式のリクエストを受信したときにServletコンテナによって自動で実行される
 - doPost
 - ブラウザに対する応答処理を行うためのメソッド
 - オーバーライドしておくと、Webブラウザから「HTTP POST」形式のリクエストを受信したときにServletコンテナによって自動で実行される

Webブラウザからのリクエスト形式(1)

- HTTP GET形式のリクエストとは？
 - WebブラウザでURLを直接指定したときのリクエスト



URLを直接指定(入力など)

- 画面上のリンクをクリックした場合のリクエスト

リンクをクリック



Webブラウザからのリクエスト形式(2)

- HTTP POST形式のリクエストとは？
 - Webブラウザでサーブレットを送信先に指定したフォーム等に入力して送信ボタンをクリックしたときのリクエスト

エックス(Knowledge-ex.) - Mozilla Firefox

移動(G) ブックマーク(B) ツール(T) ヘルプ(H)

PageRank <https://www.knowledge-ex.co.jp/inquiry/inquiry> 移動

お問い合わせ内容(必須)
最近発売された新製品についてのカタログを送付していただくことは可能でしょうか？

会社名(必須) 有限会社ナレッジエックス

部署名 開発部

氏名(必須) ナレッジ 太郎

氏名(ふりがな) なれっじ たろう

メールアドレス(必須) taro.knowledge@knowledge-ex.co.jp

電話番号 03-9999-8888

都道府県 東京都

住所 大田区羽田旭町7番1号

確認

送信ボタンをクリック

※フォーム入力してのリクエスト送信を、「HTTP GET」で行うことも可能です(後述)

Webブラウザからのリクエスト形式(3)

- Servletに対するURL指定
 - WebブラウザからServletを実行させるためのURL

http://www.knowledge-ex.jp/myshop/catalogview

↑
プロトコル指定+ホスト名(ポート番号)

↑
コンテキストパス

↑
サーブレットパス

Servletに対するURL

=「プロトコル指定+ホスト名(ポート番号)」+「コンテキストパス」+「サーブレットパス」

Webブラウザからのリクエスト形式(4)

- コンテキストパス
 - Webアプリケーション単位で与えられるパス
 - ひとつのサーブレットコンテナで、複数のアプリケーションを動かすことが可能なため、パスを分けて管理する必要がある
 - 開発者が任意の名称をつけられる
- サーブレットパス
 - 各サーブレットごとに与えられるパス
 - あらかじめそれぞれのサーブレットクラスに対してサーブレットパスを割り当てておく

Webブラウザからのリクエスト形式(5)

- GETとPOSTの違い

- HTTP GET

- Webブラウザから送信したパラメータが、全てWebブラウザ（のURL表示部）に表示されてしまう

- HTTP POST

- Webブラウザから送信したパラメータはWebブラウザ（のURL表示部）には表示されない

- フォームからデータを送信する画面で、HTTP GETを使うことも可能だが...
- ユーザーが重要な情報を入力するような画面では、HTTP GETは使用すべきでない
- 通常は、フォーム送信を伴うリクエストでは、HTTP POSTを使う（明示的に指定）

Servletの実装(5)

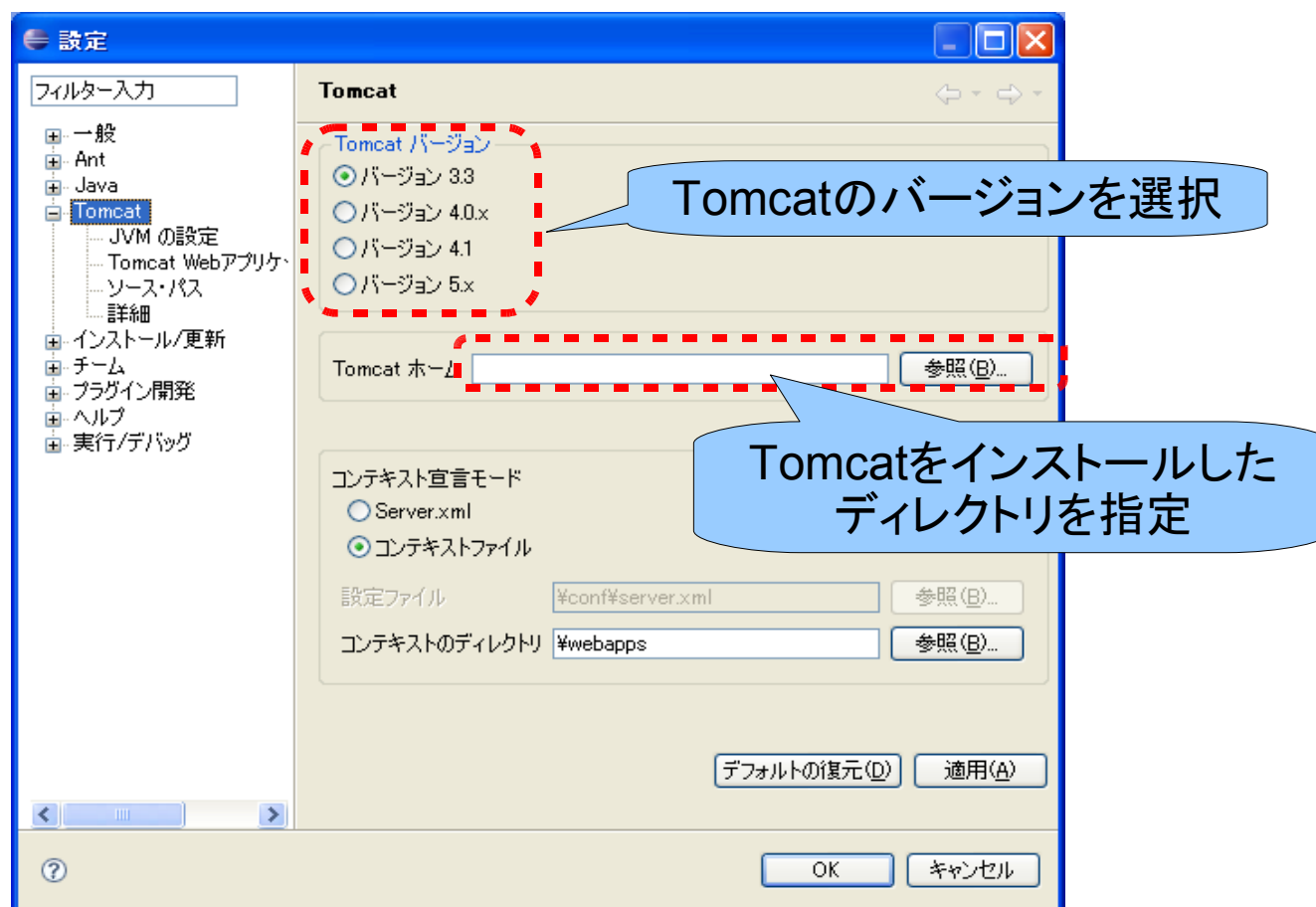
- HttpServletの主なメソッド
 - service
 - HTTP GET形式、HTTP POST形式どちらのリクエストに対しても応答する処理を行うためのメソッド
 - オーバーライドしておく、GET/POSTどちらのリクエストがあったときもServletコンテナによって自動で実行される
 - destroy
 - サーブレットの終了処理を行うためのメソッド
 - オーバーライドしておく、サーブレットが終了する直前に、Servletコンテナによって自動で実行される
 - getInitParameter
 - サーブレットの初期化パラメータを取得するためのメソッド
※ 詳しくは後述

Servletを作ってみよう(1)

- これから作成するサーブレットの仕様を決定
 - ブラウザからURLを入力して実行する
 - サーブレットが実行されると、ブラウザに「Hello,Servlet!」などの文字が表示される
 - クラス名(パッケージ名含む)は、「web.FirstServlet」
 - アプリケーション全体のコンテキストパスは「/kx」

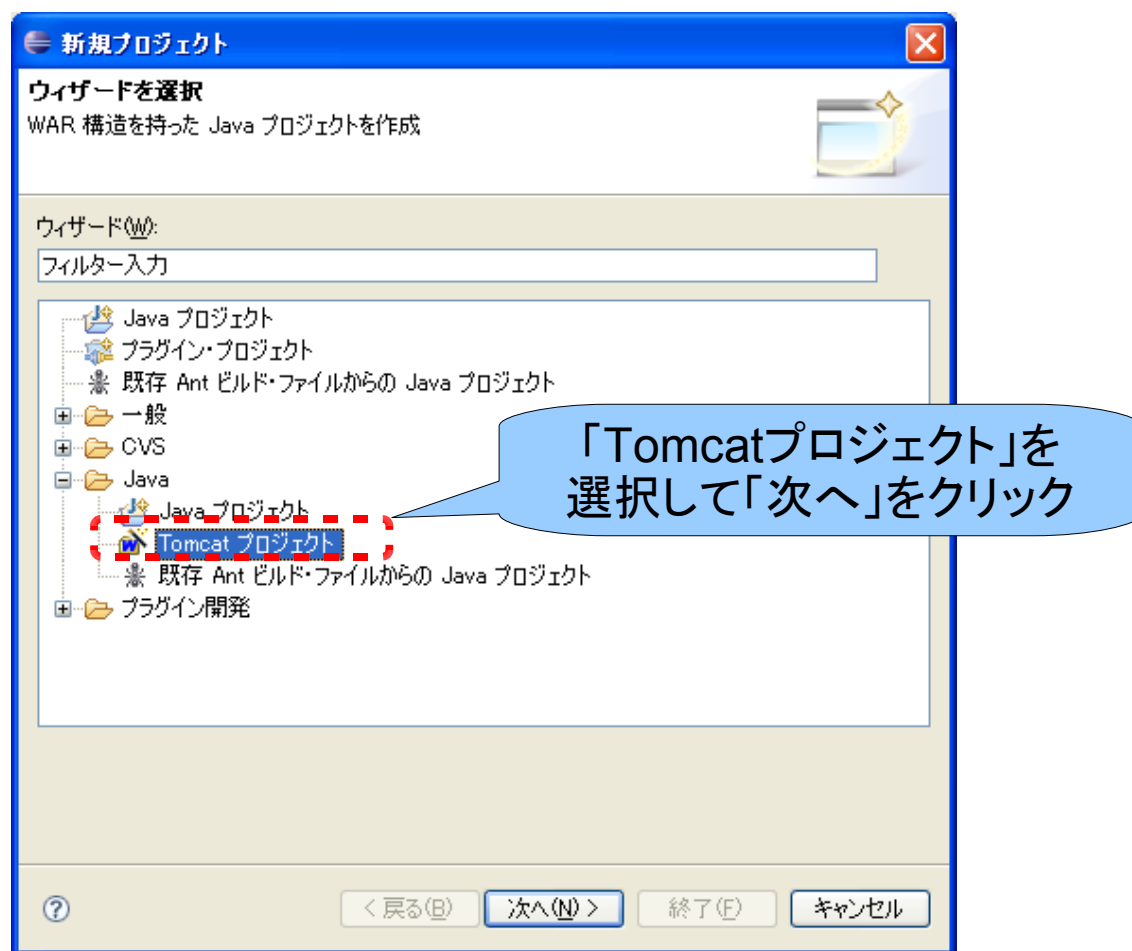
Servletを作ってみよう(2)

- Eclipse(Tomcat Plug-in)の設定
 - メニュー「ウィンドウ」→「設定」→「Tomcat」を選択



Servletを作ってみよう(3)

- 「Tomcatプロジェクト」の作成
 - メニュー「ファイル」→「新規」→「プロジェクト」を選択



Servletを作ってみよう(4)

- プロジェクト名とロケーションの設定

新規 Tomcat プロジェクト

Java プロジェクトの設定
プロジェクト名とプロジェクトの場所の入力

プロジェクト名(P):

☒ デフォルト・ロケーションの使用(D)

ロケーション(L): C:/eclipse/3.2.1/workspace 参照(B)...

プロジェクトの作成場所を自分で決めたい場合は、
このチェックを外し、「ロケーション」にパスを入力

「プロジェクト名」(任意)を入力し、
「次へ」をクリック

< 戻る(B) 次へ(N) > 終了(F) キャンセル

Servletを作ってみよう(5)

- コンテクストパスとドキュメントルートの設定

新規 Tomcat プロジェクト

Tomcat プロジェクト 設定

このプロジェクト用に Tomcat のコンテキスト定義を追加または更新

コンテキスト名

☒ コンテキスト定義の更新を可能にする (server.xml または コンテキスト・ファイル)

Web アプリケーション・ルートとするサブディレクトリ (オプション)

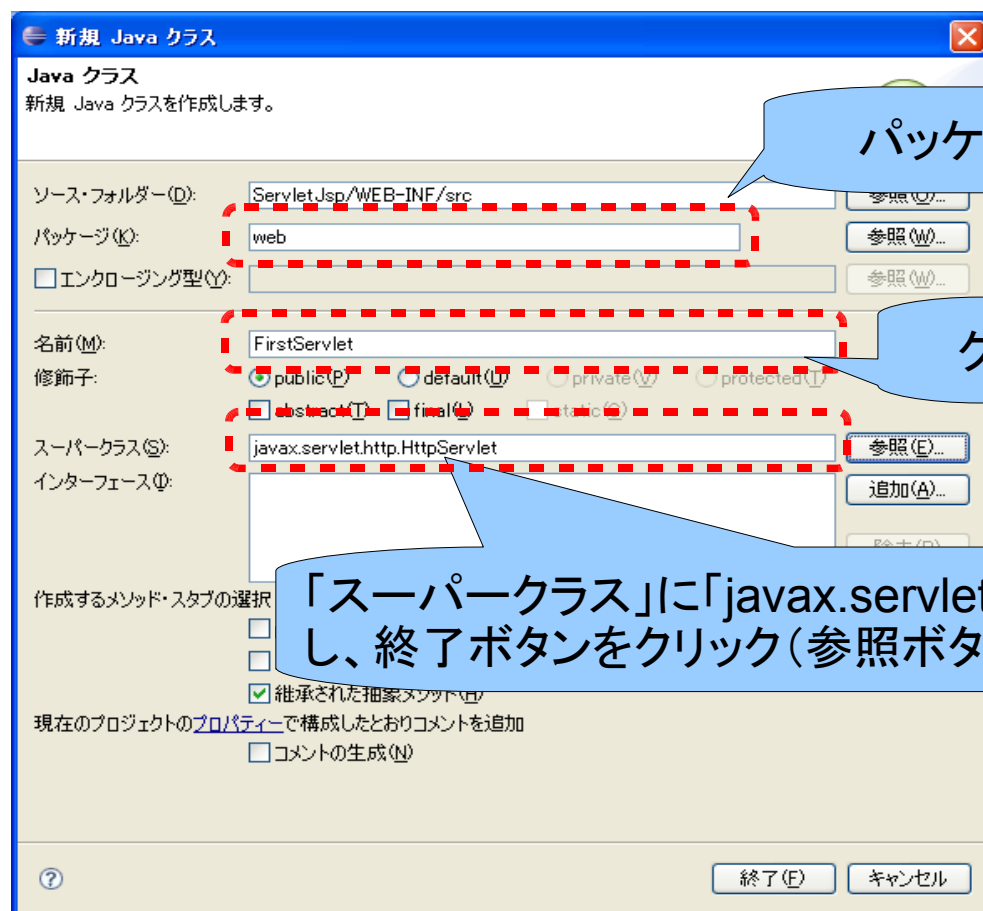
コンテクストパス(任意)を入力し、「終了」をクリック

ドキュメントルートの変更したい場合は、ここに任意のパスを入力

? < 戻る(B) 次へ(N) > 終了(F) キャンセル

Servletを作ってみよう(6)

- サーブレットクラスの作成
 - クラス作成ダイアログで、親クラスにHttpServletを指定



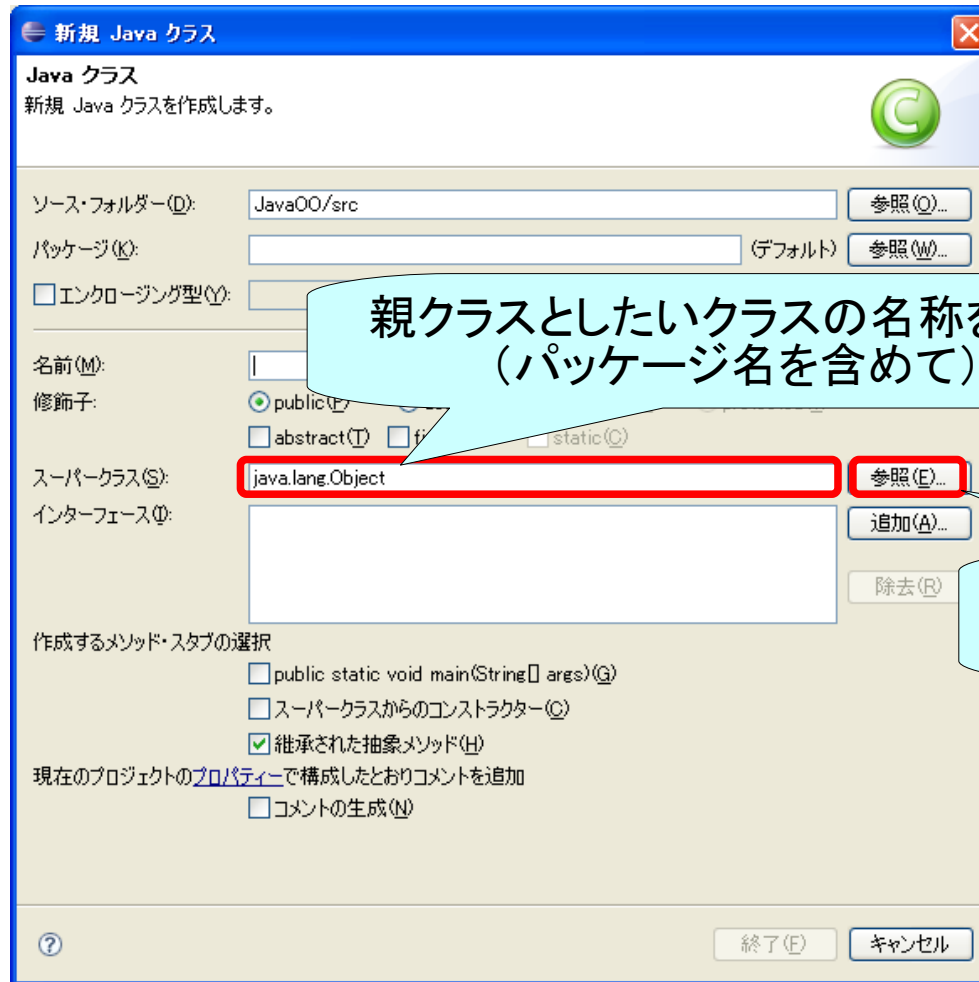
パッケージ名(任意)を入力

クラス名(任意)を入力

「スーパークラス」に「javax.servlet.http.HttpServlet」を指定し、終了ボタンをクリック(参照ボタンを使うと便利)

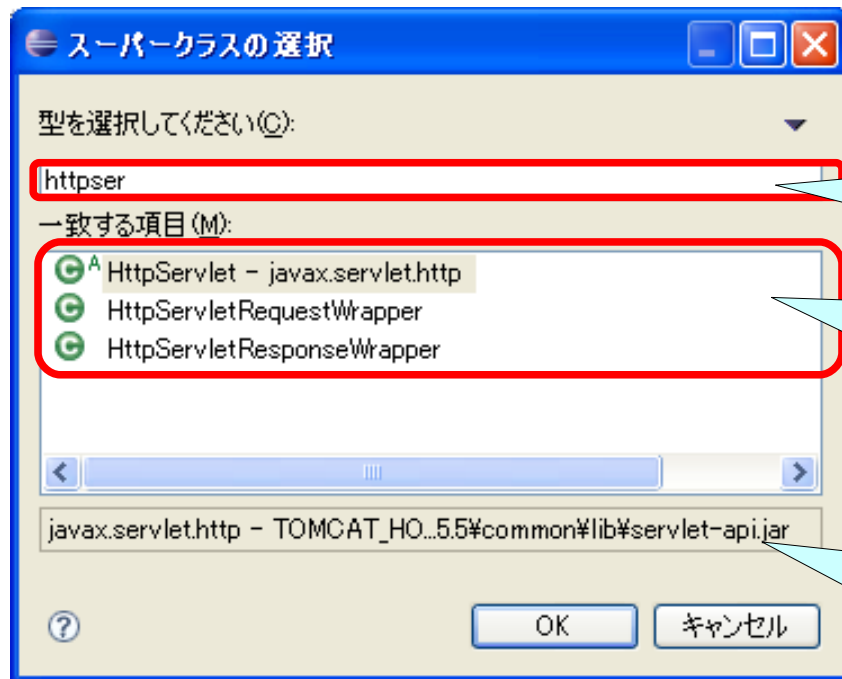
継承を用いたクラスの作成(1)

- クラス作成ダイアログで親クラスを指定可能



継承を用いたクラスの作成(2)

- 「参照」ボタンで一覧から親クラスを選択



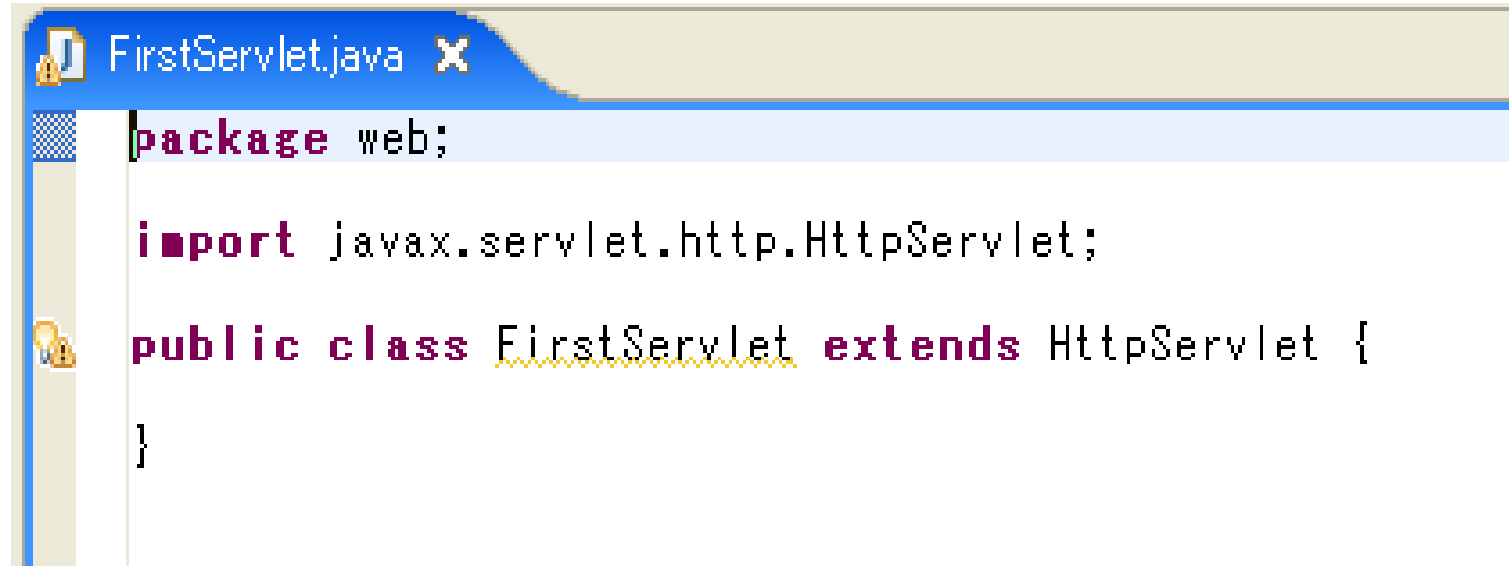
①親クラスとしたいクラスの名称の先頭から何文字かを入力する

②入力した文字で始まるクラスが絞り込まれて表示されるので、選択して「OK」をクリック

選択されているクラスの配置場所やパッケージ名が表示される

Servletを作ってみよう(7)

- サーブレットクラスが作成される
 - 中身が空のサーブレットクラスができる



```
FirstServlet.java X
package web;

import javax.servlet.http.HttpServlet;

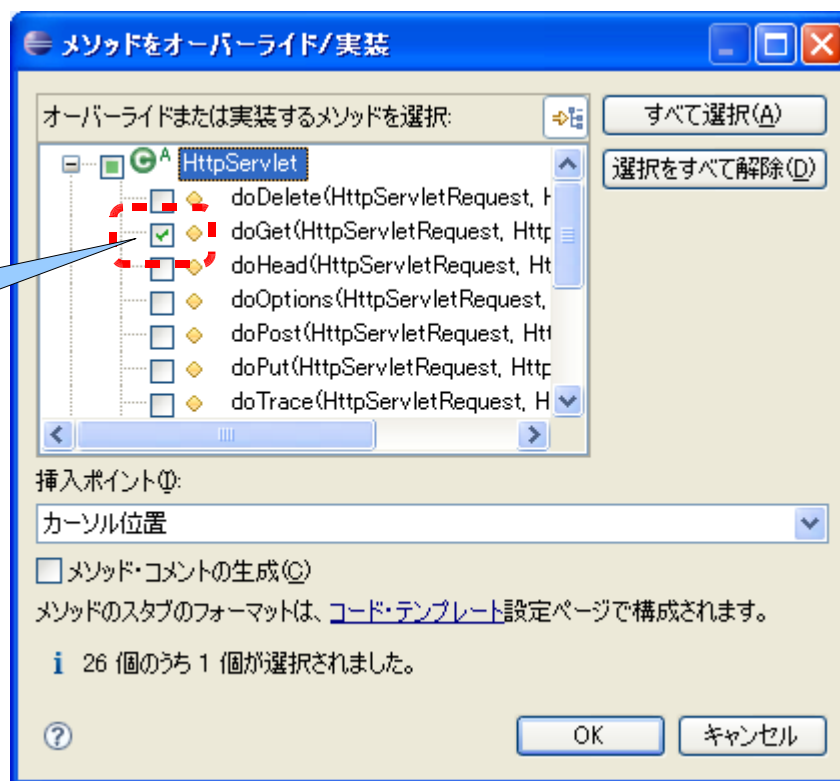
public class FirstServlet extends HttpServlet {
}
```

Servletを作ってみよう(8)

- オーバーライドするメソッド
 - ブラウザでURLを指定して実行＝doGetメソッド
 - 親クラスのメソッドをオーバーライドするには、Eclipseの「メソッドのオーバーライド/実装」機能を利用すると便利

Servletを作ってみよう(9)

- doGetメソッドをオーバーライドする
 - メニュー「ソース」→「メソッドのオーバーライド/実装」を選択
 - オーバーライドしたいメソッドをチェックし、「OK」をクリック



Servletを作ってみよう(10)

- doGetメソッドが追加された
 - ソースに、doGetメソッドの定義が追加された
 - メソッド内部のコメントとsuper呼び出しは削除して良い
 - 引数の名称は機械的に「arg0」「arg1」・・・と命名されるので、わかりやすい名称に変更すると良い
 - 「arg0」→「request」
 - 「arg1」→「response」

```
@Override
protected void doGet(HttpServletRequest arg0, HttpServletResponse
// TODO 自動生成されたメソッド・スタブ
super.doGet(arg0, arg1);
}
```

わかりやすい名前に変更

特に必要なければ削除

Servletを作ってみよう(11)

- ブラウザへ応答を出力する手順
 - doGet(doPost)メソッドの第二引数HttpServletResponseを利用
 - setContentType()メソッドでMIMEタイプとエンコーディングを決定
 - getWriter()メソッドで出力用のストリームを取得
 - ストリームに対しprintln()やwrite()等のメソッドを用いて応答内容を出力
 - 出力終了後はストリームを閉じる

Servletを作ってみよう(12)

- HttpServletResponse#setContentType()メソッド
 - 出力するHTML文書の「MIMEタイプ」と「エンコーディング」を指定する
 - サーブレットからの出力エンコーディングのデフォルトは「ISO-8859-1」(後述)なので、指定を省略すると日本語が正しく出力されない

凡例

```
HttpServletResponse#setContentType(String);
```

↑
"MIMEタイプ; charset=エンコーディング名"

コード例

```
response.setContentType("text/html; charset=Windows-31J");
```

Servletを作ってみよう(13)

- MIMEタイプとは
 - インターネット上でやりとりされる文書や画像、動画などの形式を指定するもの
 - HTMLを表示する場合は「text/html」を指定する

MIMEタイプ名	内容
text/plain	プレーンテキスト
text/html	HTMLドキュメント
text/css	スタイルシート
image/gif	GIF画像
image/jpg	JPEG画像
application/pdf	PDF文書

MIMEタイプの一例

Servletを作ってみよう(14)

- エンコーディングとは
 - 文字とコード番号を対応づけるコード体系
 - エンコーディングを指定しないと、「8859_1(半角英数と記号のみ)」となり、日本語が正しく表示できない

エンコーディング名	内容	用途
Windows-31J	シフトJISコード	Windowsで主に使われる(IEのデフォルト)
EUC_JP	EUCコード	Unix系OSで主に使われる
ISO-2022-JP	JISコード	電子メールで主に使われる
UTF8	Unicode	国際化対応のアプリケーションで使われる
8859_1	Latin1コード	ASCIIコード(半角英数と記号のみ)

エンコーディングの一例

※ 「Shift_JIS」は、WindowsのシフトJISコードと完全に一致していないため、現在では「Windows-31J」を使うのが一般的です。

Servletを作ってみよう(15)

- HttpServletResponse#getWriter()メソッド
 - ブラウザに文字列(HTMLドキュメント)を出力するための出力ストリーム(java.io.PrintWriter)を取得する
 - 取得したストリームを変数に格納して使用する

コード例

```
PrintWriter out = response.getWriter();
```

※ 上記コード例では「java.io.PrintWriter」をimport宣言でimportしてあるものとします

Servletを作ってみよう(16)

- `PrintWriter#println()`メソッド
 - ストリームに文字列を出力するためのメソッド
 - `System.out.println()`と使い方は同じ
 - HTMLのタグも含めてそのまま文字列で出力する

コード例

```
PrintWriter out = response.getWriter();  
out.println("<h1>Hello, Servlet!</h1>");  
out.println("<p>サーブレットの出力サンプルです</p>");
```


Servletを作ってみよう(17)

- 出力内容をコーディングしてみよう
 - 下記のHTML文書を出力するようにPrintWriterを使ってコーディングしてみましょう

出力したいHTML文書

```
<html>
<head><title>First Servlet</title></head>
<body>
<h1>First Servlet</h1>
<p>サーブレットの出力サンプルです</p>
</body>
</html>
```

Servletを作ってみよう(18)

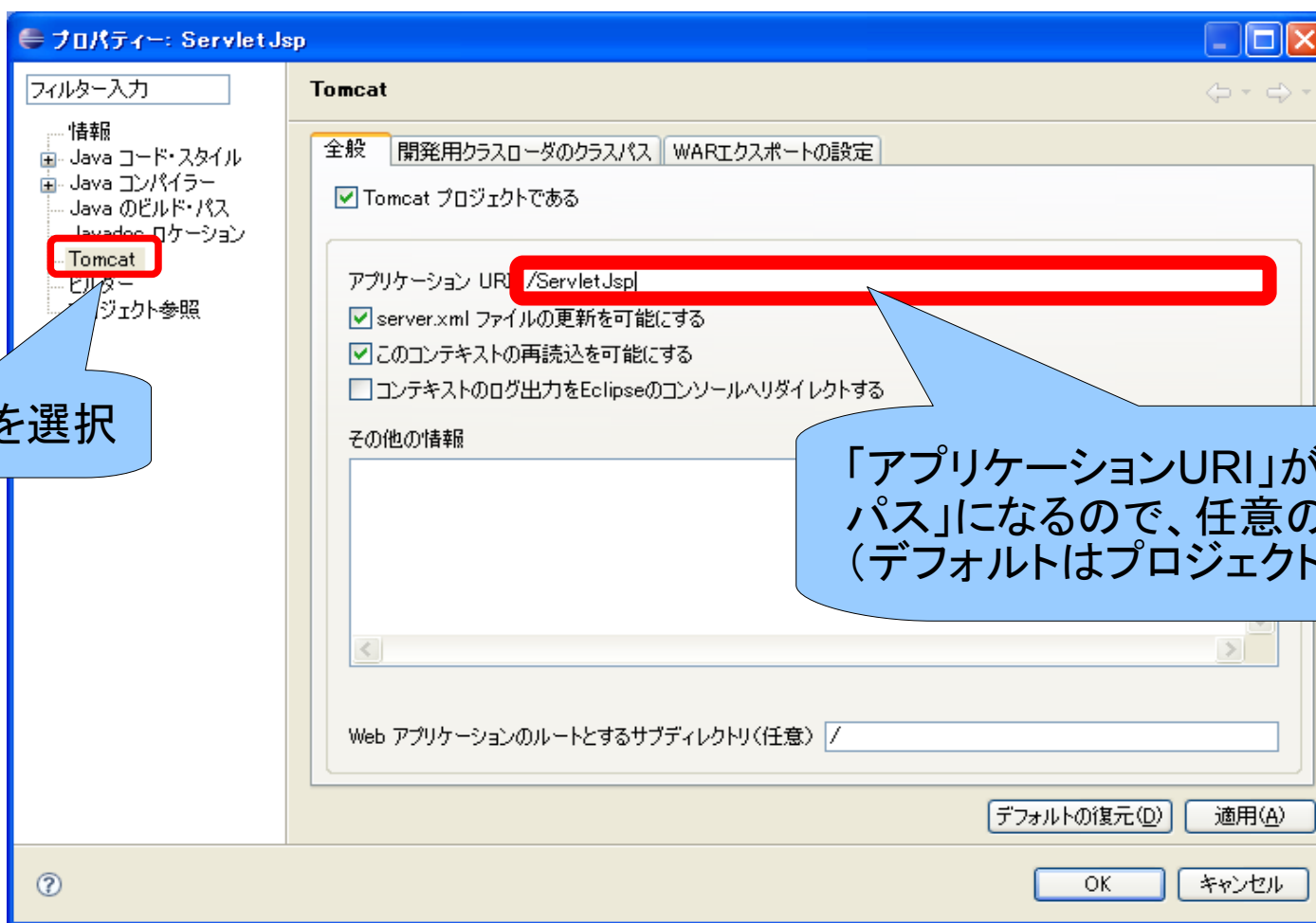
- PrintWriter#close()メソッド
 - ストリームを閉じる
 - ストリームを閉じると全ての出力結果がWebブラウザに送られる
 - doGet()やdoPost()が終了すると自動でストリームが閉じられるので、close()メソッドは明示的に発行しなくても良い

コード例

```
PrintWriter out = response.getWriter();  
out.println("<p>サーブレットの出力サンプルです</p>");  
:  
out.close();
```

参考:コンテクトパスの変更

- 作成したプロジェクトに対するコンテクトパスを変更するにはプロジェクト名にカーソルを合わせ、右クリックして「プロパティ」を選択

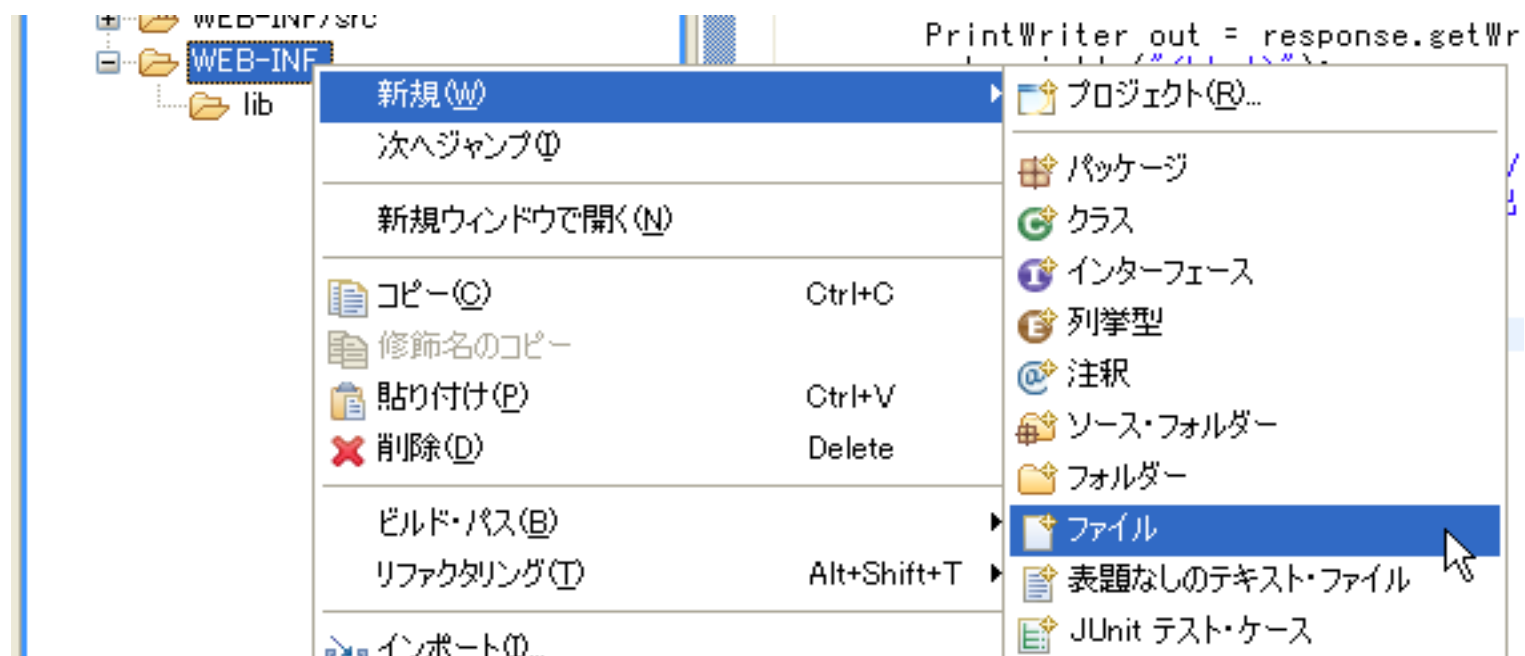


「Tomcat」を選択

「アプリケーションURI」が「コンテクトパス」になるので、任意の値に変更
(デフォルトはプロジェクト名と同じ)

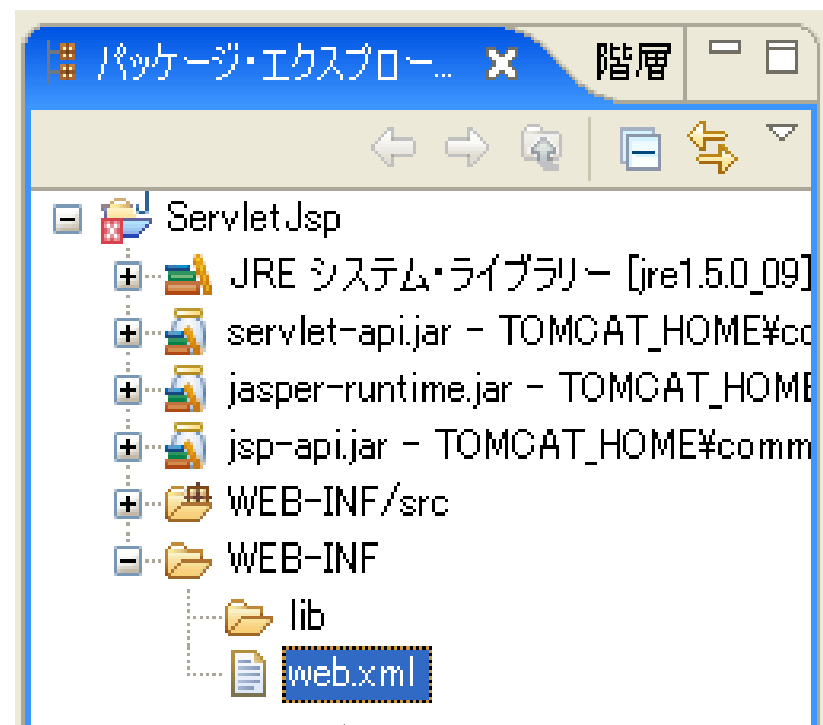
サーブレットパスを設定しよう(1)

- web.xmlファイルの作成
 - サーブレットパスその他の設定は、「WEB-INF」ディレクトリの直下に「web.xml」ファイルを作成して行う(詳しくは後述)
 - 「WEB-INF」フォルダを選択して右クリックし、「新規」→「ファイル」を選択



サーブレットパスを設定しよう(2)

- ファイル名の指定



サーブレットパスを設定しよう(3)

- ファイルが作成される
 - 中央にエディターが開くので、web.xmlの内容を記述していく

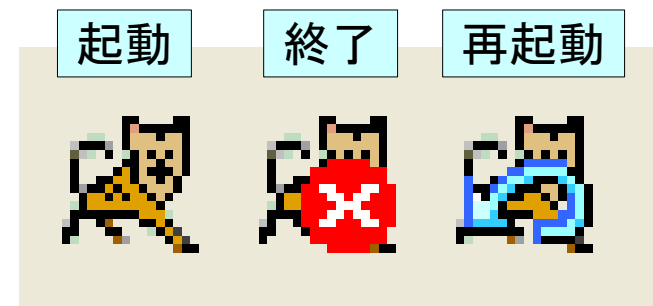
```
<?xml version="1.0" encoding="Windows-31J"?>
<web-app>
  <servlet>
    <servlet-name>first</servlet-name>
    <servlet-class>web.FirstServlet</servlet-class>
  </servlet>
  <servlet-mapping>
    <servlet-name>first</servlet-name>
    <url-pattern>/first</url-pattern>
  </servlet-mapping>
</web-app>
```

Servletを実行してみよう(1)

- Tomcatの起動
 - Servletの実行にはServletコンテナが必要
 - Servletコンテナを内蔵するTomcatを起動する



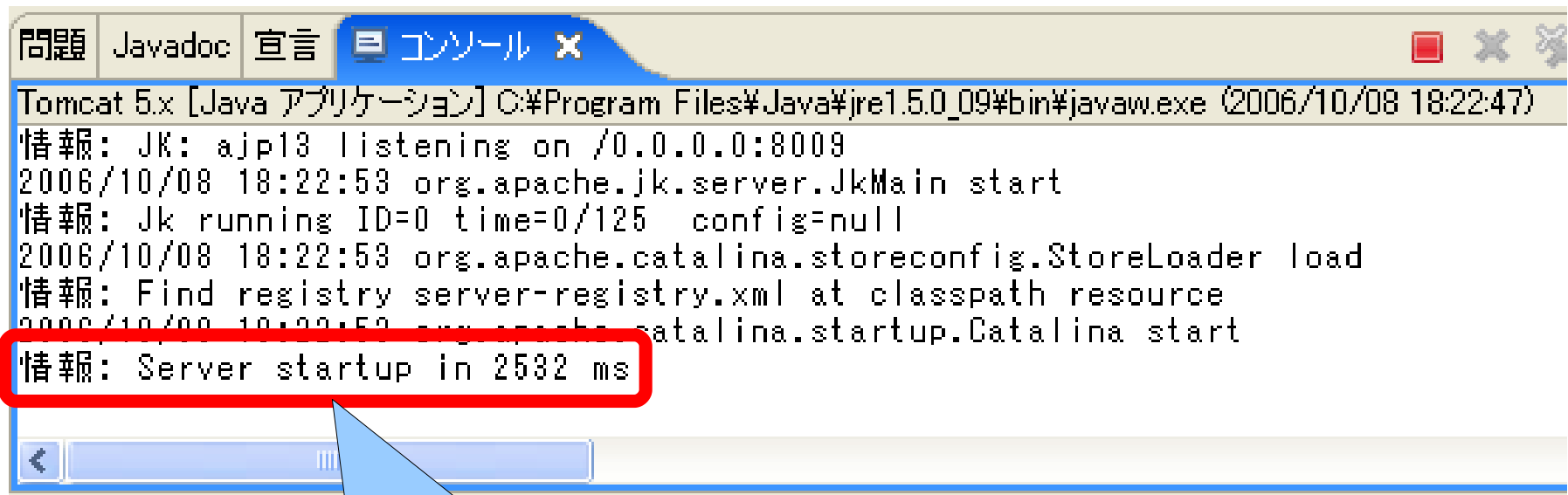
「Tomcat起動」ボタンを押下



Servletを実行してみよう(2)

- Tomcatの起動(続き)

- Tomcatが起動すると、Eclipseの「コンソール」ビューにTomcatのログが表示される
- 「情報: Server startup in xxxx ms」が表示されたら起動完了



```
Tomcat 5.x [Java アプリケーション] C:\Program Files\Java\jre1.5.0_09\bin\javaw.exe (2006/10/08 18:22:47)
情報: JK: ajp13 listening on /0.0.0.0:8009
2006/10/08 18:22:53 org.apache.jk.server.JkMain start
情報: Jk running ID=0 time=0/125 config=null
2006/10/08 18:22:53 org.apache.catalina.storeconfig.StoreLoader load
情報: Find registry server-registry.xml at classpath resource
2006/10/08 18:22:53 org.apache.catalina.startup.Catalina start
情報: Server startup in 2532 ms
```

このメッセージが表示
されたら起動完了

Servletを実行してみよう(3)

- Servletの呼び出し

- ブラウザからサーブレットを呼び出すためのURLを入力すると、サーブレットが呼び出され結果がWebブラウザに表示される



コード例

• FirstServletコード例

```
package web;

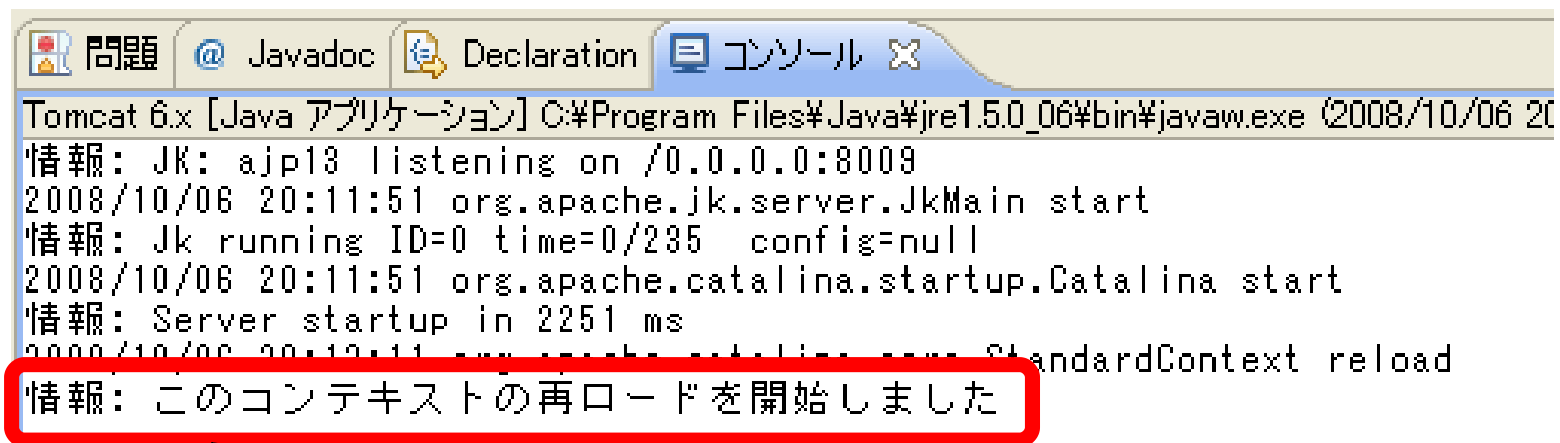
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class FirstServlet extends HttpServlet {

    protected void doGet(HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException {
        response.setContentType("text/html;charset=Windows-31J");
        PrintWriter out = response.getWriter();
        out.println("<html>");
        out.println("<head><title>First Servlet</title></head>");
        out.println("<body>");
        out.println("<h1>First Servlet</h1>");
        out.println("<p>サーブレットの出力サンプルです</p>");
        out.println("</body>");
        out.println("</html>");
        out.close();
    }
}
```

コードの変更とTomcatの再起動

- Tomcatプロジェクトでは、デフォルトで変更内容の再読込が有効になっており、Servletのソースコードや、web.xmlファイルを変更した場合も、基本的にTomcatを再起動する必要はない



```
Tomcat 6.x [Java アプリケーション] C:\Program Files\Java\jre1.5.0_06\bin\javaw.exe (2008/10/06 20:11:51)
情報: JK: ajp13 listening on /0.0.0.0:8009
2008/10/06 20:11:51 org.apache.jk.server.JkMain start
情報: Jk running ID=0 time=0/235 config=null
2008/10/06 20:11:51 org.apache.catalina.startup.Catalina start
情報: Server startup in 2251 ms
2008/10/06 20:12:11 org.apache.catalina.core.StandardContext reload
情報: このコンテキストの再ロードを開始しました
```

Tomcatが変更された内容を再読込したことを示すログ

web.xmlファイルの概要

株式会社ナレッジエックス
<http://www.knowledge-ex.jp/>

web.xmlとは

- Servletコンテナ上のアプリケーションに関する設定を記述するファイル
- 記述形式はXML
- 保管場所は「WEB-INF」ディレクトリの直下
- 記述できる内容(の一部)
 - サーブレットの定義
 - サーブレット名、クラス名、初期化パラメータ(後述)
 - サーブレットパスの定義
 - フィルタの定義(後述)
 - セッションタイムアウトの定義(後述)

<servlet>タグ

- <servlet>タグ

- 作成したサーブレットに対して、クラス名とは別に「サーブレット名」をつけて管理したり、初期化パラメータを設定する

- <servlet-name>

- サーブレット名

- <servlet-class>

- サーブレットのクラス名（パッケージ名を含めて記述）

- <init-param>

- 初期化パラメータを定義（スライド#80を参照）

<servlet-mapping>タグ

- <servlet-mapping>タグ
 - サーブレットに対するサーブレットパスを定義するタグ
 - <servlet-name>
 - サーブレット名。<servlet>タグで定義されたものを選んで指定
 - <url-pattern>
 - このサーブレットに対して割り当てるサーブレットパス
 - 「*」を用いたワイルドカード指定も可能
 - 例)「*.do」と指定→末尾が「.do」の全てのURLに対応

パラメータの送受信

株式会社ナレッジエックス
<http://www.knowledge-ex.jp/>

ブラウザとの間でパラメータを送受信する

- サーブレットではブラウザから送信されたパラメータを受信して処理を行うこともできる
- これにより、ユーザーの操作（入力内容）に応じた表示結果を返す、インタラクティブなアプリケーションを作ることが可能になる

パラメータの送信手順(1)

- Webブラウザからパラメータを送信する手順
 - ① HTTP GET形式の場合
 - URLの末尾を「?」で区切り、パラメータを追加して送信
 - パラメータを複数送信する場合は、「&」で区切り追加する

凡例

`http://ホスト名/パス?パラメータ名1=パラメータ値1[&パラメータ名2=パラメータ値2・・・]`

記述例

`http://localhost:8080/web/app?name=Taro&age=28`

パラメータの送信手順(2)

- Webブラウザからパラメータを送信する手順
 - ② HTTP POST形式の場合
 - <form>タグの送信先(action属性)をサーブレットに指定
 - <input>タグなどを使ってユーザーにパラメータを入力させる
 - 送信ボタンを押下するとパラメータが送信される

パラメータの送信手順(3)

- **<form>タグの書式**
 - action属性に送信先(サーバレット)のURLを指定
 - method属性には通常「POST」を指定
 - HTTP POST形式で送信するため
 - 指定を省略するとGET形式になるがパラメータがURL表示部に表示されてしまう

凡例

```
<form action="送信先のURL" method="POST">  
  :  
  (<input>などのフォーム要素を記述)  
  :  
</form>
```

パラメータの送信手順(4)

- <input>タグ等の書式
 - <input>タグの場合、type属性に入力の種別を指定
 - name属性に「パラメータ名」を指定

コード例

```
<form action="送信先のURL" method="POST">  
  :  
  <input type="text" name="username">  
  <input type="password" name="userpass">  
  :  
  <input type="submit" value="送信する">  
  :  
</form>
```

参考: フォームでよく使用するタグ

名称	タグ表記	内容
テキストフィールド	<input type="text" ...>	1行のテキストを入力するフォーム要素
パスワード	<input type="password" ...>	1行のテキストをマスクして入力するフォーム要素
チェックボックス	<input type="checkbox" ...>	複数選択可能なフォーム要素
ラジオボタン	<input type="radio" ...>	複数から一つだけを選択可能なフォーム要素
サブミットボタン	<input type="submit" ...>	フォームの入力内容を送信するボタン
リセットボタン	<input type="reset" ...>	フォームの入力内容を初期状態(画面が表示された時点)に戻すボタン
隠しパラメータ	<input type="hidden" ...>	画面に表示されないが送信したいパラメータを記述できるフォーム要素
ボタン	<input type="button" ...>	フォームの入力内容を送信しないボタン

パラメータの受信(1)

- Webブラウザから送信されるパラメータを受信する手順
 - doGet()またはdoPost()メソッドの引数HttpServletRequestを利用
 - setCharacterEncoding()メソッドで送信されてくるパラメータのエンコーディングを指定
 - getParameter()メソッドでパラメータを文字列として受信


パラメータの受信(2)

- HttpServletRequest#setCharacterEncoding()メソッド
 - 送信されてくるパラメータのエンコーディングを指定
 - InternetExplorerでは日本語文字列は「シフトJIS」コードで送信するのがデフォルトになっている
 - そのため他のブラウザも基本的にそれに追従している

凡例

```
HttpServletRequest.setCharacterEncoding(String);
```

エンコーディング名



コード例


```
request.setCharacterEncoding("Windows-31J");
```


パラメータの受信(3)

- HttpServletRequest#getParameter()メソッド
 - 受信したパラメータを文字列として取り出す
 - 引数にパラメータ名を指定
 - 戻り値はString型

凡例

```
HttpServletRequest#getParameter(String);
```



パラメータ名

コード例

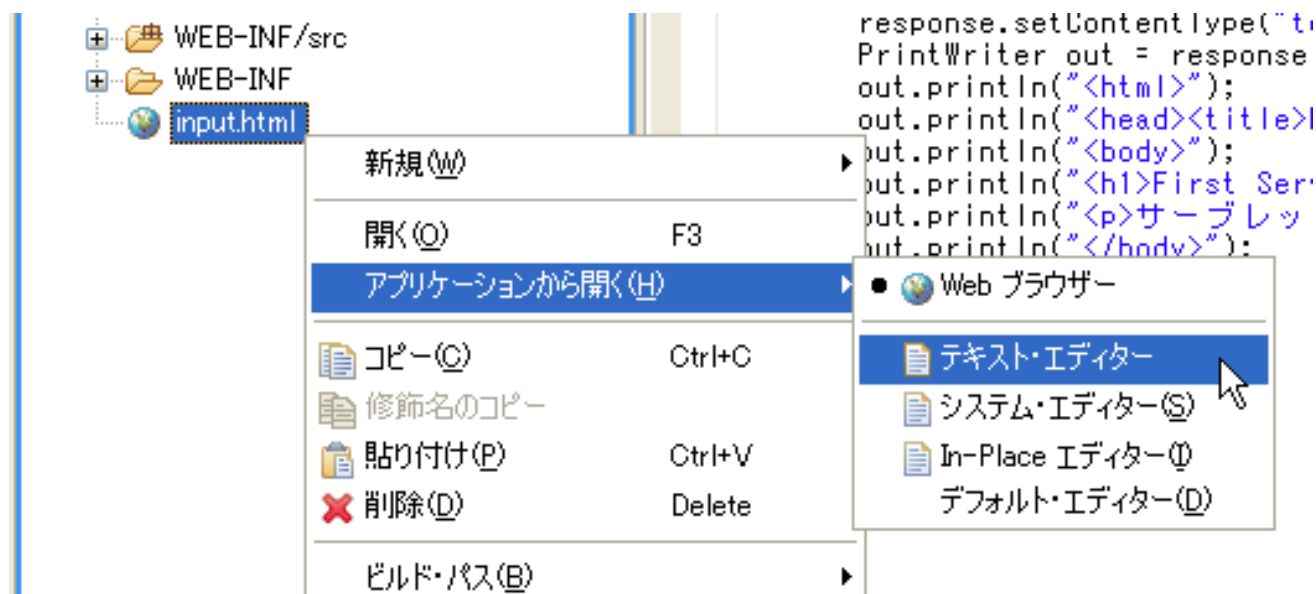
```
String param = request.getParameter("username");
```

演習：パラメータ送受信を行うコードの作成

- 作成するHTMLファイル、サーブレットの仕様
 - HTMLファイル(input.html)
 - <form>タグを使い「HTTP POST」形式で送信する
 - action属性は「http://localhost:8080/kx/second」と指定
 - <input>タグ(テキストフィールド)を使いパラメータを送信する
 - パラメータ名は「param」とする
 - サーブレット(web.SecondServletクラス)
 - doPost()メソッドをオーバーライドして作成
 - HttpServletRequest#setCharacterEncoding()メソッドでエンコーディングをWindows-31Jに指定
 - HttpServletRequest#getParameter()メソッドでパラメータを取得
 - その内容を<h1>タグで囲んでブラウザに表示する
 - web.xmlファイルにサーブレットの設定を記述
 - スライド#87を参照

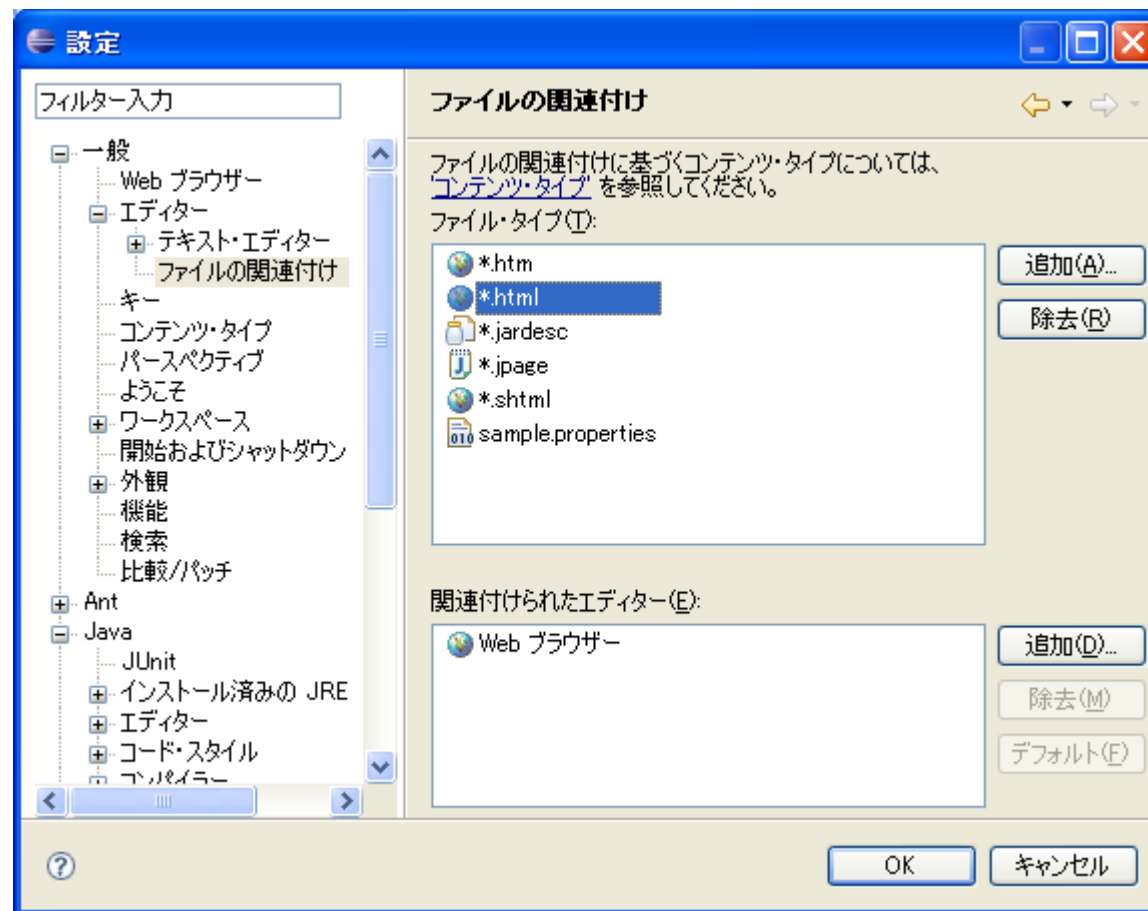
パラメータ送信HTMLファイルの作成

- web.xmlファイルの作成と同じ要領で作成
 - 「プロジェクト名」を選択し右クリック→「新規」→「ファイル」を選択
 - ファイル名に、任意のHTMLファイル名を指定(拡張子は、.htmlとする)
 - 編集するには、ファイルを右クリック→「アプリケーションから開く」→「テキスト・エディター」を選択
 - もしくは、ファイルの関連づけを変更(次スライド参照)



参考: ファイルの関連づけの設定

- 拡張子と使用するエディターの関連づけ
 - メニュー「ウィンドウ」→「設定」を選択
 - ツリーから「一般」→「エディター」→「ファイルの関連付け」を選択



input.htmlのコード例

```
<html>
<head><title>SecondServlet</title></head>
<body>
<h1>Second Servlet</h1>
<p>パラメータを入力してください</p>
<form action="http://localhost:8080/kx/second" method="post">
<input type="text" name="param">
<input type="submit" value="送信する">
</form>
</body>
</html>
```

サーブレットのコード例

```
package web;
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class SecondServlet extends HttpServlet {

    protected void doPost(HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException {

        request.setCharacterEncoding("Windows-31J");
        String param = request.getParameter("param");

        response.setContentType("text/html;charset=Windows-31J");
        PrintWriter out = response.getWriter();
        out.println("<html>");
        out.println("<head><title>Second Servlet</title></head>");
        out.println("<body>");
        out.println("あなたの入力した内容は、");
        out.println("<h1>"+param+"</h1>");
        out.println("ですね");
        out.println("</body>");
        out.println("</html>");
        out.close();
    }

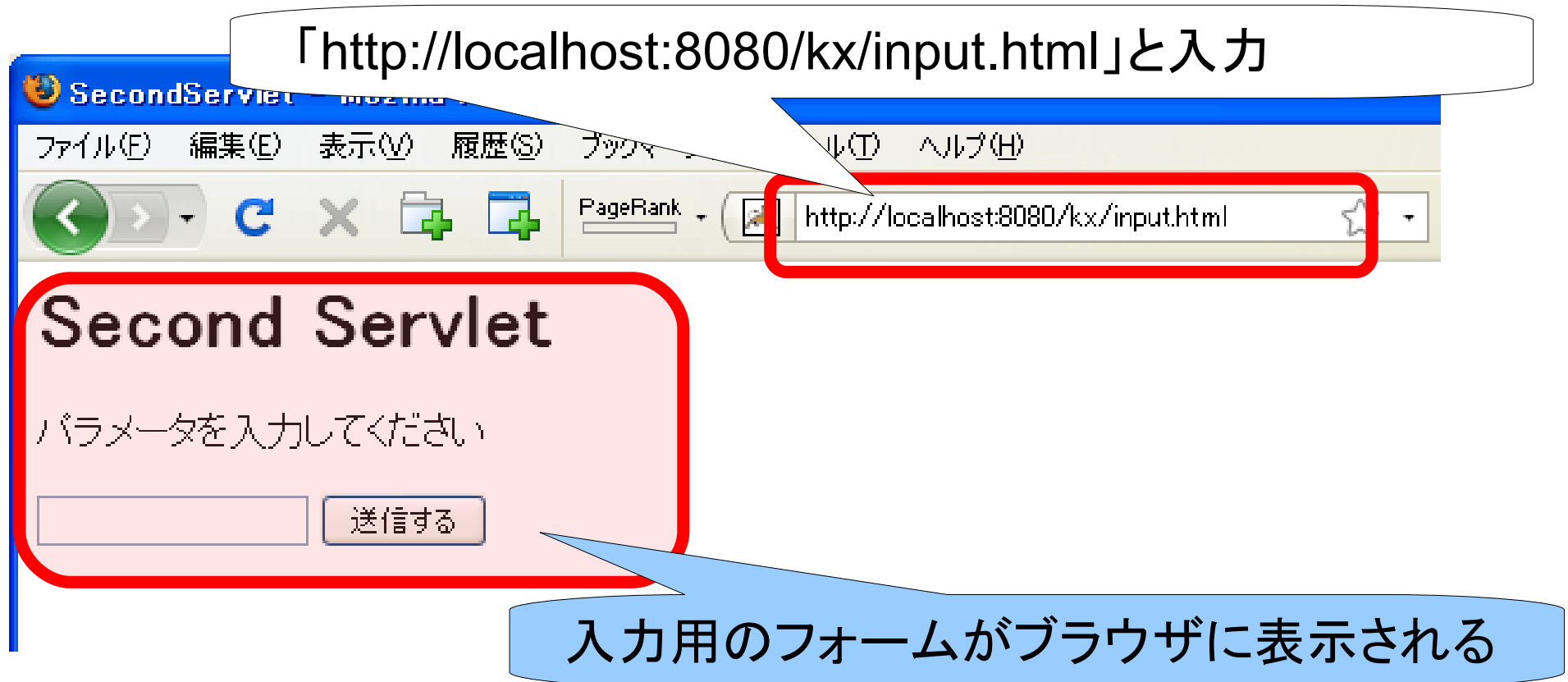
}
```

web.xmlのコード例

```
<?xml version="1.0" encoding="Windows-31J"?>
<web-app>
  <servlet>
    <servlet-name>first</servlet-name>
    <servlet-class>web.FirstServlet</servlet-class>
  </servlet>
  <servlet>
    <servlet-name>second</servlet-name>
    <servlet-class>web.SecondServlet</servlet-class>
  </servlet>
  <servlet-mapping>
    <servlet-name>first</servlet-name>
    <url-pattern>/first</url-pattern>
  </servlet-mapping>
  <servlet-mapping>
    <servlet-name>second</servlet-name>
    <url-pattern>/second</url-pattern>
  </servlet-mapping>
</web-app>
```

実行例

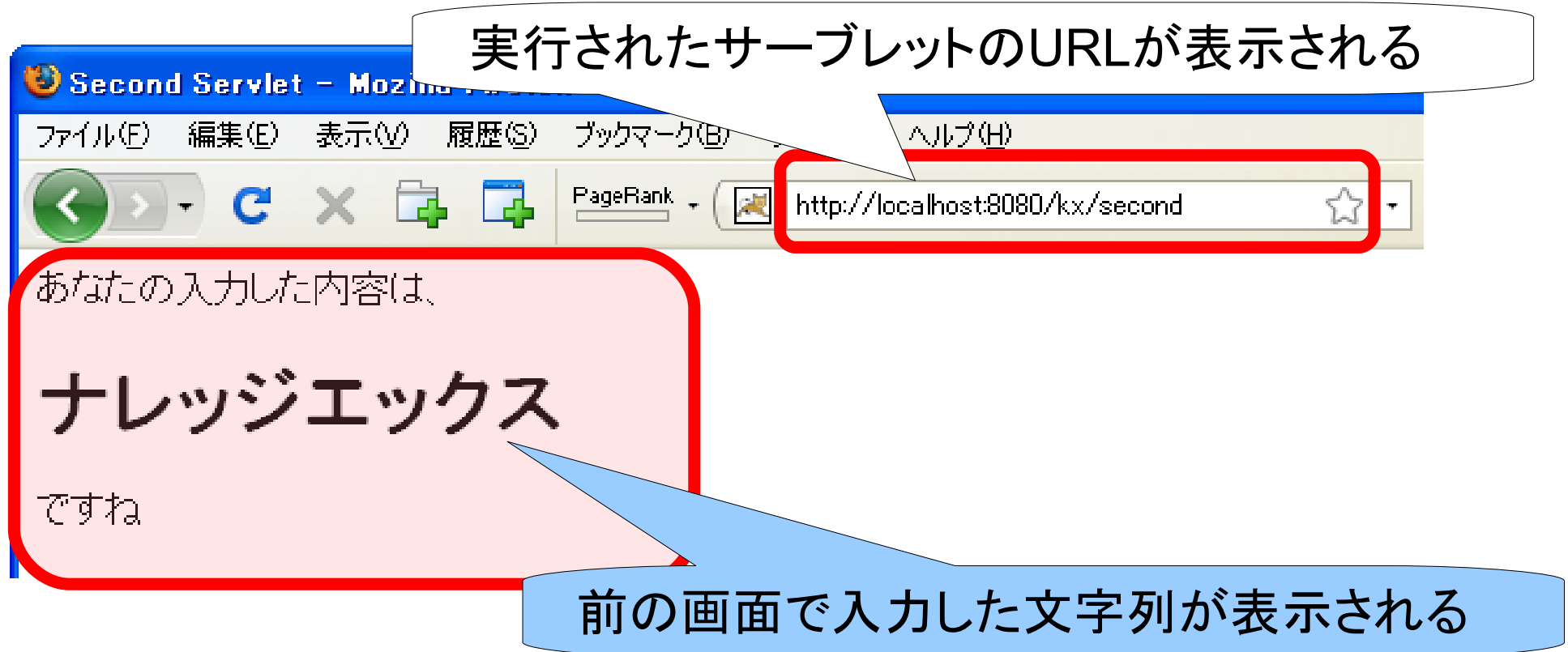
- HTMLファイルの呼び出し
 - ブラウザからHTMLファイルを呼び出すためのURLを入力



実行例

- サーブレットの実行

- 「送信する」ボタンを押下するとサーブレットが実行され、結果がブラウザに表示される



初期化パラメータの設定


- サーブレットに対し「初期化パラメータ」をweb.xmlに設定することができる
 - 初期化パラメータを設定することで、コードの内容は変更せずに、サーブレットに関連する設定情報などを変更することが可能
 - 例) データベースへの接続情報(URL、ID、パスワード)

初期化パラメータの記述

- 初期化パラメータはweb.xmlに記述
 - <servlet>タグ内に、<init-param>タグを追加
 - <init-param>タグの中に
 - <param-name>タグを使って「パラメータ名」
 - <param-value>タグを使って「パラメータ値」を記述
 - <init-param>タグはパラメータが複数ある場合は繰り返し記述することが可能

記述例

```
<servlet>
  <servlet-name>first</servlet-name>
  <servlet-class>web.FirstServlet</servlet-class>
  <init-param>
    <param-name>username</param-name>
    <param-value>Taro</param-value>
  </init-param>
  <init-param>
    <param-name>userage</param-name>
    <param-value>30</param-value>
  </init-param>
</servlet>
```




初期化
パラメータ

初期化パラメータの読み込み

- HttpServlet#getInitParameter()メソッド
 - HttpServletがもつgetInitParameter()メソッドを実行する
 - 引数にパラメータ名を指定すると、初期化パラメータを文字列で取得できる

凡例

```
HttpServlet#getInitParameter(String) ;
```



初期化パラメータ名

コード例

```
String initParam = getInitParameter("username") ;
```

演習

- 初期化パラメータを読み込んで表示するサーブレットを作成しましょう
 - getInitParameter()で初期化パラメータを読み込んで、それをブラウザ上に表示するサーブレットクラスを作成
 - 作成したサーブレットに対する設定(初期化パラメータ含む)をweb.xmlに追加
 - 実行して結果を確認

サーブレットのコード例

```
package web;
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class InitParamServlet extends HttpServlet {

    protected void doGet(HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException {
        String param = getInitParameter("param");

        response.setContentType("text/html;charset=Windows-31J");
        PrintWriter out = response.getWriter();
        out.println("<html>");
        out.println("<head><title>InitParam Servlet</title></head>");
        out.println("<body>");
        out.println("初期化パラメータは、");
        out.println("<h1>"+param+"</h1>");
        out.println("ですね");
        out.println("</body>");
        out.println("</html>");
        out.close();
    }
}
```

web.xmlのコード例

```
<?xml version="1.0" encoding="Windows-31J"?>
<web-app>
  <servlet>
    <servlet-name>initparam</servlet-name>
    <servlet-class>web.InitParamServlet</servlet-class>
    <init-param>
      <param-name>param</param-name>
      <param-value>Knowledge-ex.</param-value>
    </init-param>
  </servlet>
  <servlet-mapping>
    <servlet-name>initparam</servlet-name>
    <url-pattern>/initparam</url-pattern>
  </servlet-mapping>
</web-app>
```

Servletのエラーとその対処

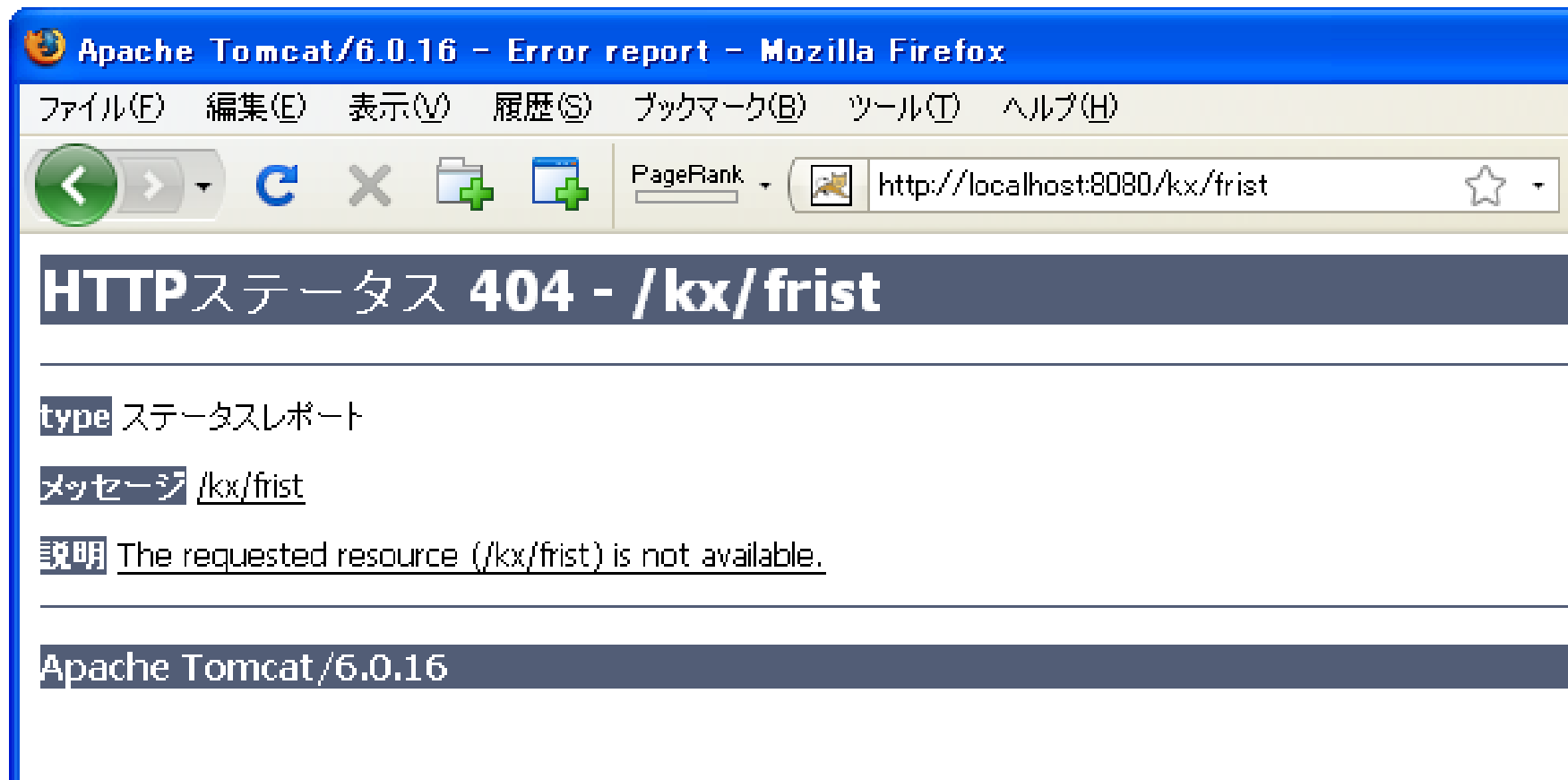
株式会社ナレッジエックス
<http://www.knowledge-ex.jp/>

Servlet実行時のエラーについて

- Servletを実行した際のエラーについては、ブラウザに表示される内容から下記のように分類できる
 - ブラウザに表示されるエラー
 - 404エラー
 - 500エラー
 - 405エラー
 - 「サーバが見つからない」エラー
 - Tomcatのログに表示される例外
 - SAXParseException
 - BindException

404エラー

- 「Not Found」エラーとも呼ばれ、リクエストした内容がサーバに存在しなかったことを示す



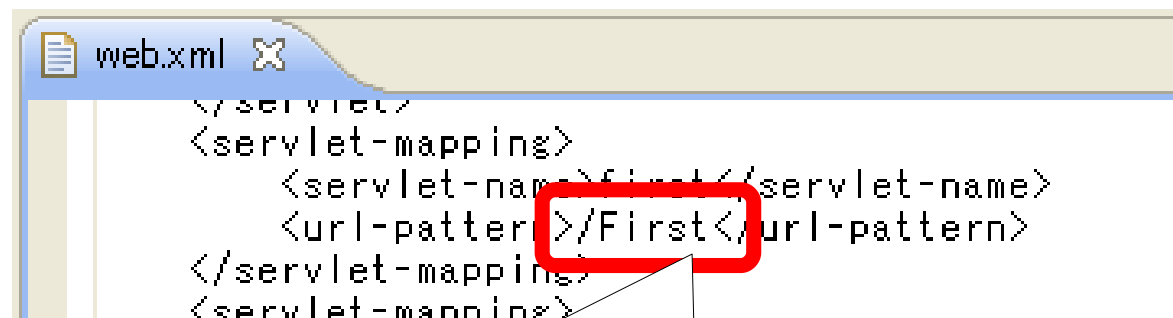
404エラーの原因(1)

- 指定したURLに誤りがある



「/first」を「/frist」と入力してしまった

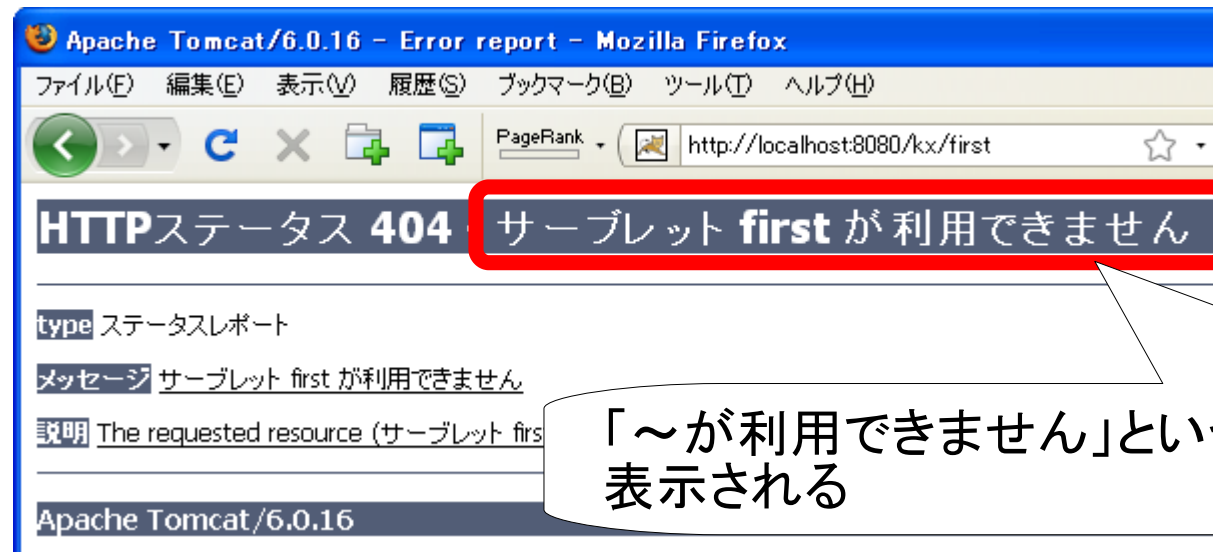
- web.xmlのサーブレットパスの記載が誤っている



「/first」を「/First」と記載してしまった

404エラーの原因(2)

- 以前の実行でサーブレットが異常な状態となり実行不可能となった



対策

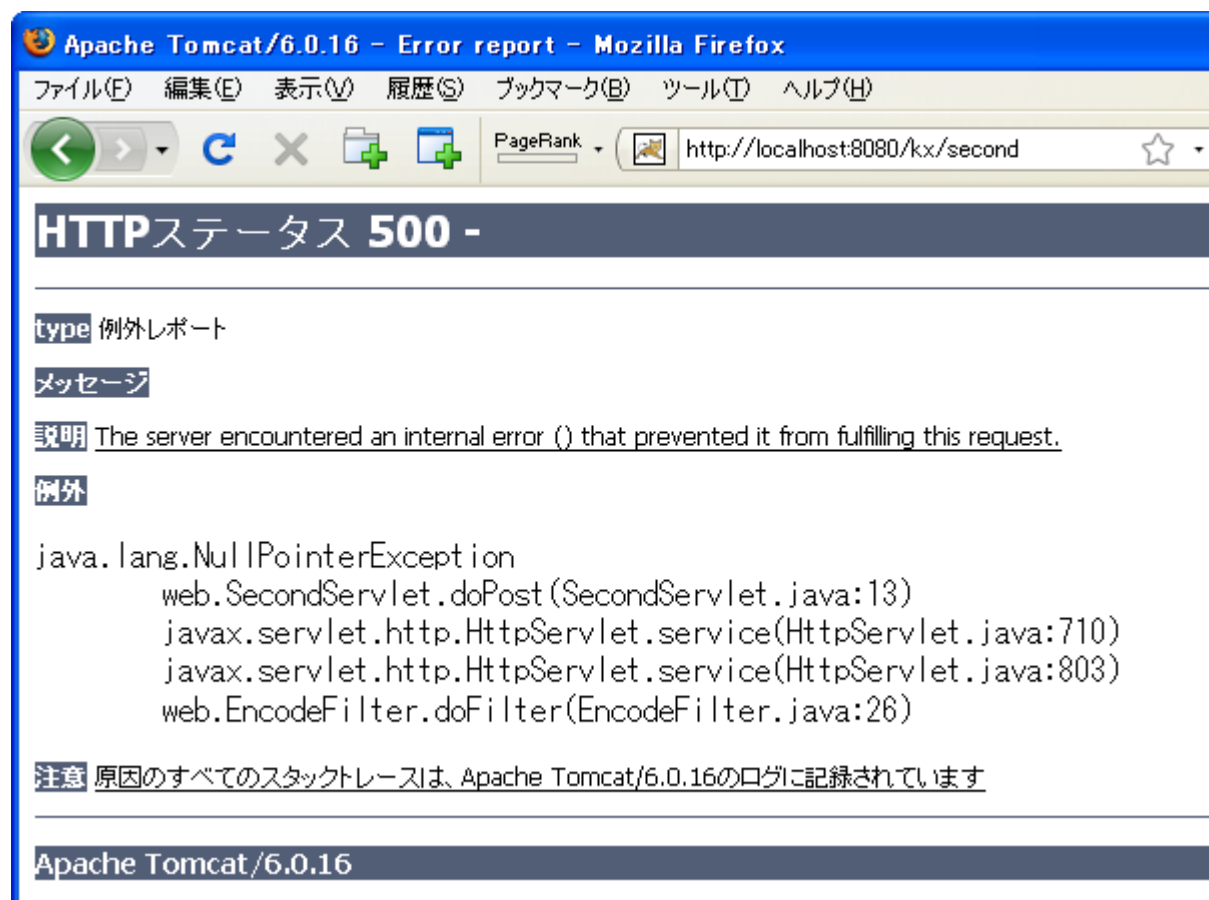
サーブレットのコードを正しく修正し、
Tomcatを再起動して再度実行する

404エラーの原因(3)

- それまで動いていたものも全て動かなくなった場合は、Tomcatの起動ログを調査する(後述)

500エラー

- 「Internal Server Error」とも呼ばれ、サーバ内部でエラーが発生したことを示す



500エラーの原因

- 実行中のサーブレットから例外が送出された

Apache Tomcat/6.0.16 - Error report - Mozilla Firefox

ファイル(F) 編集(E) 表示(V) 履歴(S) ブックマーク(B) ツール(T) ヘルプ(H)

PageRank http://localhost:8080/kx/second

HTTPステータス 500 -

type 例外レポート

メッセージ

説明 The server encountered an internal error () that prevented it from fulfilling the request.

例外

```
java.lang.NullPointerException
    web.SecondServlet.doPost(SecondServlet.java:13)
    javax.servlet.http.HttpServlet.service(HttpServlet.java:710)
    javax.servlet.http.HttpServlet.service(HttpServlet.java:803)
    web.EncodeFilter.doFilter(EncodeFilter.java:26)
```

注意 原因のすべてのスタックトレースは、Apache Tomcat/6.0.16のログに記録されています

Apache Tomcat/6.0.16

発生した例外のスタックトレースが表示される

405エラー

- 「Method Not Allowed」エラーとも呼ばれ、リクエストできない形式(GET/POST)であることを示す



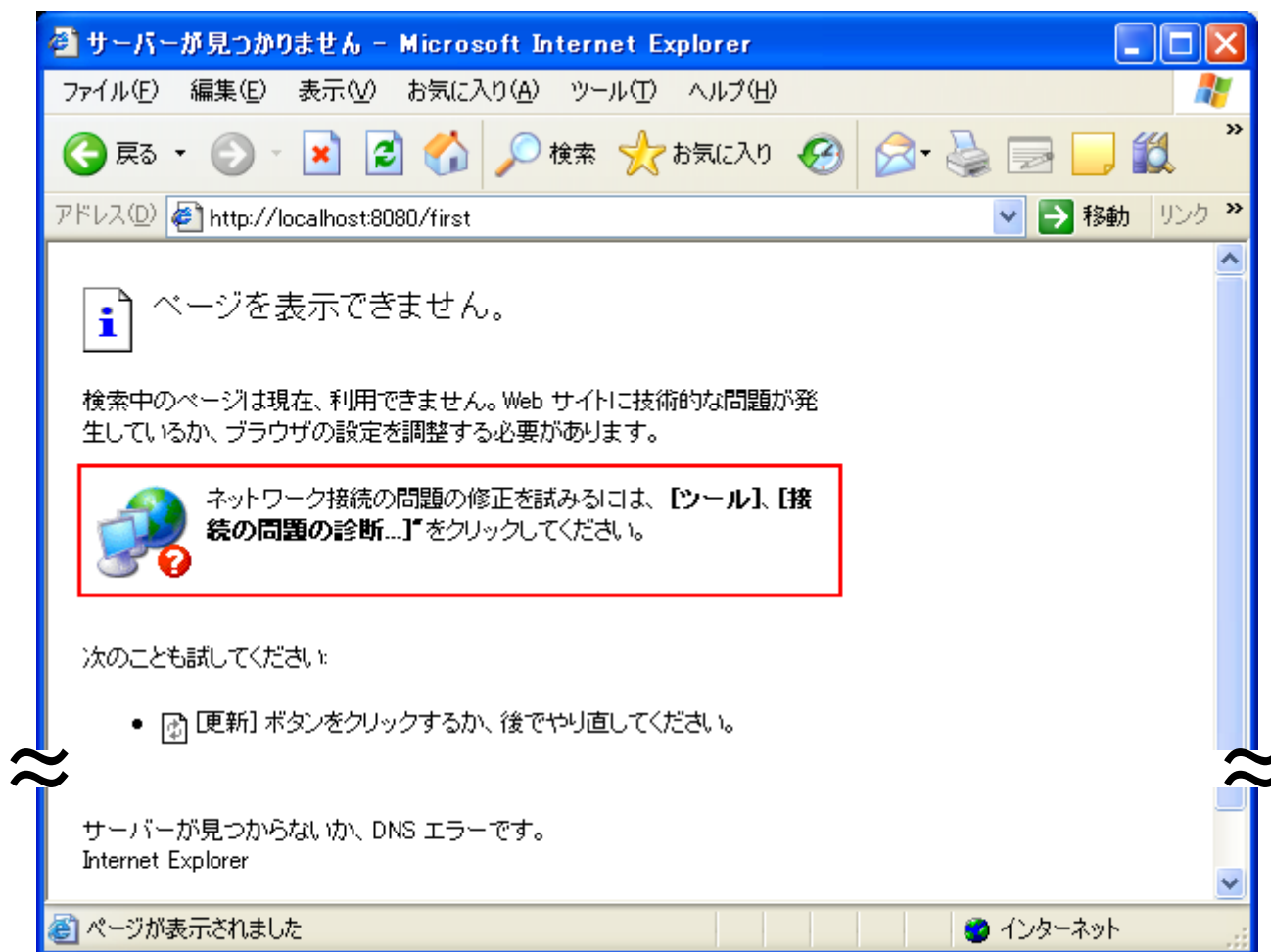
405エラーの原因

- doGetメソッドのないサーブレットをGETリクエストで実行しようとした(URL直接指定やリンク)
- doPostメソッドのないサーブレットをPOSTリクエストで実行しようとした(フォームからの送信)



「サーバが見つからない」エラー

- Internet Explorerの場合



「サーバが見つからない」エラー

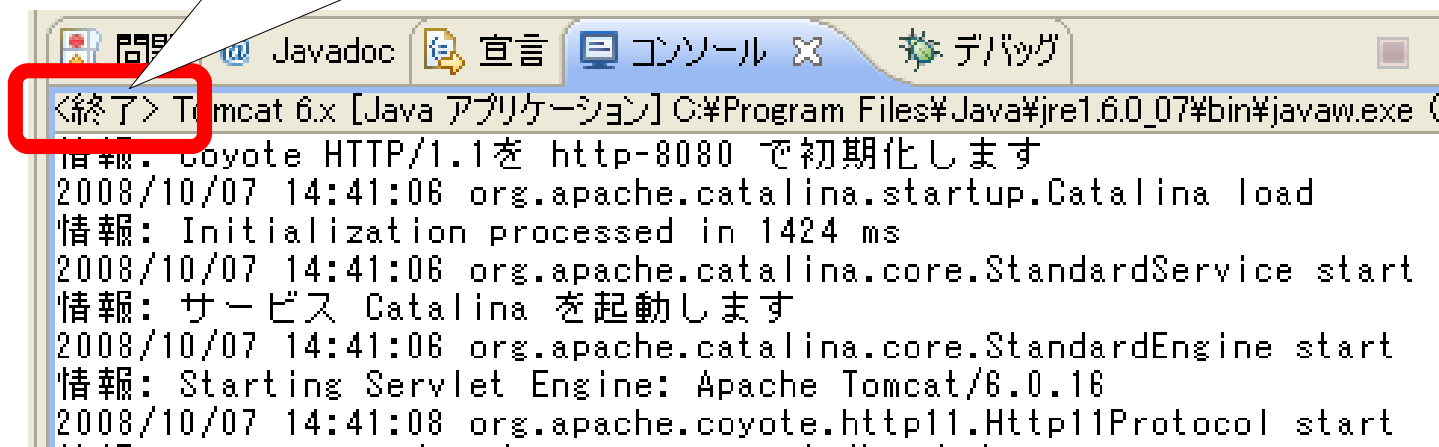
- Firefoxの場合



「サーバが見つからない」エラーの原因

- Tomcatが起動していないことが原因
 - 再起動中の場合は、起動が完了してから再度実行
 - もともとTomcatを起動させていない場合は起動させる

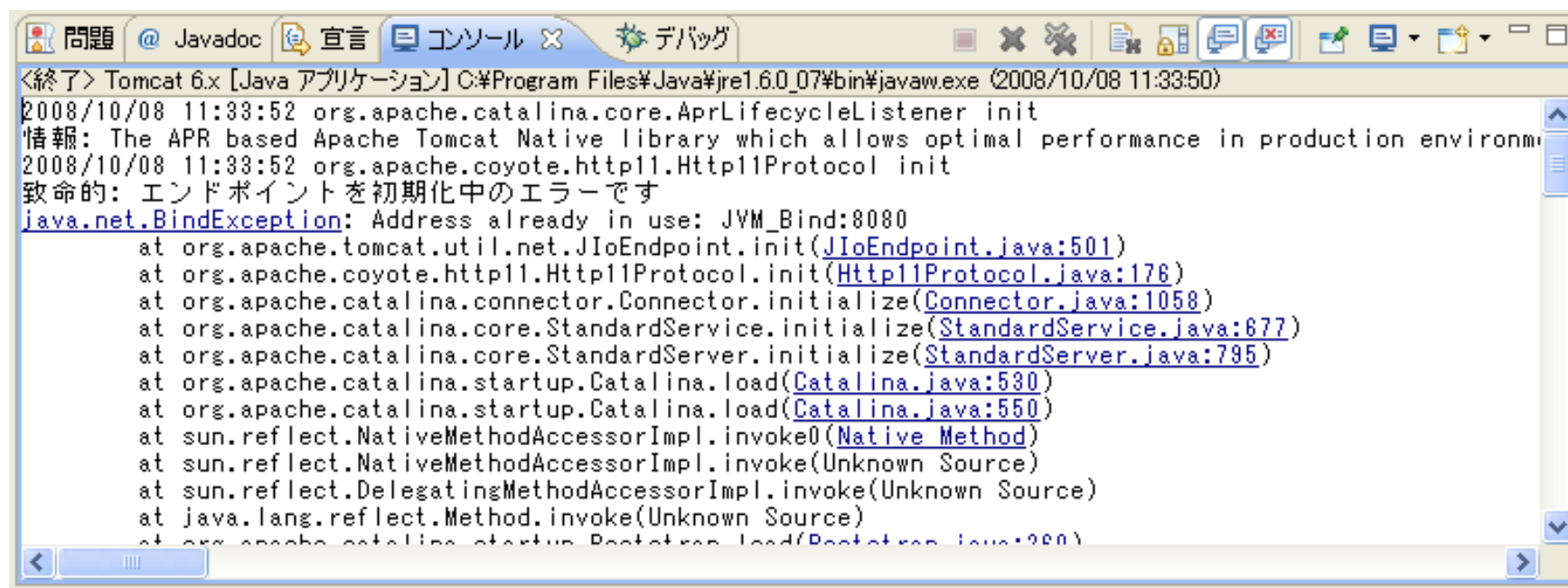
Tomcatが終了していることを示す表示



```
<終了> Tomcat 6.x [Java アプリケーション] C:\Program Files\Java\jre1.6.0_07\bin\javaw.exe (
情報: Coyote HTTP/1.1を http-8080 で初期化します
2008/10/07 14:41:06 org.apache.catalina.startup.Catalina load
情報: Initialization processed in 1424 ms
2008/10/07 14:41:06 org.apache.catalina.core.StandardService start
情報: サービス Catalina を起動します
2008/10/07 14:41:06 org.apache.catalina.core.StandardEngine start
情報: Starting Servlet Engine: Apache Tomcat/6.0.16
2008/10/07 14:41:08 org.apache.coyote.http11.Http11Protocol start
```

Tomcatのログに表示される例外(1)

- java.net.BindException
 - ポートが使用中で割り当てができなかったことを示す

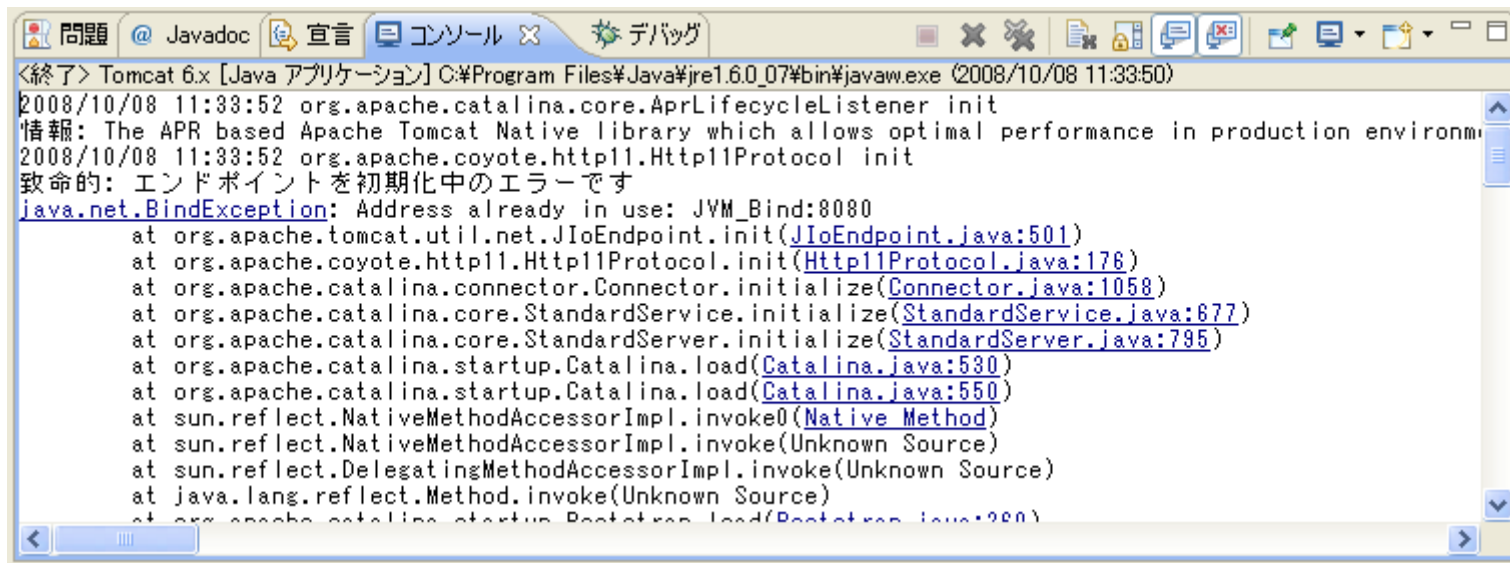


The screenshot shows a Java IDE console window with the following content:

```
<終了> Tomcat 6.x [Java アプリケーション] C:\Program Files\Java\jre1.6.0_07\bin\javaw.exe (2008/10/08 11:33:50)
2008/10/08 11:33:52 org.apache.catalina.core.AprLifecycleListener init
情報: The APR based Apache Tomcat Native library which allows optimal performance in production environm
2008/10/08 11:33:52 org.apache.coyote.http11.Http11Protocol init
致命的: エンドポイントを初期化中のエラーです
java.net.BindException: Address already in use: JVM_Bind:8080
    at org.apache.tomcat.util.net.JIoEndpoint.init(JIoEndpoint.java:501)
    at org.apache.coyote.http11.Http11Protocol.init(Http11Protocol.java:176)
    at org.apache.catalina.connector.Connector.initialize(Connector.java:1058)
    at org.apache.catalina.core.StandardService.initialize(StandardService.java:677)
    at org.apache.catalina.core.StandardServer.initialize(StandardServer.java:795)
    at org.apache.catalina.startup.Catalina.load(Catalina.java:530)
    at org.apache.catalina.startup.Catalina.load(Catalina.java:550)
    at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
    at sun.reflect.NativeMethodAccessorImpl.invoke(Unknown Source)
    at sun.reflect.DelegatingMethodAccessorImpl.invoke(Unknown Source)
    at java.lang.reflect.Method.invoke(Unknown Source)
    at org.apache.catalina.startup.Bootstrap.load(Bootstrap.java:280)
```

BindExceptionの原因(1)

- すでにTomcatを起動している状態で、もうひとつ別のTomcatを起動しようとした



The screenshot shows a Java IDE console window with the following text:

```
<終了> Tomcat 6.x [Java アプリケーション] C:\Program Files\Java\jre1.6.0_07\bin\javaw.exe (2008/10/08 11:33:50)
2008/10/08 11:33:52 org.apache.catalina.core.AprLifecycleListener init
情報: The APR based Apache Tomcat Native library which allows optimal performance in production environm
2008/10/08 11:33:52 org.apache.coyote.http11.Http11Protocol init
致命的: エンドポイントを初期化中のエラーです
java.net.BindException: Address already in use: JVM_Bind:8080
    at org.apache.tomcat.util.net.JIoEndpoint.init(JIoEndpoint.java:501)
    at org.apache.coyote.http11.Http11Protocol.init(Http11Protocol.java:176)
    at org.apache.catalina.connector.Connector.initialize(Connector.java:1058)
    at org.apache.catalina.core.StandardService.initialize(StandardService.java:677)
    at org.apache.catalina.core.StandardServer.initialize(StandardServer.java:795)
    at org.apache.catalina.startup.Catalina.load(Catalina.java:530)
    at org.apache.catalina.startup.Catalina.load(Catalina.java:550)
    at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
    at sun.reflect.NativeMethodAccessorImpl.invoke(Unknown Source)
    at sun.reflect.DelegatingMethodAccessorImpl.invoke(Unknown Source)
    at java.lang.reflect.Method.invoke(Unknown Source)
    at org.apache.catalina.startup.Bootstrap.load(Bootstrap.java:280)
```

BindExceptionの原因(2)

- 終了済みのログを消去し、起動中のTomcatが存在していることを確認

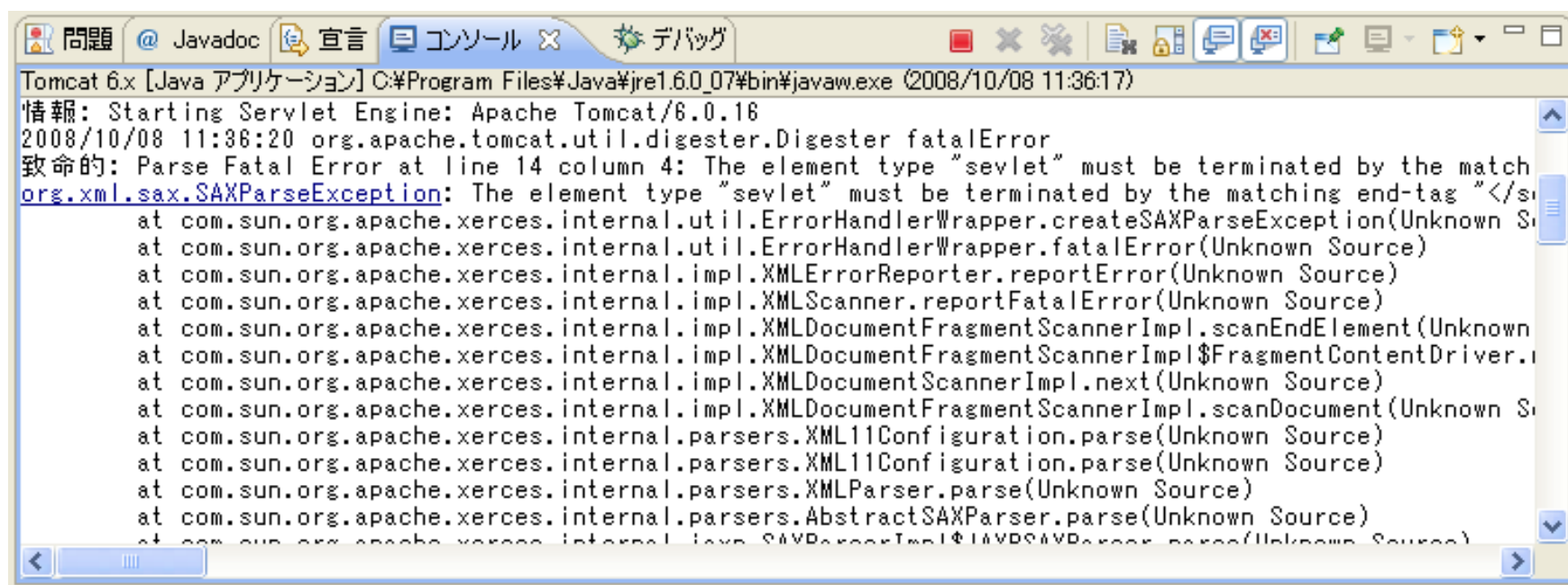
The screenshot shows two instances of the Tomcat console window. The top window displays a `java.net.BindException: Address already in use: JVM_Bind:8080` error. A red circle highlights the 'X' icon in the toolbar, which is used to clear the log. A callout bubble points to this icon with the text: 「終了した全ての起動を除去」ボタンをクリックし、起動中のTomcatのログだけを表示させる. The bottom window shows the console after the logs have been cleared, displaying the startup sequence of Tomcat 6.0.16, including messages like 'Coyote HTTP/1.1を http-8080 で初期化します' and 'Server startup in 10460 ms'. A blue arrow points from the top window to the bottom one. A light blue callout bubble on the right side of the bottom window contains the text: 起動中のTomcatのログに表示が切り替わる.

「終了した全ての起動を除去」ボタンをクリックし、起動中のTomcatのログだけを表示させる

起動中のTomcatのログに表示が切り替わる

Tomcatのログに表示される例外(2)

- org.xml.sax.SAXParseException
 - XMLドキュメントのパーズ(解釈)に失敗したことを示す



The screenshot shows a Windows-style console window titled "Tomcat 6.x [Java アプリケーション]". The window contains the following text:

```
Tomcat 6.x [Java アプリケーション] C:\Program Files\Java\jre1.6.0_07\bin\javaw.exe (2008/10/08 11:36:17)
情報: Starting Servlet Engine: Apache Tomcat/6.0.16
2008/10/08 11:36:20 org.apache.tomcat.util.digester.Digester fatalError
致命的: Parse Fatal Error at line 14 column 4: The element type "sevlet" must be terminated by the match
org.xml.sax.SAXParseException: The element type "sevlet" must be terminated by the matching end-tag "</s
at com.sun.org.apache.xerces.internal.util.ErrorHandlerWrapper.createSAXParseException(Unknown Source)
at com.sun.org.apache.xerces.internal.util.ErrorHandlerWrapper.fatalError(Unknown Source)
at com.sun.org.apache.xerces.internal.impl.XMLErrorReporter.reportError(Unknown Source)
at com.sun.org.apache.xerces.internal.impl.XMLScanner.reportFatalError(Unknown Source)
at com.sun.org.apache.xerces.internal.impl.XMLDocumentFragmentScannerImpl.scanEndElement(Unknown Source)
at com.sun.org.apache.xerces.internal.impl.XMLDocumentFragmentScannerImpl$FragmentContentDriver.next(Unknown Source)
at com.sun.org.apache.xerces.internal.impl.XMLDocumentScannerImpl.next(Unknown Source)
at com.sun.org.apache.xerces.internal.impl.XMLDocumentFragmentScannerImpl.scanDocument(Unknown Source)
at com.sun.org.apache.xerces.internal.parsers.XML11Configuration.parse(Unknown Source)
at com.sun.org.apache.xerces.internal.parsers.XML11Configuration.parse(Unknown Source)
at com.sun.org.apache.xerces.internal.parsers.XMLParser.parse(Unknown Source)
at com.sun.org.apache.xerces.internal.parsers.AbstractSAXParser.parse(Unknown Source)
at com.sun.org.apache.xerces.internal.impl.SAXParserImpl$SAXParser.parse(Unknown Source)
```


SAXParseExceptionの原因(1)

- Tomcatが設定に必要とするXMLドキュメント (web.xmlまたはserver.xml) に文法の誤りがある
 - 例外の詳細メッセージや前後のログ内容で判断

①例外直前のログ

致命的: Parse Fatal Error at line 14 column 4:

解析に失敗した箇所が14行目の4カラム目である

②例外の詳細メッセージ

The element type "sevlet" must be terminated by the matching end-tag "</sevlet>".

(「servlet」要素がこれにマッチする「</servlet>」で閉じられていなければならない)

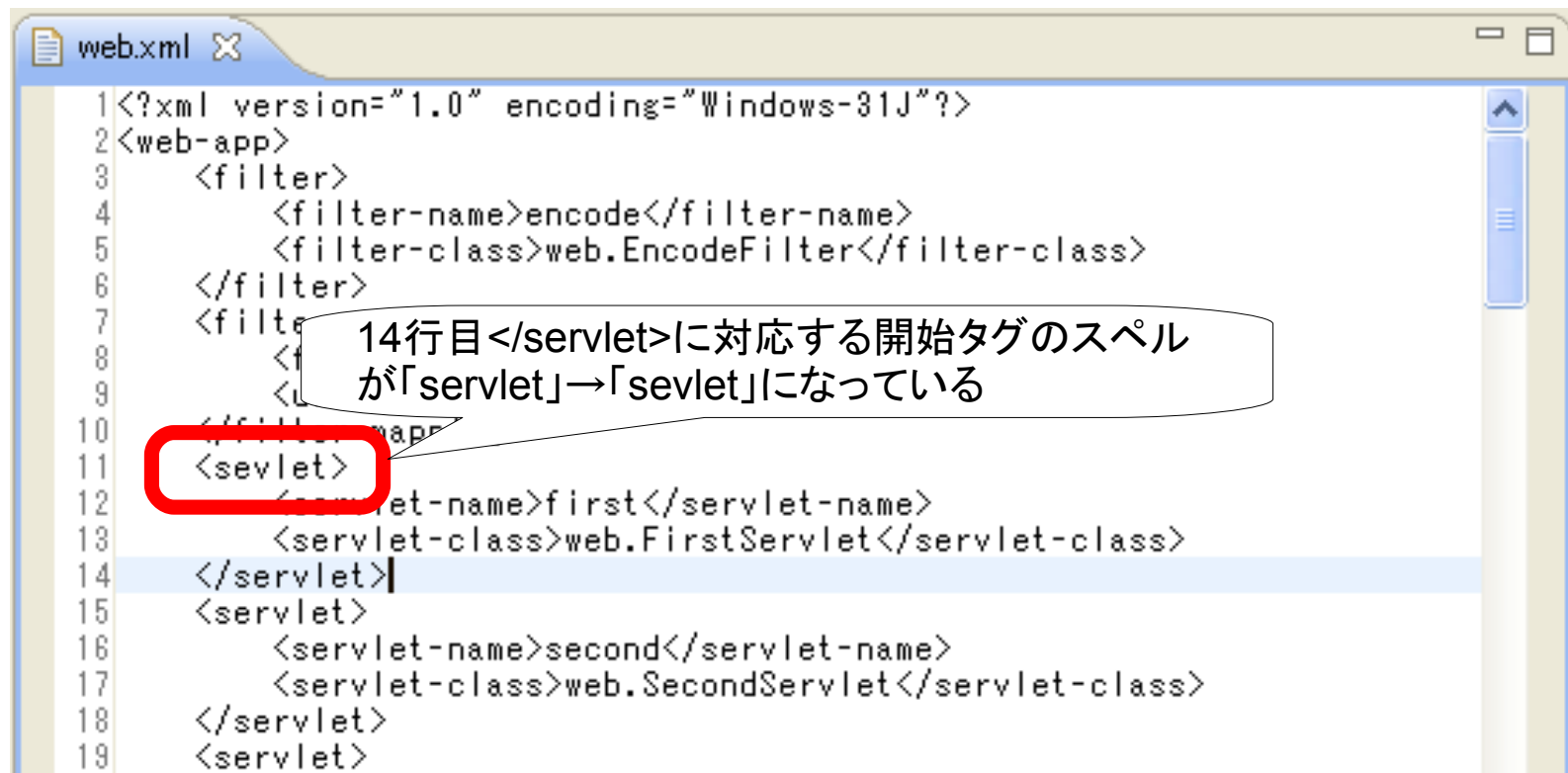
③例外直後のログ

致命的: アプリケーションの web.xml ファイル jndi:/localhost/kx/WEB-INF/web.xml の解析エラーです

解析に失敗したファイルはweb.xmlである

SAXParseExceptionの原因(2)

- web.xmlのタグの記述に誤りがあった
 - 誤りを修正し、Tomcatを再起動して、同様の例外がログに表示されないことを確認



The screenshot shows a code editor window titled 'web.xml'. The XML content is as follows:

```
1<?xml version="1.0" encoding="Windows-31J"?>
2<web-app>
3  <filter>
4    <filter-name>encode</filter-name>
5    <filter-class>web.EncodeFilter</filter-class>
6  </filter>
7  <filter>
8    <filter-name>first</filter-name>
9    <filter-class>web.FirstFilter</filter-class>
10 </filter>
11 <filter>
12 </filter>
13 <filter>
14 </filter>
15 <filter>
16 </filter>
17 <filter>
18 </filter>
19 </web-app>
```

A red circle highlights the tag `<sevlet>` on line 11. A callout box points to line 14, stating: "14行目</servlet>に対応する開始タグのスペルが「servlet」→「sevlet」になっている" (The spelling of the start tag corresponding to line 14's </servlet> is 'servlet' → 'sevlet').

SAXParseExceptionが起きると・・・

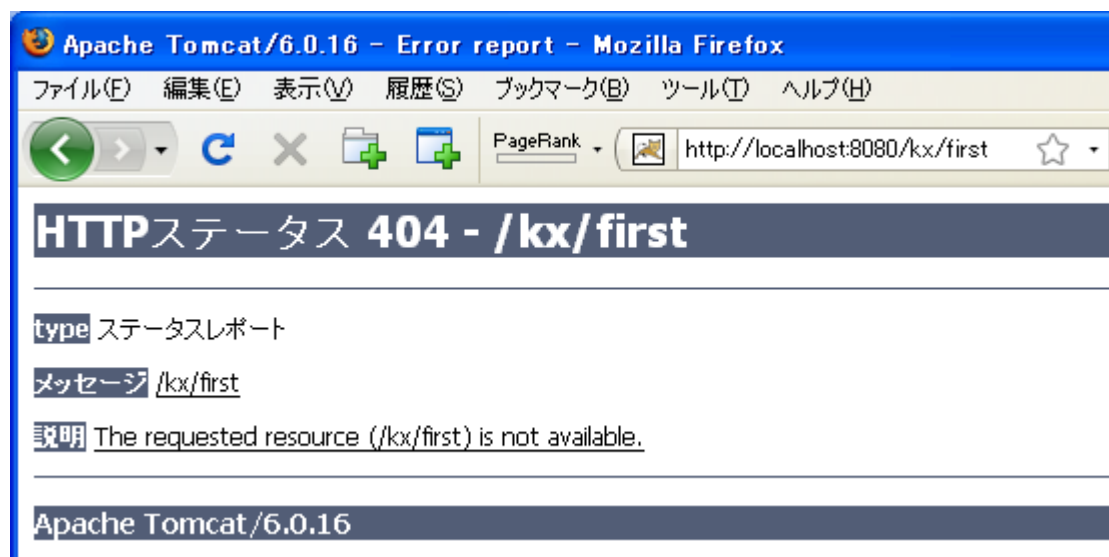
- SAXParseExceptionが発生すると、該当するアプリケーション全体が使用不能になるので、ブラウザからアクセスしても404エラーになる



```
Tomcat 6.x [Java アプリケーション] C:\Program Files\Java\jre1.6.0_07\bin\javaw.exe (2008/10/08 11:36:17)
2008/10/08 11:36:20 org.apache.catalina.startup.ContextConfig applicationWebConfig
致命的: 14行の4列目で発生しました
致命的: 前のエラーのためにこのアプリケーションは利用できないようにマークします
2008/10/08 11:36:20 org.apache.catalina.startup.ContextConfig
致命的: Error getConfigured
2008/10/08 11:36:20 org.apache.catalina.core.StandardContext start
致命的: 以前のエラーのためにコンテキストの起動が失敗しました [/kx]
2008/10/08 11:36:21 org.apache.coyote.http11.Http11Protocol start
情報: Coyote HTTP/1.1を http-8080 で起動します
2008/10/08 11:36:21 org.apache.jk.common.ChannelSocket init
情報: JK: ajp13 listening on /0.0.0.0:8009
2008/10/08 11:36:21 org.apache.jk.server.JkMain start
情報: Jk running ID=0 time=0/62 config=null
2008/10/08 11:36:21 org.apache.catalina.startup.Catalina start
情報: Server startup in 1699 ms
```

web.xmlの解析に失敗したため、アプリケーション全体が使用不能となったことを示すログ

この状態でブラウザからアクセスしても、404エラーになってしまう



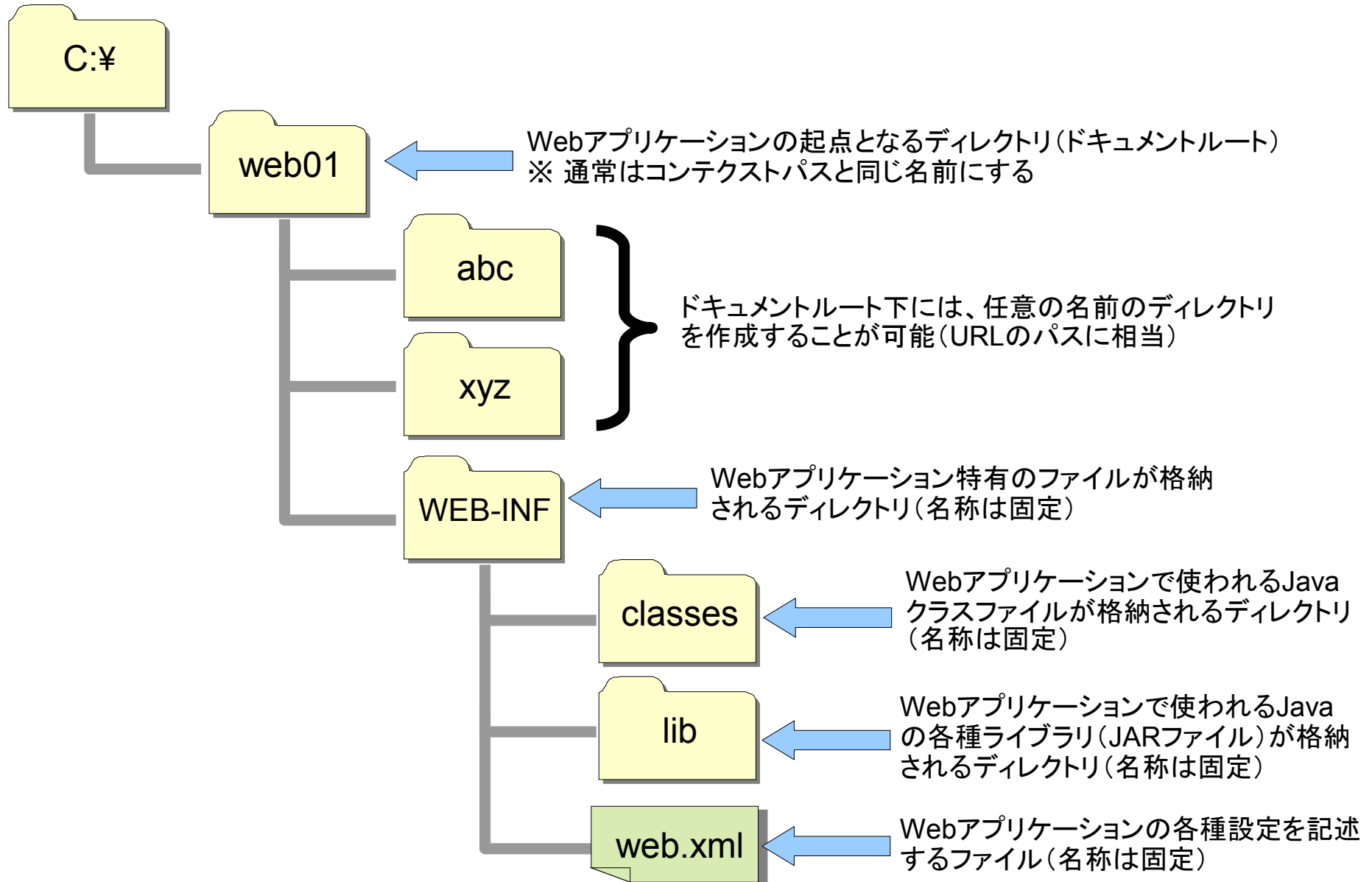
コンテナへのデプロイ

株式会社ナレッジエックス
<http://www.knowledge-ex.jp/>

コンテナへのデプロイとは？

- コンテナへのデプロイとは？
 - 作成したWebアプリケーションを、コンテナ上で動作するようにセッティングすること(=配備)
- デプロイの手段は2種類
 - ディレクトリ単位でのデプロイ
 - WARファイル単位でのデプロイ

Webアプリケーションのディレクトリ構成



ディレクトリ単位のデプロイ(1)

- ディレクトリ単位のデプロイの方法
 - Webアプリケーションを格納したディレクトリをコンテナに認識させる
 - コンテナがWebアプリケーション格納用として用意したディレクトリに移動
 - コンテナの設定ファイルにディレクトリの位置を記述して認識させる

ディレクトリ単位のデプロイ(2)

- ①コンテナが用意したディレクトリへ移動
 - Apache Tomcatを利用する場合のディレクトリ
 - [Tomcatのインストールディレクトリ]/webapps
 - そのディレクトリに配置すれば、Tomcatが自動的に認識してくれる
 - アプリケーションのコンテキストパスは、ディレクトリ名と同じになる

ディレクトリ単位のデプロイ(3)

- ②コンテナの設定ファイルに位置を記述
 - Apache Tomcatを利用する場合のファイル名
 - [Tomcatのインストールディレクトリ]/conf/server.xml
 - このファイルにディレクトリ位置を登録すれば、Tomcatが自動的に認識してくれる
 - アプリケーションのコンテキストパスはディレクトリ名と同じでなくても良い

ディレクトリ単位のデプロイ(4)

- ②コンテナの設定ファイルに位置を記述
 - EclipseでTomcat Plug-inを利用する場合
 - プロジェクトの設定
 - プロジェクト名を選択して右クリックし「プロパティ」を選択
 - ツリーから「Tomcat」を選択
 - 「アプリケーションURI」がコンテキストパスなので、任意の名称に変更
 - server.xmlへの反映
 - プロジェクト名を選択して右クリックし「Tomcatプロジェクト」→「Tomcatのコンテキストを更新」を選択
 - 「更新に成功しました」と出れば反映は成功
 - Tomcatが起動中の場合は、再起動する

ディレクトリ単位のデプロイ(5)

- server.xmlに追加される内容(一部)
 - <Context>タグ
 - path属性
 - コンテキストパスの指定
 - docBase属性
 - ドキュメントルートのディレクトリ位置の指定
 - workDir属性
 - Tomcatが一時的に作成するファイルを配置するディレクトリの指定
 - reloadable属性
 - サーブレットやJSPの内容が更新されたとき、Tomcatにその内容をリロードして反映するかどうかを指定(true|false)

WARファイル単位のデプロイ(1)

- WARファイル(Web ARchiveファイル)とは
 - Webアプリケーションのディレクトリ構成を1つのファイルにまとめ圧縮したもの
 - 拡張子は「.war」

WARファイル単位のデプロイ(2)

- WARファイルの作成方法

- EclipseでTomcat Plug-inを利用する場合

- プロジェクトの設定

- プロジェクト名を選択して右クリックし「プロパティ」を選択
 - ツリーから「Tomcat」を選択
 - タブ「WARエクスポートの設定」を選択
 - ファイル名のところに、ディレクトリ名とファイル名を入力
 - ファイル名がコンテキストパスになるので注意

- WARファイルの生成

- プロジェクト名を選択して右クリックし「Tomcatプロジェクト」→「プロジェクト設定に従いWARファイルを作成」を選択
 - 「操作に成功しました」と出れば作成は成功
 - エクスプローラなどで、ファイルができていることを確認してみる

WARファイル単位のデプロイ(3)

- WARファイルのデプロイ方法
 - ①所定のディレクトリに直接ファイルを置く
 - [Tomcatのインストールディレクトリ]/webapps にWARファイルを置く
 - Tomcatが起動していれば、一定時間後にWARファイルを認識し、内容を展開してくれる
 - Tomcatが停止中の場合は起動すると認識される
 - 同じWARファイルを再度デプロイしたい場合
 - Tomcatを停止する
 - webappsディレクトリ内にあるWARファイルと同名のディレクトリを削除する
 - WARファイルを置き換える
 - Tomcatを起動する

WARファイル単位のデプロイ(4)

- WARファイルのデプロイ方法
 - ②Tomcatの管理画面からアップロードする
 - Tomcatが起動していることを確認
 - Webブラウザから「http://localhost:8080/」にアクセス
 - 「Tomcat Manager」リンクをクリック
 - IDとパスワードは、「admin」「パスワードなし」を入力（インストール時にデフォルトの設定でインストールした場合）
 - 画面下の「WARファイルの配備」の「参照」ボタンを押下し、作成しておいたWARファイルを選択
 - 「配備」ボタンを押下するとWARファイルがアップロードされる
 - [Tomcatのインストールディレクトリ]/webappsに内容が展開される

WARファイル単位のデプロイ(5)

- Tomcatの管理画面で再度デプロイしたい場合
 - 「アプリケーション」の一覧から、自分のアプリケーション名の右にある「配備解除」リンクをクリック
 - 「アプリケーション」から自分のアプリケーションが消え、WARファイルやディレクトリも削除される
 - 前スライドの手順で再度デプロイを行う

演習

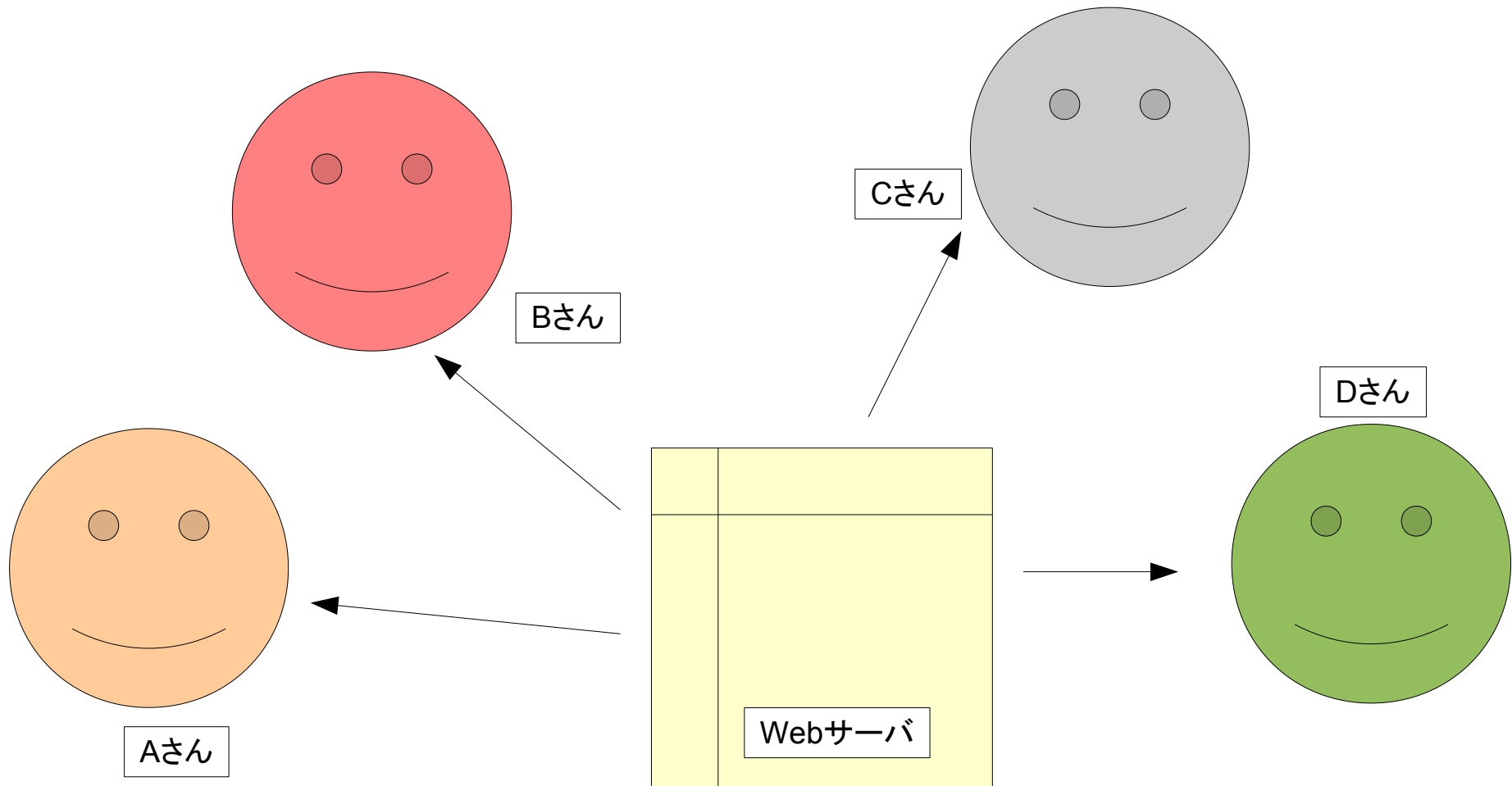
- Tomcatの管理画面を使ったWARファイルのデプロイを行ってみましょう
 - 今まで使っていたプロジェクトでWARファイルを作成します
 - ファイル名は、コンテキストパスと異なる名称にしてください
 - Tomcatを起動します
 - 起動中の場合はそのまま結構です
 - Tomcatの管理画面からWARファイルをデプロイしてください
 - デプロイしたアプリケーションが正しく動作することを確認します
 - 動作が確認できたら「配備解除」を行ってアプリケーションを削除しましょう

セッション管理

株式会社ナレッジエックス
<http://www.knowledge-ex.jp/>

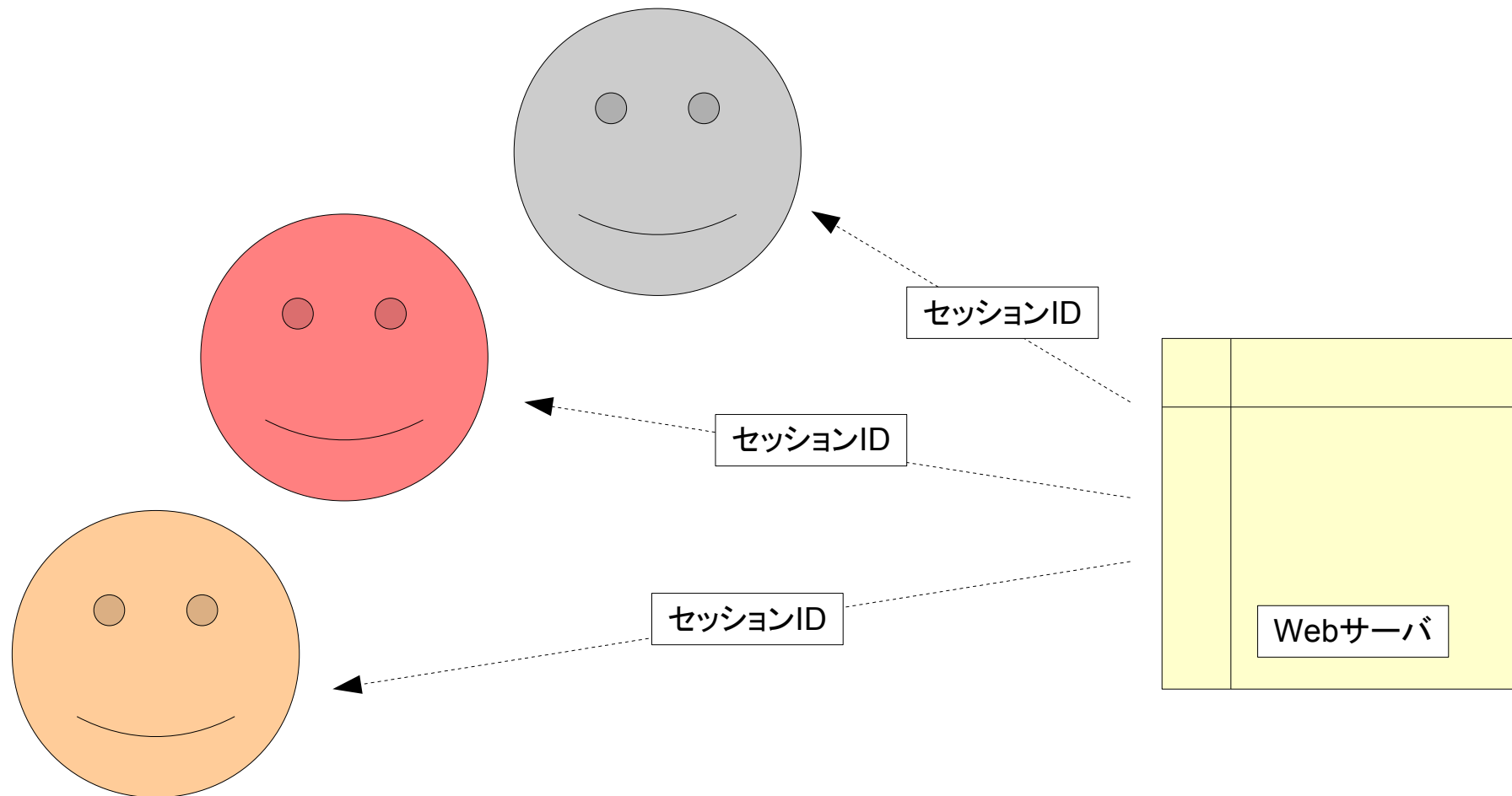
セッション管理とは

- Webアプリケーションを利用する不特定多数のユーザーを識別するためのしくみ



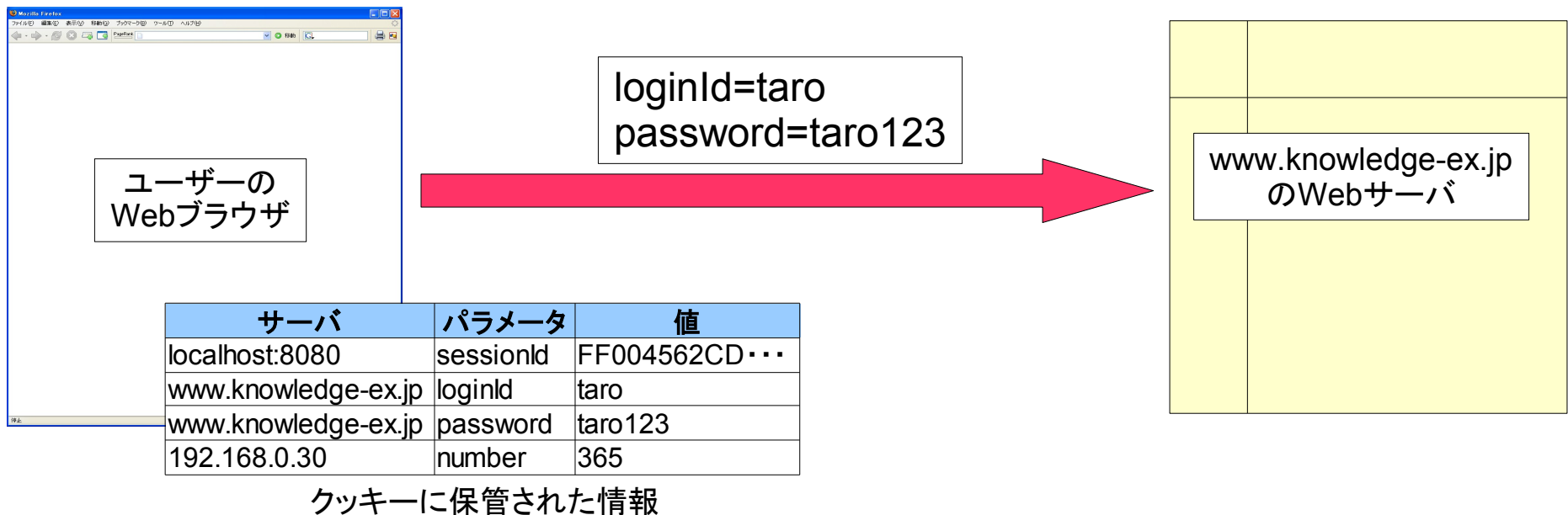
セッション管理のしくみ

- それぞれのユーザーに対して「セッションID」という識別のためのIDを与える



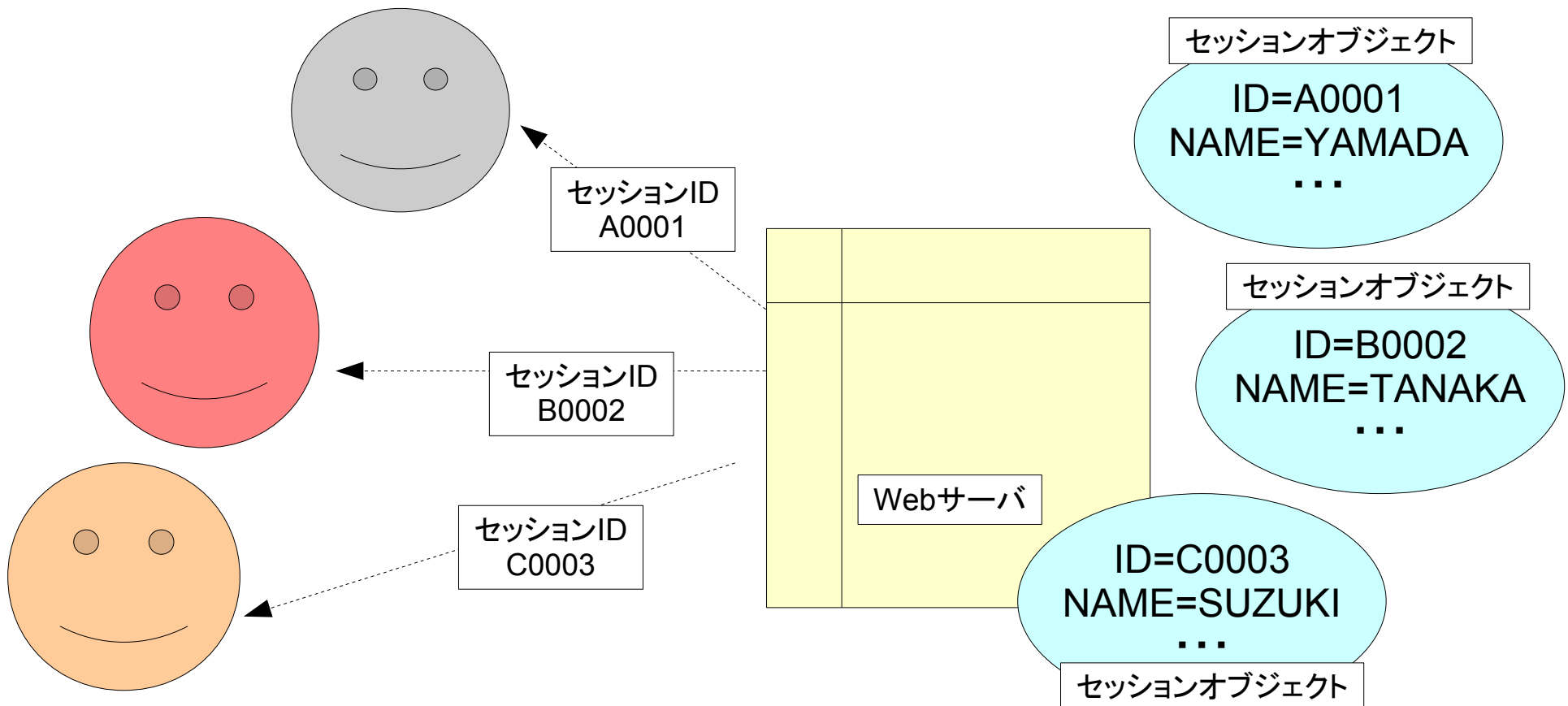
セッションIDの保管

- ブラウザには、一時的な情報を保管する「クッキー(Cookie)」という仕組みがある
 - サーバからクッキーに情報が保管されると、Webブラウザは同じサーバに対してその情報を常に送信する
 - セッションIDをクッキーに入れておけば、次からは自動でセッションIDが送られてくるので、ユーザーが識別できる



セッションオブジェクト

- セッションIDによってユーザーの識別が可能になる
- さらにユーザーごとに固有の情報を保持するために「セッションオブジェクト」というオブジェクトを用意することができる



セッションの開始と終了

- セッションの開始

- あるユーザー(Webブラウザ)にセッションIDを与え、対応するセッションオブジェクトを新たに生成すること

- セッションの終了

- あるユーザー(Webブラウザ)に与えたセッションIDを無効化し、対応するセッションオブジェクトを消滅させること

- セッションの継続

- あるユーザー(Webブラウザ)が有効なセッションIDを保持しているかどうか(セッションオブジェクトが存在するかどうか)確認すること

セッションの開始

- HttpServletRequest#getSession()メソッド
 - セッションオブジェクトを取得するためのメソッド
 - 戻り値は javax.servlet.http.HttpSession型
 - セッションを開始する場合は、引数をtrueにする

凡例

```
HttpServletRequest#getSession(boolean);
```

①引数を「true」にした場合

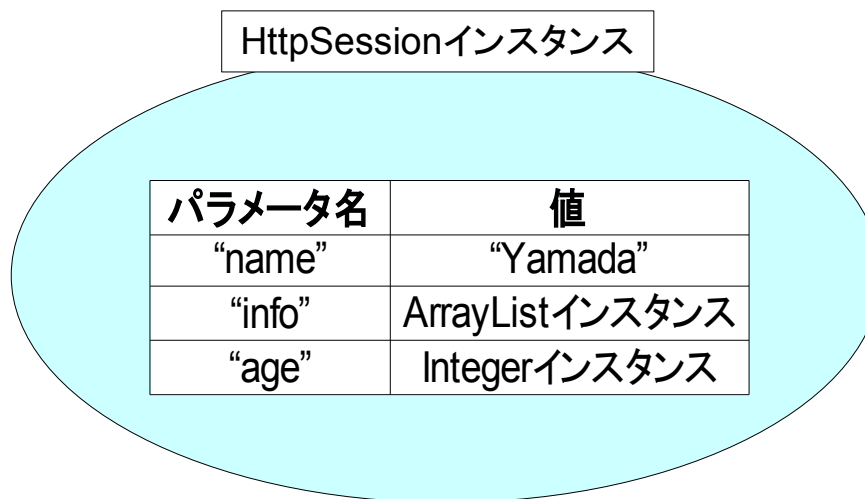
その時点でセッションが開始していない(セッションオブジェクトがない)場合は、新たなセッションIDを発行し、セッションオブジェクトを生成して返す
すでにセッションが開始済みの場合は、既存のセッションオブジェクトを返す

②引数を「false」にした場合

その時点でセッションが開始していない(セッションオブジェクトがない)場合は、nullを返す
すでにセッションが開始済みの場合は、既存のセッションオブジェクトを返す

HttpSessionオブジェクト

- javax.servlet.http.HttpSessionオブジェクト
 - セッションオブジェクトを表現したオブジェクト
 - Object型のデータを格納しておくことができる
 - 格納時にはString型のパラメータ名で識別する
 - HashMapと類似の構造
 - インスタンスはセッションが終了するまで保持される



HttpSessionへのデータ格納・取り出し

- データの格納・・・setAttribute()メソッド
 - 格納できるデータはクラス型のインスタンスのみ
 - 基本データ型は直接格納できないので注意

凡例

```
HttpSession#setAttribute(String, Object);
```

↑
格納する
オブジェクト
につける名称

↑
格納したい
オブジェクト

コード例

```
HttpSession session = request.getSession(true);  
session.setAttribute("loginUser", "yamada");  
session.setAttribute("age", new Integer(30));
```

HttpSessionへのデータ格納・取り出し

- データの取り出し・・・getAttribute()メソッド
 - 戻り値はObject型のため、変数などに格納する場合は、元の型にキャストしてから扱う必要があるので注意

凡例

```
HttpSession#getAttribute(String);
```

↑
取り出したいオブジェクトにつけておいた名称

コード例

```
HttpSession session = request.getSession(true);  
String name = (String)session.getAttribute("loginUser");  
Integer age = (Integer)session.getAttribute("age");
```

セッション継続の確認

- セッション継続の必要性
 - セッション継続が前提の画面を表示する場合に、セッションが継続しているかどうか確認する必要がある
 - 例) ログインしないと見ることのできないページ
 - セッション継続を確認しないと「なりすまし」による不正アクセスを許すことになってしまう
 - セッション継続は、HttpServletRequest.getSession()メソッドの引数をfalseにして戻り値がnullでないことで確認できる

コード例

```
HttpSession session = request.getSession(false);  
if (session == null) {  
    セッションが継続していないのでエラー処理を行う  
    return;  
}  
セッションが正常に継続している場合の処理
```

セッションを終了する

- HttpSession#invalidate()メソッド
 - 現在のユーザーに対するセッションIDを無効化し、セッションオブジェクトを消滅させるメソッド

凡例

```
HttpSession#invalidate()
```

コード例

```
HttpSession session = request.getSession(false);  
if (session != null) {  
    session.invalidate();  
}
```

セッションタイムアウト

- セッションの終了はinvalidate()メソッドを使うと明示的に行われるが、一定時間アクセスのないセッションは、自動で無効化されるようになっている(セッションタイムアウト)
 - デフォルトのセッションタイムアウトは(一般的に)30分
 - web.xmlにセッションタイムアウトを設定すると、タイムアウト時間を変更することが可能(<session-config>タグ)

コード例

```
<web-app>
  <servlet>
    :
  </servlet>
  <servlet-mapping>
    :
  </servlet-mapping>
  <session-config>
    <session-timeout>45</session-timeout>
  </session-config>
</web-app>
```

セッションタイムアウトを45分に
設定(数値の単位は分)

演習(1)

- セッションの開始、継続、終了を行うサーブレットをそれぞれ作成してみましょう

演習(2)

- クラス「web.Session1Servlet」
 - サーブレットパスは「/session1」
 - doGetメソッドをオーバーライドし処理を記述
 - 新規セッションを開始
 - セッションオブジェクトに適切なパラメータを格納
 - 「セッションを開始しました」というメッセージをブラウザに表示



演習(3)

- クラス「web.Session2Servlet」
 - サーブレットパスは「/session2」
 - doGetメソッドをオーバーライドし処理を記述
 - セッションの継続を確認
 - セッションが継続していれば、「セッションが継続しています」というメッセージと、Session1Servletで格納しておいたパラメータの内容をブラウザに表示
 - セッションが継続していない場合は、「セッションが開始されていません」というメッセージをブラウザに表示



演習(4)

- クラス「web.Session3Servlet」
 - サーブレットパスは「/session3」
 - doGetメソッドをオーバーライドし処理を記述
 - セッションを終了させる
 - ブラウザに「セッションを終了しました」というメッセージを表示



演習(5)

- 作成したら、以下の動作を確認
 - 「Session1Servlet」→「Session2Servlet」→「Session3Servlet」の順に実行し動作を確認
 - 「Session3Servlet」の後に「Session2Servlet」を再度実行しセッションが継続していないことを確認

解答例(1)

- web.SessionServlet1

```
package web;

import java.io.IOException;
import java.io.PrintWriter;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import javax.servlet.http.HttpSession;

public class SessionServlet1 extends HttpServlet {
    protected void doGet(HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException {
        HttpSession session = request.getSession(true);
        session.setAttribute("loginUser", "KnowledgeTaro");
        response.setContentType("text/html;charset=Windows-31J");
        PrintWriter out = response.getWriter();
        out.println("<html>");
        out.println("<head><title>Session Servlet 1</title></head>");
        out.println("<body>");
        out.println("<h1>Session Servlet 1</h1>");
        out.println("<p>セッションを開始しました</p>");
        out.println("</body>");
        out.println("</html>");
    }
}
```

解答例(2)

- web.SessionServlet2

```
package web;

import java.io.IOException;
import java.io.PrintWriter;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import javax.servlet.http.HttpSession;

public class SessionServlet2 extends HttpServlet {
    protected void doGet(HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException {
        response.setContentType("text/html;charset=Windows-31J");
        PrintWriter out = response.getWriter();
        out.println("<html>");
        out.println("<head><title>Session Servlet 2</title></head>");
        out.println("<body>");
        out.println("<h1>Session Servlet 2</h1>");
        HttpSession session = request.getSession(false);
        if (session == null) {
            out.println("<p>セッションが開始されていません</p>");
            out.println("</body>");
            out.println("</html>");
            return;
        }
        out.println("<p>セッションが継続しています</p>");
        out.println("<p>" + session.getAttribute("loginUser") + "</p>");
        out.println("</body>");
        out.println("</html>");
    }
}
```

解答例(3)

- web.Session3Servlet

```
package web;

import java.io.IOException;
import java.io.PrintWriter;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import javax.servlet.http.HttpSession;

public class SessionServlet3 extends HttpServlet {
    protected void doGet(HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException {
        HttpSession session = request.getSession(false);
        if (session != null) {
            session.invalidate();
        }
        response.setContentType("text/html;charset=Windows-31J");
        PrintWriter out = response.getWriter();
        out.println("<html>");
        out.println("<head><title>Session Servlet 3</title></head>");
        out.println("<body>");
        out.println("<h1>Session Servlet 3</h1>");
        out.println("<p>セッションを終了しました</p>");
        out.println("</body>");
        out.println("</html>");
    }
}
```

解答例(3)

- web.xml(関連部分のみ抜粋)

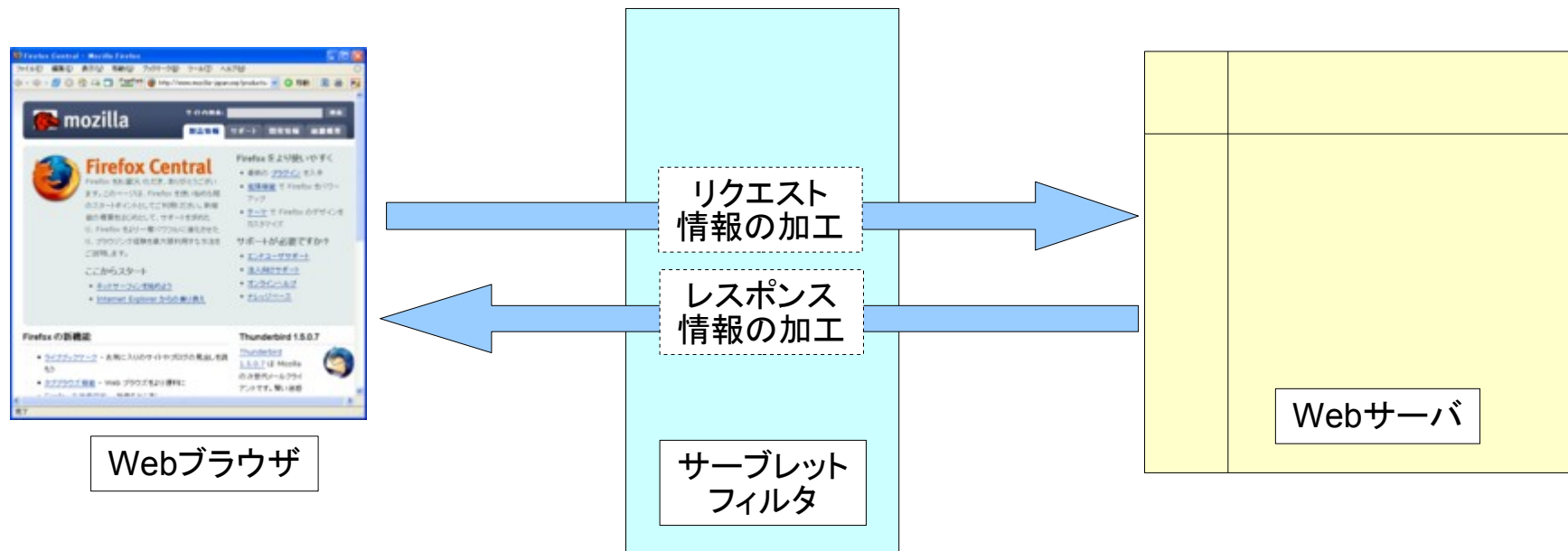
```
<?xml version="1.0" encoding="Windows-31J"?>
<web-app>
  <servlet>
    <servlet-name>session1</servlet-name>
    <servlet-class>web.SessionServlet1</servlet-class>
  </servlet>
  <servlet>
    <servlet-name>session2</servlet-name>
    <servlet-class>web.SessionServlet2</servlet-class>
  </servlet>
  <servlet>
    <servlet-name>session3</servlet-name>
    <servlet-class>web.SessionServlet3</servlet-class>
  </servlet>
  <servlet-mapping>
    <servlet-name>session1</servlet-name>
    <url-pattern>/session1</url-pattern>
  </servlet-mapping>
  <servlet-mapping>
    <servlet-name>session2</servlet-name>
    <url-pattern>/session2</url-pattern>
  </servlet-mapping>
  <servlet-mapping>
    <servlet-name>session3</servlet-name>
    <url-pattern>/session3</url-pattern>
  </servlet-mapping>
</web-app>
```

サーブレットフィルタ

株式会社ナレッジエックス
<http://www.knowledge-ex.jp/>

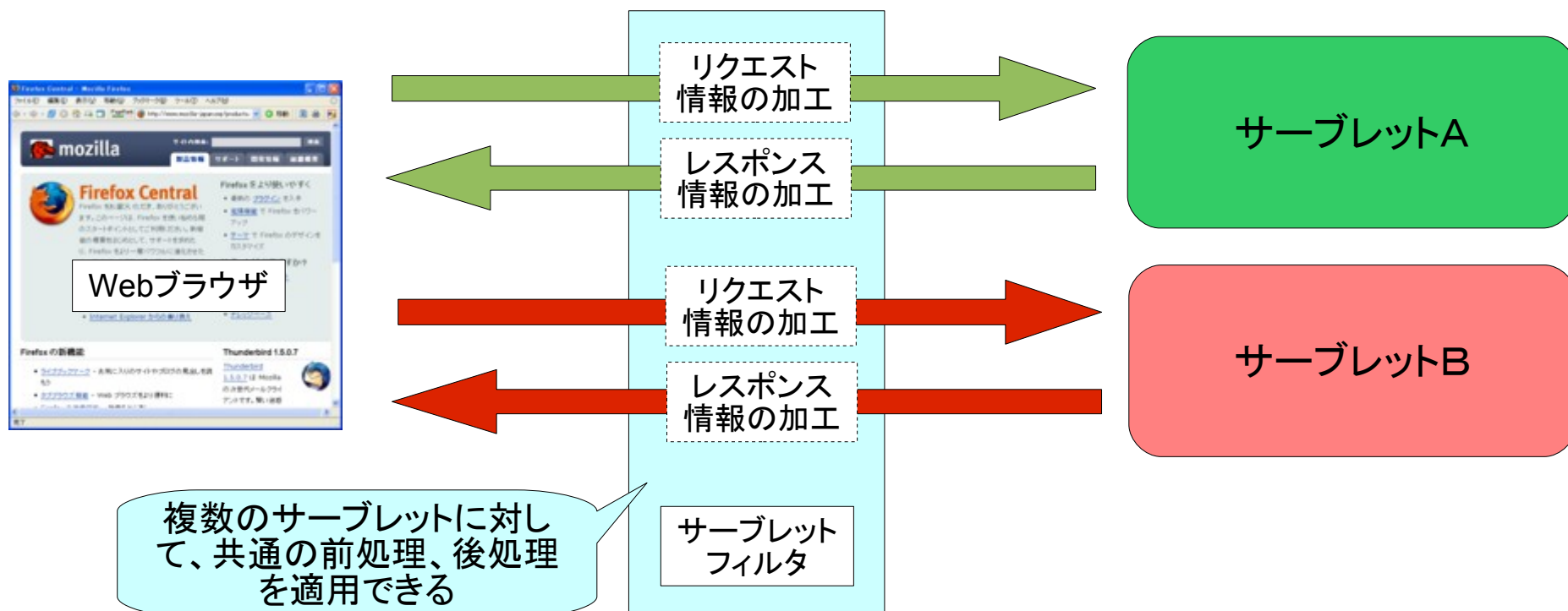
サージレットフィルタとは

- サージレットフィルタとは
 - サージレットに対するリクエスト・レスポンスをフィルタリングする
 - サージレットに到達する前にリクエスト情報を加工する
 - ブラウザに到達する前にレスポンス情報を加工する



サードレットフィルタのメリット

- URLパターンの指定によって、複数種類のサードレットに対して同じフィルタを適用できるため、共通な前処理・後処理がある場合などに便利
 - 例)リクエストパラメータのエンコーディング設定、特定条件でのサードレットへのアクセス制限など



サーブレットフィルタクラスの作成

- javax.servlet.Filterインターフェースを実装するクラスを作成
 - doFilterメソッドをオーバーライドすると、フィルタ処理を実行することが可能
 - doFilterメソッドは、サーブレットが実行される前に呼ばれる
 - doFilterメソッドの引数 (HttpServletRequestと HttpServletResponse) に対して必要な前処理を実行する
 - フィルタ処理が終わったら、引数FilterChainに対して、doFilterメソッドを実行すると、サーブレットが実行される
 - サーブレット実行後に後処理を行いたい場合は、FilterChain#doFilter()の後に記述する

サーブレットフィルタの作成

- サーブレットフィルタクラスのコード例

```
package web;
import java.io.*;
import javax.servlet.*;

public class EncodeFilter implements Filter {

    public void init(FilterConfig config) throws ServletException {}

    public void doFilter(ServletRequest request, ServletResponse response,
        FilterChain chain) throws IOException, ServletException {

        request.setCharacterEncoding("Windows-31J");
        chain.doFilter(request, response);

    }

    public void destroy() {}

}
```

サーブレットフィルタの動作設定

- 「web.xml」にサーブレットフィルタの設定を記述
 - <filter>タグ・・・サーブレットフィルタの定義
 - <filter-name>タグ
 - サーブレットフィルタの名称を定義
 - <filter-class>タグ
 - サーブレットフィルタクラス名を指定(パッケージ名含む)
 - <filter-mapping>タグ
 - <filter-name>タグ
 - 動作させたいサーブレットフィルタの名称を指定
 - <url-pattern>タグ
 - このフィルタを動作させたいURLパターンを指定
 - サーブレットパス設定の<url-pattern>タグと同様
 - 複数のURLパターンに対応させたい場合は、このタグを複数記述

サーブレットフィルタの作成

- コード例

```
<web-app>
  <filter>
    <filter-name>encode</filter-name>
    <filter-class>web.EncodeFilter</filter-class>
  </filter>
  <filter-mapping>
    <filter-name>encode</filter-name>
    <url-pattern>/*</url-pattern>
  </filter-mapping>
  <servlet>
    :
  </servlet>
  <servlet-mapping>
    :
  </servlet-mapping>
</web-app>
```

サーブレットフィルタクラスをパッケージ名込みで指定

「/*」は全てのサーブレットに対してこのフィルタを使用することを示す

演習

- サーブレットフィルタクラスを作成してみましょう
- web.xmlにサーブレットフィルタの設定を追加し、フィルタクラスの動作を確認してみましょう

JSPの基礎

株式会社ナレッジエックス
<http://www.knowledge-ex.jp/>

Agenda

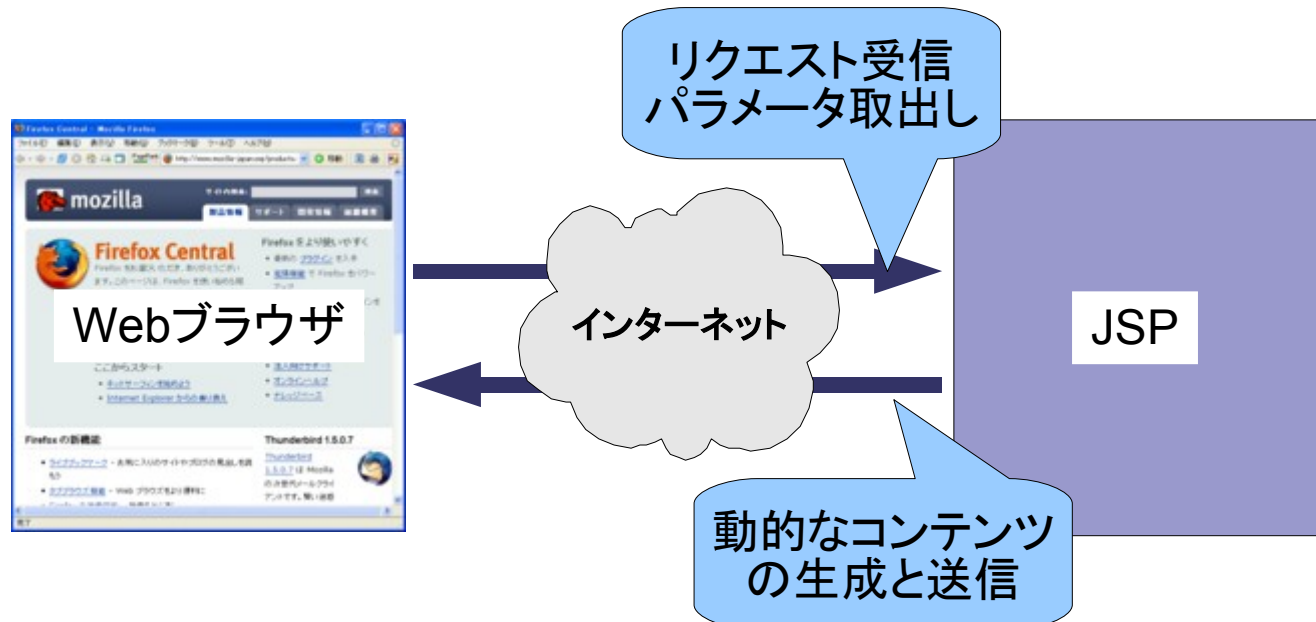
- JSPとは
 - JSPのしくみ
- JSPをつくる
 - 開発手順
 - JSPタグ紹介
 - 標準タグ
 - 標準拡張タグ
 - JavaBeansとの連携
 - 暗黙オブジェクト
 - タグライブラリ

JSPとは

株式会社ナレッジエックス
<http://www.knowledge-ex.jp/>

JSPとは

- JSP(JavaServer Pages)
 - HTMLに動的な処理を行うタグ(JSPタグ)を追加し、表示内容を動的に変更できるWebページを作成するためのAPI
 - サーブレットはJavaコード上で実現しているが、JSPではHTMLコード上で動的な処理を実現している



JSPでの動的処理の表現

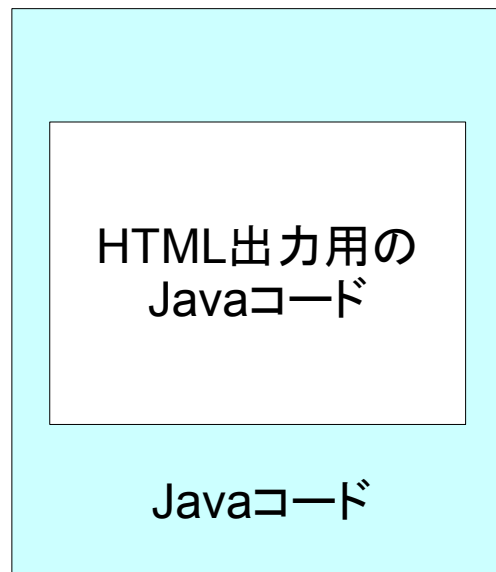
- 「JSPタグ」を使用
 - Javaを用いた動的な処理を実現するためのタグ群

```
<html>
<body>
<% int a = 32,b = 68; %>
<h1><%= a %> + <%= b %> = <%= a+b %></h1>
</body>
</html>
```

JSPタグを用いたコードの例

ServletとJSPの違い

- Servlet
 - Javaコードに対して、HTML出力のためのコードを追加
- JSP
 - HTMLコードに対して、Java処理のためのコードを追加



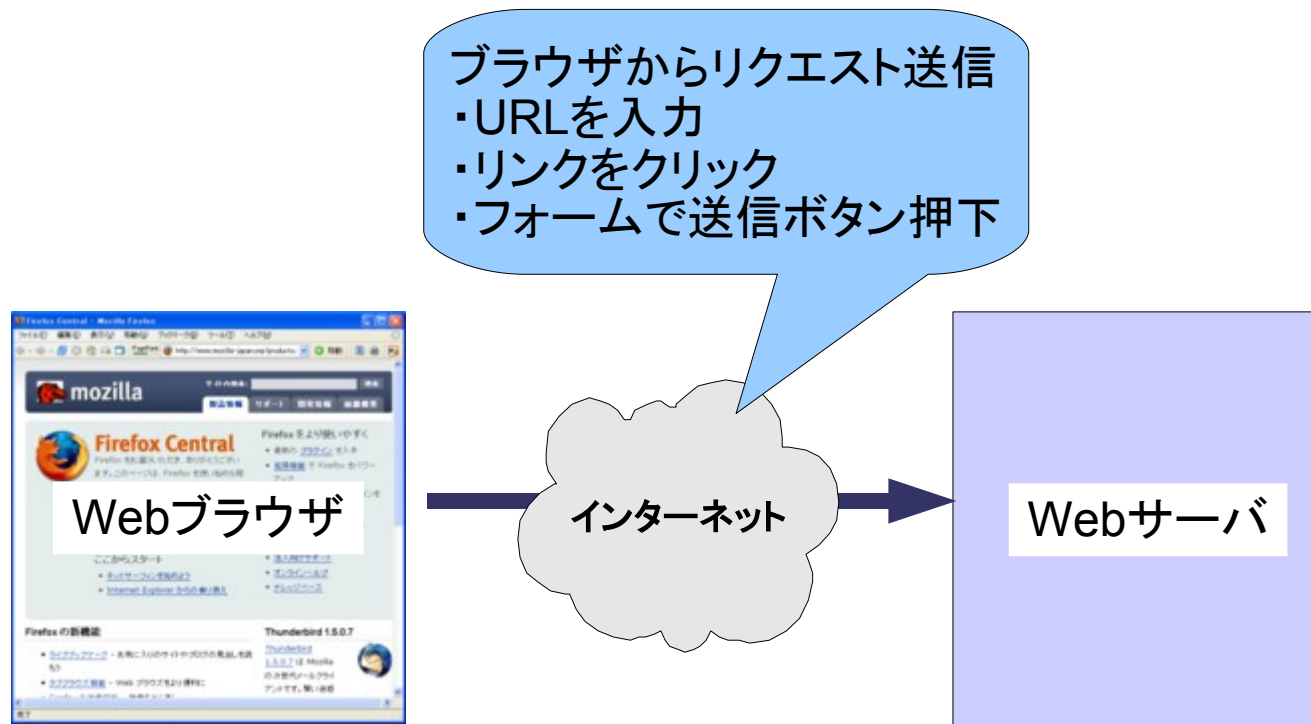
Servlet



JSP

JSPのしくみ(1)



- WebブラウザとJSPの通信手順
 - ①Webブラウザは基本的にWebサーバとしか通信できないため、ブラウザからのリクエストはいったんWebサーバに送られる

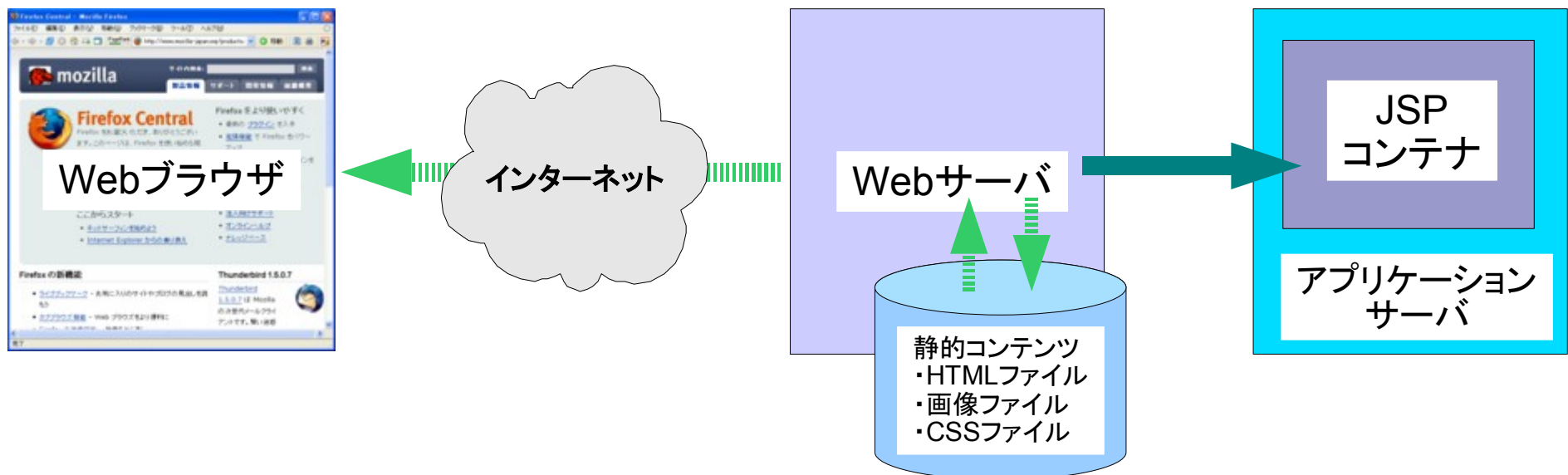


JSPのしくみ(2)

- WebブラウザとJSPの通信手順

- ②Webサーバではリクエスト内容を解析し

- A)静的コンテンツ(HTMLファイルや、画像など)に対するリクエストであれば、自身で応答を送信()
 - B)JSPに対するリクエストであれば、「JSPコンテナ」にそのリクエストを転送()

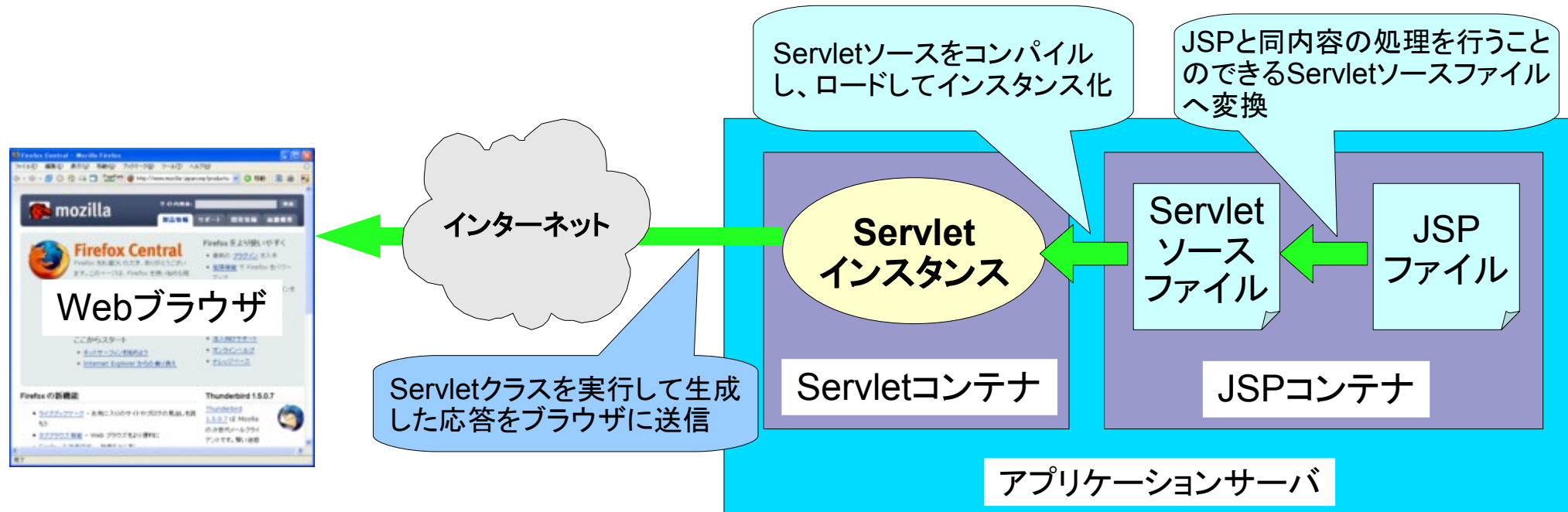


JSPのしくみ(3)

• WebブラウザとJSPの通信手順

– ③JSPコンテナでリクエストに指定されたJSPを実行

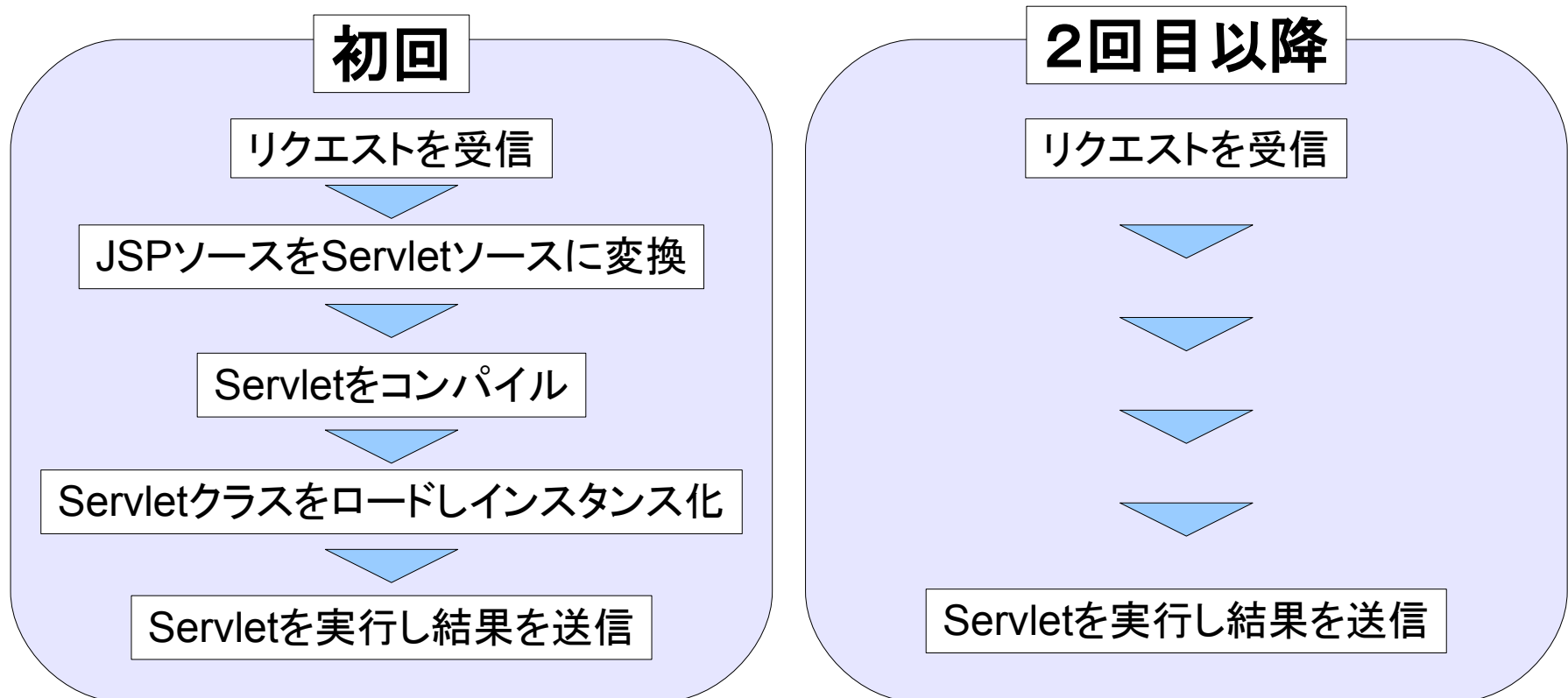
- 1) 指定されたJSPと同じ処理を行うServletソースが生成される
- 2) 生成されたServletソースがコンパイルされる
- 3) コンパイルされたServletクラスがServletコンテナで実行される



JSPのしくみ(4)

- JSPを繰り返し実行した場合

- いったんServletに変換され、Servletコンテナにインスタンスが存在している場合はJSPからServletへの変換は行われず、対応するServletが直接実行される



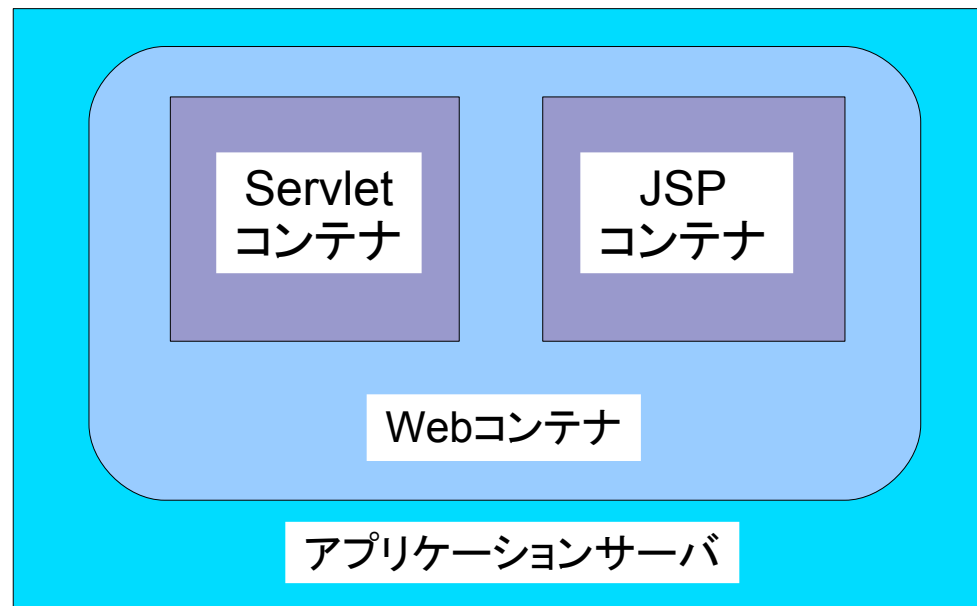
JSPコンテナとは

- JSPを管理するミドルウェア
 - 管理の内容
 - JSPのライフサイクルを管理
 - JSPからServletへの変換・コンパイル
 - Servletコンテナとの連携処理
 - リクエスト・レスポンス情報の提供
 - WebサーバやWebブラウザとの通信
 - Webサーバからリクエスト情報の受信
 - Webブラウザに対してレスポンス情報の送信
 - Java用のWebアプリケーションサーバに含まれている
 - TomcatやWebSphere、WebLogicなどに含まれる

※「コンテナ」=「オブジェクトの容れ物」という意味でよく使われます

ServletコンテナとJSPコンテナ

- ServletコンテナとJSPコンテナは通常同じアプリケーションサーバに含まれており連携して動作
- ServletコンテナとJSPコンテナを総称して「Webコンテナ」と呼ぶ



JSPをつくる

株式会社ナレッジエックス
<http://www.knowledge-ex.jp/>

JSPタグ

- JSPタグの分類
 - 基本(Core)タグ
 - ディレクティブ
 - スタンダードアクション
- タグ以外の要素
 - 暗黙オブジェクト
 - EL式

JSPの作成手順

- JSPファイルの作成
 - 拡張子「.jsp」のテキストファイルを作成
 - ドキュメントルート以下の任意の場所に作成可能
 - フォルダを作成してその下に作成することも可能
 - JSPファイルをコーディング
 - テキストエディタのみでコーディング可能
 - コンパイルは実行時に行うので作成時には不要

Eclipseでフォルダを新規作成するには、プロジェクト名を選択して右クリックし、「新規」→「フォルダ」を選択します。

JSPの実行

- JSPの実行手順
 - URL指定は、通常のHTMLファイルと同様
 - サーブレットの「サーブレットパス」のような定義は不要
- JSPの実行順序
 - 原則として記述された順に解釈・実行される
 - HTMLタグはそのまま表示
 - JSPタグは実行された結果が表示される
 - ただしJSPタグには表示結果を伴わないものもある

JSPタグの分類

- JSPタグの分類

- `<%` と `%>` で囲まれるタグ
 - 単独で動作（開始タグ、終了タグの区別がない）
- `<jsp:xxx` で始まるタグ
 - 開始タグ、終了タグがある
 - 間に別のタグを含めることができる
 - 単独で終了するタグは、「/」をつけて閉じる
 - XMLの記述文法を踏襲している

基本(Core)タグ

- 基本(Core)タグ
 - Comment
 - Expression
 - Scriptlet
 - Declaration

Comment(1)

- Comment
 - JSPにコメントを記述するためのタグ
 - JSP実行時には無視されブラウザにも出力されない

Syntax

```
<%-- コメント内容 --%>
```

Comment(2)

- Sample

- HTMLコメントタグ(<!-- ~ -->)との動作の違いを確認してみましょう(実行後、ブラウザで「ソース」を表示)

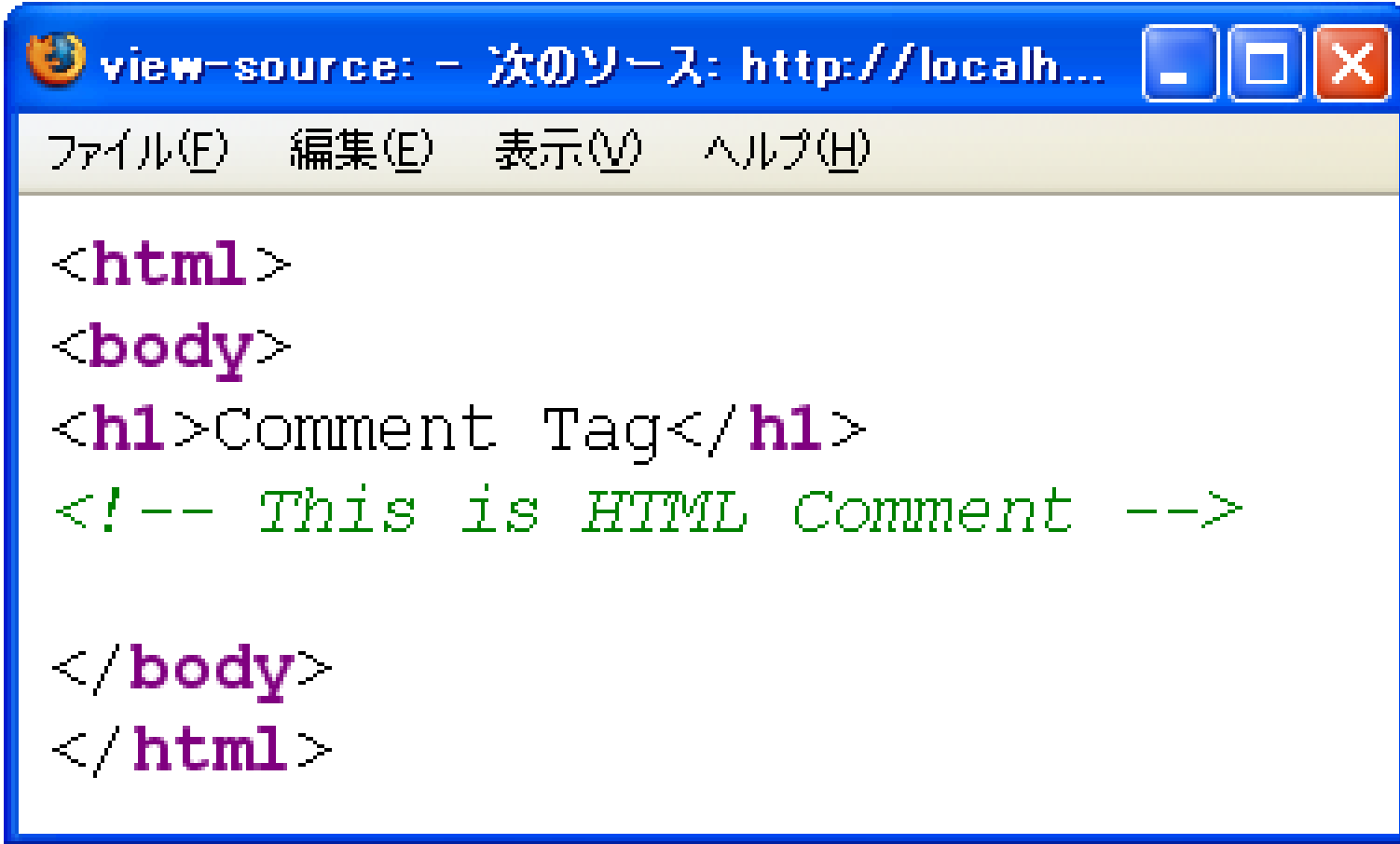
comment.jsp

```
<html>
<body>
<h1>Comment Tag</h1>
<!-- This is HTML Comment -->
<%-- This is JSP Comment --%>
</body>
</html>
```

注)本テキストでは、これ以降、特に指示がなければ、JSPファイルはドキュメントルート下に/pageフォルダを作成し、その下に作成してください。

Comment(3)

- comment.jspの実行結果(ソースを表示)



The screenshot shows a web browser window titled "view-source: - 次のソース: http://localh...". The window has a menu bar with "ファイル(F)", "編集(E)", "表示(V)", and "ヘルプ(H)". The main content area displays the following HTML code:

```
<html>
<body>
<h1>Comment Tag</h1>
<!-- This is HTML Comment -->

</body>
</html>
```

Expression(1)

- Expression
 - 式の評価結果を文字列に変換しブラウザに出力する
 - 文は記述できないので、末尾に「;」はつけない
 - 戻り値が定義されたメソッド呼び出しを指定することも可能

Syntax

```
<%= 式 %>
```

or

```
<jsp:expression>式</jsp:expression>
```

Expression(2)

- Sample(<%= ... %>を使用)
 - Expressionタグの箇所が式の評価結果となって表示されていることを確認しましょう

expression.jsp

```
<html>
<body>
<h1>Expression Tag</h1>
<p><%= 5*5*Math.PI %></p>
</body>
</html>
```

Expression(3)

- expression.jspの実行結果



Expression(4)

- Sample(<jsp:expression>を使用)
 - 前スライド(Expression(2))と同じ結果になることを確認しましょう

expression2.jsp

```
<html>
<body>
<h1>Expression Tag</h1>
<p>
<jsp:expression>5*5*Math.PI</jsp:expression>
</p>
</body>
</html>
```


Scriptlet(1)

- Scriptlet
 - Javaコードを実行するためのタグ
 - タグ内に書かれたJavaコードがそのまま実行される
 - メソッド定義は不要
 - タグが離れていても、{ ... }の対応は保持される
 - ひとつのタグに、複数の文を記述することが可能

Syntax

```
<% 実行したいJavaコード %>
```

Scriptlet(2)

- Sample

- 2つのScriptletタグに含まれる「{」と「}」が対応していることを確認しましょう。

scriptlet.jsp

```
<html>
<body>
<h1>Scriptlet Tag</h1>
<% for(int i=0;i<10;i++) { %>
<%= i %>
<% } %>
</body>
</html>
```

Scriptlet(3)

- scriptlet.jspの実行結果



Declaration(1)

- Declaration
 - インスタンス変数を記述するためのタグ
 - 変数の値はアプリケーションが終了、再起動するか、JSPが再コンパイルされるまで保持される
 - 複数の定義を行を分けて記述することも可能

Syntax

```
<%! インスタンス変数宣言 %>
```

or

```
<jsp:declaration>  
    インスタンス変数宣言  
</jsp:declaration>
```

Declaration(2)

- Sample(<%! ... %>を使用)
 - 実行後、リロードボタンを繰り返しクリックし、どのような結果となるか確認してみましょう

declaration.jsp

```
<html>
<body>
<h1>Declaration Tag</h1>
<%! int a = 0; %>
<%= a++ %>
</body>
</html>
```

Declaration(3)

- declaration.jspの実行結果



Declaration(4)

- Sample(<jsp:declaration>を使用)
 - Declaration(2)のdeclaration.jspと同じ結果になることを確認しましょう

declaration2.jsp

```
<html>
<body>
<h1>Declaration Tag</h1>
<jsp:declaration>
int a = 0;
</jsp:declaration>
<%= a++ %>
</body>
</html>
```

ディレクティブ

- ディレクティブ
 - Page Directive
 - Include Directive
 - Taglib Directive (タグライブラリの項で紹介)

Page Directive(1)

- Page Directive
 - ページに関する設定を行うためのタグ
 - 各種の属性を指定することで設定を行う

Syntax

```
<%@ page 属性指定 %>
```

or

```
<jsp:directive.page 属性指定 />
```

Page Directive(2)

- 主な属性

- contentType属性

- 出力するページのMIMEタイプとエンコーディングを指定する
 - HttpServletResponse#setContentType()と同様の役割
 - 下記のpageEncoding属性を省略した場合は、JSPファイルそのもののエンコーディングと一致している必要あり
 - 例) `contentType="text/html;charset=Windows-31J"`

- pageEncoding属性

- JSPファイルそのもののエンコーディングを指定する
 - 主に、ソースファイルのエンコーディングと、出力のエンコーディングが異なる場合に用いられる

Page Directive(3)

- 主な属性

- import属性

- Javaクラスのインポートを指定する
 - クラス定義におけるimport文と同様の役割
 - Page Directiveのimport属性では複数の指定をカンマで区切って行うことが可能
 - 例) `import="java.util.*,java.text.*"`

Page Directive(4)

- 主な属性

- session属性

- ページ内でセッションオブジェクトを使用するかを指定する
 - session="true"の場合
 - セッションオブジェクトを使用する
 - その時点でセッションが開始されていなければ、セッションを開始し、セッションオブジェクトを新規に生成
 - session="false"の場合
 - セッションオブジェクトを使用しない
 - ページ内に、セッションオブジェクトにアクセスするコードがあると、実行時(サーブレットへの変換時)にエラーとなり実行できない
 - デフォルト(指定を省略した場合)はsession="true"とみなされる

Page Directive(5)

- 主な属性

- errorPage属性

- ページ内で例外が発生した場合に代わりに表示するエラー用のページを指定する
 - 例) `errorPage="/web/jsp/error.jsp"`

- isErrorPage属性

- このページが、errorPage属性で指定されたエラー用のページかどうかを指定する
 - `isErrorPage="true"`とすると、exceptionという暗黙オブジェクト(後述)に、前のページで発生した例外インスタンスが格納され、例外の内容を解析することが可能となる
 - isErrorPage属性を省略しても、エラー用のページとして利用することは可能

Page Directive(6)

- 主な属性

- isELIgnored属性

- ページ内にあるEL式(後述)を解釈するか、無視するかの指定
 - `isELIgnored="true"`とすると、EL式があっても無視される
 - `isELIgnored="false"`とすると、EL式を解釈して実行する

Page Directive(7)

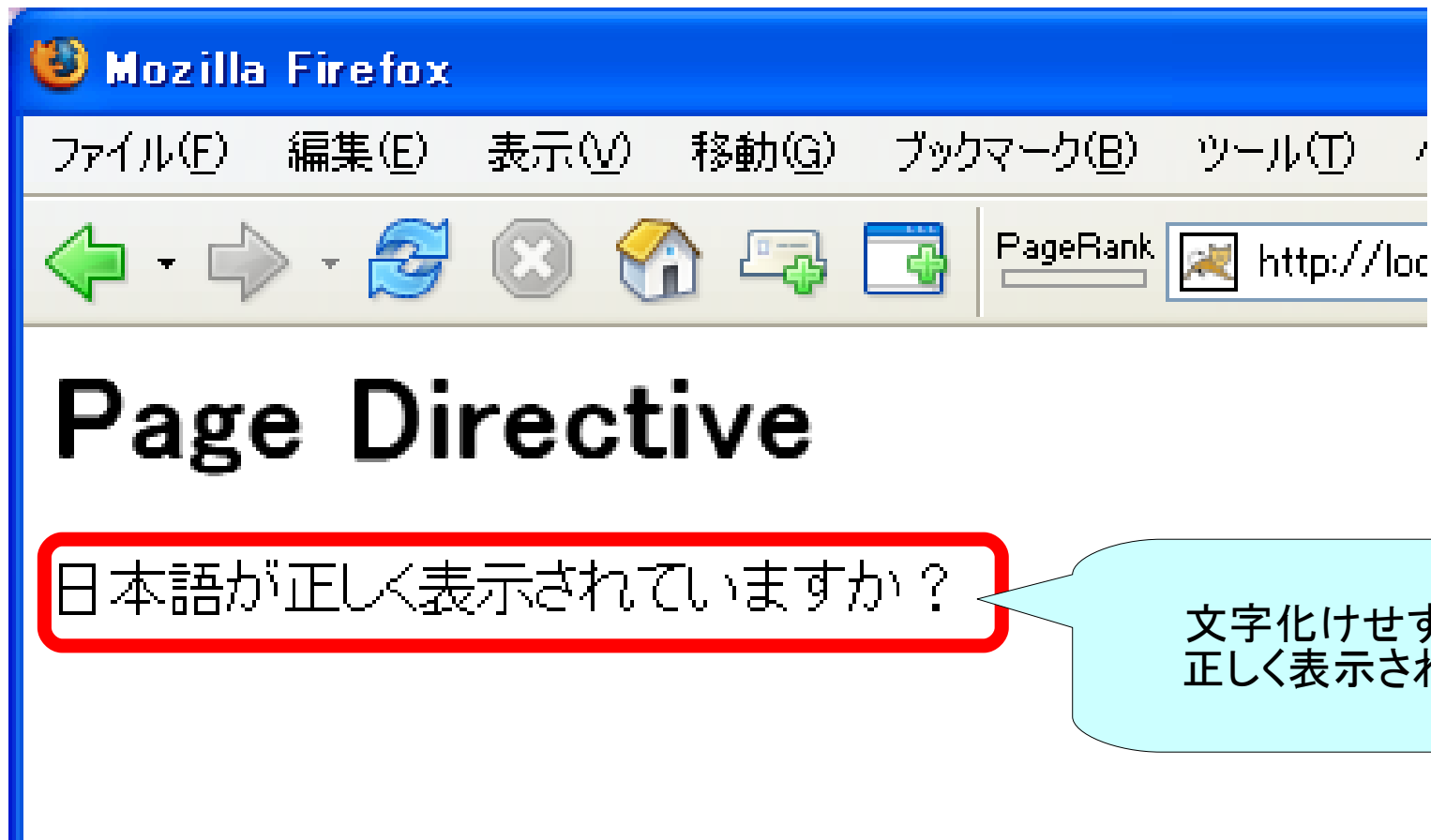
- Sample(contentType属性を使用)
 - 日本語を含むJSPがcontentType属性を使用することで正しく表示できることを確認しましょう

page.jsp

```
<%@page contentType="text/html; charset=Windows-31J" %>
<html>
<body>
<h1>Page Directive</h1>
<p>日本語が正しく表示されていますか？</p>
</body>
</html>
```

Page Directive(8)

- page.jspの実行結果



Page Directive(9)

- Sample(contentType/pageEncoding属性を併用)
 - contentType属性とpageEncoding属性を併用することで、ファイルの文字コードと異なるエンコーディングで出力ができることを確認しましょう

page2.jsp

```
<%@page contentType="text/html; charset=EUC_JP"
      pageEncoding="Windows-31J" %>
<html>
<body>
<h1>Page Directive</h1>
<p>日本語が正しく表示されていますか？</p>
</body>
</html>
```

Page Directive(10)

- Sample(<jsp:directive.page>を使用)
 - 「Page Directive(7)」のサンプルソースと同じ結果になることを確認しましょう

page3.jsp

```
<jsp:directive.page
    contentType="text/html; charset=Windows-31J" />
<html>
<body>
<h1>Page Directive</h1>
<p>日本語が正しく表示されていますか？</p>
</body>
</html>
```

Include Directive(1)

- Include Directive

- 他のページをインクルードするためのタグ
- Include Directiveの書かれた場所に、指定されたページの内容がそのままインクルードされた後に実行される
- 同一アプリケーション以外のファイルはインクルードできない

Syntax

```
<%@ include file="インクルードしたいファイルのURL" %>
```

or

```
<jsp:directive.include file="インクルードしたいファイルのURL" />
```

※ URLは相対パス指定

Include Directive(2)

- Sample

- 下記2つのファイルを作成し、include.jspを実行して、結果を確認しましょう

include.jsp

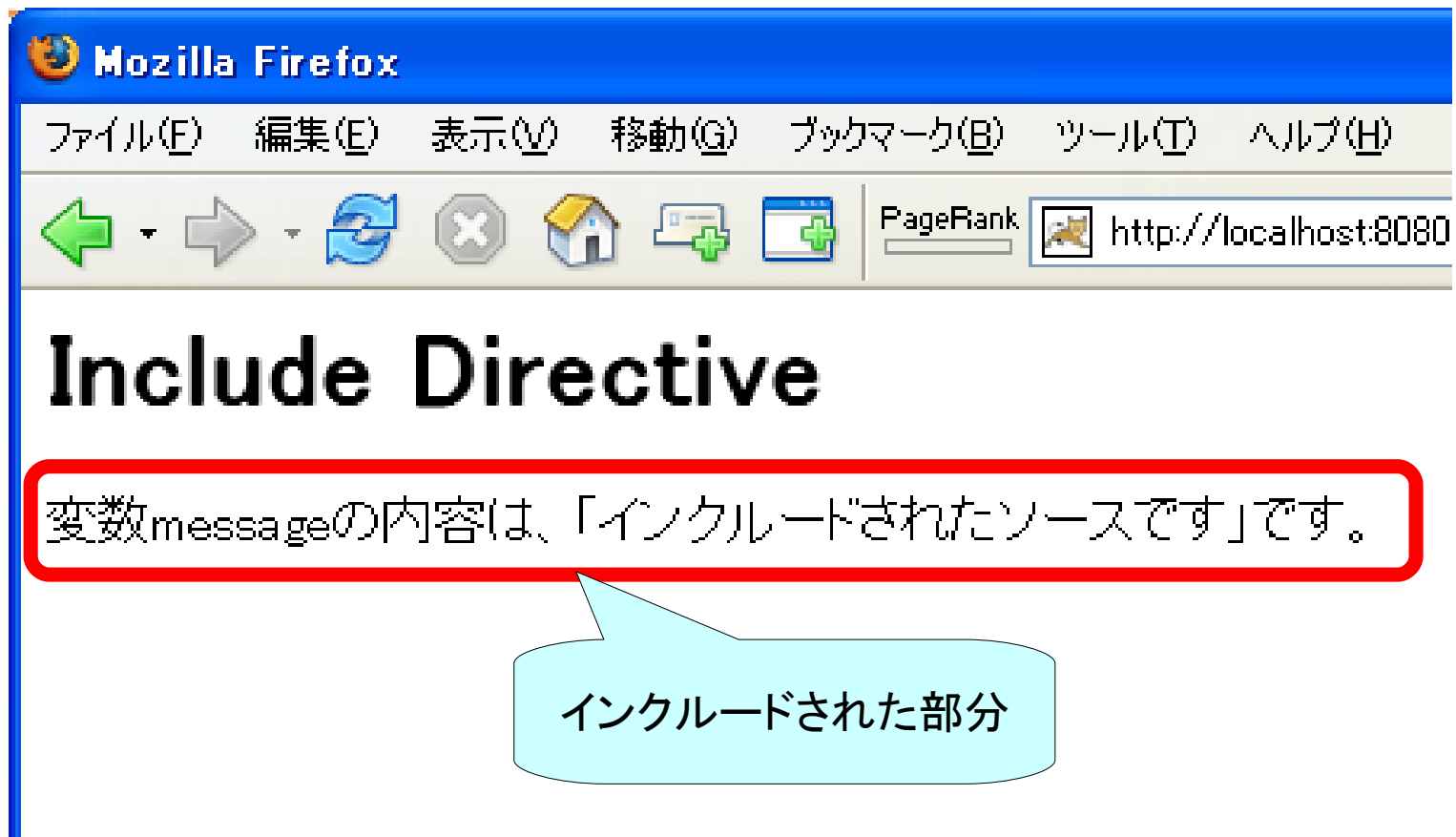
```
<%@page contentType="text/html; charset=Windows-31J" %>
<html>
<body>
<h1>Include Directive</h1>
<% String message="インクルードされたソースです"; %>
<%@ include file="fragment.jsp" %>
</body>
</html>
```

fragment.jsp

```
<%@page contentType="text/html; charset=Windows-31J" %>
変数messageの内容は、「<%= message %>」です。
```

Include Directive(3)

- include.jspの実行結果



スタンダードアクション

•スタンダードアクション

- <jsp:forward>
- <jsp:param>
- <jsp:include>
- <jsp:useBean>
- <jsp:getProperty>
- <jsp:setProperty>

「ServletとJSPの連携」で解説

<jsp:forward>(1)

- <jsp:forward>
 - 現在のページから、他のページへ転送するタグ
 - ブラウザには転送元のページ内容は一切表示されない
 - リクエスト、レスポンス情報は転送元から転送先へそのまま引き継がれる
 - 同一アプリケーション以外のリソースへは転送できない

Syntax

```
<jsp:forward page="転送したいページのURL" />
```

※ URLは相対パス指定か、ドキュメントルートからの絶対パス指定

<jsp:forward>(2)

- リクエストパラメータを追加して転送
 - <jsp:param>タグを使うと、転送先にリクエストパラメータを追加して転送することができる
 - <jsp:forward>タグを開始タグと終了タグに分け、両者の間に<jsp:param>タグを含めて記述

Syntax

```
<jsp:param name="パラメータ名" value="パラメータ値" />
```


<jsp:forward>(3)

- Sample(1)

- 下記2つのファイルを作成し、forward.jspを実行して、結果を確認しましょう

forward.jsp

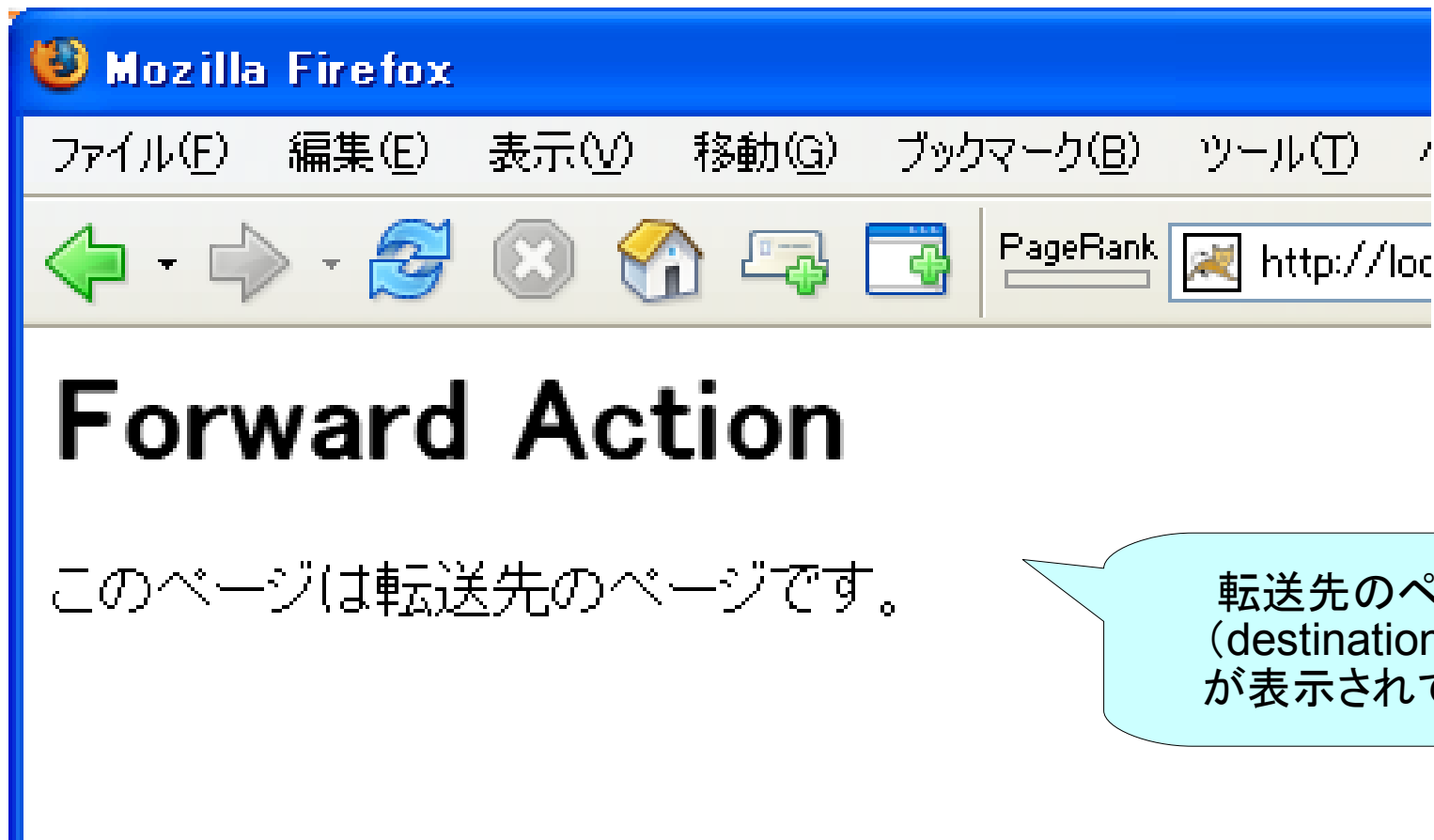
```
<%@page contentType="text/html; charset=Windows-31J" %>
<html>
<body>
<h1>Forward Action</h1>
<jsp:forward page="destination.jsp"/>
</body>
</html>
```

destination.jsp

```
<%@page contentType="text/html; charset=Windows-31J" %>
<html>
<body>
<h1>Destination Page</h1>
<p>このページは転送先のページです。</p>
</body>
</html>
```

<jsp:forward>(4)

- forward.jspの実行結果



<jsp:forward>(5)

- Sample(2) (<jsp:param>タグを使用)
 - 下記2つのファイルを作成し、URLの末尾に「?param1=abcde」をつけてforward2.jspを実行して結果を確認しましょう

forward2.jsp

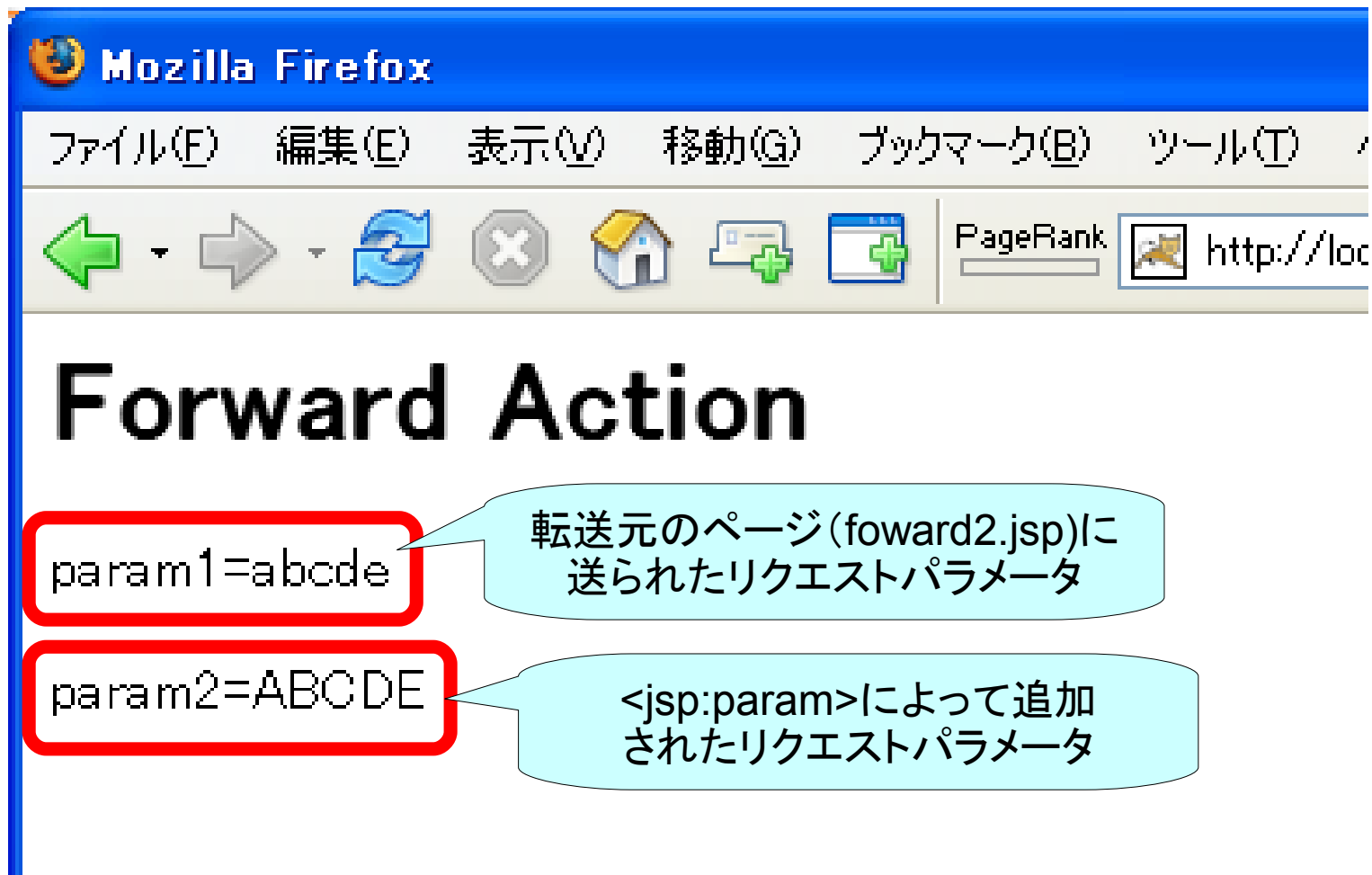
```
<%@page contentType="text/html; charset=Windows-31J" %>
<html>
<body>
<jsp:forward page="destination2.jsp">
    <jsp:param name="param2" value="ABCDE"/>
</jsp:forward>
</body>
</html>
```

destination2.jsp

```
<%@page contentType="text/html; charset=Windows-31J" %>
<html>
<body>
<h1>Forward Action</h1>
<p>param1=<%= request.getParameter("param1") %></p>
<p>param2=<%= request.getParameter("param2") %></p>
</body>
</html>
```

<jsp:forward>(6)

- foward2.jspの実行結果



<jsp:include>(1)

- <jsp:include>
 - 現在のページに他のページの実行結果をインクルードするタグ
 - ソースをインクルードするのではなく、実行結果をインクルードする
 - 同一アプリケーション以外のリソースはインクルードできない
 - <jsp:forward>と同じく<jsp:param>タグも使用可能

Syntax

```
<jsp:include page="インクルードしたいページのURL" />
```

※ URLは相対パス指定か、ドキュメントルートからの絶対パス指定

<jsp:include>(2)

- Sample

- 下記2つのファイルを作成し、includeAction.jspを実行して結果を確認しましょう

includeAction.jsp

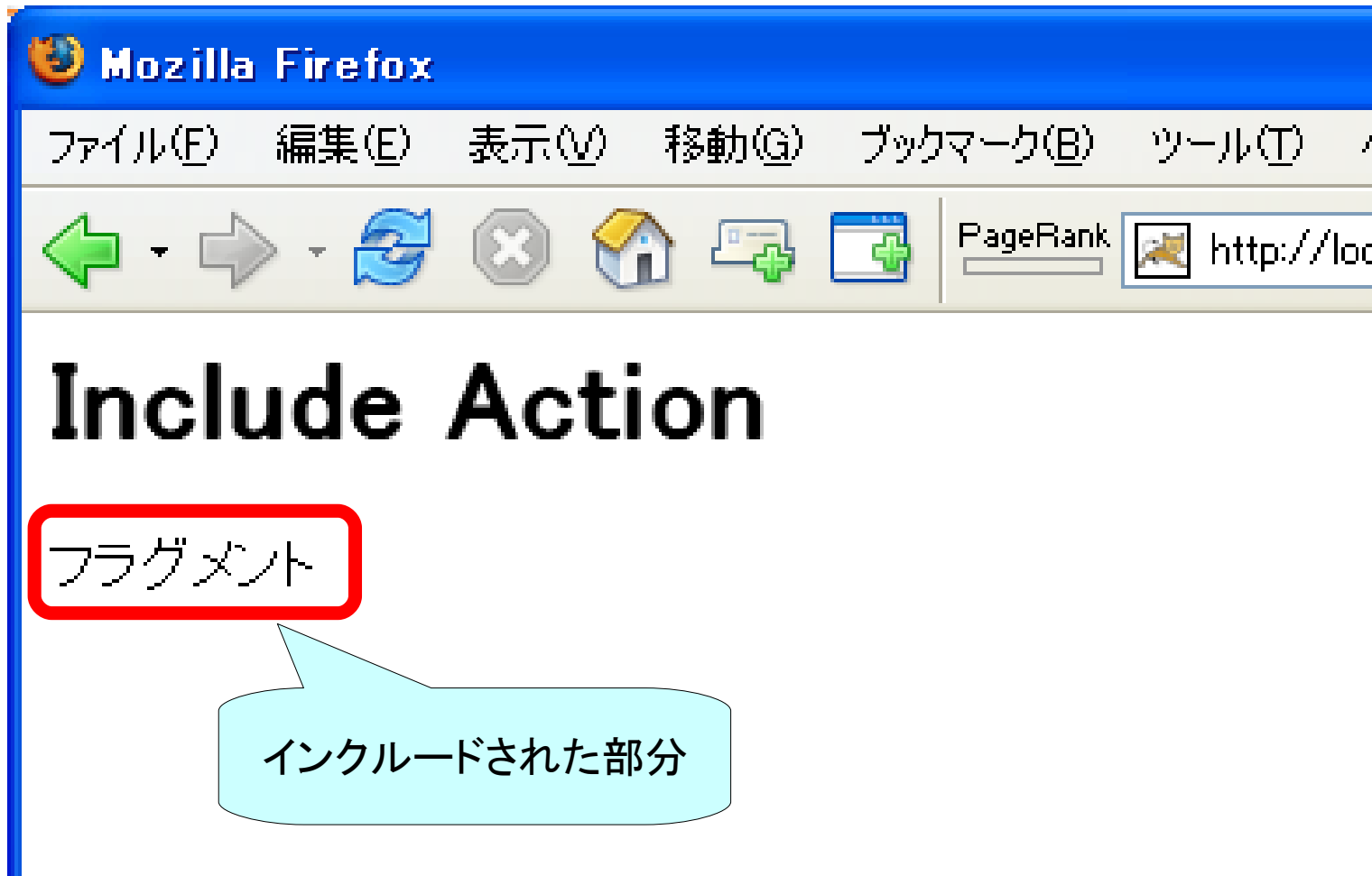
```
<%@page contentType="text/html; charset=Windows-31J" %>
<html>
<body>
<h1>Include Action</h1>
<jsp:include page="fragment2.jsp"/>
</body>
</html>
```

fragment2.jsp

```
<%@page contentType="text/html; charset=Windows-31J" %>
<html>
<body>
<p>フラグメント</p>
</body>
</html>
```

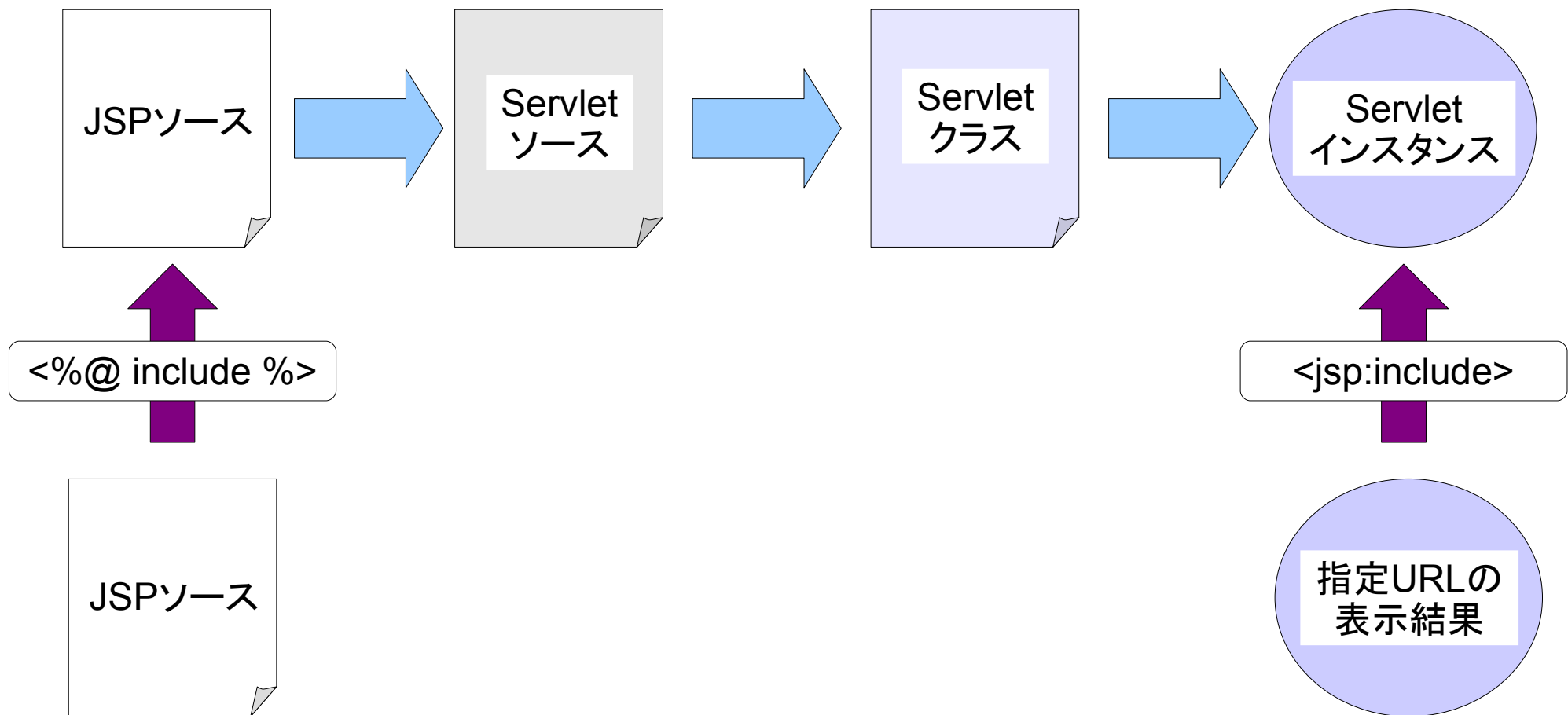
<jsp:include>(3)

- includeAction.jspの実行結果



<jsp:include>と<%@include %>の違い

- インクルードされるタイミングが異なっている



暗黙オブジェクト(1)

- 暗黙オブジェクトとは
 - 定義をしなくてもあらかじめ使える状態になっているオブジェクトのこと
- 主な暗黙オブジェクト
 - request
 - response
 - session
 - out
 - config
 - exception

暗黙オブジェクト(2)

- request
 - javax.servlet.HttpServletRequest型のインスタンス
 - ServletのdoGet/doPostメソッドの引数と同様に使える
- response
 - javax.servlet.ServletResponse型のインスタンス
 - ServletのdoGet/doPostメソッドの引数と同様に使える
- session
 - javax.servlet.http.HttpSession型のインスタンス
 - Servletのセッションオブジェクトと同じものを共用できる

暗黙オブジェクト(2)

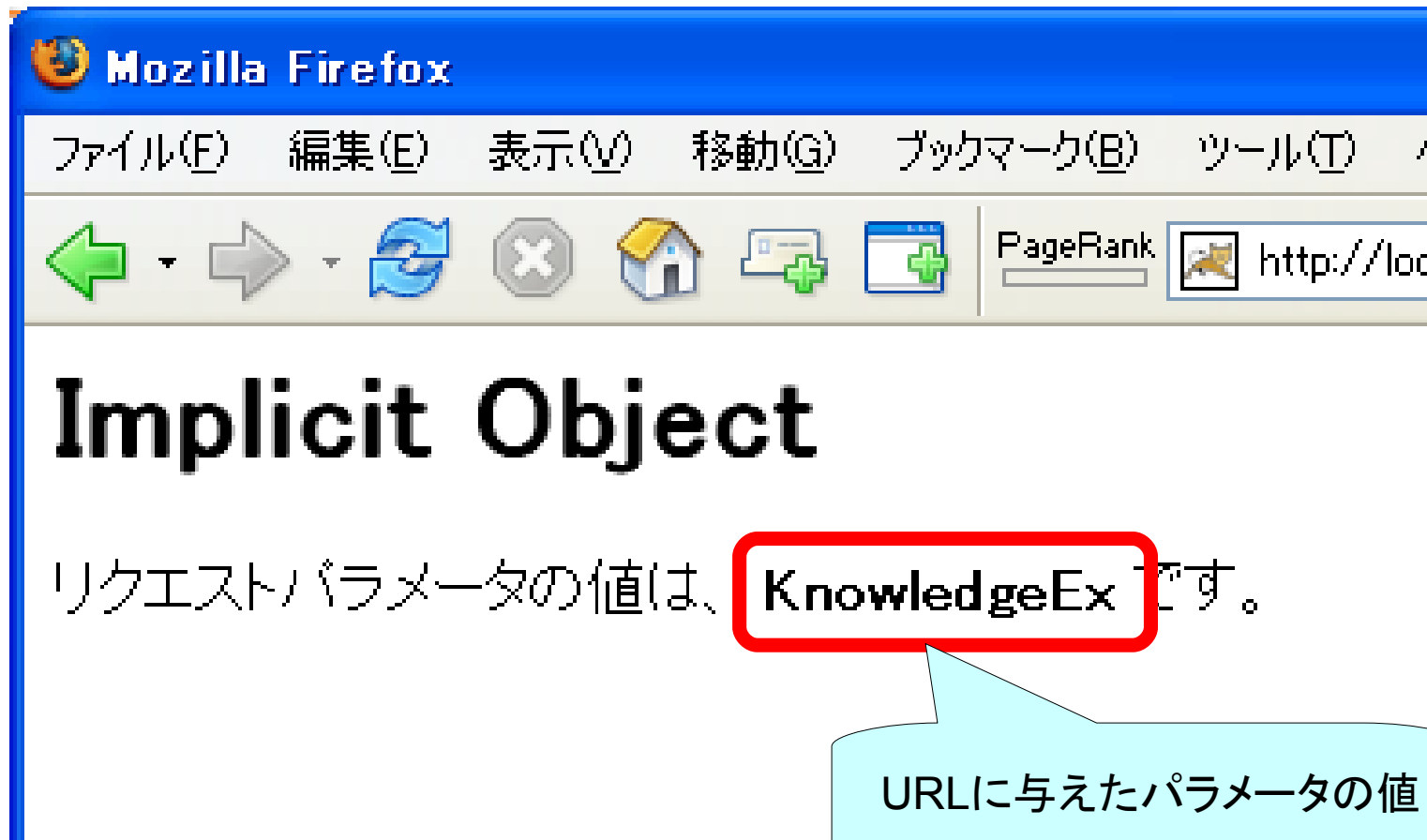
- out
 - javax.servlet.jsp.JspWriter型のインスタンス
 - Servletで出力に使うPrintWriterと同様に使える
- config
 - javax.servlet.ServletConfig型のインスタンス
- exception
 - Page DirectiveでerrorPageに指定された場合にのみ使うことができるオブジェクト
 - 前のページで発生した例外のインスタンスが格納されている

演習

- 暗黙オブジェクトrequestを使って、リクエストパラメータを取出し、暗黙オブジェクトoutでその内容をprintlnメソッドで表示するJSPを作ってみましょう
 - JSPの実行時に、URLの末尾に「?パラメータ名=パラメータ値」の形式でパラメータを与えます
 - パラメータ名は「param」とします

実行例

- 実行結果



解答例

implicit.jsp

```
<%@page contentType="text/html; charset=Windows-31J" %>
<html>
<body>
<h1>Implicit Object</h1>
リクエストパラメータの値は、
<% out.println("<b>" + request.getParameter("param") + "</b>"); %>
です。
</body>
</html>
```

タグライブラリ

株式会社ナレッジエックス
<http://www.knowledge-ex.jp/>

タグライブラリ

- タグライブラリとは
 - 標準のJSPには含まれない拡張タグ仕様
 - 開発者が独自にタグを開発することも可能

標準タグライブラリ

- 標準タグライブラリとは
 - Java標準として公式に仕様が定められたタグライブラリ
 - JSTL(JavaServer Pages Standard Tag Library)と呼ばれる

JSTLの入手

- J2EE(Java EE) SDKに含まれている
 - <http://java.sun.com/javaee/>
- Jakarta Projectで開発され配布されている
 - <http://jakarta.apache.org/taglibs/>
 - 「JCP Standardized Tag Libraries」をダウンロード



JSTL

- JSTLに含まれるタグ
 - Core
 - 基本機能(制御、代入など)を提供するタグライブラリ
 - XML
 - XML文書の操作機能を提供するタグライブラリ
 - Internationalization(I18N)
 - ロケールやフォーマット、メッセージなどの機能を提供するタグライブラリ
 - SQL
 - データベース操作機能を提供するタグライブラリ
 - Functions
 - EL式(後述)の内部で使える関数群

JSTLの利用手順(1)

- (1)JSTLを実装したJARファイルを配置
 - /WEB-INF/libに配置する
 - Jakarta版を入手した場合は「jstl.jar」「standard.jar」をアーカイブから取り出して/WEB-INF/libに配置

JSTLの利用手順(2)

- (2)各JSPで、Taglib Directiveを使ってJSTLの利用を定義

Syntax

```
<%@ taglib uri="TLDファイルのURI" prefix="プレフィックス" %>
```

uri属性…TLDファイル(タグライブラリの定義を記述したXMLファイル)の位置を指定
prefix属性…タグ名の前に「:」で区切って指定する名称を指定

例) prefix="mytaglib" と定義し、「loop」という名前のタグを利用したい場合



```
<mytaglib:loop . . . />
```

JSTLの利用手順(2)

- JSTLを使う場合のuri属性・prefix属性の指定
 - 規定値が定められているのでそれを利用する

ライブラリ	URI	prefix
Core	<code>http://java.sun.com/jsp/jstl/core</code>	c
XML	<code>http://java.sun.com/jsp/jstl/xml</code>	x
Internationalization	<code>http://java.sun.com/jsp/jstl/fmt</code>	fmt
SQL	<code>http://java.sun.com/jsp/jstl/sql</code>	sql
Functions	<code>http://java.sun.com/jsp/jstl/functions</code>	fn

Coreタグ

- <c:set>/<c:remove>
- <c:out>
- <c:if>
- <c:choose>/<c:when>/<c:otherwise>
- <c:forEach>
- <c:import>

<c:set>タグ(1)

- データをリクエストオブジェクト・セッションオブジェクト・ページコンテキストなどに格納する
 - 通常の変数への格納ではないので、変数と同じように使うことはできないことに注意

Syntax

```
<c:set var="オブジェクトの名称" value="格納したい値"  
       scope="格納したいスコープ" />
```

or

```
<c:set var="オブジェクトの名称" scope="格納したいスコープ">  
  格納したい値  
</c:set>
```

スコープ…オブジェクトを格納する場所。

リクエストオブジェクト=request、セッションオブジェクト=session
などを指定できる。省略するとページコンテキスト(それぞれのページ用に
用意された格納場所=pageと表記)とみなされる

<c:set>タグ(2)

- Sample

- "name"という名前で、"KnowledgeTaro"という文字列をセッションオブジェクトに格納するコード例

c-set.jsp

```
<%@taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>
<html>
<body>
<h1>Core Tag (set)</h1>
<c:set var="name" value="KnowledgeTaro" scope="session"/>
</body>
</html>
```

<c:out>タグ(1)

- リクエストオブジェクト・セッションオブジェクト・ページコンテキストなどのデータを文字列として表示する
 - 通常の変数が対象ではないので、通常の変数をこのタグで表示することはできないので注意

Syntax

```
<c:out value="表示したい値"  
        default="デフォルト値"  
        escapeXml="true|false"/>
```

デフォルト値…表示したい値がnullだった場合に、代わりに表示する値

escapeXml …「<」「>」「&」「'」「"」を記号と解釈せず、文字としてそのまま表示するかどうか

true=文字として表示する、false=記号と解釈して表示する

デフォルト(指定を省略した場合)は、escapeXml="true"とみなされる

<c:out>タグ(2)

- Sample

- セッションオブジェクトに"name"という名前で格納しておいた文字列を表示するコード例

c-out.jsp

```
<%@taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>
<%@ page isELIgnored ="false" %>
<html>
<body>
<h1>Core Tag (out)</h1>
<c:set var="name" value="<h3>KnowledgeTaro</h3>" scope="session"/>
<c:out value="${name}" />
<c:out value="${name}" escapeXml="false"/>
<c:out value="${age}" default="99"/>
</body>
</html>
```

<c:out>タグ(3)

- c-out.jspの実行結果



EL式(1)

- EL式(式言語)
 - 演算結果をページに埋め込むための表記
 - ページのどの場所でも利用することができる
 - 利用するには、Page DirectiveでisELIgnored="false"を指定する必要がある

Syntax

```
${ 式表現 }
```

Page Directiveの指定

```
<%@ page isELIgnored = "false" %>
```

EL式(2)

- EL式で利用できる式表現
 - 通常の演算子を用いた式
 - 暗黙オブジェクトを用いた式
 - 各スコープ(リクエストオブジェクト、セッションオブジェクト、ページコンテキストなど)に格納された値
 - スコープを明示してオブジェクトを指定する場合は、「requestScope.」「sessionScope.」を前につける(省略可)
 - 「.」で区切ってオブジェクトのプロパティ値(後述)を指定
 - [...] で配列やMapオブジェクトの要素を指定

<c:remove>タグ(1)

- リクエストオブジェクト・セッションオブジェクト・ページコンテキストなどに格納されているデータを削除する
 - 通常の変数が対象ではないので、通常の変数をこのタグで削除することはできないので注意

Syntax

```
<c:remove var="削除したいオブジェクトの名称"  
          scope="スコープ" />
```

<c:remove>タグ(2)

- Sample

- セッションオブジェクトに"name"という名前で格納しておいたデータを削除するコード例

c-remove.jsp

```
<%@taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>
<%@ page isELIgnored ="false" %>
<html>
<body>
<h1>Core Tag (remove)</h1>
<c:set var="name" value="KnowledgeTaro" scope="session"/>
name=<c:out value="${name}" /><br>
<c:remove var="name" scope="session"/>
name=<c:out value="${name}" default="Deleted!" />
</body>
</html>
```


<c:remove>タグ(3)

- c-remove.jspの実行結果



<c:if>タグ(1)

- if文をタグ表記で実現する

Syntax

```
<c:if test="評価したい条件式(EL式で記述)"  
      var="評価結果を格納するオブジェクトの名称"  
      scope="オブジェクトのスコープ">
```

条件式がtrueの場合に実行したい処理

```
</c:if>
```

<c:if>タグ(2)

- Sample

- セッションオブジェクトに"a"、"b"という名前で格納しておいたデータを比較するコード例

c-if.jsp

```
<%@taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>
<%@ page isELIgnored ="false" %>
<html>
<body>
<h1>Core Tag (if)</h1>
<c:set var="a" value="100" scope="session"/>
<c:set var="b" value="200" scope="session"/>

<c:if test="${a<b}" var="result">
  <c:out value="${a}"/> is smaller than <c:out value="${b}"/>
  <br>
  result = <c:out value="${result}"/>
</c:if>
</body>
</html>
```

注) \${a<b}は文字の辞書式比較で真偽を判定しています

<c:if>タグ(2)

- c-if.jspの実行結果



XMLタグ

- `<x:parse>`
- `<x:transform>`
- `<x:set>`
- `<x:out>`
- `<x:if>`
- `<x:choose>/<x:when>/<x:otherwise>`
- `<x:forEach>`

Internationalizationタグ

- <fmt:formatDate>
- <fmt:formatNumber>
- <fmt:message>
- <fmt:param>
- <fmt:parseDate>
- <fmt:parseNumber>
- <fmt:requestEncoding>

<fmt:formatDate>タグ(1)

- 日付データをフォーマットして表示・格納するタグ

Syntax

```
<fmt:formatDate value="フォーマットしたい時刻の値"  
                pattern="表示フォーマット指定"  
                var="結果を格納するオブジェクトの名称"  
                scope="オブジェクトを格納したいスコープ" />
```

※ 表示フォーマット指定は、java.text.DateFormatクラスなどの指定と同様

var/scope属性を省略すると、結果がブラウザに出力され、
var/scope属性を指定すると、結果はブラウザには出力されない

<fmt:formatDate>タグ(2)

- Sample

- セッションオブジェクトに格納した現在日付を「yyyy/MM/dd」というフォーマットで表示するコード例

fmt-formatDate.jsp

```
<%@taglib uri="http://java.sun.com/jsp/jstl/fmt" prefix="fmt" %>
<%@ page isELIgnored ="false" %>
<html>
<body>
<h1>Fmt Tag (formatDate)</h1>
<% session.setAttribute("nowDate",new java.util.Date()); %>
<fmt:formatDate value="${nowDate}" pattern="yyyy/MM/dd"/>
</body>
</html>
```


<fmt:formatDate>タグ(3)

- fmt-formatDate.jspの実行結果



<fmt:formatNumber>タグ(1)

- 数値データをフォーマットして表示するタグ

Syntax

```
<fmt:formatNumber value="フォーマットしたい数値の値"  
                  type="number|currency|percent"  
                  pattern="フォーマット指定"  
                  currencySymbol="通貨記号"  
                  groupingUsed="true|false"  
                  maxIntegerDigits="整数部の最大桁数"  
                  minIntegerDigits="整数部の最小桁数"  
                  maxFractionDigits="小数部の最大桁数"  
                  minFractionDigits="小数部の最大桁数"  
                  var="結果を格納したいオブジェクトの名称"  
                  scope="オブジェクトを格納したスコープ"/>
```

※ 表示フォーマット指定は、java.text.DecimalFormatクラスなどの指定と同様
type属性を指定すると、それぞれのデフォルトのパターンが適用される
var/scope属性を省略すると、結果がブラウザに出力され、
var/scope属性を指定すると、結果はブラウザには出力されない

<fmt:formatNumber>タグ(2)

- Sample

- セッションオブジェクトに格納した数値を、様々なフォーマットで表示するコード例

fmt-formatNumber.jsp

```
<%@page contentType="text/html; charset=Windows-31J" %>
<%@taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>
<%@taglib uri="http://java.sun.com/jsp/jstl/fmt" prefix="fmt" %>
<%@ page isELIgnored ="false" %>
<html>
<body>
<h1>Fmt Tag (formatNumber)</h1>
<c:set var="num1" value="12345.6789"/>
<c:set var="num2" value="0.34567"/>
a) <fmt:formatNumber value="${num1}" type="number"/><br>
b) <fmt:formatNumber value="${num1}" type="currency" currencySymbol="\\"/><br>
c) <fmt:formatNumber value="${num1}" type="number" groupingUsed="false"/><br>
d) <fmt:formatNumber value="${num1}" maxFractionDigits="4"/><br>
e) <fmt:formatNumber value="${num1}" maxIntegerDigits="3"/><br>
f) <fmt:formatNumber value="${num1}" minIntegerDigits="6" groupingUsed="false"/><br>
g) <fmt:formatNumber value="${num2}" type="percent"/><br>
</body>
</html>
```

<fmt:formatNumber>タグ(3)

- fmt-formatNumber.jspの実行結果



SQLタグ

- <sql:setDataSource>
- <sql:query>
- <sql:param>
- <sql:dateParam>
- <sql:update>
- <sql:transaction>

Function

- fn:contains() / fn:containsIgnoreCase()
- fn:startsWith() / fn:endsWith()
- fn:escapeXml()
- fn:indexOf()
- fn:join() / fn:replace() / fn:split() / fn:trim()
- fn:length()
- fn:substring()
- fn:substringAfter() / fn:substringBefore()
- fn:toLowerCase() / fn:toUpperCase()

ServletとJSPの連携

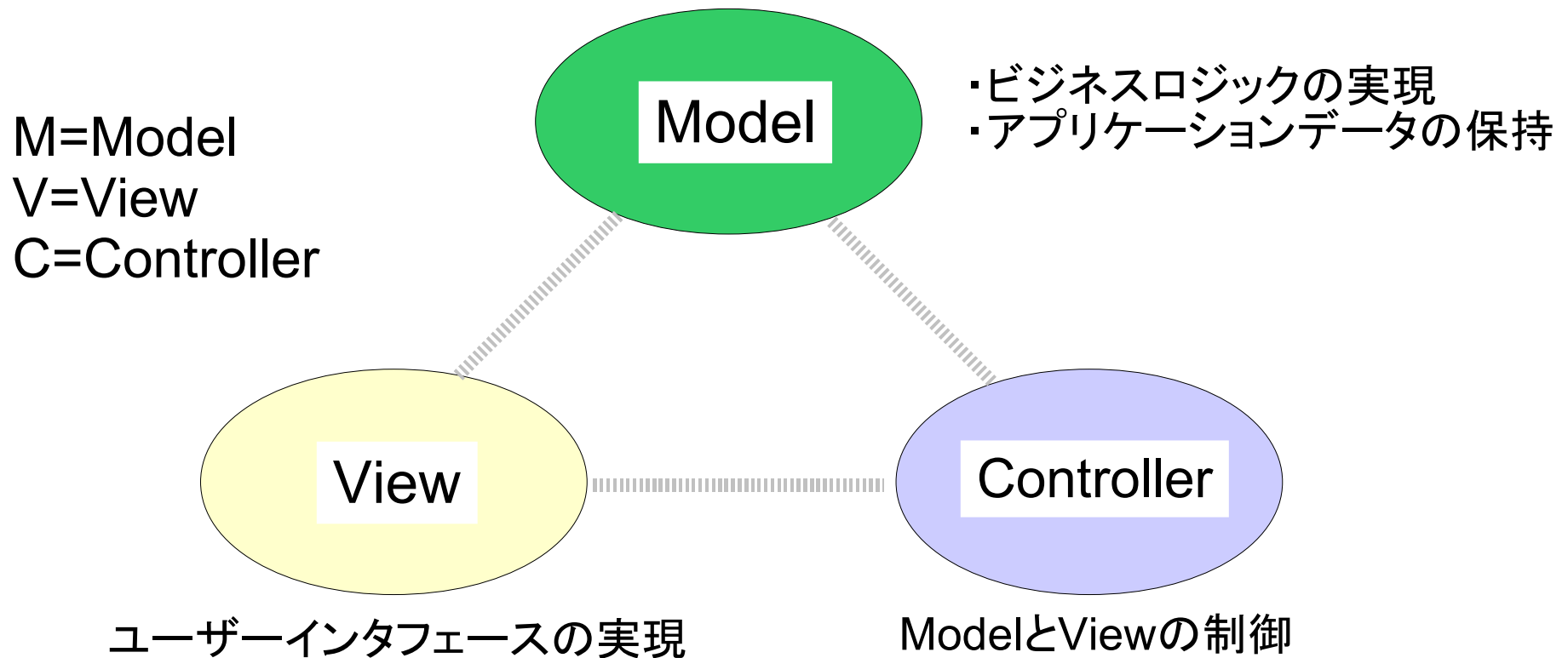
株式会社ナレッジエックス
<http://www.knowledge-ex.jp/>

Agenda

- ServletとJSPの連携
 - MVCアーキテクチャ
 - それぞれの役割分担
 - 連携に用いる各API

MVCアーキテクチャ(1)

- アプリケーションを3つの要素に分けて設計する
アーキテクチャ

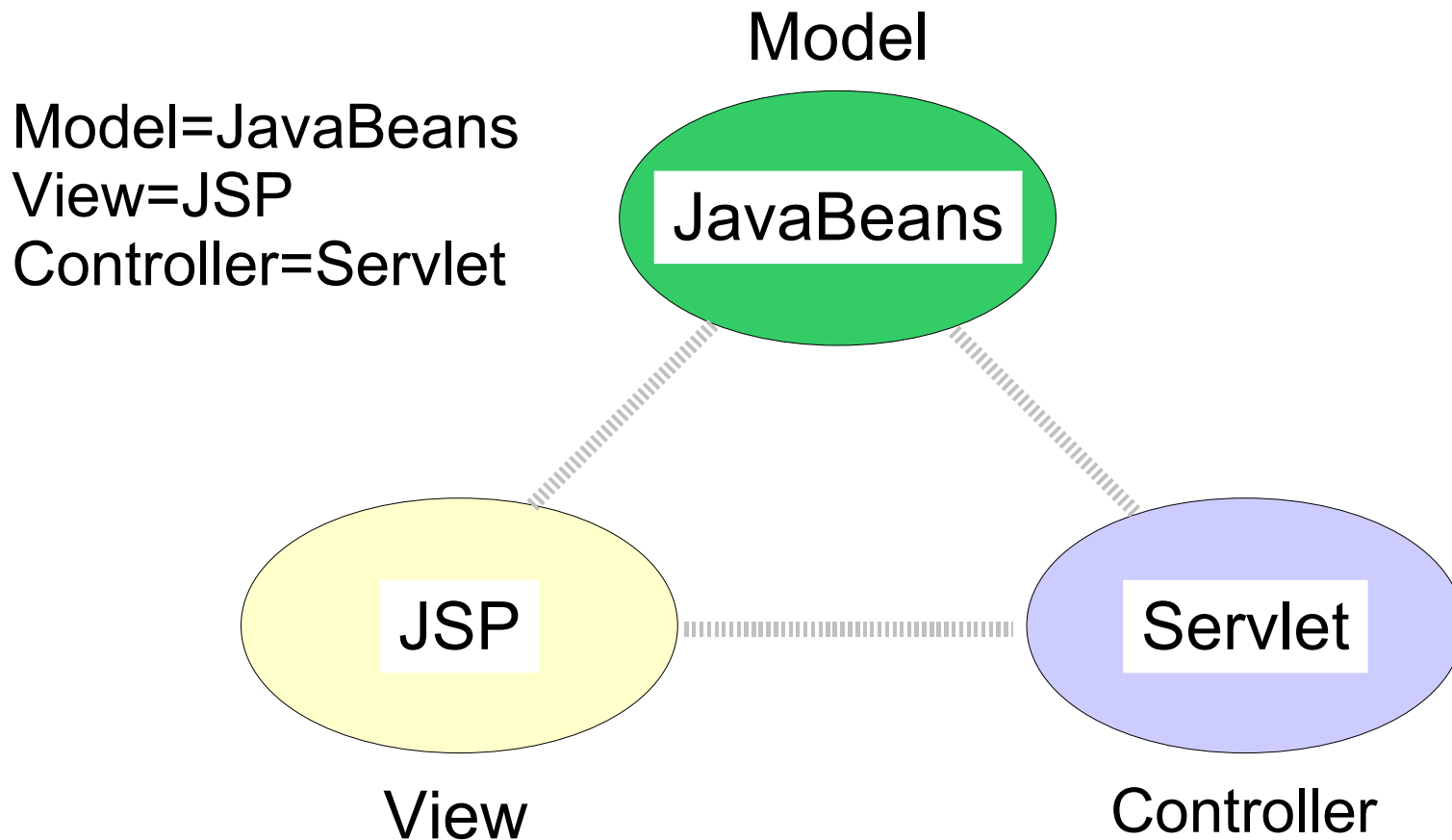


MVCアーキテクチャ(2)

- Model
 - ビジネスロジックを実現する
 - アプリケーションデータを格納・保持する
- View
 - ユーザーインターフェースを実現する
 - 画面の表示を行う
 - 画面からの入力を受け付ける
- Controller
 - ModelとViewを制御する
 - Viewからのリクエストに応じたModelを選択し実行
 - Modelの実行結果に応じたViewを選択し表示

それぞれの役割分担

- JavaのWebアプリケーションにMVCを適用すると...



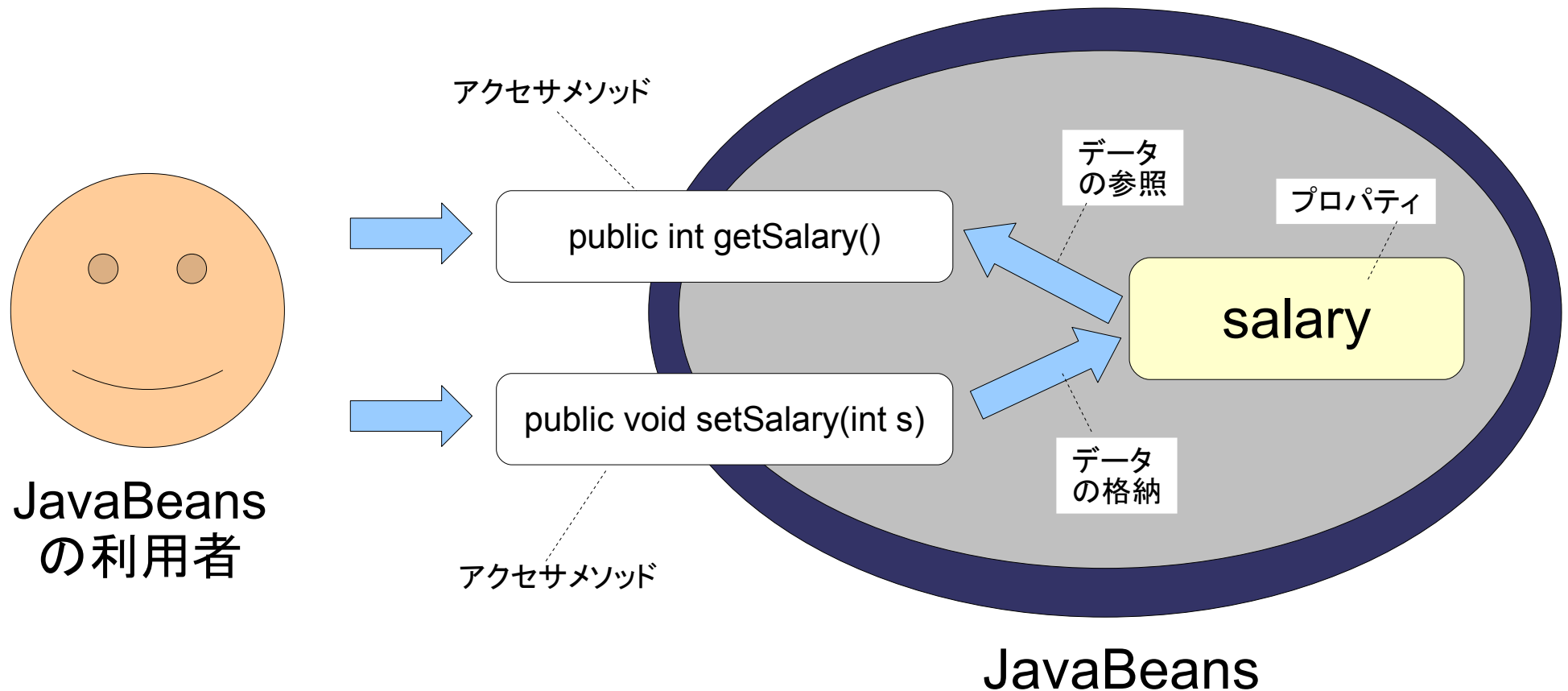
JavaBeansとは

- Javaソフトウェアの部品化を行うための形態
 - あるクラスがJavaBeansであるための条件(※)
 - デフォルトコンストラクタがコールできること
 - `java.io.Serializable`インターフェースを実装していること
 - 必須ではないが慣例となっているルール
 - プロパティという概念をサポートすること
 - `getter/setter`というアクセサメソッドによって値のやりとりを行う

(※)このほか、JavaBeansの仕様においては、JavaBeansとは「ビルダー・ツールで、ビジュアルに操作することが出来る、再利用可能なソフトウェア・コンポーネント」と定められていますが、サーバサイドアプリケーションにおいてはGUIを意識することは少ないので、あまりこの定義は重視されていません

プロパティとは

- getter/setterというアクセサメソッドによって値のやりとりを行う形態



プロパティに関するルール(1)

- プロパティ
 - 通常、プロパティ名の先頭は小文字で定義する
 - プロパティの値はカプセル化されている
 - 値を格納する変数名はプロパティ名と同じでなくとも良い
 - プロパティを格納する変数は必ずしも存在しなくても良い
 - プロパティは1クラスに複数定義可能

プロパティに関するルール(2)

- アクセサメソッド

- 値の参照＝getterメソッド

- メソッド名は「get+プロパティ名」
 - プロパティ名の先頭は大文字で記述
 - 引数は、指定しなくて良い
 - 戻り値は、プロパティのデータ型

- 値の更新＝setterメソッド

- メソッド名は「set+プロパティ名」
 - プロパティ名の先頭は大文字で記述
 - 引数は、プロパティのデータ型
 - 戻り値はvoid型

JavaBeansの例

- Sample
 - salaryというプロパティをもつJavaBeansクラスの例

SampleBean.jsp

```
package web;

import java.io.Serializable;

public class SampleBean implements Serializable {

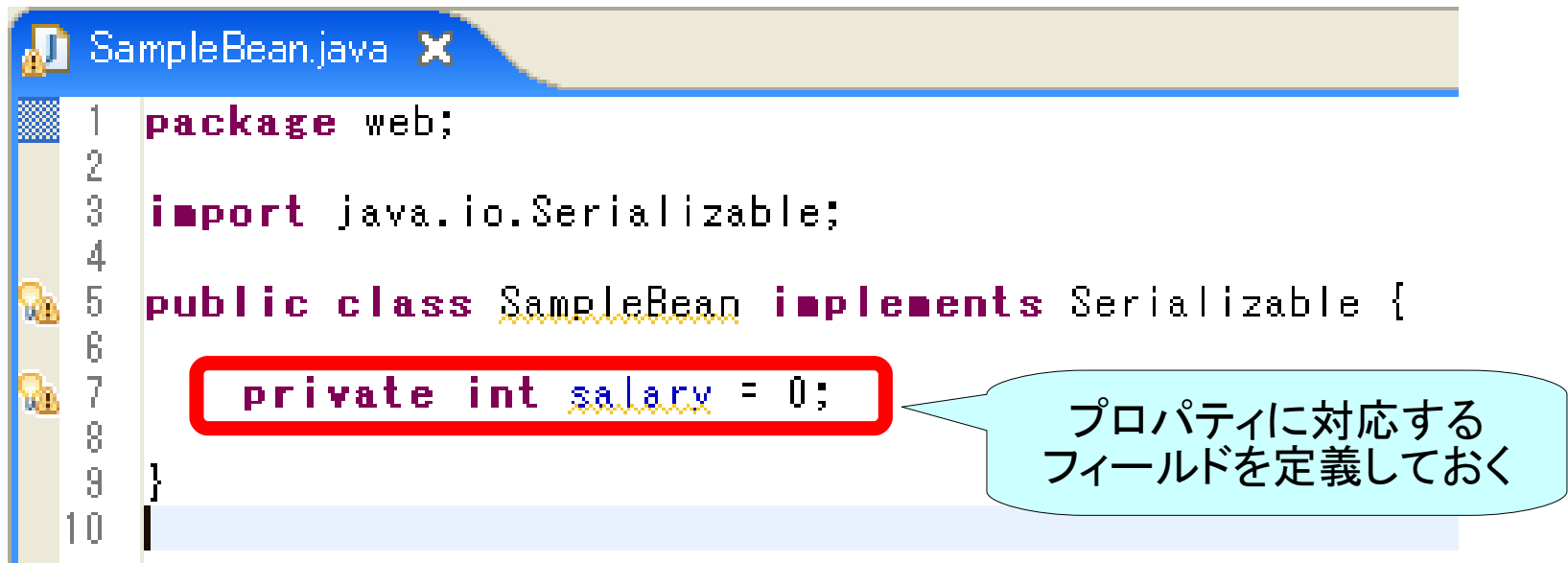
    private int salary = 0;

    public int getSalary() {
        return salary;
    }

    public void setSalary(int salary) {
        this.salary = salary;
    }
}
```


Eclipseでアクセサメソッドを作る(1)

- Eclipseでは、プロパティに対応するフィールドを定義しておけばメニュー操作でアクセサメソッドを追加できる

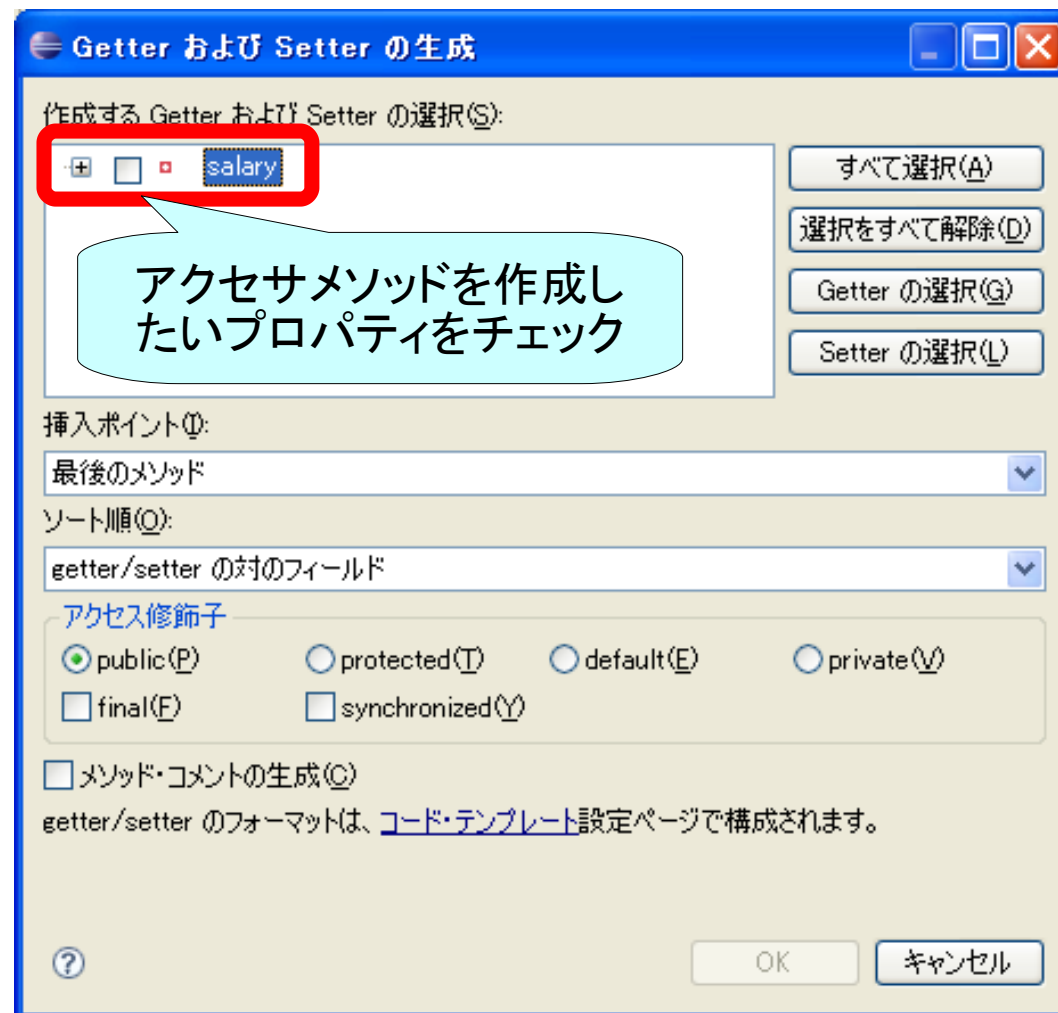


```
SampleBean.java x
1 package web;
2
3 import java.io.Serializable;
4
5 public class SampleBean implements Serializable {
6     private int salary = 0;
7
8 }
9
10
```

プロパティに対応するフィールドを定義しておく

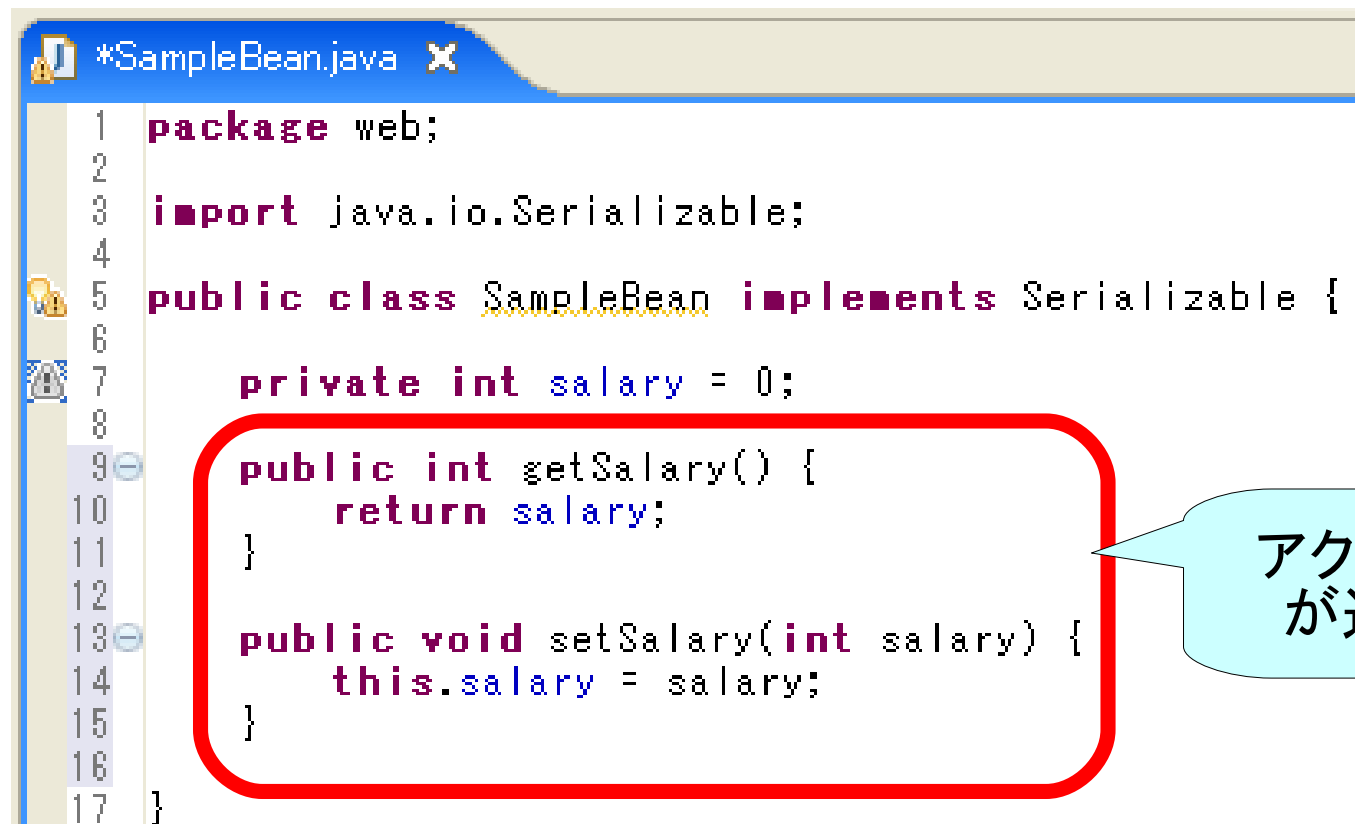
Eclipseでアクセサメソッドを作る(2)

- メニュー「ソース」→「GetterおよびSetterの生成」を選択し、表示されるダイアログで作成したいプロパティ名をチェックし、「OK」をクリック



Eclipseでアクセサメソッドを作る(3)

- ソースファイルに、選択したプロパティに対するアクセサメソッドが追加されている



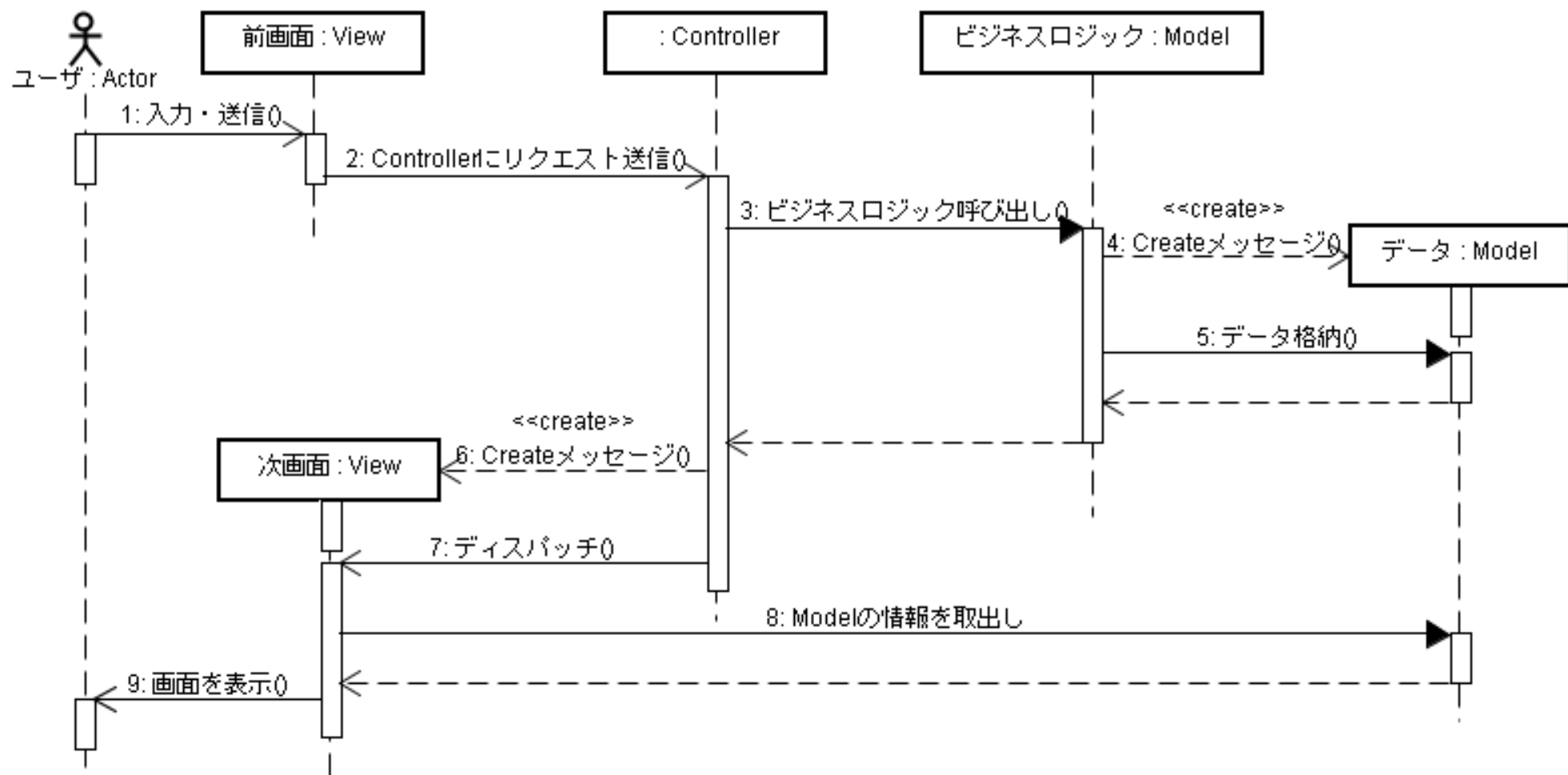
```
1 package web;
2
3 import java.io.Serializable;
4
5 public class SampleBean implements Serializable {
6
7     private int salary = 0;
8
9     public int getSalary() {
10         return salary;
11     }
12
13     public void setSalary(int salary) {
14         this.salary = salary;
15     }
16
17 }
```

アクセサメソッド
が追加された



int型の「salary」というプロパティを持つSampleBeanクラスを作成してみましょう

MVCアーキテクチャのシーケンス



連携に用いる各API

- Servlet
 - ServletからJSPを呼び出し実行する
 - リクエストオブジェクトにインスタンスを格納する
 - セッションオブジェクトにインスタンスを格納する
- JSP
 - JavaBeansの内容を取り出して操作する

ServletからJSPを呼び出す

- javax.servlet.RequestDispatcherオブジェクトを使用
 - ServletRequest#getRequestDispatcher()で取得
 - RequestDispatcher#forward()でリクエスト・レスポンスを指定されたリソースへ転送(ディスパッチ)
 - ServletRequestからRequestDispatcherを取得した場合、同一コンテキストのリソースのみ転送可能

Syntax

```
ServletRequest#getRequestDispatcher(転送先URL指定)  
RequestDispatcher#forward(ServletRequest, ServletResponse)
```

コード例

```
RequestDispatcher rd = request.getRequestDispatcher("/test.jsp");  
rd.forward(request, response);
```

RequestDispatcher使用例

- Sample

- 同一コンテキストの/page/test.jspへ転送するサーブレットのコード例

DispatchServlet.java

```
package web;

import java.io.IOException;

import javax.servlet.RequestDispatcher;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

public class DispatchServlet extends HttpServlet {

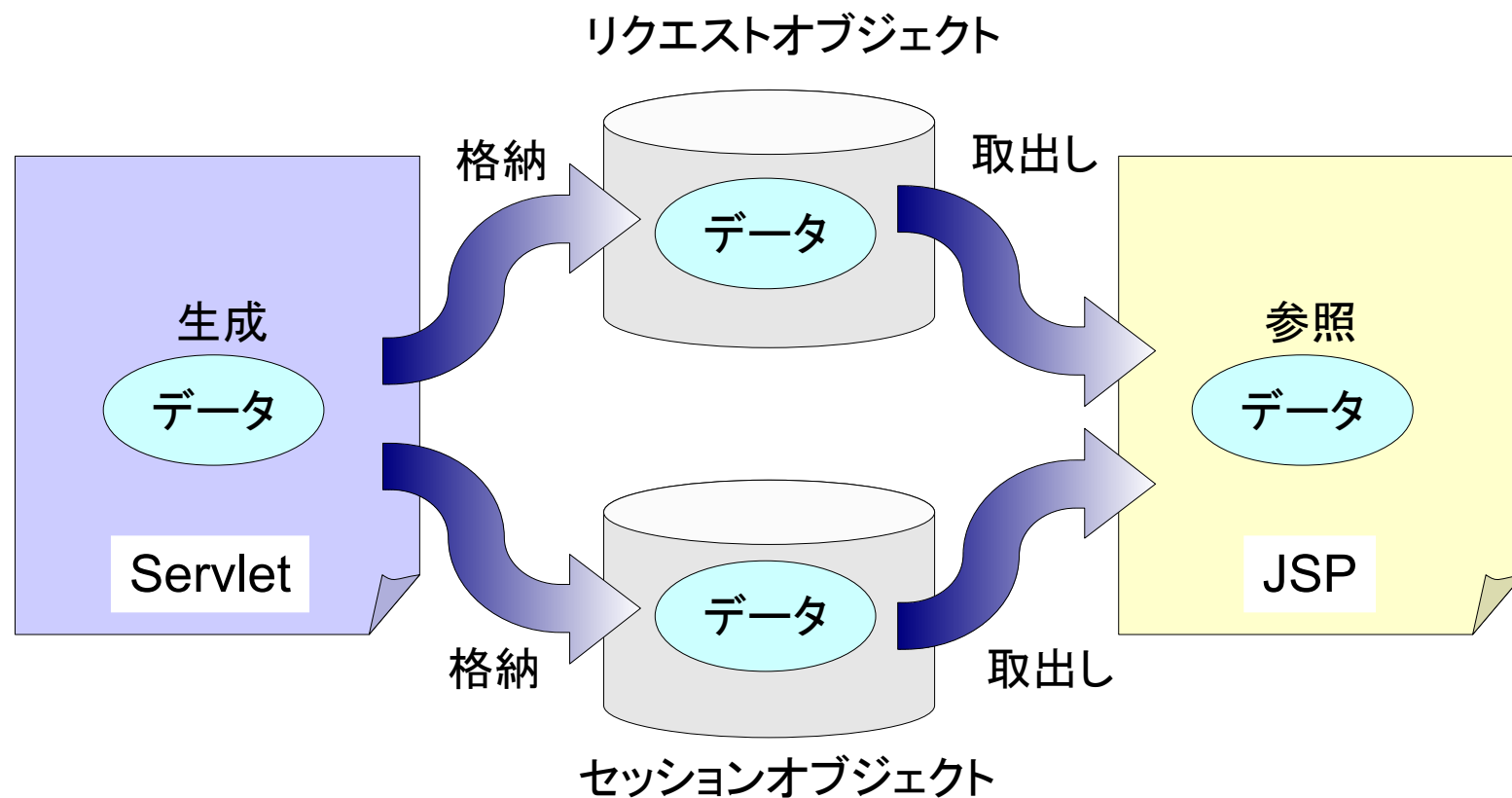
    protected void doGet(HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException {
        RequestDispatcher rd = request.getRequestDispatcher("/page/test.jsp");
        rd.forward(request, response);
    }

}
```

このサーブレットのサーブレットパスは「/dispatch」としてください

ServletとJSPでデータを共有する(1)

- ServletとJSPではリクエストオブジェクトやセッションオブジェクトを共用することができる
 - Servletでリクエスト・セッションオブジェクトにデータを格納しておけば、JSPで取り出して利用することができる



ServletとJSPでデータを共有する(2)

- Servletでのデータ格納
 - リクエストオブジェクト・セッションオブジェクトともにsetAttribute()メソッドでオブジェクトを格納できる

Syntax

```
HttpServletRequest#setAttribute(オブジェクト名, インスタンス)  
HttpSession#setAttribute(オブジェクト名, インスタンス)
```

コード例

```
HttpSession session = request.getSession(true);  
session.setAttribute("name", "KnowledgeTaro");  
request.setAttribute("age", new Integer(20));
```

ServletとJSPでデータを共有する(3)

- Sample

- リクエスト、セッションオブジェクトにデータを格納して
/page/dispatch.jspへ転送するサーブレットのコード例

SetAttributeServlet.java

```
package web;

import java.io.IOException;

import javax.servlet.*;
import javax.servlet.http.*;

public class SetAttributeServlet extends HttpServlet {

    protected void doGet(HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException {

        HttpSession session = request.getSession(true);
        session.setAttribute("name", "KnowledgeTaro");
        request.setAttribute("age", 20);
        RequestDispatcher rd = request.getRequestDispatcher("/page/dispatch.jsp");
        rd.forward(request, response);
    }

}
```

このサーブレットのサーブレットパスは「/setattr」としてください

ServletとJSPでデータを共有する(4)

- JSPでのデータ取り出し
 - 暗黙オブジェクト(request/session)のgetAttributeメソッドで取り出す
 - <jsp:useBean>/<jsp:getProperty>で取り出す(後述)
 - JSTLの各種タグやEL式で取り出す

ServletとJSPでデータを共有する(5)

- Sample

- リクエスト、セッションオブジェクトに格納済みのデータを取り出して表示するJSPのコード例(暗黙オブジェクトとExpression利用)

dispatch.jsp

```
<%@page contentType="text/html; charset=Windows-31J" %>
<html>
<body>
<h1>Dispatched JSP</h1>
session(name) = <%= session.getAttribute("name") %><br>
request(age)   = <%= request.getAttribute("age") %>
</body>
</html>
```

ServletとJSPでデータを共有する(6)

- SetAttributeServletの実行結果



ServletとJSPでデータを共有する(6)

- Sample

- リクエスト、セッションオブジェクトに格納済みのデータを取り出して表示するJSPのコード例(EL式使用)

dispatch2.jsp

```
<%@page contentType="text/html; charset=Windows-31J" %>
<%@ page isELIgnored ="false" %>
<html>
<body>
<h1>Dispatched JSP</h1>
session(name) = ${name}<br>
request(age) = ${age}
</body>
</html>
```

JavaBeans連携のためのタグ

- JavaBeans連携のためのタグ

- <jsp:useBean>
- <jsp:getProperty>
- <jsp:setProperty>

<jsp:useBean>

- Servletなどでリクエスト/セッションオブジェクトに格納済のJavaBeansをScriptlet・Expressionで利用できるように定義する
- またはページ内でJavaBeansのインスタンスを新しく作成する

Syntax

```
<jsp:useBean id="格納されたJavaBeansの名称"  
             scope="page|request|session|application"  
             class="クラス名(パッケージ名含む)"  
             type="クラス名(パッケージ名含む)"/>
```

※ id属性の値は、リクエストオブジェクトやセッションオブジェクトのsetAttribute()メソッドの第一引数と同じにする

<jsp:getProperty>

- <jsp:useBean>で定義したJavaBeansのプロパティを取得し、ブラウザに出力する
 - 内部で指定されたJavaBeansのgetterメソッドが実行され、値が取得される

Syntax

```
<jsp:getProperty name="定義したJavaBeansの名称"  
                  property="取り出したいプロパティ名" />
```

※ name属性の値は、<jsp:useBean>タグのid属性の値と同じにする

<jsp:setProperty>

- <jsp:useBean>で定義したJavaBeansのプロパティの値を変更する
 - 内部で指定されたJavaBeansのsetterメソッドが実行され、値が変更される

Syntax

```
<jsp:setProperty name="定義したJavaBeansの名称"  
                 property="変更したいプロパティ名"  
                 value="格納したい値" />
```

※ name属性の値は、<jsp:useBean>タグのid属性の値と同じにする

<jsp:useBean>/<jsp:getProperty>の使用例(1)

- Sample(JavaBeans)
 - 従業員の氏名、年齢を格納するJavaBeansの例

EmpBean.java

```
package web;

public class EmpBean {

    private String name = null;
    private int age = 0;

    public int getAge() {
        return age;
    }
    public void setAge(int age) {
        this.age = age;
    }
    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }
}
```

<jsp:useBean>/<jsp:getProperty>の使用例(2)

- Sample(Servlet)

- リクエストオブジェクトにEmpBeanのデータを格納して/page/useBean.jspにディスパッチするServletのコード例

UseBeanServlet.java

```
package web;

import java.io.IOException;
import javax.servlet.*;
import javax.servlet.http.*;

public class UseBeanServlet extends HttpServlet {

    protected void doGet(HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException {

        EmpBean bean = new EmpBean();
        bean.setName("KnowledgeTaro");
        bean.setAge(20);
        request.setAttribute("emp", bean);
        RequestDispatcher rd = request.getRequestDispatcher("/page/test.jsp");
        rd.forward(request, response);
    }
}
```

このサーブレットのサーブレットパスは「/useBean」としてください

<jsp:useBean>/<jsp:getProperty>の使用例(3)

- Sample(Servlet)

- <jsp:useBean>と<jsp:getProperty>を使って、JavaBeansの内容を表示するJSPのコード例

useBean.jsp

```
<%@page contentType="text/html; charset=Windows-31J" %>
<jsp:useBean id="emp" scope="request" class="web.EmpBean"/>
<html>
<body>
<h1>UseBean</h1>
<table border>
  <tr>
    <td>Name</td>
    <td><jsp:getProperty name="emp" property="name"/></td>
  </tr>
  <tr>
    <td>Age</td>
    <td><jsp:getProperty name="emp" property="age"/></td>
  </tr>
</table>
</body>
</html>
```

<jsp:useBean>/<jsp:getProperty>の使用例(4)

- UseBeanServletの実行結果



演習(1)

- MVCアーキテクチャを用いたアプリケーションを作成しましょう

「占いアプリケーション」の仕様

- ① 最初に表示される画面で、自分の名前と、生年月日を入力
- ② 送信ボタンを押下すると、次の画面が表示される
- ③ 次画面ではその人の金運、恋愛運、仕事運、健康運、全体運が表示される

演習(2)

• 画面構成

入力画面

占いアプリケーション

氏名

生年月日 年 月 日

結果画面

ナレツジ太郎さんの運勢

金運 ★

恋愛運 ★★

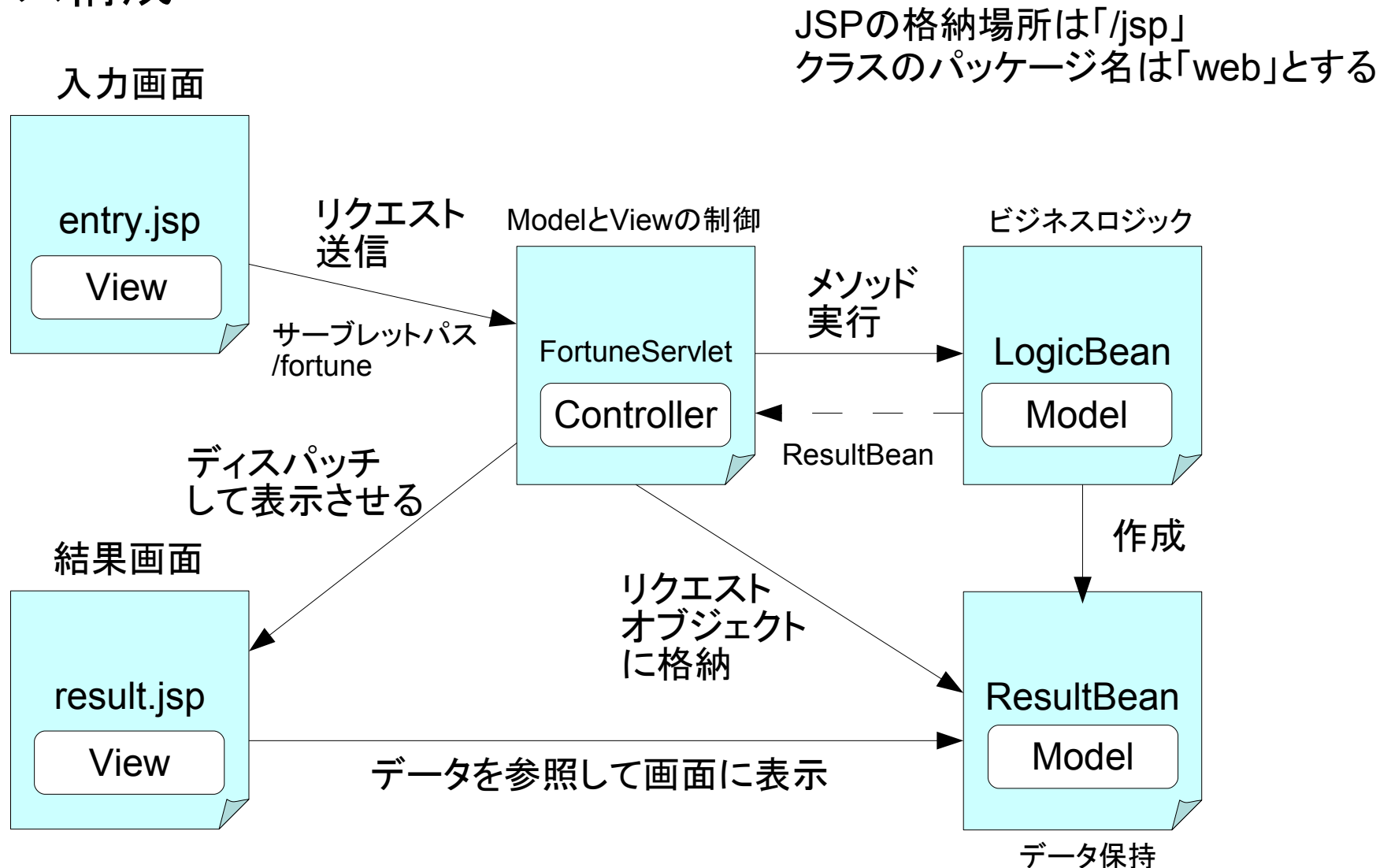
仕事運 ★★★★★

健康運 ★★★★★

全体運 ★★★

演習(3)

- クラス構成



演習(4)

• entry.jspの作成

入力画面

占いアプリケーション

氏名

生年月日 年 月 日

- ① 氏名の入力
テキストフィールドで表示
パラメータ名は「name」
- ② 生年月日の入力
コンボボックスで表示
パラメータ名はそれぞれ「year」「month」「date」
- ③ フォーム内容の送信先
 - ・`http://localhost:8080/kx/fortune`
(FortuneServletのURL)とする
 - ・`method="POST"`を指定する

演習(5)

- ResultBeanの作成

「money」「love」「work」「health」「total」という
プロパティ(すべてString型)を持つBeanを作成

演習(6)

• LogicBeanの作成

- ① execute()メソッドを定義
 - ・引数は、String name,int year,int month,int date、戻り値は、ResultBean型
- ② 引数nameの文字数を取得し、変数に格納しておく(以下「文字数」と表記)
例) int len = name.length();
- ③ それぞれの運勢を次の式で計算
 - ・金運 = 「文字数」×「年」×「月」×「日」を5で割った余り+1
 - ・恋愛運 = (「文字数」+「年」)×「月」×「日」を5で割った余り+1
 - ・仕事運 = (「文字数」+「年」+「月」)×「日」を5で割った余り+1
 - ・健康運 = 「文字数」+「年」+「月」+「日」を5で割った余り+1
 - ・全体運 = 上記4つの平均値(小数点以下切捨て)
- ④ ResultBeanのインスタンスを作成し、それぞれの運勢の数値を「★」の文字列に変えてプロパティに格納(金運=4なら、「★★★★」という文字列を格納)
 - ・金運 = setMoney()メソッドで格納
 - ・恋愛運 = setLove()メソッドで格納
 - ・仕事運 = setWork()メソッドで格納
 - ・健康運 = setHealth()メソッドで格納
 - ・全体運 = setTotal()メソッドで格納
- ⑤ インスタンスをreturnで戻り値に指定

①のメソッドをまず定義し、そのメソッドの中に②～⑤の内容を実装してください

演習(7)

• FortuneServletの作成

- ① doPost()メソッドをオーバーライド
- ② リクエストパラメータ「name」「year」「month」「date」をそれぞれ取り出して、変数に格納しておく
 - ・year,month,dateは、Integer.parseInt()メソッドを使って、整数に変換する
例) `int year = Integer.parseInt(request.getParameter("year"));`
- ③ LogicBeanのインスタンスを作成
- ④ LogicBeanのインスタンスに対して、execute()メソッドを実行
 - 引数は、②で格納した変数を「name」「year」「month」「date」の順に指定
- ⑤ execute()メソッドの戻り値として受け取ったResultBeanのインスタンスを、「result」という名称で、HttpServletRequestにsetAttribute()メソッドで格納
- ⑥ 次の画面のJSP(/jsp/result.jsp)を、RequestDispatcherを使って実行

演習(8)

• result.jspの作成

結果画面

ナレッジ太郎さんの運勢

金運 ★
恋愛運 ★★
仕事運 ★★★★★
健康運 ★★★★★
全体運 ★★★

- ① 「～さんの運勢」の表示
名前は、リクエストパラメータ「name」を表示
- ② それぞれの運勢を表示
<jsp:useBean>を使ってJavaBeansを定義
 - ・id属性は「result」
 - ・class属性は「web.ResultBean」
 - ・scope属性は「request」<jsp:getProperty>を使って、ResultBeanのそれぞれのプロパティ値を表示

演習(9)

- 実行例(入力画面)

Mozilla Firefox

ファイル(F) 編集(E) 表示(V) 移動(G) ブックマーク(B) ツール(T) /

← → ↺ × 🏠 ➕ 📖 PageRank http://loc

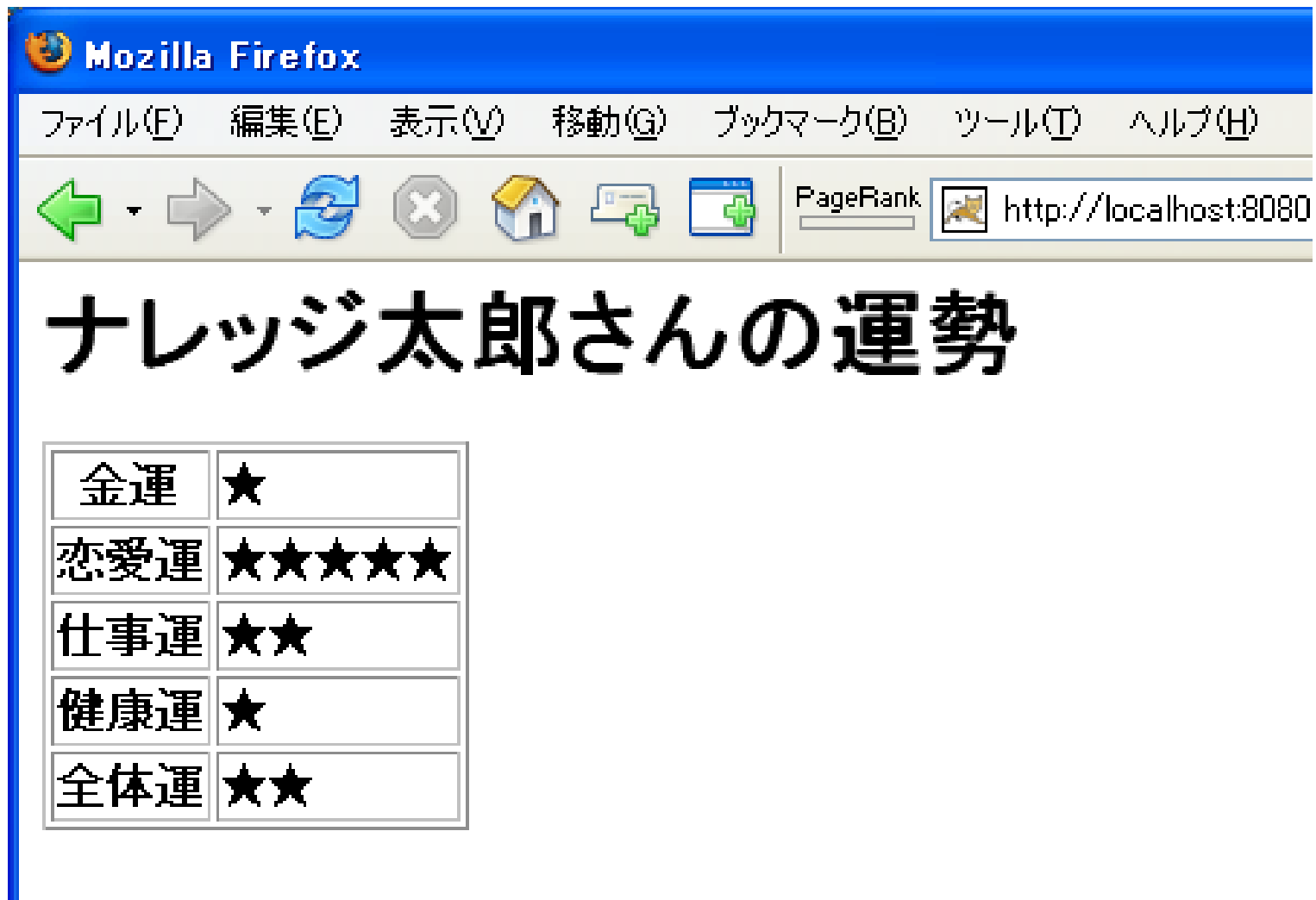
占いアプリケーション

氏名

生年月日 年 月 日

演習(10)

- 実行例(結果画面)



回答例(1)

- entry.jsp

```
<%@page contentType="text/html; charset=Windows-31J" %>
<html>
<body>
<h1>占いアプリケーション</h1>
<form action="http://localhost:8080/web/fortune" method="POST">
<table>
<tr>
<th>氏名</th>
<td><input type="text" name="name"></td>
</tr>
<tr>
<th>生年月日</th>
<td>
<select name="year">
<% for(int i=1900;i<=2010;i++) { %>
<option value="<%= i %>"><%= i %></option>
<% } %>
</select>年
```

前半

```
<select name="month">
<% for(int i=1;i<=12;i++) { %>
<option value="<%= i %>"><%= i %></option>
<% } %>
</select>
月
<select name="date">
<% for(int i=1;i<=31;i++) { %>
<option value="<%= i %>"><%= i %></option>
<% } %>
</select>
日
</td>
</tr>
</table>
<input type="submit" value="占う！">
</form>
</body>
</html>
```

続き

回答例(2)

• ResultBean

```
package web;

import java.io.Serializable;

public class ResultBean
    implements Serializable {

    private String money = "";
    private String love = "";
    private String work = "";
    private String health = "";
    private String total = "";

    public String getHealth() {
        return health;
    }
    public void setHealth(String health) {
        this.health = health;
    }
    public String getLove() {
        return love;
    }
}
```

```
    public void setLove(String love) {
        this.love = love;
    }
    public String getMoney() {
        return money;
    }
    public void setMoney(String money) {
        this.money = money;
    }
    public String getTotal() {
        return total;
    }
    public void setTotal(String total) {
        this.total = total;
    }
    public String getWork() {
        return work;
    }
    public void setWork(String work) {
        this.work = work;
    }
}
```

回答例(3)

- LogicBean

```
package web;

import java.io.Serializable;

public class LogicBean implements Serializable {
    String[] star = {"", "★", "★★", "★★★", "★★★★", "★★★★★", "★★★★★★"};
    public ResultBean execute(String name, int year, int month, int date) {
        int len = name.length();
        int money = (len*year*month*date)%5 + 1;
        int love = ((len+year)*month*date)%5 + 1;
        int work = ((len+year+month)*date)%5 + 1;
        int health = (len+year+month+date)%5 + 1;
        int total = (int) ((money+love+work+health)/4);

        ResultBean bean = new ResultBean();
        bean.setMoney(star[money]);
        bean.setLove(star[love]);
        bean.setWork(star[work]);
        bean.setHealth(star[health]);
        bean.setTotal(star[total]);
        return bean;
    }
}
```

回答例(4)

- FortuneServlet

```
package web;

import java.io.IOException;
import javax.servlet.*;
import javax.servlet.http.*;

public class FortuneServlet extends HttpServlet {

    protected void doPost(HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException {
        request.setCharacterEncoding("Windows-31J");
        String name = request.getParameter("name");
        int year = Integer.parseInt(request.getParameter("year"));
        int month = Integer.parseInt(request.getParameter("month"));
        int date = Integer.parseInt(request.getParameter("date"));
        LogicBean logic = new LogicBean();
        ResultBean bean = logic.execute(name, year, month, date);
        request.setAttribute("result", bean);
        RequestDispatcher rd = request.getRequestDispatcher("/jsp/result.jsp");
        rd.forward(request, response);
    }
}
```

回答例(5)

- result.jsp

```
<%@page contentType="text/html; charset=Windows-31J" %>
<jsp:useBean id="result" class="web.ResultBean" scope="request"/>
<html>
<body>
<h1><%= request.getParameter("name") %>さんの運勢</h1>
<table border>
<tr>
<th>金運</th><td><jsp:getProperty name="result" property="money"/></td>
</tr>
<tr>
<th>恋愛運</th><td><jsp:getProperty name="result" property="love"/></td>
</tr>
<tr>
<th>仕事運</th><td><jsp:getProperty name="result" property="work"/></td>
</tr>
<tr>
<th>健康運</th><td><jsp:getProperty name="result" property="health"/></td>
</tr>
<tr>
<th>全体運</th><td><jsp:getProperty name="result" property="total"/></td>
</tr>
</table>
</body>
</html>
```

回答例(6)

- web.xml(演習に関連する部分のみ抜粋)

```
<?xml version="1.0" encoding="Windows-31J"?>
<web-app>

    <servlet>
        <servlet-name>fs</servlet-name>
        <servlet-class>web.FortuneServlet</servlet-class>
    </servlet>

    <servlet-mapping>
        <servlet-name>fs</servlet-name>
        <url-pattern>/fortune</url-pattern>
    </servlet-mapping>

</web-app>
```

「Webアプリケーション開発基礎」テキスト

2007年3月13日	v0.9.001	公開
2007年10月19日	v0.9.002	公開
2008年9月24日	v0.9.003	公開
2008年11月25日	v0.9.004	公開
2009年3月10日	v0.9.005	公開
2009年9月26日	v0.9.006	公開

株式会社ナレッジエクス
