

JavaEEの基本機能



JavaEEの基本機能

- 本章では、JavaEEの中でもWebアプリケーションを作成するための基本機能に絞って、その動作をハンズオンを通して体験していきます。
 - サークレット
 - JSP
 - サークレット・フィルタ

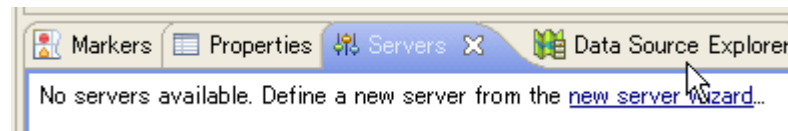
- JSPは既に古くなった技術ですが依然として既存システムでは広く使われており、保守の際に仕組みを理解している必要があること、Webフレームワークの歴史を理解する上で必要な知識であるため解説します。
新規システムでは使用すべきではありません。

準備

- 本章では、Apache Software Foundationが提供する、Tomcat(7.0.25)というサーバを使用します。
- ここでは開発のために、TomcatをEclipseに登録します。
- Eclipse IDE for Java EE Developersの3.7.2を使用します。
- Eclipseを起動したら、新しくワークスペースを作成してください。

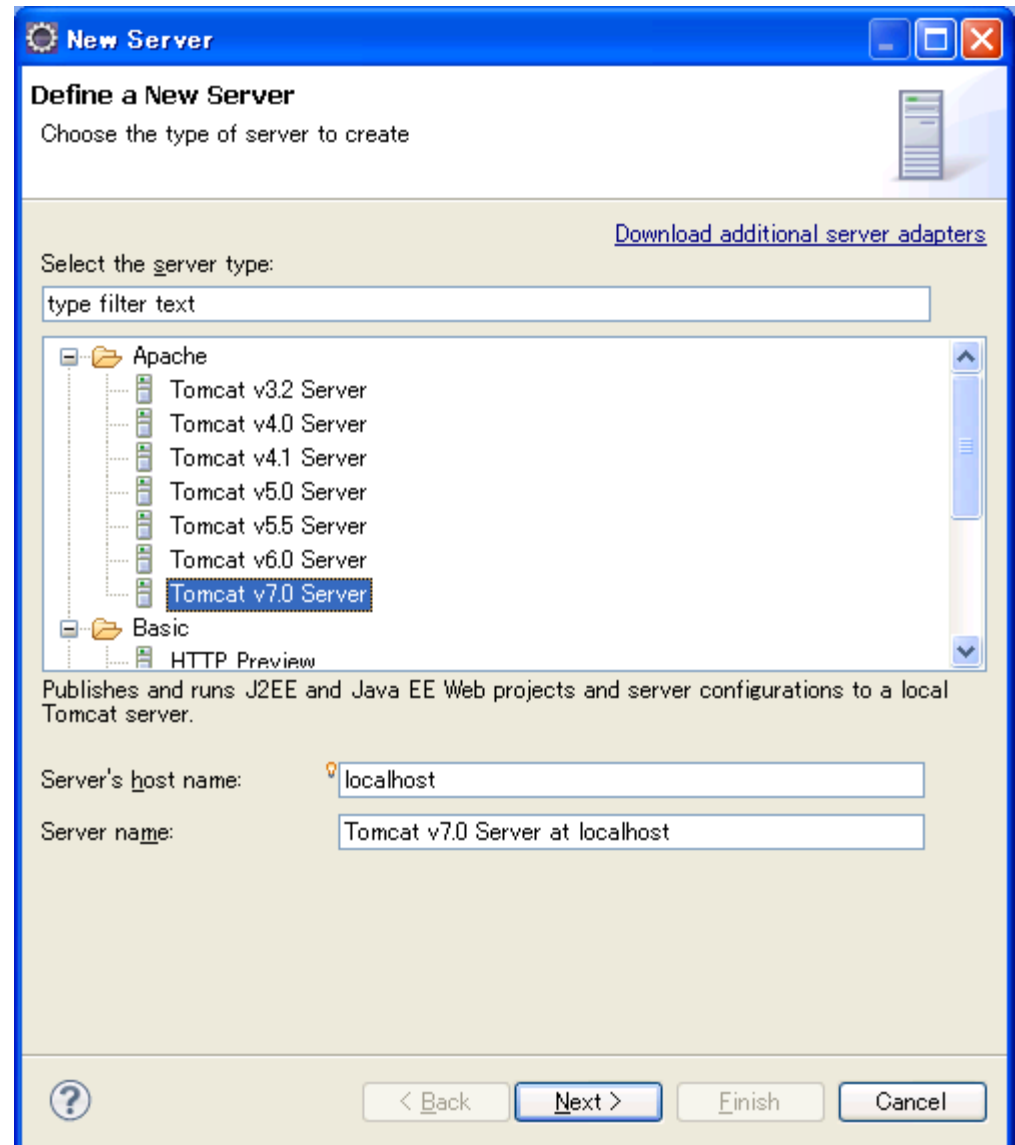
準備

- 起動するとデフォルトで、JavaEEパースペクティブになっているはずですが(右上にJavaEEと表示されているはず)、なっていないければ変更してください。
- 画面の下に「Server」というタブがあるので、new server wizardをクリックします。



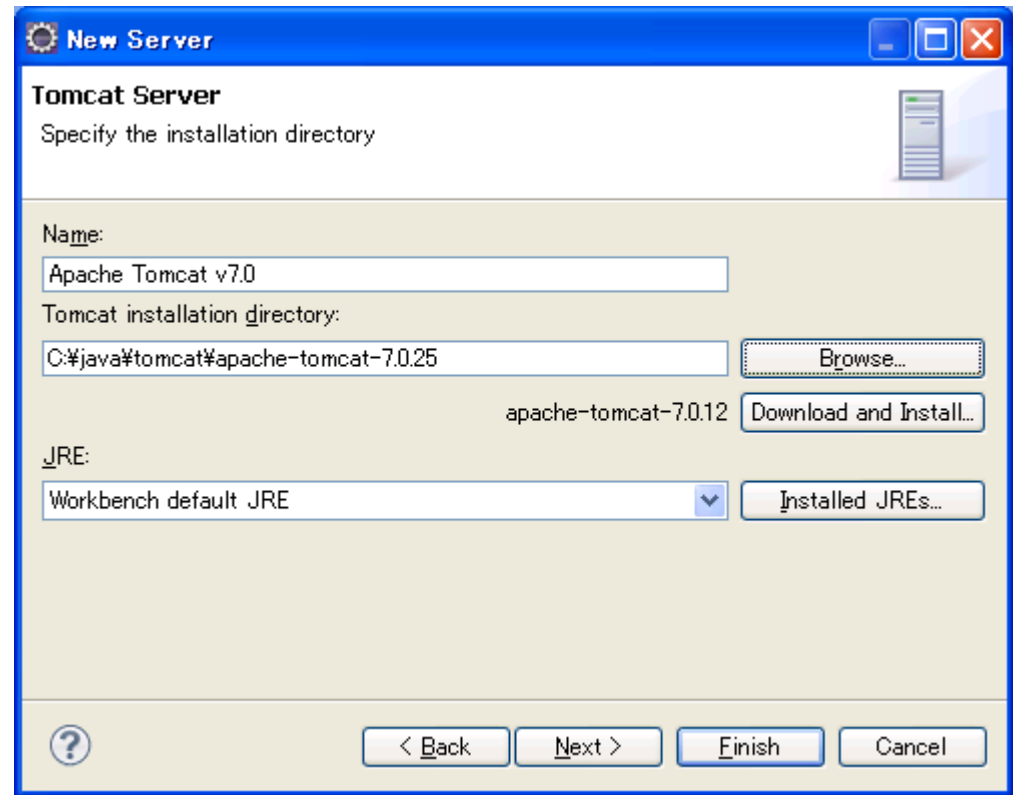
準備

- Apacheの下、Tomcat v7.0 Serverを選び、Nextをクリック。

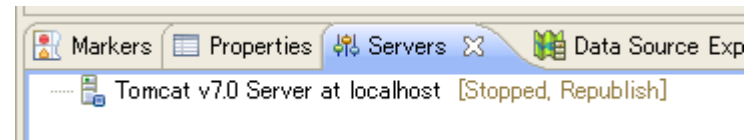


準備

- Browseボタンを押して、Tomcatをインストールした場所を指定します。
- Finishをクリックします。

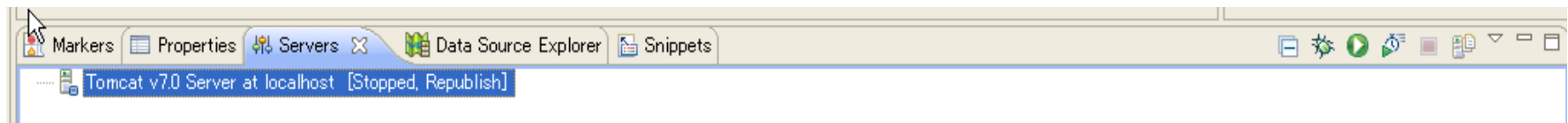
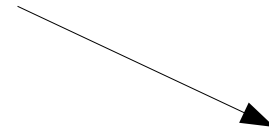


- Tomcatが作成、登録されます。



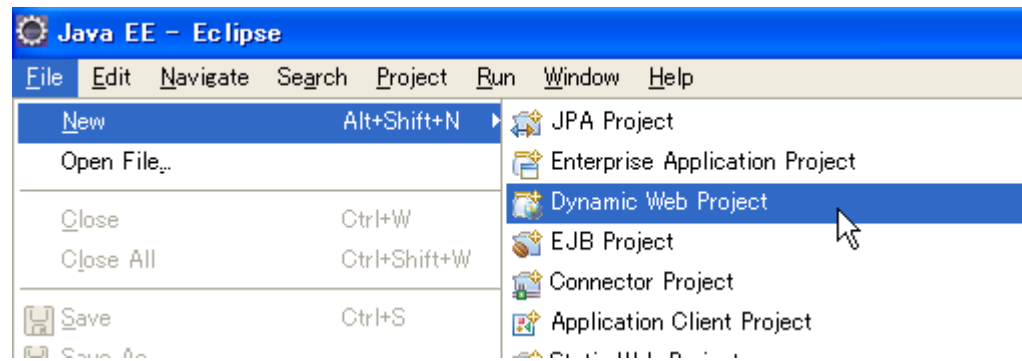
準備

- Tomcatを選択してから、右クリックして開始、あるいは、右の方にある実行ボタン(緑のボタン)をクリックします。
- 「Started」になれば、正しく起動しています。



サーブレット

- 前章で解説したサーブレットを作成してみましょう。
- File->New->Dynamic Web Projectをクリックします。



サーブレット

- プロジェクト名
に”servlet01”と入力し
て、Finishをクリックしま
す。

New Dynamic Web Project

Dynamic Web Project
Create a standalone Dynamic Web project or add it to a new or existing Enterprise Application.

Project name:

Project location
☒ Use default location
Location:

Target runtime

Dynamic web module version

Configuration

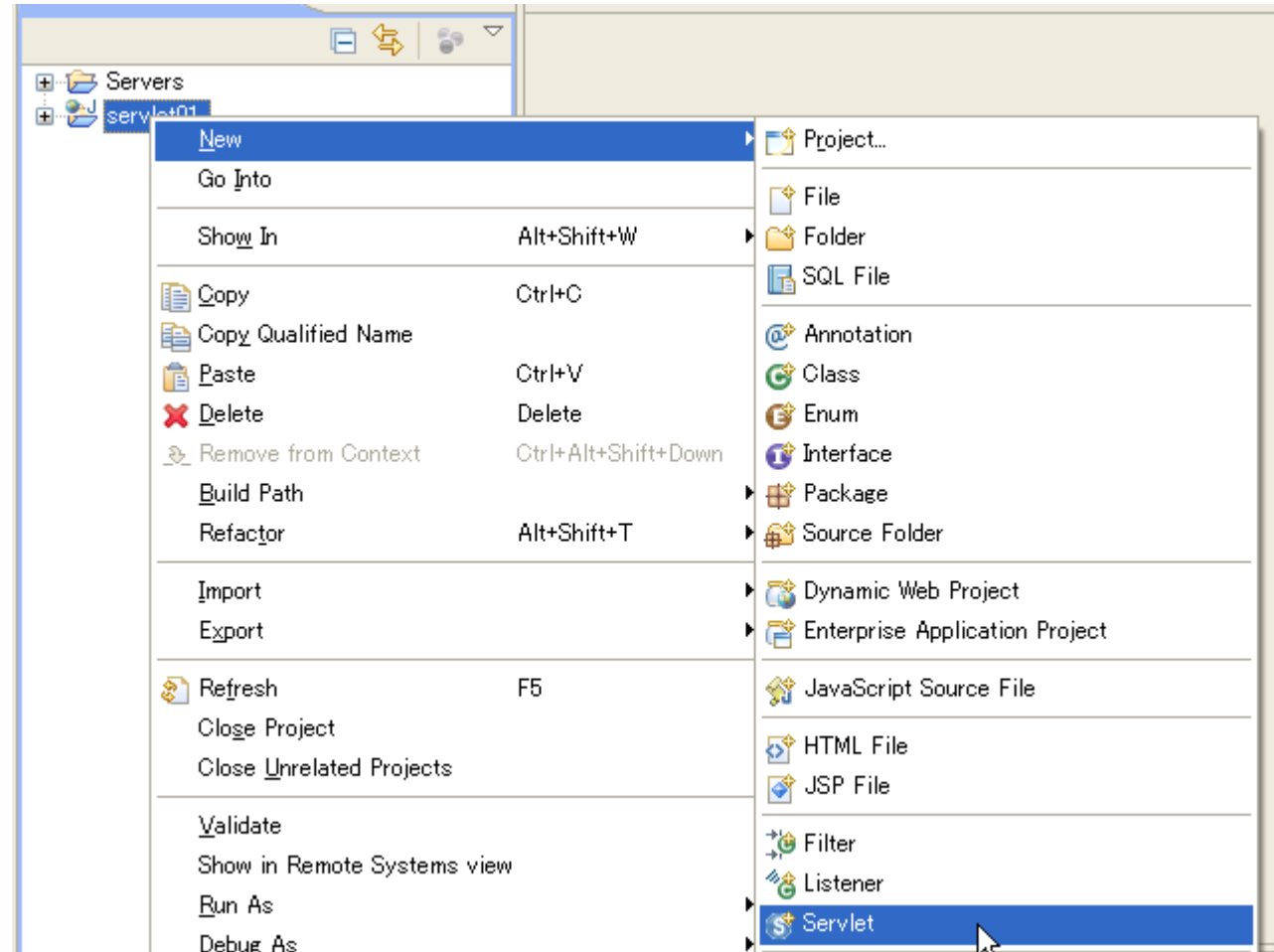
A good starting point for working with Apache Tomcat v7.0 runtime. Additional facets can later be installed to add new functionality to the project.

EAR membership
☐ Add project to an EAR
EAR project name:

Working sets
☐ Add project to working sets
Working sets:

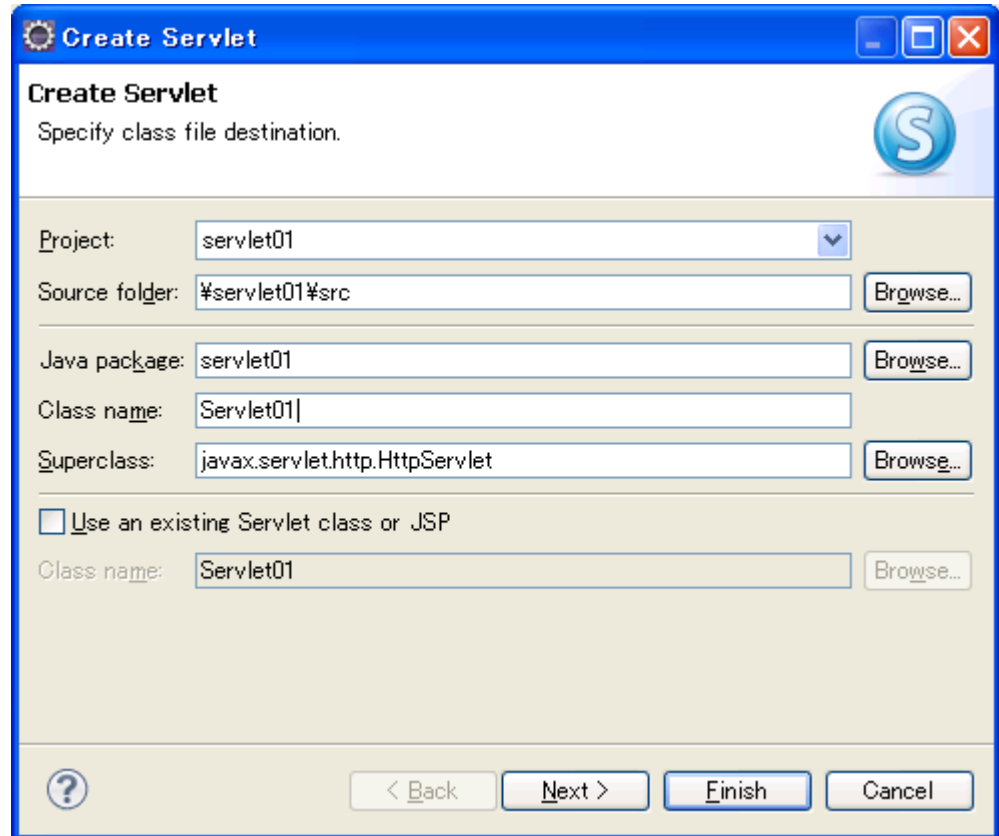
サーブレット

- エンタープライズ・エクスプローラの中に、servlet01というプロジェクトが作成されるので、ここを右クリックし、New->Servletを選びます。



サーブレット

- 以下を入力して終了をクリックします。
- Javaパッケージ:
servlet01
- クラス名:
Servlet01



The image shows a 'Create Servlet' dialog box from an IDE. The title bar says 'Create Servlet'. The main title is 'Create Servlet' and the subtitle is 'Specify class file destination.' There is a blue 'S' icon in the top right. The dialog has several input fields and buttons:

- Project:** A dropdown menu showing 'servlet01'.
- Source folder:** A text field with '#servlet01/src' and a 'Browse...' button.
- Java package:** A text field with 'servlet01' and a 'Browse...' button.
- Class name:** A text field with 'Servlet01'.
- Superclass:** A text field with 'javax.servlet.http.HttpServlet' and a 'Browse...' button.
- ☐ **Use an existing Servlet class or JSP**
- Class name:** A text field with 'Servlet01' and a 'Browse...' button.

At the bottom, there is a question mark icon and four buttons: '< Back', 'Next >', 'Finish', and 'Cancel'.

サーブレット

- 生成されたソースコードには、doGet()というメソッドとdoPost()というメソッドがあることが分かります。

doGet()は、GETメソッドを処理するメソッド。doPost()は、POSTメソッドを処理するメソッドです。

```
/**
 * @see HttpServlet#doGet(HttpServletRequest request, HttpServletResponse response)
 */
protected void doGet(HttpServletRequest request, HttpServletResponse response)
    throws ServletException, IOException
{
    // TODO Auto-generated method stub
}

/**
 * @see HttpServlet#doPost(HttpServletRequest request, HttpServletResponse response)
 */
protected void doPost(HttpServletRequest request, HttpServletResponse response)
    throws ServletException, IOException
{
    // TODO Auto-generated method stub
}
```

サーブレット

- GETメソッドに処理を追加してみましょう。

以下のように入力します。これでブラウザに対して”Hello, World”という文字列を返すことができます。

```
protected void doGet(HttpServletRequest request, HttpServletResponse response)
    throws ServletException, IOException
{
    PrintWriter pw = response.getWriter();
    pw.println("Hello, World");
}
```

PrintWriterでエラーが出たら、電球をクリックしてインポートしてください。

- ブラウザにテキストを返す場合、このようにdoGet()で受けとったresponseオブジェクトのgetWriter()メソッドを呼び出して取り出した、PrintWriterオブジェクトを通して行います。

サブレット

- 実行してみます。

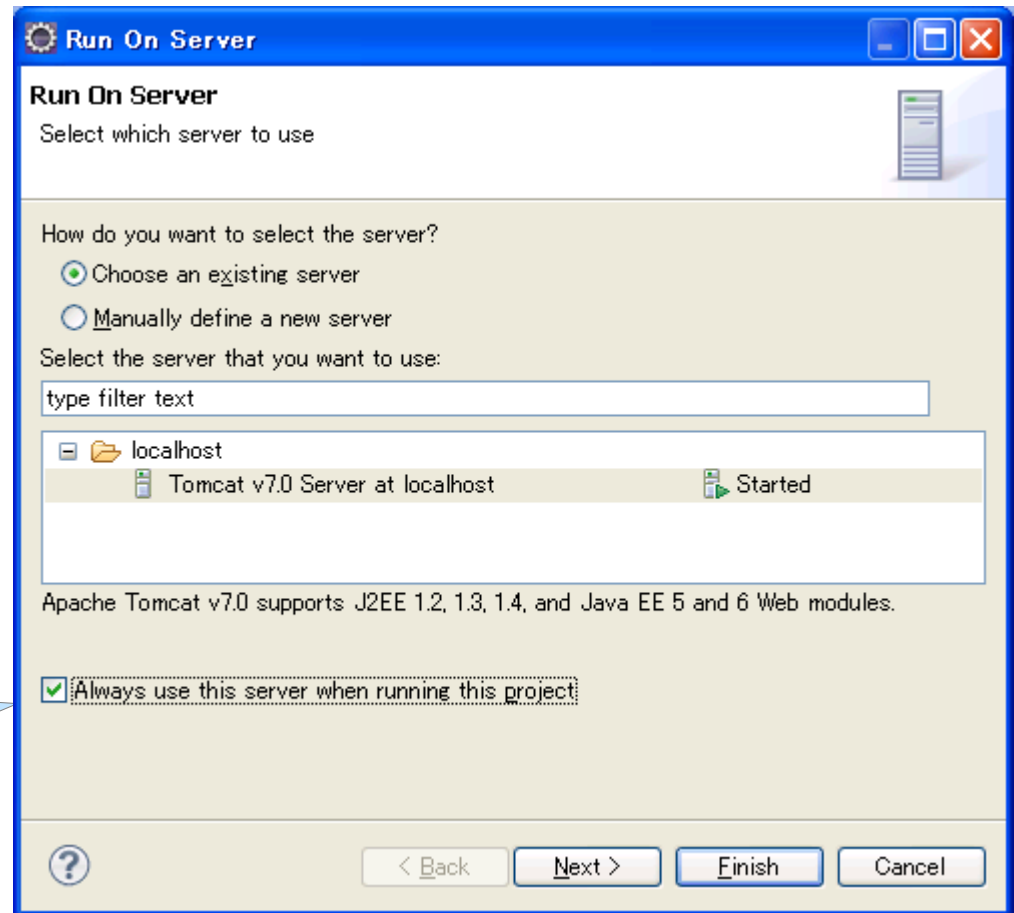
ソース上で、右クリックして、Run As->Run onServerで実行をクリックします。



サーブレット

- 実行するサーバを選びます。
今回は、Tomcatを使用するので、TomcatをクリックしてFinishをクリックします。

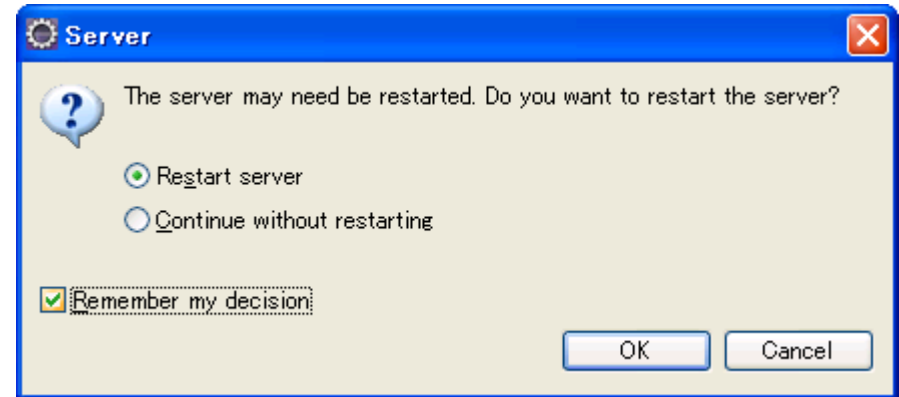
ここをクリックしておくと、次回以降、このステップを省略できます。



サーバレット

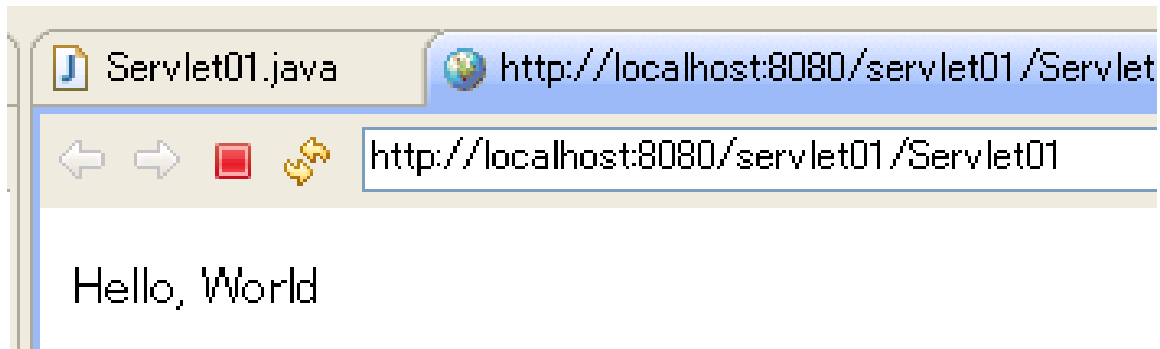
- このダイアログが表示された場合は、OKをクリックします。

ここをクリックしておくと、次回以降、このステップを省略できます。



サーブレット

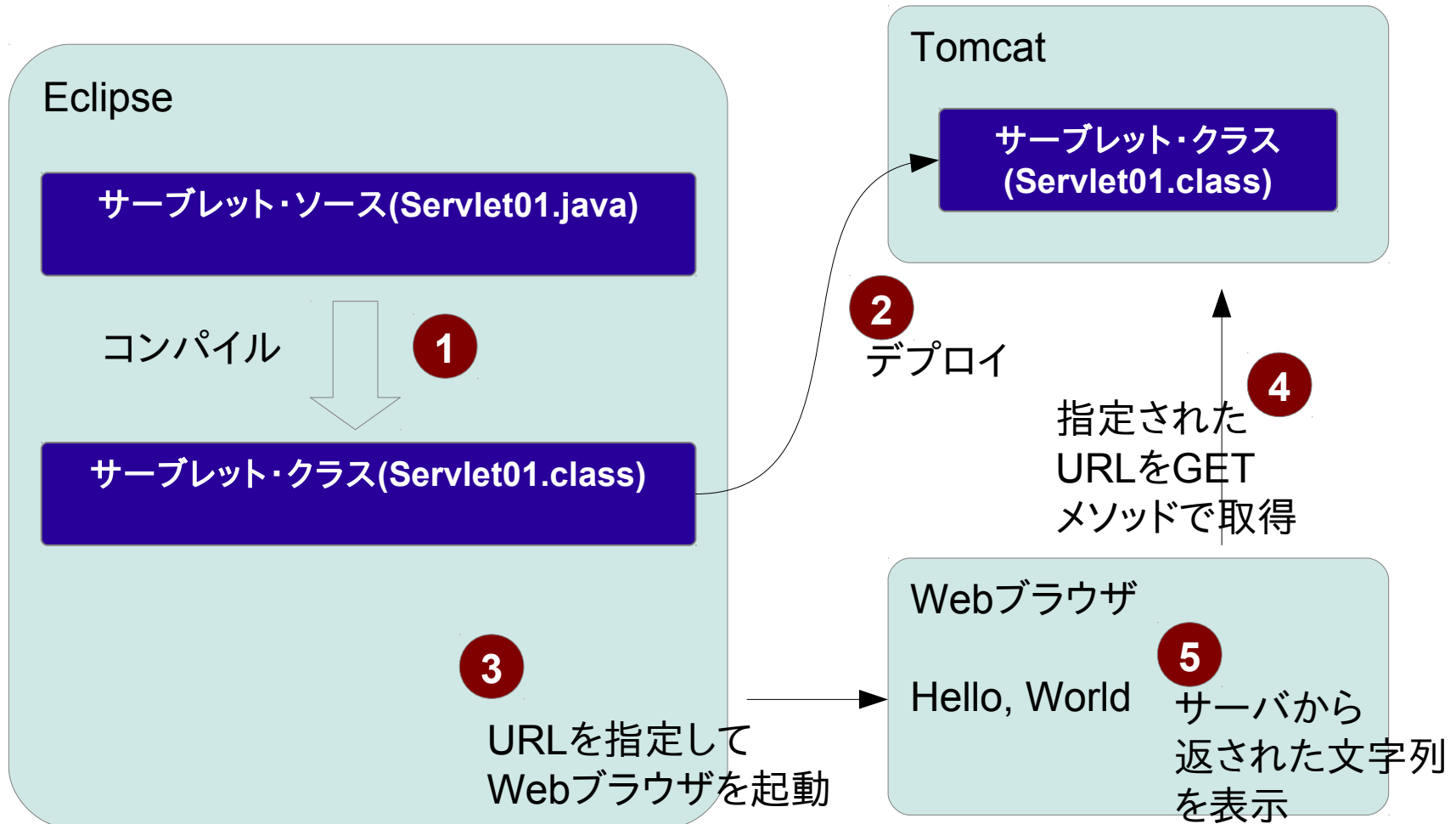
- ブラウザが起動して以下のように表示されます。



Servlet01に指定した、Hello, Worldが表示された。

サーブレット

- これらの手順は、Eclipseで簡単に実行できますが、実際には裏では以下のようなことが行われています。

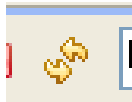


演習:サーブレット

- 同じようにして、Servlet02を作成してください。
- 表示文字列は、Servlet01とは違うものにしてください。
- Servlet02を作成した後も、Servlet01が実行できることを確認してください。

演習: サーブレット

- 同じようにして、もう1つ動的Webプロジェクトを作成します。
 - プロジェクトの名前は、servlet02にします。
 - パッケージ名、クラス名は、これまでと同じservlet01、Servlet01としてください。
 - 表示文字列は、これまでと違うものにしてください。
-
- この段階で次の3つのサーブレットが作成されました。どれも実行可能であることを確認してください。この時、Eclipseから実行するだけでなく、ブラウザの各タブで再読み込みボタン(あるいはF5キー)を押して結果を確認します。これによりデプロイし直すことなく3つのサーブレットが共存して同時に実行できることを確認してください。
 - servlet01プロジェクトのservlet01.Servlet01クラス。
 - servlet01プロジェクトのservlet01.Servlet02クラス。
 - servlet02プロジェクトのservlet01.Servlet01クラス。



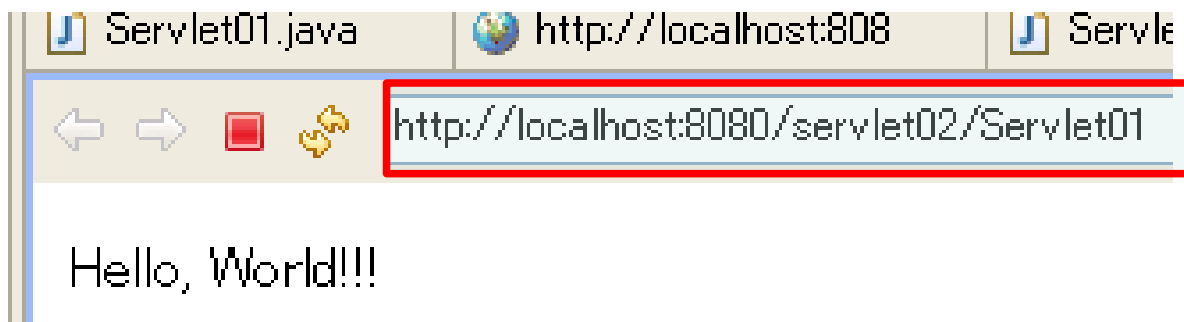
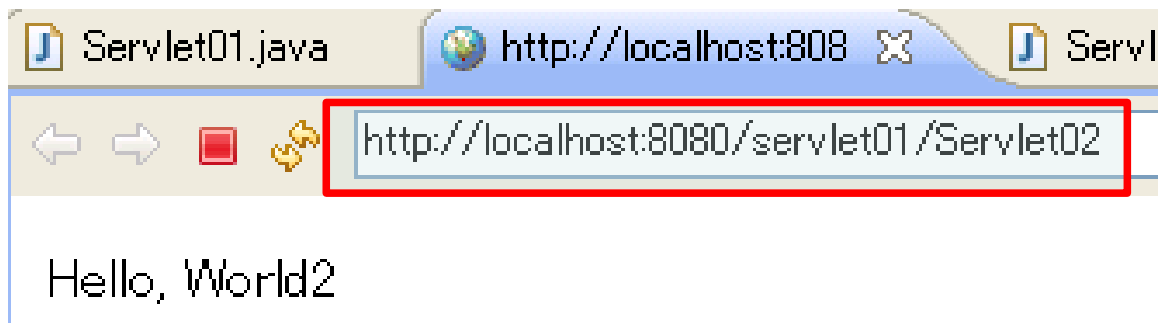
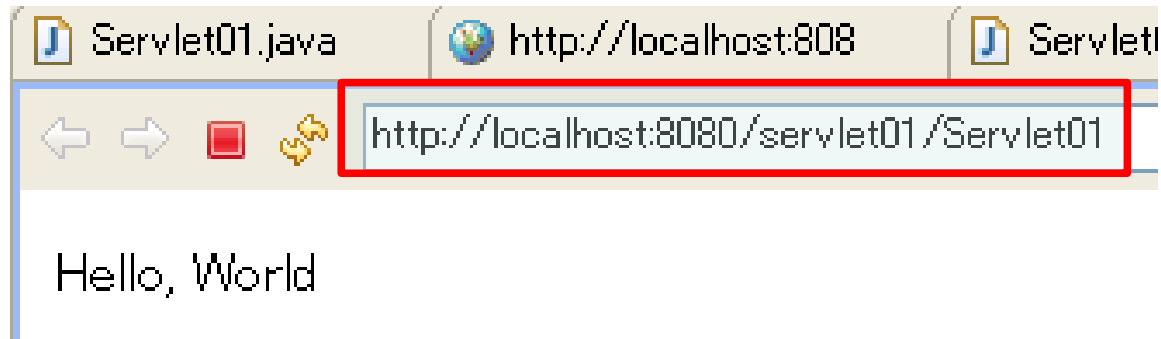
再読み込みボタン。

アプリケーションのデプロイとコンテキストパス

- 演習の結果を振り返ってみてください。
 - Servlet01とServlet02とで異なる表示がされていますが、ブラウザはどうやってこれらの2つを見分けているのでしょうか？
 - servlet01プロジェクトと、servlet02プロジェクトとで全く同一の名前のクラス (servlet01.Servlet01) が1つのTomcatサーバに同居しており、これらが違う結果を返しています。なぜこんなことが可能なのでしょうか？

ヒントは、ブラウザに指定されているURLにあります。

アプリケーションのデプロイとコンテキストパス



アプリケーションのデプロイとコンテキストパス

- 名前は、大きく以下の3つの部分に分けられます。

`http://localhost:8080/servlet01/Servlet01`

Tomcatのサーバごとに割り振られる名前。

Eclipseのプロジェクトごとに割り振られる名前。

サーブレットごとに割り振られる名前。

アプリケーションのデプロイとコンテキストパス

- localhost:8080の部分は、Tomcatサーバの場所を表しています。
- localhostというのは自分自身を表しています。今回はTomcatを同じPCの中で動かしているため、localhostを指定します。
- 8080というのはポート番号です。TCP/IPではポート番号を使うことで通信先を区別できます。
 - - 例えば、同じPCの上で2つのTomcatサーバを動かした場合、ポート番号が無いとどちらもlocalhostという同じ名前になってしまって区別できません。
 - この場合、ポート番号を変えることで(例えば1つ目は、8080、もう1つは8081にする)、2つのサーバを区別することができるようになります。

アプリケーションのデプロイとコンテキストパス

- servlet01、servlet02の 部分はコンテキストパスと呼ばれます。
- JavaEEサーバは、このコンテキストパスによってアプリケーションを分けます。このアプリケーションの単位を「Webモジュール」と呼びます。
- これによって全く別のところで入手した2つのアプリケーションを、1つのJavaEEサーバ上に同居させることができます。演習で見たように全く同一のパッケージ、クラス名のクラスが複数のアプリケーションに存在していても衝突することなく同居できます。
- コンテキストパスはデプロイの時に指定します(今回は、Eclipseのウィザードが自動設定しました)。変更するには、サーバビューで、Tomcatをダブルクリックし、「Modules」タブを選択し、編集したいWebモジュールを選択してからEditボタンをクリックします。
- アプリケーションサーバは、Javaのクラスローダを用いてコンテキストごとにアプリケーションを分離します。これによって全く同一のクラス名を持った異なるクラスが、1つのJVM上で動作することが可能になっています(内容が高度になるので詳細は割愛しますが、興味のある方は <http://tomcat.apache.org/tomcat-7.0-doc/class-loader-howto.html> を見てみてください)。

アプリケーションのデプロイとコンテキストパス

- 最後のServlet01, Servlet02の部分はソースコードか設定ファイルで指定します。今回は以下のようにサーブレットクラスのソースコードの先頭で指定されています。
- この指定のことを「サーブレット・マッピング」と呼びます。

```
/**
 * Servlet implementation class Servlet01
 */
@WebServlet("/Servlet01")
public class Servlet01 extends HttpServlet {
    private static final long serialVersionUID = 1L
```

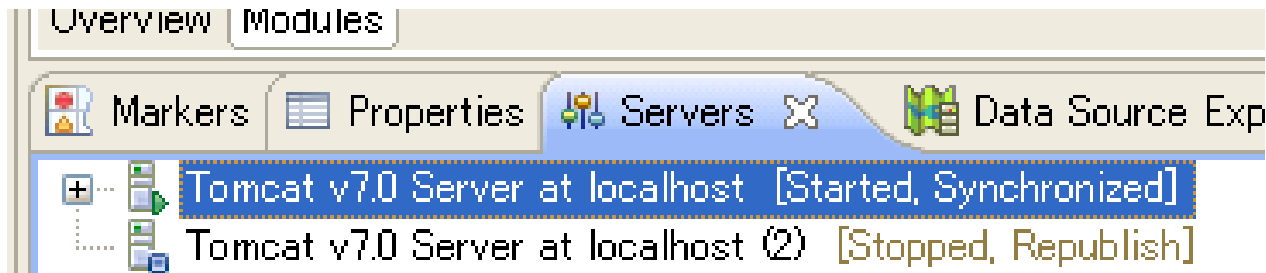
設定ファイルで指定する方法は複雑なので基礎編では省略します。

興味のある方

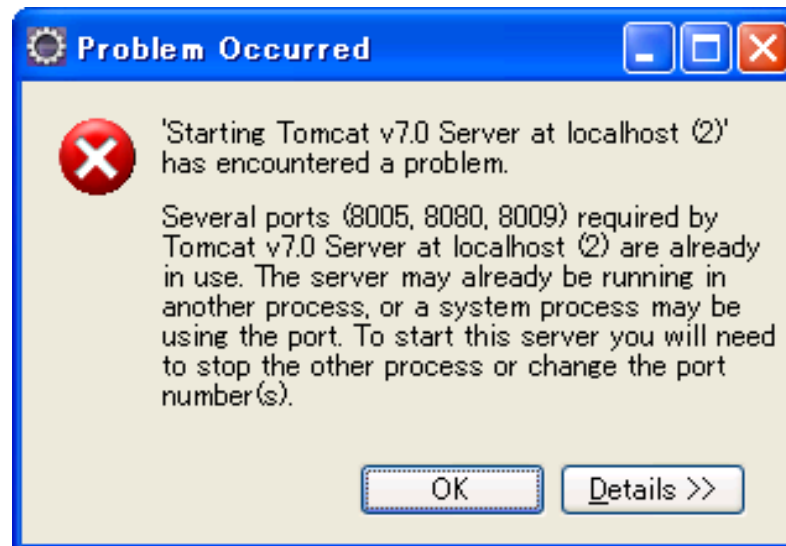
は、<http://jcp.org/aboutJava/communityprocess/final/jsr315/index.html> を参照してください。

演習: 複数のサーバ

- これまでと同じ手順で、もう1つサーバを作成してください。サーバビューで右クリックして New->Serverをクリックしてデフォルトのままで作成すれば作成できます。



- これまでの手順で、新しく作成したサーバを起動して、以下のエラーが表示されることを確認してください。






演習: 複数のサーバ

- ひとつのPC上で複数のサーバを起動する場合は、ポート番号を分けなければなりません。
今回作成したサーバをダブルクリックします。
- ポートのところを以下のように、元の値+1に設定し、保存します。



▼ Ports

Modify the server ports.

Port Name	Port Number
 Tomcat admin port	8006
 HTTP/1.1	8081
 AJP/1.3	8010

- 起動が成功することを確認してください。

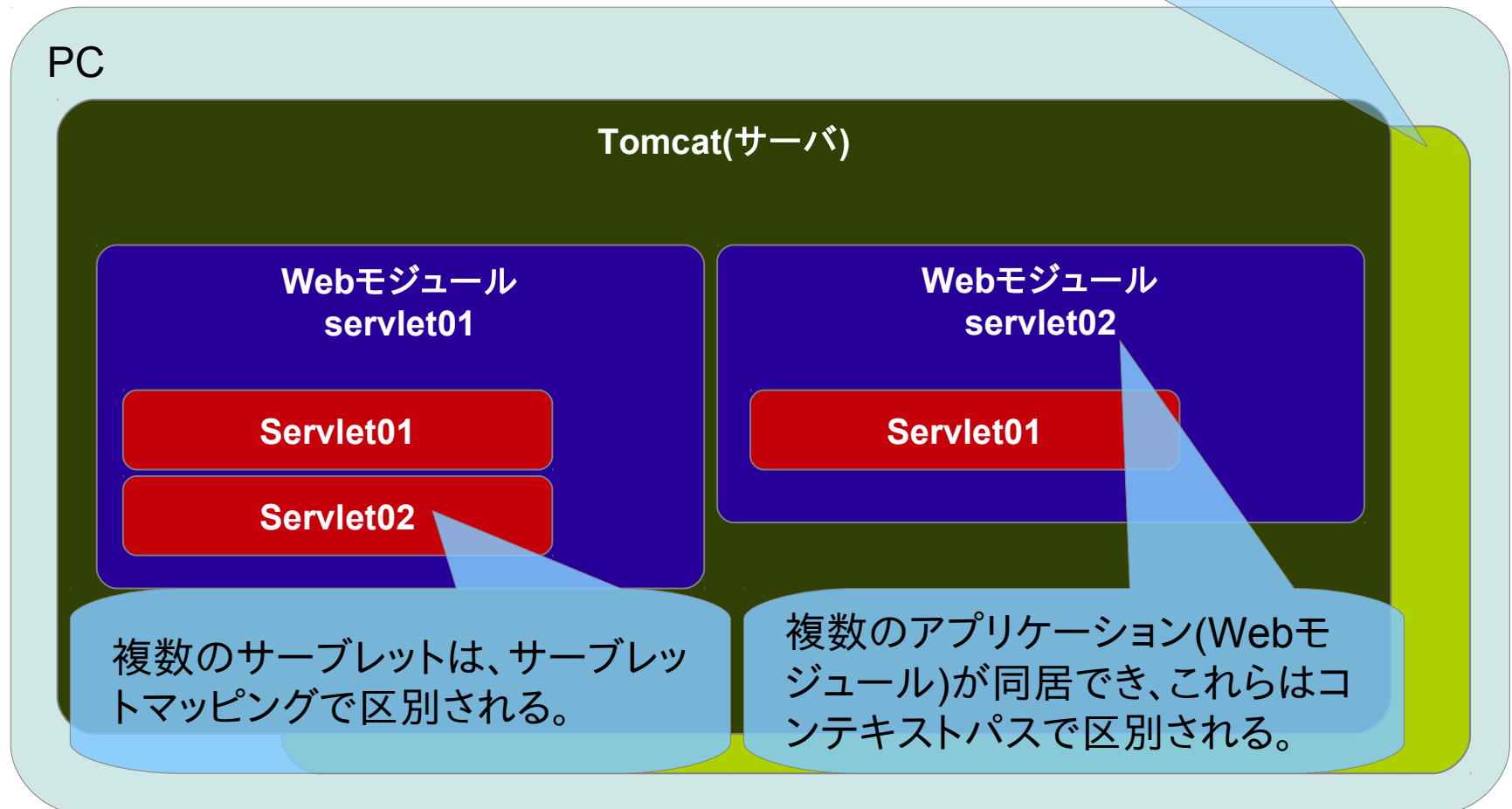
演習: 複数のサーバ

- 以下の手順で、新しいサーバにアプリケーションをデプロイします。
- 今回作成したサーバをダブルクリックします。
- Modulesタブをクリックします。
- Add Web Module...ボタンを押して、servlet01とservlet02を追加します。これで新しいサーバにアプリケーションがデプロイされます。
- 保存します。
- 今回作成したサーバを右クリックして、Restartを選びます。
- ブラウザで結果を確認してください。localhost:8080と、localhost:8081のどちらでも結果が表示されることを確認してください。

アプリケーションのデプロイとコンテキストパス

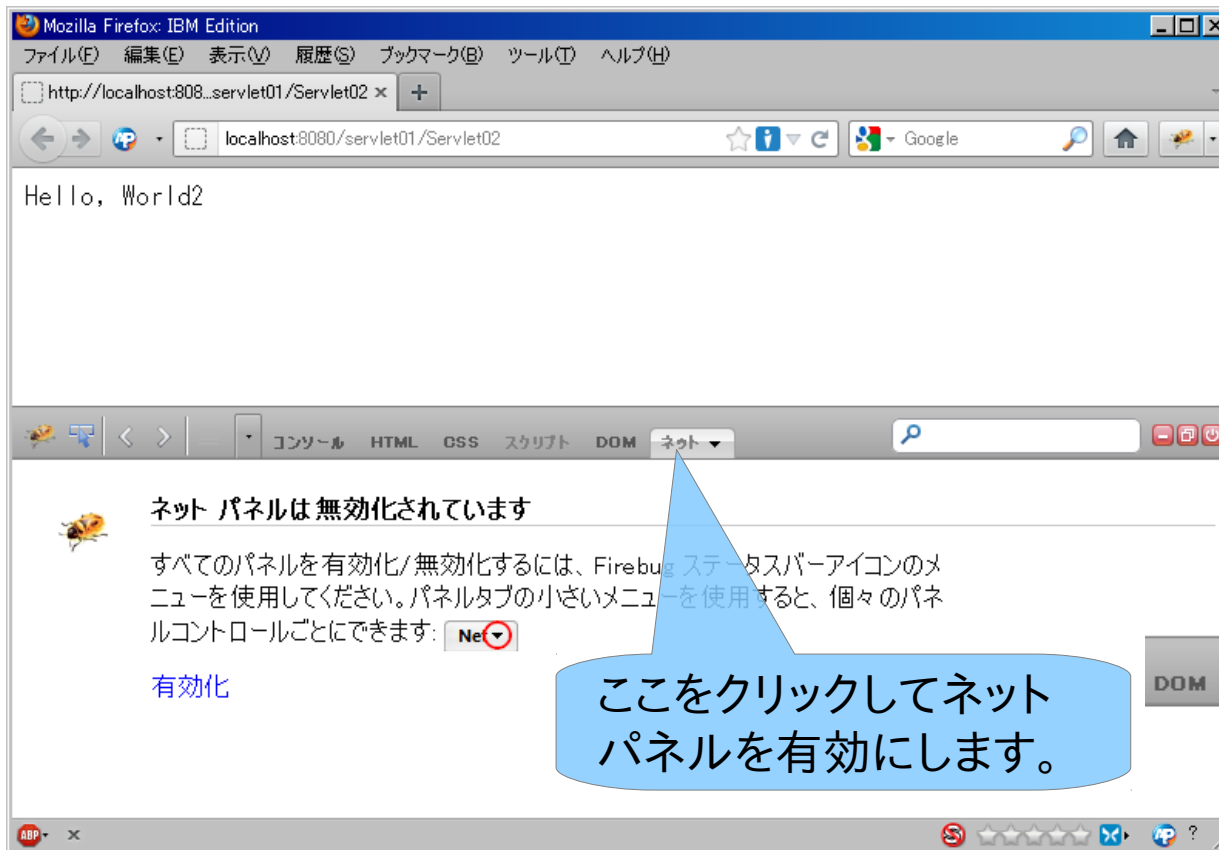
- もう一度おさらいしましょう。

サーバが同一PC上に複数ある場合は、ポート番号で分ける。



httpの通信内容を確認する

- httpの通信内容を確認しましょう。
- Firebugというデバッガを使用します。ブラウザの右上の虫のアイコンをクリックして起動します(ホタルです。ゴキブリじゃありません!)



httpの通信内容を確認する

- Firebugを起動したら、F5キーを押して再度リクエストを行います。

この+をクリックして展開します。

ここをクリックして、ソースを表示します。

ここをクリックして、ソースを表示します。

URL	ステータス	ドメイン	サイズ	リモート IP	タイムライン
GET Servlet0	200 OK	localhost:8080	15 B	127.0.0.1:8080	

ヘッダ レスポンス キャッシュ

レスポンスヘッダ [ソースを表示](#)

Content-Length 15
Date Tue, 21 Feb 2012 00:32:26 GMT
Server Apache-Coyote/1.1

リクエストヘッダ [ソースを表示](#)

Accept text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Encoding gzip, deflate
Accept-Language ja,en-us;q=0.7,en;q=0.3
Cache-Control max-age=0
Connection keep-alive
Host localhost:8080
User-Agent Mozilla/5.0 (Windows NT 5.1; rv:10.0.1) Gecko/20100101 Firefox/10.

1 件のリクエスト 15 B 16ms (onload: 144ms)

httpの通信内容を確認する

- この図は、localhost:8080のservlet01/Server02を調べた例です。

The screenshot shows the 'Network' tab in a web browser's developer tools. The selected item is a GET request to `/servlet01/Servlet02` on `localhost:8080`. The response status is `200 OK`. The response headers are visible, including `Server: Apache-Coyote/1.1` and `Content-Length: 15`. The request headers are also visible, including `User-Agent: Mozilla/5.0 (Windows NT 5.1; rv:10.0.1) Gecko/20100101 Firefox/10.0.1` and `Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8`.

Callouts from the image:

- ステータスコード200が返っていることが分かります。
- GETリクエストが送られていることが分かります。
- レスポンスタブをクリックして、レスポンスの内容も確認してください。

http通信では、実際にはこのようなテキストをWebブラウザとサーバとでやりとりしています。

HTMLの表示

- 今回は、Hello, Worldというテキストを表示しましたが、通常はHTMLを表示します。
- どこでも良いのでWebサイトを開いて、ブラウザを右クリックし「ページのソースを表示」を選んでください。ここで表示されるのがHTMLです。
- 実際のWebサイトで使用されているHTMLは非常に複雑ですが、今回は以下のような簡単な画面を表示させてみましょう。



タイトルバーに、Hello World Applicationと表示。

大きな文字でHello, Worldと表示。

HTMLの表示

- 前ページの画面を表示するには、ブラウザに以下のようなhtmlを返すことで実現できます。

```
<html>  
  <head>  
    <title>Hello World Application</title>  
  </head>  
  <body>  
    <h1>Hello, World</h1>  
  </body>  
</html>
```

文書全体をhtmlタグで囲みます。

ヘッダ部。タイトルを指定しています。

ボディ部。実際のブラウザの画面に表示されます。

ソースを変更して保管すると、Tomcatに自動的にデプロイされます。この時コンソールビューに「**INFO: このコンテキストの再ロードが完了しました**」と表示されます。

演習: HTMLの表示

- 前ページのHTMLを返すように、Servlet01を修正してください。
- 修正したら結果を確認してください。

演習: HTMLの表示(解答例)

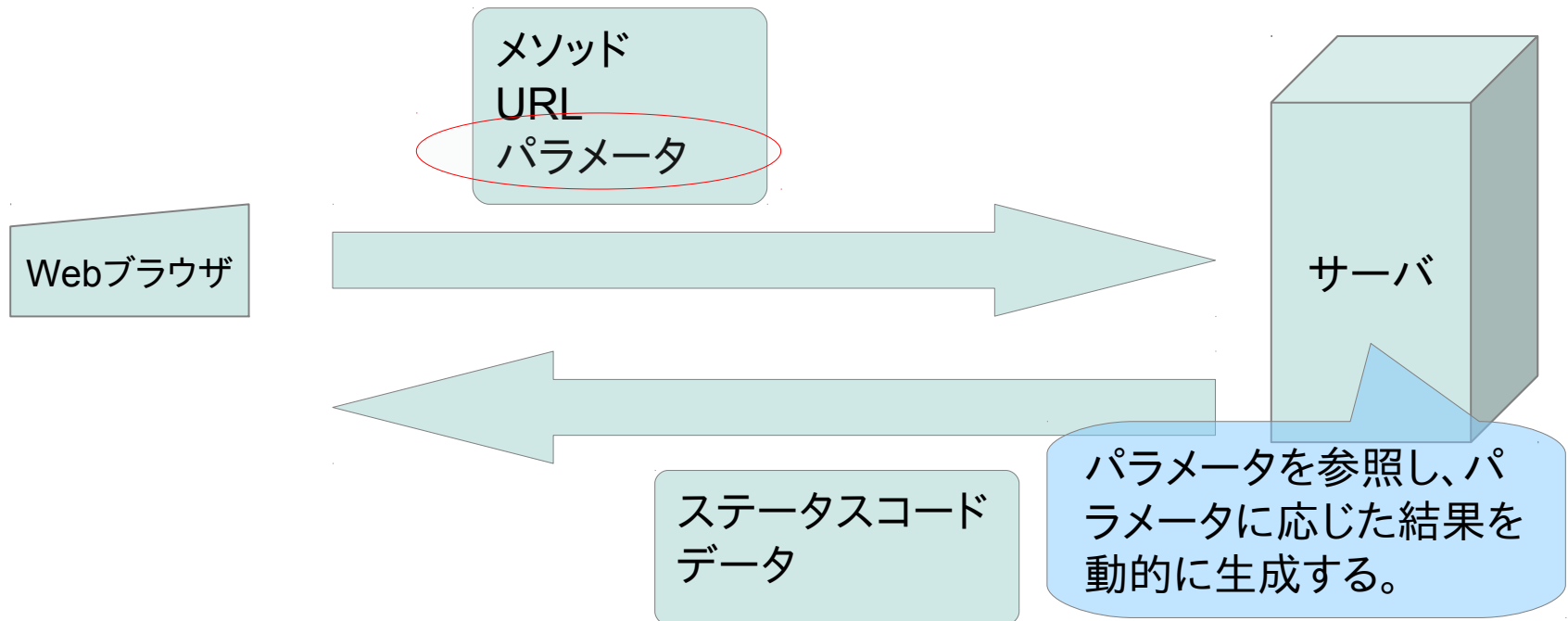
```
protected void doGet(HttpServletRequest request, HttpServletResponse r
    throws ServletException, IOException
{
    PrintWriter pw = response.getWriter();
    pw.println("<html>");
    pw.println("    <head>");
    pw.println("        <title>Hello World Application</title>");
    pw.println("    </head>");
    pw.println("    <body>");
    pw.println("        <h1>Hello, World</h1>");
    pw.println("    </body>");
    pw.println("</html>");
}
```

HTML

- 省略...

リクエストパラメータ

- 本研修の冒頭で、Webアプリケーションは「Webブラウザをクライアントとして利用し、サーバ側で処理を行うことで、動的に結果を得る」ものであると解説しました。
- これまでのサーブレットは静的な結果しか返していません。動的な結果を生成する1つの方法はブラウザから与えられたパラメータに応じた結果を返すことです。
- このためにはリクエストパラメータを使用します。



リクエストパラメータ

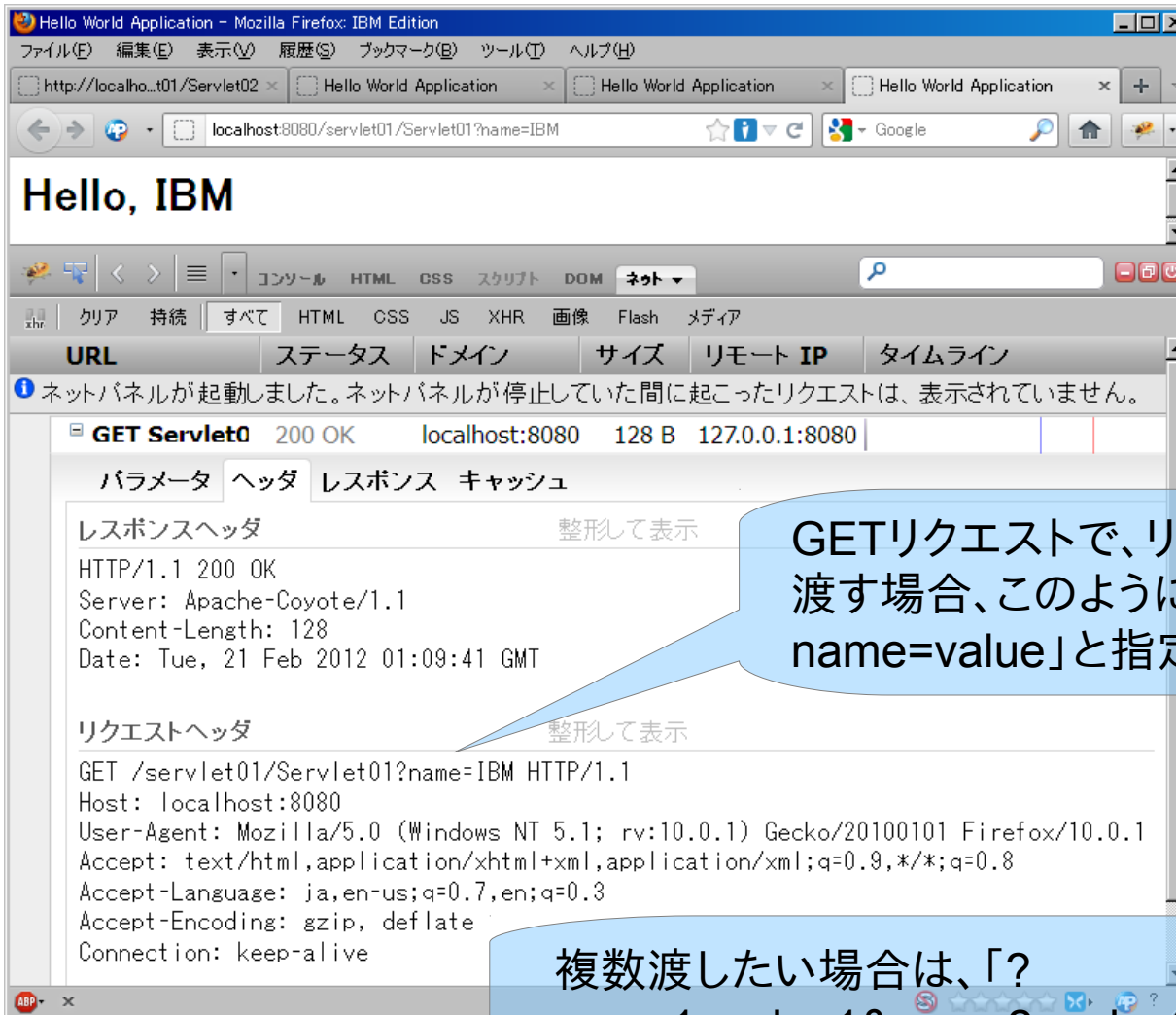
- Servlet01の内容を以下のように書き換えます。

```
//  
protected void doGet(HttpServletRequest request, HttpServletResponse  
    throws ServletException, IOException  
{  
    PrintWriter pw = response.getWriter();  
    pw.println("<html>");  
    pw.println("    <head>");  
    pw.println("        <title>Hello World Application</title>");  
    pw.println("    </head>");  
    pw.println("    <body>");  
  
    String name = request.getParameter("name");  
    if (name != null) {  
        pw.println("        <h1>Hello, " + name + "</h1>");  
    }  
    pw.println("    </body>");  
    pw.println("</html>");  
}
```

リクエストパラメータを取り出すには、requestオブジェクトの、getParameter()を使用します。

リクエストパラメータ

- ブラウザで、URLの最後に、「?name=IBM」を付け足してください。



GETリクエストで、リクエストパラメータを渡す場合、このようにURLの最後に、「?name=value」と指定します。

複数渡したい場合は、「?name1=value1&name2=value2...」と指定します。

formタグ

- URLの最後にリクエストパラメータを指定する方法は分かりにくく、面倒で、間違えやすいものです。一般にはformタグを使用します。

```
PrintWriter pw = response.getWriter();  
pw.println("<html>");  
pw.println("  <head>");  
pw.println("    <title>Hello World Application</title>");  
pw.println("  </head>");  
pw.println("  <body>");
```

```
String name = request.getParameter("name");  
if (name != null) {  
    pw.println("    <h1>Hello, " + name + "</h1>");  
}
```

```
else {  
    pw.println("      <form action='" + request.getRequestURI() + "'>");  
    pw.println("        <label>Input Name: <input type='text' name='name'></input></label>");  
    pw.println("        <input type='submit'></input>");  
    pw.println("      </form>");  
}  
pw.println("    </body>");  
pw.println("</html>");
```

リクエスト先をaction属性に指定します。requestオブジェクトの、getRequestURI()を呼ぶと、自分自身のURLを取得できます。

ラベルタグで囲んで、ラベルを指定します。

送信ボタンは、inputタグ(type='submit')を使用します。

文字列の入力域は、inputタグ(type='text')を使用します。name属性で、リクエストパラメータの名前を指定します。

実際に試してみてください。

formタグ

- ブラウザで表示すると、以下のようにフォームが表示されます。



inputタグ(type='submit')は、このように送信ボタンとして表示されます。

inputタグ(type='text')は、このようにテキスト入力域として表示されます。

ラベルタグで指定した内容が表示されます。ラベルタグと一緒に囲んだ入力域と関連付けられています。このラベル部分をクリックすると、入力域にフォーカスが移動することを確認してください。

formタグ

Hello World Application - Mozilla Firefox: IBM Edition

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)

localhost:8080/servlet01/Servlet01?name=IBM

Hello, IBM

コンソール HTML CSS スクリプト DOM ネット

URL ステータス ドメイン サイズ リモート IP タイムライン

URL	ステータス	ドメイン	サイズ	リモート IP	タイムライン
GET Servlet01	200 OK	localhost:8080	128 B	127.0.0.1:8080	

パラメータ ヘッダ レスポンス キャッシュ

レスポンスヘッダ 整形して表示

```
HTTP/1.1 200 OK
Server: Apache-Coyote/1.1
Content-Length: 128
Date: Tue, 21 Feb 2012 01:41:37 GMT
```

リクエストヘッダ 整形して表示

```
GET /servlet01/Servlet01?name=IBM HTTP/1.1
Host: localhost:8080
User-Agent: Mozilla/5.0 (Windows NT 5.1; rv:10.0.1) Gecko/20100101 Fire
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: ja,en-us;q=0.7,en;q=0.3
Accept-Encoding: gzip, deflate
Connection: keep-alive
Referer: http://localhost:8080/servlet01/Servlet01
```

1 件のリクエスト 128 B 0 (online)

リクエストURLにもパラメータが埋め込まれていることが分かります。

入力域にIBMと入れて送信ボタンを押した例です。
直接?name=IBMと指定した場合と同じ結果が得られることを確認してください。

演習:POSTリクエスト

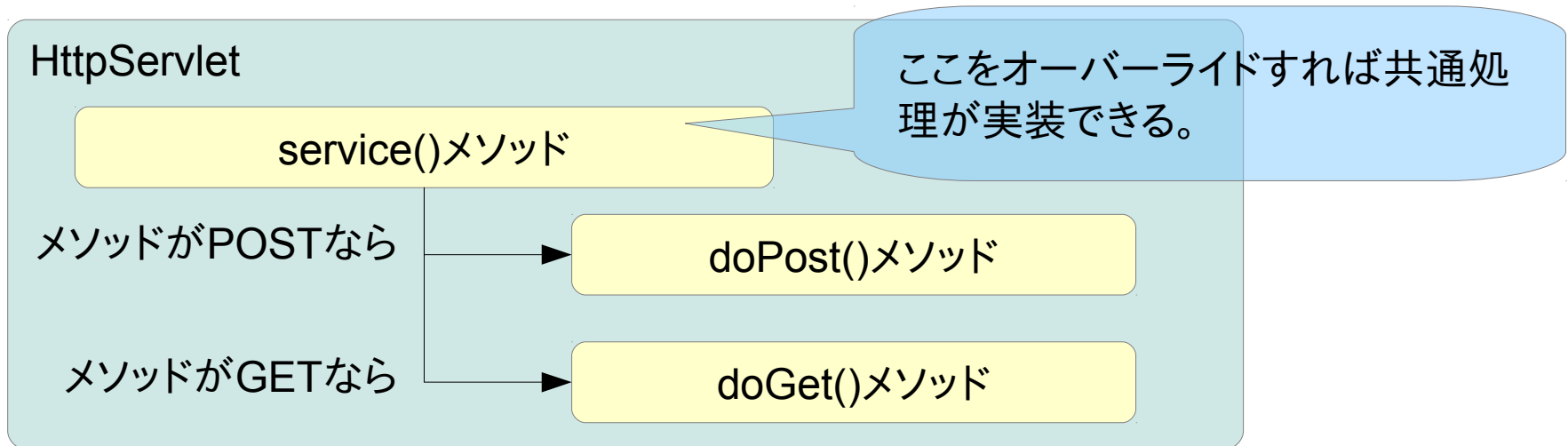
- httpリクエストとして、これまではGETリクエストを使用してきました。
- POSTリクエストも試してみましょう。
 - まずformタグに、method='post'を追加します。指定が無い場合は、method='get'がデフォルト値になります。

```
se {  
    pw.println("<form method='post' action='\" +  
    pw.println("    <label>Input Name: <input type  
    pw.println("    <input type='submit'></input>\"  
    pw.println("</form>");
```

- POSTリクエストの処理は、doPost()に記述します。
 - 処理内容はdoGet()と全く同一です。どのように修正すべきか考えてください。
- 出来上がったら、Firebugでリクエスト内容を確認してください。
- 送信した後にブラウザのURLを確認してください。GETリクエストの時と比べてどうですか？

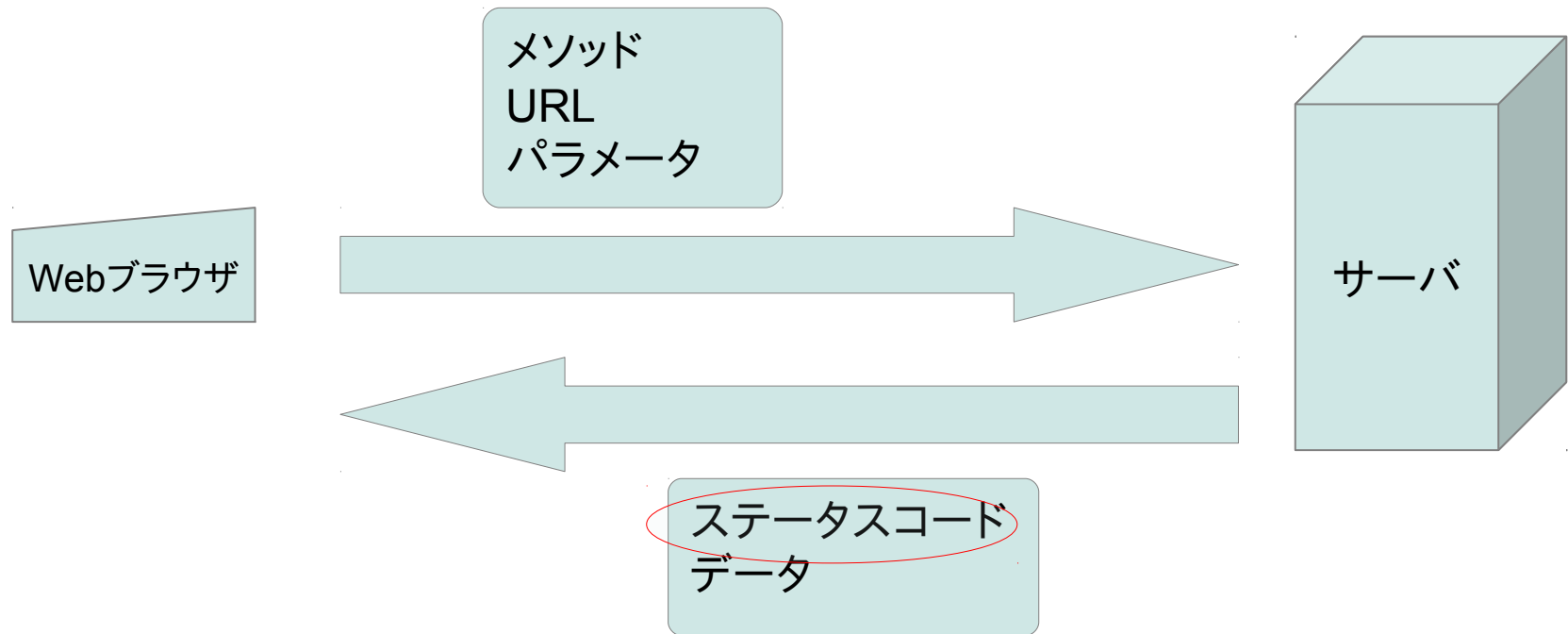
service()メソッド

- 演習の例のように、GETリクエストとPOSTリクエストで同じ処理が必要になることは良くあります。
- こういう時には、処理をメソッドにまとめておいて、そのメソッドをdoGet()とdoPost()から呼ぶようにすることも可能ですが、service()メソッドを使うことで1つにまとめることも可能です。



service()メソッドに処理を移動して、doGet()とdoPost()を削除して結果を確認してください (Eclipseでソースを右クリックして、Source->Override/Implement Method...を使います)。引数型が、HttpServletRequest, HttpServletResponseのservice()メソッドをオーバーライドします。

ステータスコード



- httpでは、サーバでの処理結果をステータスコードを用いて通知します。
- これまでの演習の例で、Firebugを使ってステータスコードを確認すると、200が返っていることが分かります。特に何もしないとデフォルトで200が返るようになっています。
- 他のコードを返したい場合には、`sendError()`メソッドを使用します。

演習:ステータスコード

- doGet()を以下のように変更して、ステータスコードを設定してください。

```
protected void doGet(HttpServletRequest request, HttpServletResponse response) throws  
    response.setCharacterEncoding("UTF-8");  
    response.sendError(HttpServletResponse.SC_SERVICE_UNAVAILABLE,  
        "混雑しています。しばらく待ってから再実行をお願いします。");  
  
    // PrintWriter pw = response.getWriter();  
    // pw.println("<html>");  
    // pw.println("<head>");
```

漢字が入る場合は、エンコーディングを指定しないと文字化けします(この行をコメントアウトして試してみてください)。

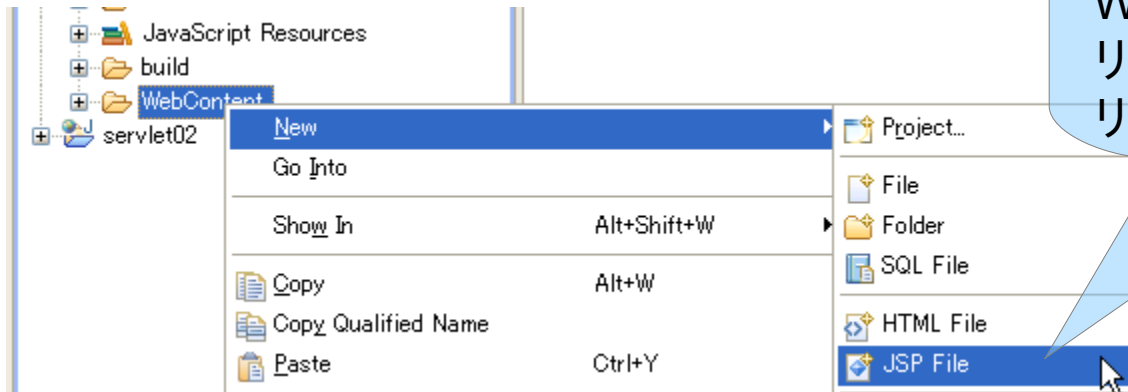
sendError()を使ってステータスコードと説明を指定します。定数 SC_SERVICE_UNAVAILABLEは、503で一時的にサービスを提供できないことを意味します。

- ブラウザで表示される画面を確認してください。
- Firebugでステータスコードを確認してください。

JSP

- 前章の演習で、HTMLを生成するサーブレットを見ました。Javaのコード内で、このようにprint文を使ってHTMLを生成するのは煩雑であることが分かります。
- JSP(JavaServer Pages)は、これを解決するための仕組みです。
- 早速JSPを作成してみましょう。

JSP



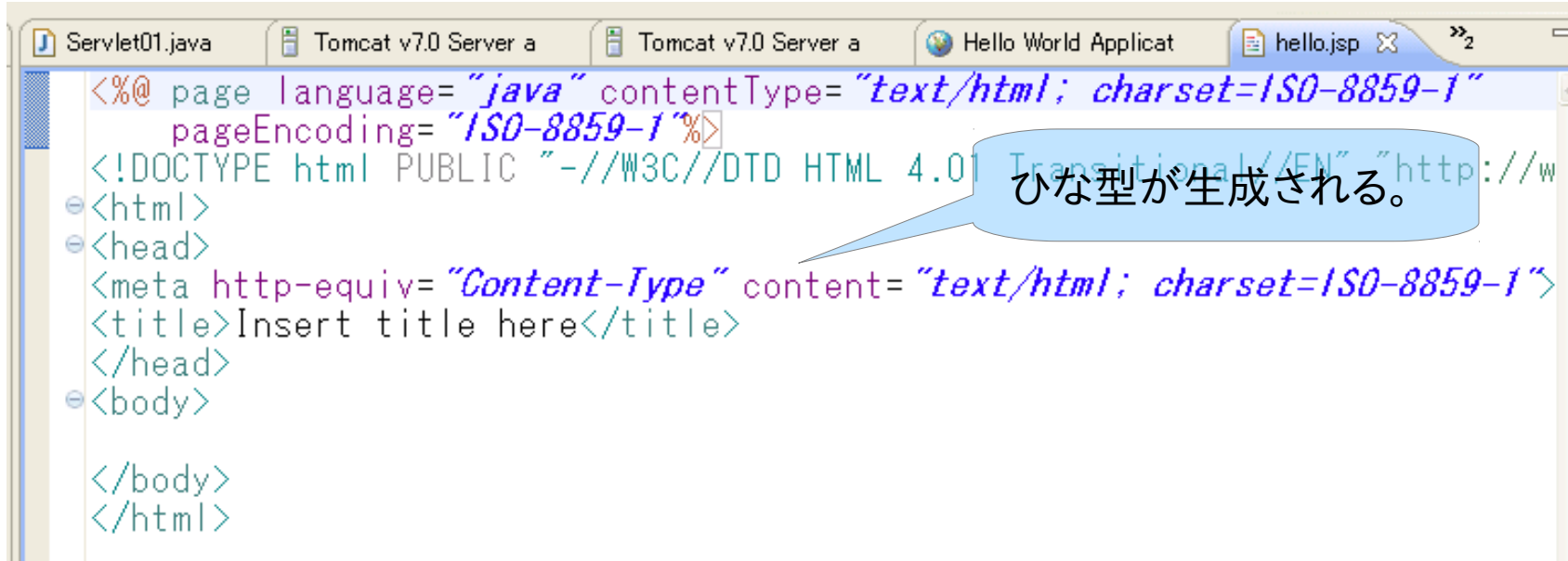
Package Explorer
で、servlet01プロジェクトの、
WebContentフォルダを右ク
リックし、New->JSP Fileをク
リックします。

JSP



名前に、hello.jspと
入れて終了。

JSP



```
<%@ page language="java" contentType="text/html; charset=ISO-8859-1"
    pageEncoding="ISO-8859-1"%>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN" "http://w
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
<title>Insert title here</title>
</head>
<body>

</body>
</html>
```

ひな型が生成される。

JSP

- 以下は、これまでと同じプログラムを、JSPで書き直したものです。

```
<body>
```

```
<%
```

```
String name = request.getParameter("name");
```

```
if (name != null) {
```

```
%>
```

```
<h1>Hello, <%= name %></h1>
```

```
<%
```

```
} else {
```

```
%>
```

```
<form action='<%= request.getRequestURI() %>'>
```

```
<label>Input Name: <input type='text' name='name'></input></label>
```

```
<input type='submit'></input>
```

```
</form>
```

```
<%
```

```
}
```

```
%>
```

```
</body>
```

JSPでは基本的に、HTMLをそのまま記述します。

<% %>で囲まれたところは、Javaのコードと認識されます。

<%= %>で囲まれたところは、Javaの式として評価されて結果が埋め込まれます。

Javaコードが少ない部分は簡潔に書ける。

Javaコードが多い部分は煩雑になる。

JSP

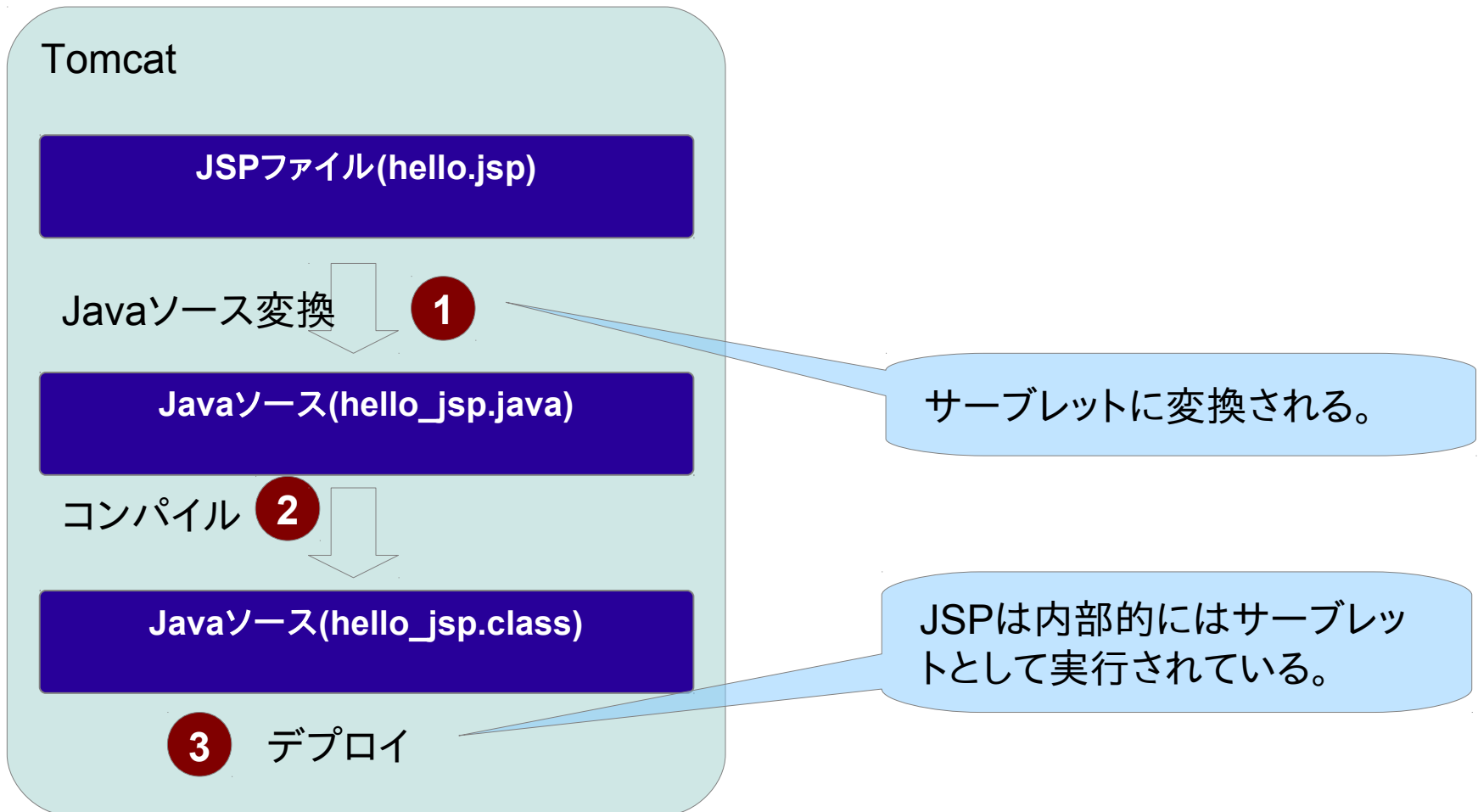
- ブラウザで、`http://localhost:8080/servlet01/hello.jsp` を開いて結果を確認してください。

- サーブレット:
 - Javaのコードの中に、HTMLを埋め込む。
 - HTML部分が多い場合には向いていない。

- JSP:
 - HTMLの中に、Javaのコードを埋め込む。Javaコード部分を「スクリプトレット」と呼ぶ。
 - ロジック部分が多い場合は向いていない。

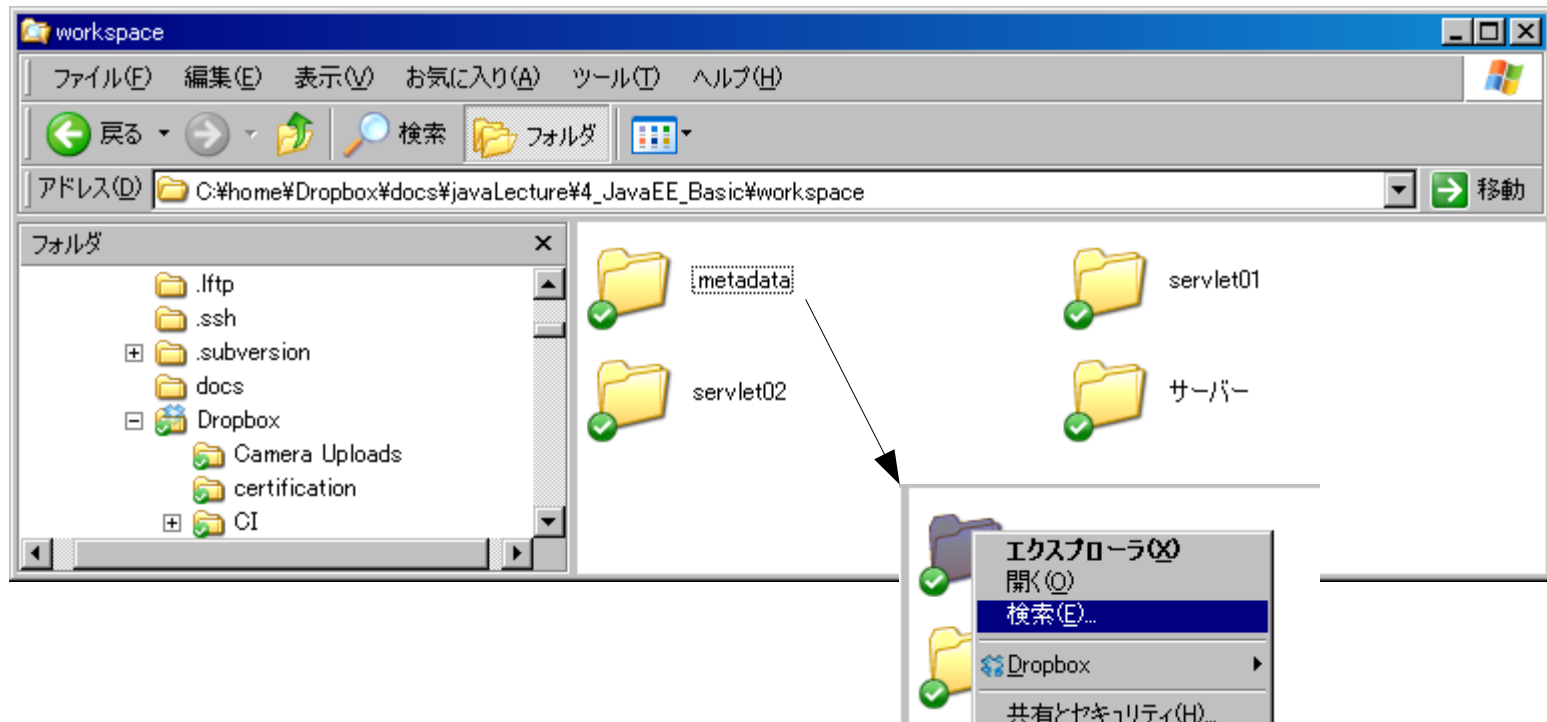
JSP

- JSPは以下のように実行されます。



JSP

- サブレットに変換後のJSPを見てみましょう。
- ワークスペースのディレクトリに、.metadataというディレクトリがあります。
右クリックして検索をクリックします。



JSP

- hello*を「ファイル名のすべてまたは一部」に入力して検索をクリックします。
- すると以下のようにサーブレット変換されたJavaソースコードが見つかります。右クリックして、プログラムから開く=>Notepadで中身を確認できます。

下の条件のいくつかまたはすべてで検索してください。

ファイル名のすべてまたは一部

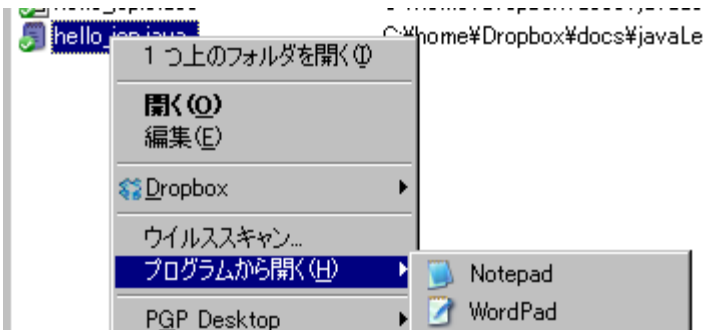
(Q):

hello*

ファイルに含まれる単語または句

(W):

名前	フォルダ名	サイズ	種類	頁
hello.jsp	C:\home\Dropbox\docs\javaLe...	1 KB	JSP ファイル	2
hello.jsp.class	C:\home\Dropbox\docs\javaLe...	5 KB	CLASS ファイル	2
hello.jsp.java	C:\home\Dropbox\docs\javaLe...	4 KB	JAVA ファイル	2



JSP

JSPから変換されたサーブレットソースコード

...

```
    out.write("<title>Hello World  
Application</title>\r\n");  
    out.write("</head>\r\n");  
    out.write("<body>\r\n");  
    out.write("\r\n");
```

```
String name = request.getParameter("name");  
if (name != null) {
```

```
    out.write("\r\n");  
    out.write(" <h1>Hello, ");  
    out.print( name );  
    out.write("</h1>\r\n");
```

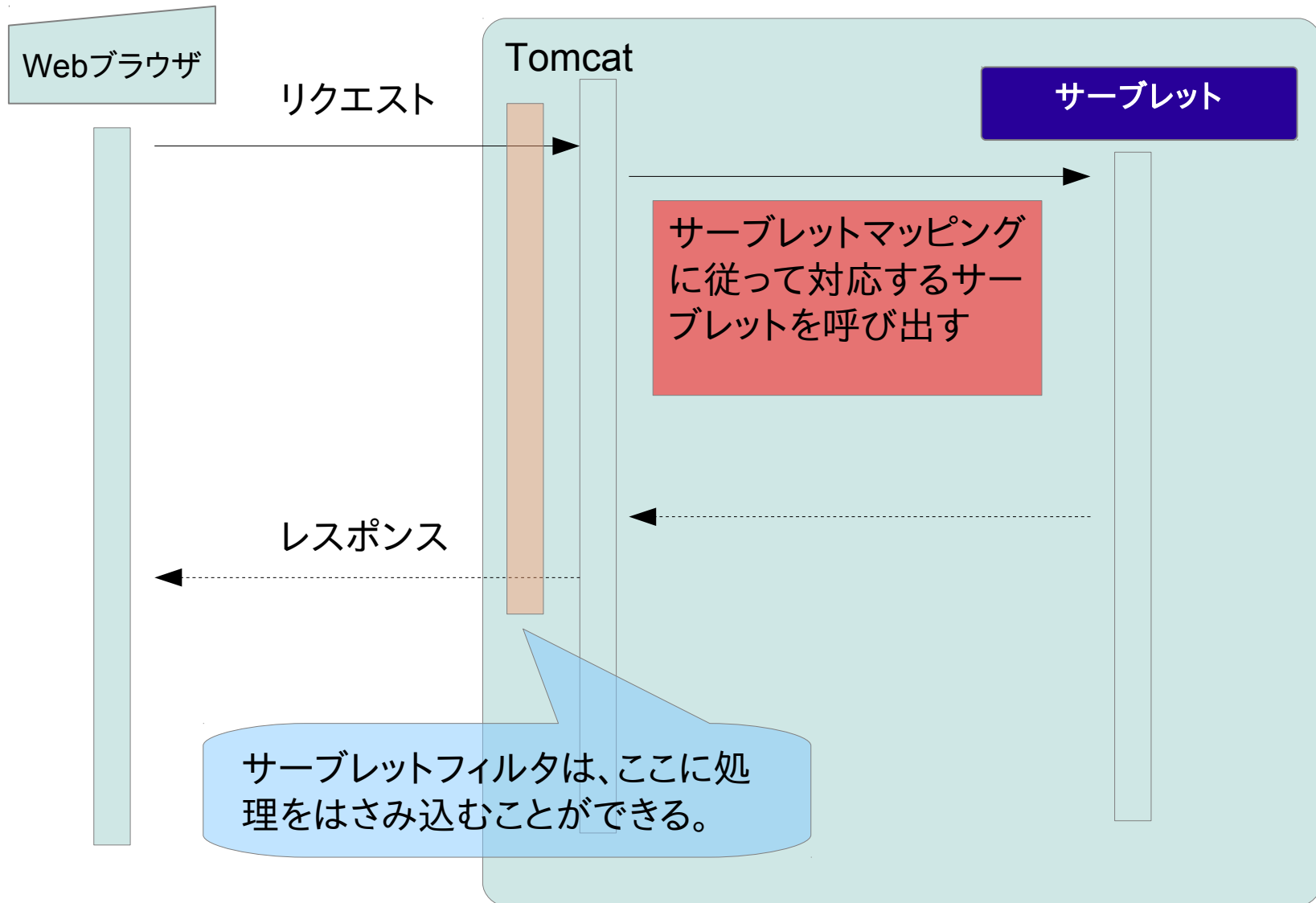
```
} else {
```

...

JSPのソースは、JSPが思い通りに動かない場合の調査や、JSPをデバッグしたい場合に必要になります

。

サーブレット・フィルタ

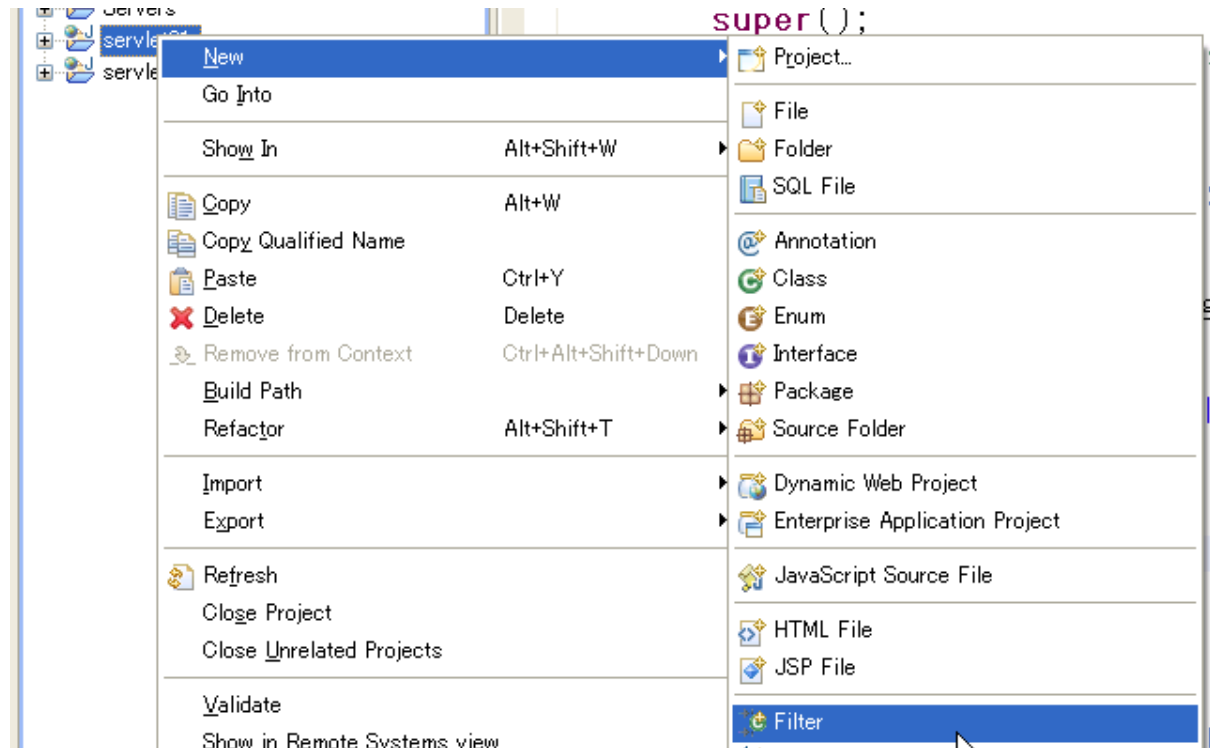


サーブレットフィルタ

- サーブレットフィルタが必要となるケース：
 - サーブレット、JSPの実行前に何らかの処理を行いたい。
 - サーブレット、JSPの実行後に何らかの処理を行いたい。
 - サーブレット、JSP実行前に割り込んで、特定の条件が成立する場合に、実行をスキップしたい。
 - サーブレット、JSP実行前に割り込んで、パラメータの内容を変更したい。
- こうした処理を、サーブレット、JSP側に変更を入れることなく実現できる。

サーブレットフィルタ

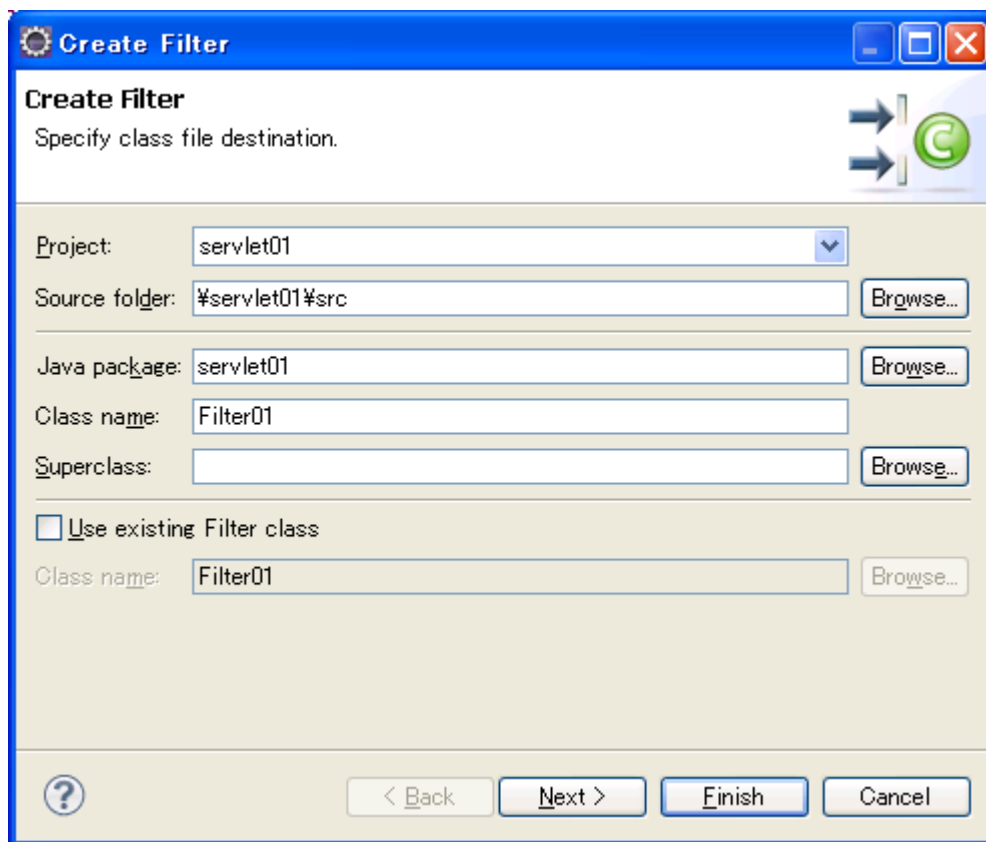
- サーブレットフィルタを使って処理時間を測ってみましょう。
- servlet01を右クリックし、New->Filterをクリックします。



サーブレットフィルタ

- 以下を入力してFinishをクリックします。

- パッケージ
servlet01
- クラス名
Filter01



The image shows a 'Create Filter' dialog box from an IDE. The title bar says 'Create Filter'. Below the title bar, it says 'Create Filter' and 'Specify class file destination.' There is a green circular icon with a 'C' on the right. The dialog has several input fields: 'Project' (servlet01), 'Source folder' (servlet01/src), 'Java package' (servlet01), 'Class name' (Filter01), and 'Superclass' (empty). There are 'Browse...' buttons next to the 'Source folder', 'Java package', and 'Superclass' fields. Below these fields, there is a checkbox labeled 'Use existing Filter class' which is unchecked. Below the checkbox, there is a 'Class name' field (Filter01) and a 'Browse...' button. At the bottom, there are four buttons: '< Back', 'Next >', 'Finish', and 'Cancel'. A help icon (?) is also present at the bottom left.

Create Filter
Specify class file destination.

Project: servlet01

Source folder: %servlet01%src Browse...

Java package: servlet01 Browse...

Class name: Filter01

Superclass: Browse...

☐ Use existing Filter class

Class name: Filter01 Browse...

? < Back Next > Finish Cancel

サーブレットフィルタ

@WebFilter("/Filter01")

public class Filter01 implements Filter {

```
public void doFilter(ServletRequest req  
    long startTime = System.nanoTime();  
    chain.doFilter(request, response);  
    System.err.printf("Elapsed %,dns%n",  
        System.nanoTime() - startTime);  
}
```

注: この要素には添付されたソースも、
た。

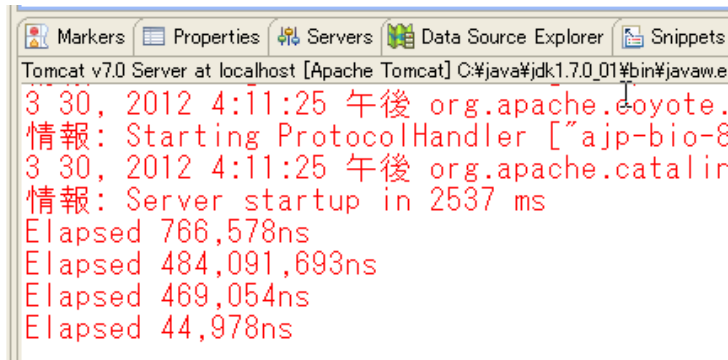
ここを、"/*"に変更します。ここはURLパターンと呼び、サーブレットマッピングと同様、このパターンに合致したリクエストがあった場合に、このフィルタが実行されます。'*'はワイルドカードと呼び「何でもマッチ」という意味です。

フィルタのURLパターンにマッチするリクエストが来ると、サーブレット、JSPの実行に先立って、このメソッドが呼ばれます。

chain.doFilter()を呼ぶと、後続処理が行われます。つまり引き続きこのリクエストに対応するサーブレット、JSPが呼び出されます。そこで、ここでは現在時刻を記録、後続処理呼び出し、再度現在時刻を取得して、その差分から処理時間を表示しています。

サーブレットフィルタ

- これまで通り、Servlet01や、hello.jspをブラウザから呼び出して、Eclipseのコンソールに以下のように経過時間が表示されることを確認してください。今回はURLマッピングを”/*”にしましたが、”/Servlet01”としたり、”*.jsp”とすると、パターンが合致した時にだけフィルタを実行できます。

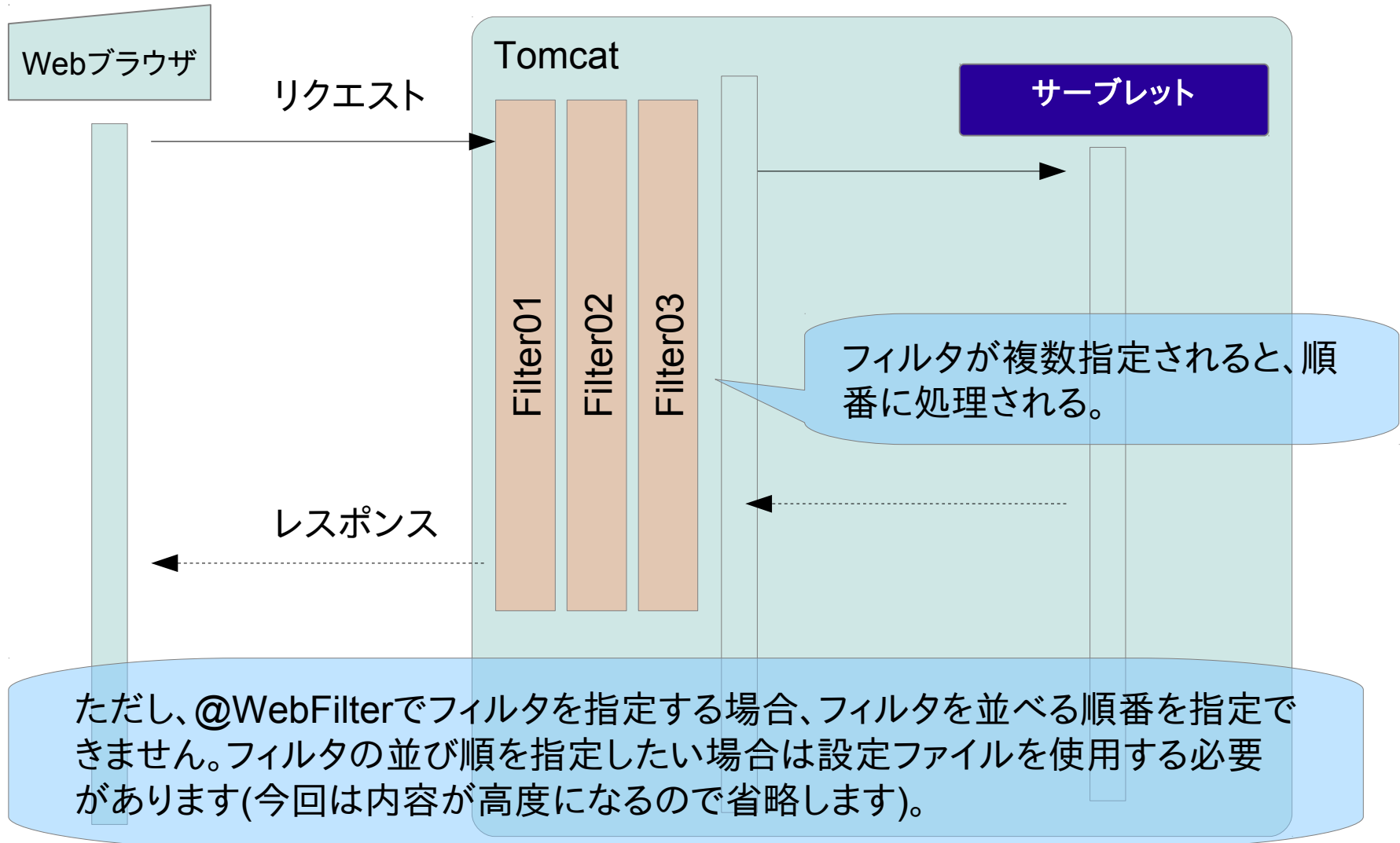


```
Tomcat v7.0 Server at localhost [Apache Tomcat] C:\java\jdk1.7.0_01\bin\javaw.e
3:30, 2012 4:11:25 午後 org.apache.coyote.
情報: Starting ProtocolHandler [\"ajp-bio-8
3:30, 2012 4:11:25 午後 org.apache.catalin
情報: Server startup in 2537 ms
Elapsed 766,578ns
Elapsed 484,091,693ns
Elapsed 469,054ns
Elapsed 44,978ns
```

- Servlet01と、hello.jspは処理内容は同一ですが、その処理時間はどうでしょうか？
 - JSPの方に時間が余計にかかるのは、主にJSPの自動コンパイルのためです。Tomcatはデフォルトで、JSPファイルが更新されたかをリクエストごとに調べて、更新されていれば、サーブレット変換、コンパイルし直しを実行します。ファイルの最終更新時刻を取得する処理は比較的重い処理なので、余計に時間がかかります。
(この設定は、development、reloadingというTomcatのパラメータで変更できますが内容が高度になるのでここでは省略します)

サーブレットフィルタ

- サーブレットフィルタは複数を数珠繋ぎにできます。これをチェーンと呼びます。



まとめ

- サーブレットでは、doGetメソッドでGETリクエストを、doPostメソッドでPOSTリクエストを処理します。
- ブラウザへのレスポンスをテキストで設定するには、リクエストオブジェクトのgetWriter()を呼び出してPrintWriterを取得して、PrintWriterのprintXXXメソッドを使って設定します。
- 1つのサーバには、複数のWebモジュールをデプロイできます。これらはコンテキストパスで区別します。
- Webモジュールが異なれば、例え同一パッケージ、クラス名のクラスであっても、サーバは別のものとして扱います。
- 複数のサーバを1つのPC上で動作させる場合は、ポート番号がぶつからないようにする必要があります。
- 1つのWebモジュール内にはサーブレットを複数用意することができます。これらをリクエストによって呼び分けるためにサーブレットマッピングを使用します。

まとめ

- ブラウザからリクエストを送る際に、リクエストパラメータを付加情報として渡すことができます。
- リクエストパラメータをサーバ側で参照するには、リクエストオブジェクトのgetParameterメソッドを使用します。
- GETリクエストでのリクエストパラメータ指定は、URLの最後に「?name1=value1&name2=value2...」と指定します。
- formタグを用いることでリクエストパラメータ付加したリクエストを簡単に生成することができます。
- サーブレットでは、Javaコード内にHTMLを埋め込むので煩雑となります。JSPを用いることでHTMLを生成するアプリケーションを簡単に記述することができます。ただしJavaコード部分が多いと逆に煩雑になります。
- JSPはアプリケーションサーバ内でサーブレットに変換されてから実行されます。

まとめ

- サーブレットフィルタを用いると、既存のJSP、サーブレットの実行前、後に別の処理をはさみ込むことができます。
- サーブレットフィルタでは、doFilterメソッドにフィルタ処理を実装します。
- FilterChain.doFilterを呼び出すことで後続処理を呼び出すことができます。
- フィルタを複数数珠繋ぎにすることができます。これをフィルタチェーンと呼びます。