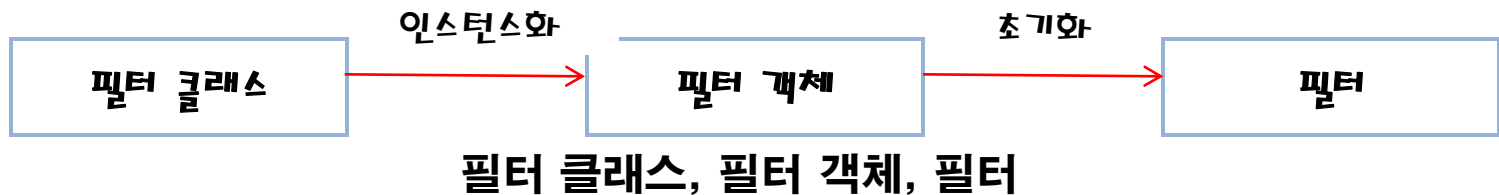


필터와 래퍼

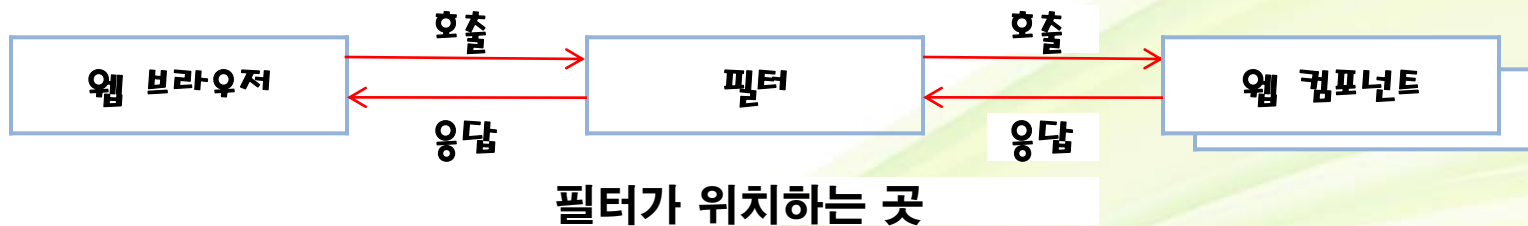


필터와 래퍼에 대하여

- 필터(filter)란 글자 그대로 여과기 역할을 하는 프로그램이다.
- 필터는 자바 클래스 형태로 구현해야 하며, 이 클래스를 필터 클래스(filter class)라고 한다.

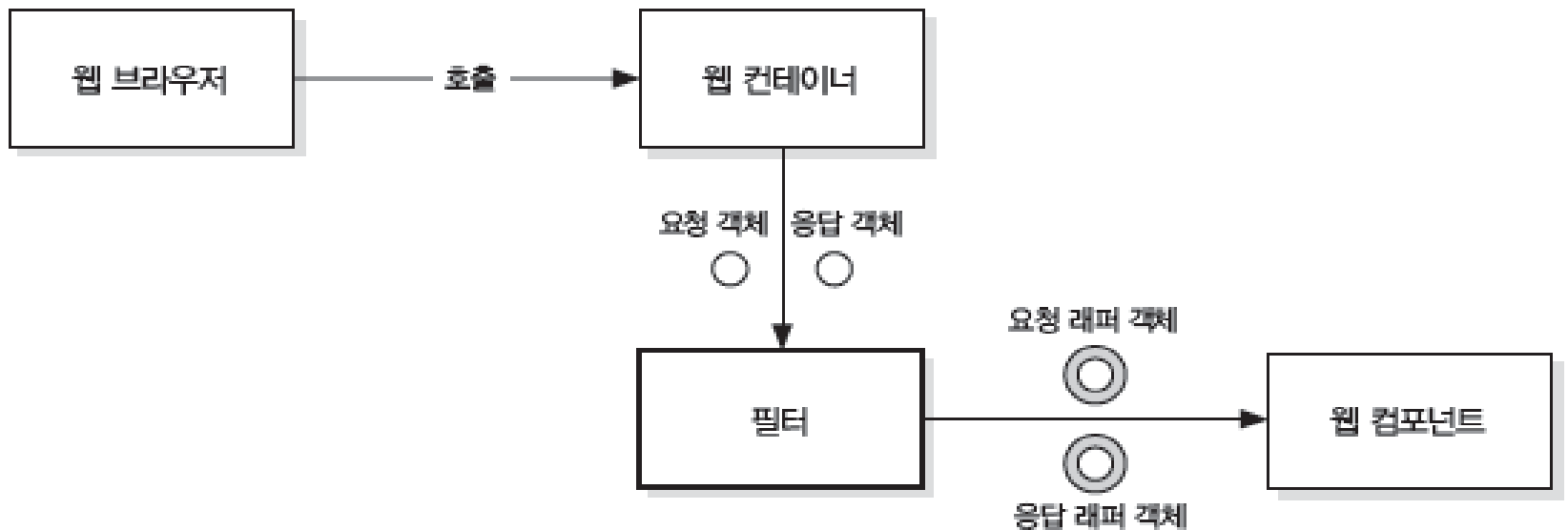


- 초기화된 필터는 웹 브라우저와 웹 컴포넌트 사이에 위치한다.



- 웹 브라우저가 웹 컴포넌트를 호출하면 필터가 대신 호출되고, 필터는 사전 작업을 수행한 다음에 웹 컴포넌트를 호출한다.
- 웹 컴포넌트가 할 일을 끝내면 실행의 제어는 다시 필터로 돌아가고, 필터는 사후 작업을 수행한 후 웹 브라우저로 응답을 보낸다.

- 래퍼(wrapper)란 웹 브라우저와 웹 컴포넌트 사이를 오가는 요청 메시지와 응답 메시지를 포장하는 프로그램이다.
- 래퍼는 래퍼 클래스 형태로 구현되는데, 포장되는 객체의 종류에 따라 요청 래퍼 클래스와 응답 래퍼 클래스로 나뉘어 진다.



요청 래퍼 객체와 응답 래퍼 객체

필터 클래스의 작성, 설치, 등록

- 필터 클래스를 작성할 때는 서블릿 규격서에 정해져 있는 규칙을 지켜야 한다. 그 중 가장 중요한 규칙은 javax.servlet.Filter 인터페이스를 구현해야 한다는 것이다.
- Filter 인터페이스에는 다음과 같은 세 개의 메서드가 있다.
 - doFilter 메서드는 웹 브라우저가 웹 컨테이너로 요청을 보냈을 때 호출되는 메서드이다.
 - init 메서드는 필터의 초기화 작업이 수행될 때 호출되는 메서드이고, destroy 메서드는 필터가 웹 컨테이너에 의해 제거되기 직전에 호출되는 메서드이다.

```
public class SimpleFilter implements Filter {  
    public void init(FilterConfig config) throws ServletException {  
    }  
  
    public void doFilter(ServletRequest request, ServletResponse response, FilterChain chain)  
        throws IOException, ServletException {  
    }  
  
    public void destroy() {  
    }  
}
```

웹 컨테이너가 필터 객체를 초기화할 때 호출되는 메서드

웹 브라우저가 웹 컨테이너로 요청을 보냈을 때 호출되는 메서드

웹 컨테이너가 필터 객체를 제거하기 직전에 호출되는 메서드

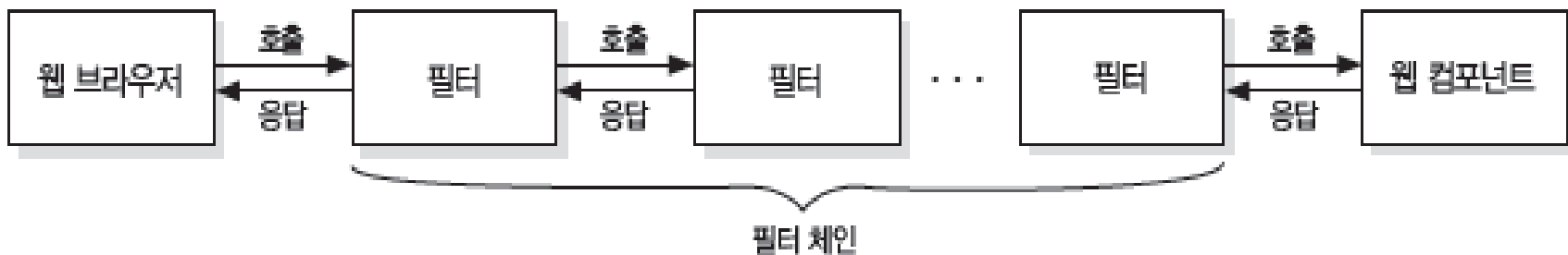
- doFilter 메서드의 첫 번째와 두 번째 파라미터는 요청 객체와 응답 객체이며, 필터가 없었더라면 이 두 객체는 웹 컨테이너가 웹 컴포넌트로 직접 넘겨주었을 것이다.

웹 컨테이너가 넘겨주는 요청 객체

웹 컨테이너가 넘겨주는 응답 객체

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

- doFilter 메서드의 세 번째 파라미터는 필터 체인을 표현하는 FilterChain 객체이다



연속해서 호출되는 필터들로 이루어지는 필터 체인

- doFilter 메서드 안에서 세 번째 파라미터에 대해 doFilter라는 이름의 메서드를 호출하면서 doFilter 메서드가 받은 첫 번째와 두 번째 파라미터를 넘겨주면 필터 체인의 다음번 멤버가 호출된다.

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

필터 체인의 다음번 멤버를 호출하는 메서드

- 다음은 웹 컴포넌트가 실행되기 전과 후에 System.out.println 메서드를 이용해서 메시지를 출력하는 필터 클래스이다.

아주 간단한 필터 클래스

```
package myfilter;
import javax.servlet.*;
import java.io.*;
public class SimpleFilter implements Filter {
    public void init(FilterConfig config) throws ServletException {
    }
    public void doFilter(ServletRequest request, ServletResponse response, FilterChain chain)
        throws IOException, ServletException {
        System.out.println(" 이제 곧 웹 컴포넌트가 시작될 것입니다. ");
        chain.doFilter(request, response);
        System.out.println(" 이제 막 웹 컴포넌트가 완료되었습니다. ");
    }
    public void destroy() {
    }
}
```

- 필터 클래스를 작성한 다음에 해야 할 일은 다음과 같다.
 - 필터 클래스를 컴파일한다.
 - 컴파일 결과물을 웹 컨테이너에 설치한다.
 - 필터 클래스를 web.xml 파일에 등록한다.

- 필터 클래스를 web.xml 파일에 등록하려면 루트 엘리먼트인 <web-app>의 아래에 <filter>와 <filter-mapping>이라는 두 개의 엘리먼트를 추가해야 한다.

```
<web-app>
  <filter>
    </filter>
  <filter-mapping>
    </filter-mapping>
</web-app>
```

← 필터를 등록하는 엘리먼트

← 필터를 적용할 웹 컴포넌트를 지정하는 엘리먼트

- <filter> 엘리먼트 안에는 <filter-name>과 <filter-class>라는 두 개의 서브엘리먼트를 써야 하며, 이 둘은 각각 필터 이름과 필터 클래스 이름을 포함하는 역할을 한다.

필터 이름

```
<filter>
  <filter-name>simple-filter</filter-name>
  <filter-class>myfilter.SimpleFilter</filter-class>
</filter>
```

필터 클래스 이름

- `<filter-mapping>` 엘리먼트 `<filter-name>`와 `<servlet-name>`, `<url-pattern>` 중 한 서브엘리먼트를 써야 한다. `<filter-name>` 이 안에는 `<filter>` 엘리먼트 안에 썼던 것과 동일한 필터 이름을 써야 하고, `<servlet-name>`과 `<url-pattern>` 중 어느 것을 써야 할지는 몇 개의 웹 컴포넌트의 해당 필터를 적용할 것인지에 따라서 결정된다.
- 필터를 특정한 한 웹 컴포넌트에만 적용하고자 할 때는 `<servlet-name>` 서브엘리먼트에 해당 웹 컴포넌트의 이름을 지정하면 된다.

```
<filter-mapping>  
  <filter-name>simple-filter</filter-name>  
  <servlet-name>hello-servlet</servlet-name>  
</filter-mapping>
```

필터 이름

필터를 적용할 웹 컴포넌트의 이름

- 필터를 여러 개의 웹 컴포넌트에 한꺼번에 적용하고자 할 때는 <url-pattern> 서브엘리먼트에 해당 웹 컴포넌트들의 URL 패턴을 쓰면 된다.
 - 필터를 같은 웹 애플리케이션 디렉터리 내에 있는 모든 웹 컴포넌트에 적용하려면 <url-pattern> 엘리먼트 안에 /*라고 쓰면 된다.

```
<filter-mapping>
  <filter-name>simple-filter</filter-name>
  <url-pattern>/*</url-pattern>
</filter-mapping>
```

같은 웹 애플리케이션 디렉터리에 있는
모든 웹 컴포넌트를 가리키는 URL 패턴

- 필터를 같은 웹 애플리케이션 디렉터리 내에 있는 모든 JSP 페이지에 적용하려면 <url-pattern> 엘리먼트 안에 *.jsp라고 쓰면 된다.

```
<filter-mapping>
  <filter-name>simple-filter</filter-name>
  <url-pattern>*.jsp</url-pattern>
</filter-mapping>
```

같은 웹 애플리케이션 디렉터리에 있는
모든 웹 컴포넌트를 가리키는 URL 패턴

- <url-pattern> 엘리먼트 안에 계층적인 URL 경로명을 쓸 수도 있다. 이 경로명은 웹 애플리케이션 디렉터리의 루트 디렉터리를 의미하는 슬래시(/) 문자로 시작해야 한다.

```
<filter-mapping>
```

```
  <filter-name>simple-filter</filter-name>
```

```
  <url-pattern>/sub1/*</url-pattern>
```

```
</filter-mapping>
```

/sub1/이라는 URL 경로명으로 시작하는
모든 웹 컴포넌트를 가리키는 URL 패턴

- <url-pattern> 엘리먼트 안에 계층적인 URL 경로명을 쓸 때는 와일드카드(*)문자와 파일 확장자를 함께 쓰면 안 된다.

```
<filter-mapping>
```

```
  <filter-name>simple-filter</filter-name>
```

```
  <url-pattern>/sub1/*.jsp</url-pattern>
```

```
</filter-mapping>
```

잘못된 URL 패턴

- <filter-mapping> 엘리먼트 안에 여러 개의 <url-pattern> 서브엘리먼트를 쓸 수도 있다.

```
<filter-mapping>
  <filter-name>simple-filter</filter-name>
  <url-pattern>/sub1/*</url-pattern>
  <url-pattern>/sub2/*</url-pattern>
</filter-mapping>
```

URL 경로명이 /sub1/ 또는 /sub2/로 시작하는 모든 웹 컴포넌트에 필터를 적용합니다

- <filter-mapping> 엘리먼트 안에 여러 개의 <servlet-name> 엘리먼트를 쓸 수도 있고, <servlet-name>과 <url-pattern> 엘리먼트를 혼용해서 쓸 수도 있다.

```
<filter-mapping>
  <filter-name>simple-filter</filter-name>
  <url-pattern>/sub1/*</url-pattern>
  <url-pattern>/sub2/*</url-pattern>
  <servlet-name>hello-servlet</servlet-name>
</filter-mapping>
```

URL 경로명이 /sub1/ 또는 /sub2/로 시작하는 웹 컴포넌트와 이름이 hello-servlet인 서블릿에 필터를 적용합니다

- 앞 페이지의 <filter-mapping> 엘리먼트는 다음과 같은 세 개의 <filter-mapping> 엘리먼트를 쓴 것과 똑같은 효과를 갖는다.

```
<filter-mapping>  
  <filter-name>simple-filter</filter-name>  
  <url-pattern>/sub1/*</url-pattern>  
</filter-mapping>
```

```
<filter-mapping>  
  <filter-name>simple-filter</filter-name>  
  <url-pattern>/sub2/*</url-pattern>  
</filter-mapping>
```

```
<filter-mapping>  
  <filter-name>simple-filter</filter-name>  
  <servlet-name>hello-servlet</servlet-name>  
</filter-mapping>
```

Servlet3.0에서 어너테이션 이용

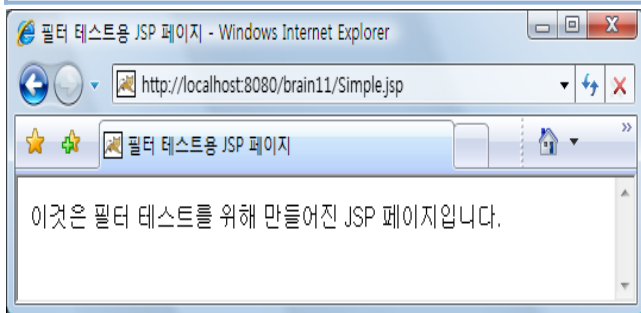
```
package myfilter;
import java.io.IOException;
import javax.servlet.Filter;
import javax.servlet.FilterChain;
import javax.servlet.FilterConfig;
import javax.servlet.ServletException;
import javax.servlet.ServletRequest;
import javax.servlet.ServletResponse;
import javax.servlet.annotation.WebFilter;
@WebFilter("/*")
public class SimpleFilter implements Filter {
    public void destroy() { }
    public void doFilter(ServletRequest request, ServletResponse response,
FilterChain chain) throws IOException, ServletException {
        System.out.println("필터가 시작 되었습니다 ");
        chain.doFilter(request, response);
        System.out.println("웹 토크전트가 완료 되었습니다");
    }
    public void init(FilterConfig fConfig) throws ServletException { }
}
```

WebFilter

```
import java.io.IOException;
import javax.servlet.*;
import javax.servlet.annotation.WebFilter;
@WebFilter(urlPatterns={"/sub1/*","/sub2/*"})
public class SimpleFilter implements Filter {
    public void doFilter(ServletRequest request,
        ServletResponse response, FilterChain chain) throws IOException,
        ServletException {
        System.out.println("필터가 시작 되었습니다");
        chain.doFilter(request, response);
        System.out.println("필터가 종료 되었습니다");
    }
    public void destroy() {
    }
    public void init(FilterConfig arg0) throws ServletException {
    }
}
```

필터를 테스트하기 위한 JSP 페이지 (1)

```
<%@page contentType= "text/html; charset=euc-kr" %>
<% System.out.println( "이것은 JSP 페이지 안에서 출력하는 메시지입니다." ); %>
<HTML>
  <HEAD><TITLE>필터 테스트용 JSP 페이지</TITLE></HEAD>
  <BODY>
    이것은 필터 테스트를 위해 만들어진 JSP 페이지입니다.
  </BODY>
</HTML>
```

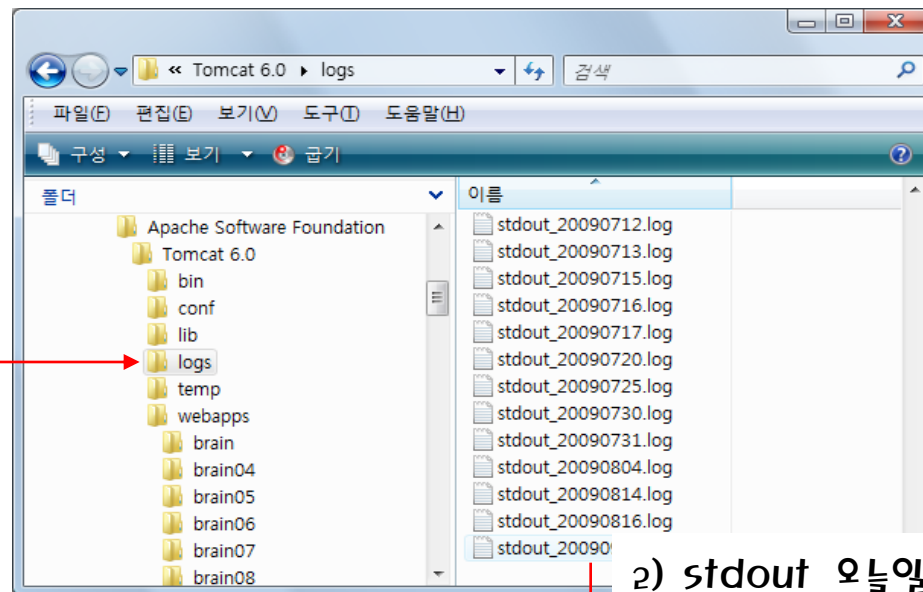


실행 결과

```
<web-app
xmlns= "http://java.sun.com/xml/ns/javaee"
version="2.5">
  <filter>
    <filter-name>simple-filter</filter-name>
    <filter-class>myfilter.SimpleFilter</filter-class>
  </filter>
  <filter-mapping>
    <filter-name>simple-filter</filter-name>
    <url-pattern>*.jsp</url-pattern>
  </filter-mapping>
</web-app>
```

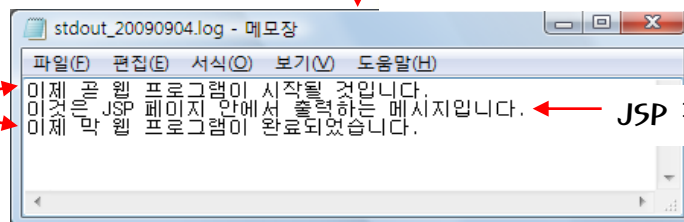
Web.xml

1) 톰캣의 logs 디렉터리로
가십시오.



2) stdout_오늘일자.log에 해당하는
파일을 텍스트 에디터로 열어보십시오.

필터에서 출력한 메시지



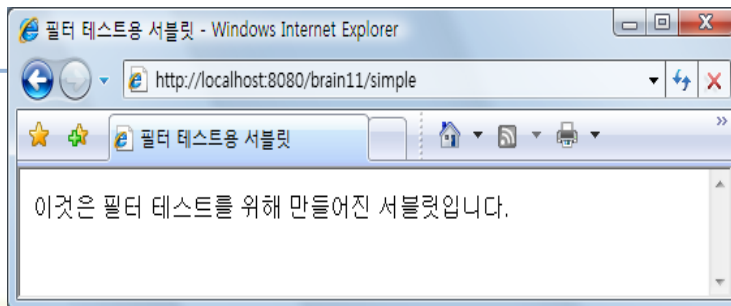
JSP 페이지에서 출력한 메시지

필터가 실행되었음을 확인하는 방법

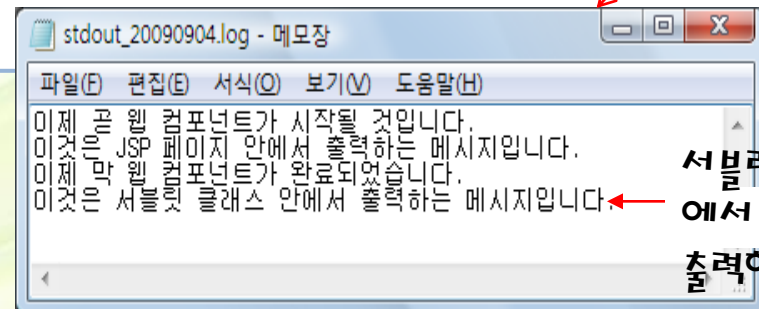
필터를 테스트하기 위한 서블릿 클래스

```
package myervlet;
import javax.servlet.http.*;
import javax.servlet.*;
import java.io.*;
public class SimpleServlet extends HttpServlet {
    public void doGet(HttpServletRequest request, HttpServletResponse response)
        throws IOException, ServletException {
        System.out.println( “이것은 서블릿 클래스 안에서 출력하는 메시지입니다.” );
        response.setContentType( “text/html;charset=euc-kr ”);
        PrintWriter out = response.getWriter();
        out.println( “<HTML><HEAD><TITLE>필터 테스트용 서블릿</TITLE></HEAD> ”);
        out.println( “<BODY> ”);
        out.println( “이것은 필터 테스트를 위해 만들어진 서블릿입니다.” );
        out.println( “</BODY> ”);
        out.println( “</HTML> ”);
    }
}
```

톰캣의 log 디렉터리에 있는
stdout_오늘일자.log 파일



실행 결과



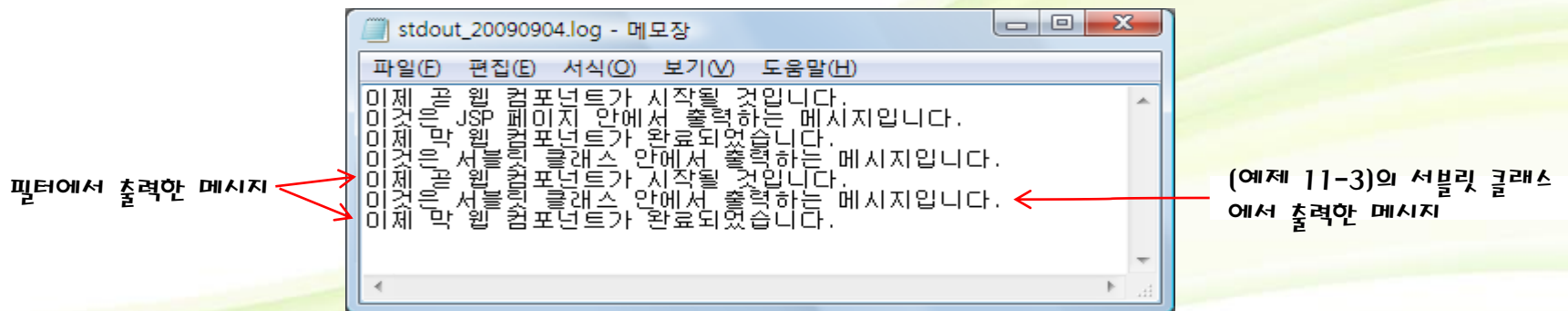
실행한 후에 기록된 로그 메시지

- web.xml 파일을 열고 <filter-mapping> 엘리먼트의 <url-pattern> 서브엘리먼트 내용을 다음과 같이 /*로 고쳐서 JSP 페이지와 서블릿 클래스에 모두 필터가 적용되도록 만들어보자.



URL 패턴을 /*로 고치세요

필터가 웹 애플리케이션의 모든 웹 컴포넌트에 적용되도록 만드는 방법

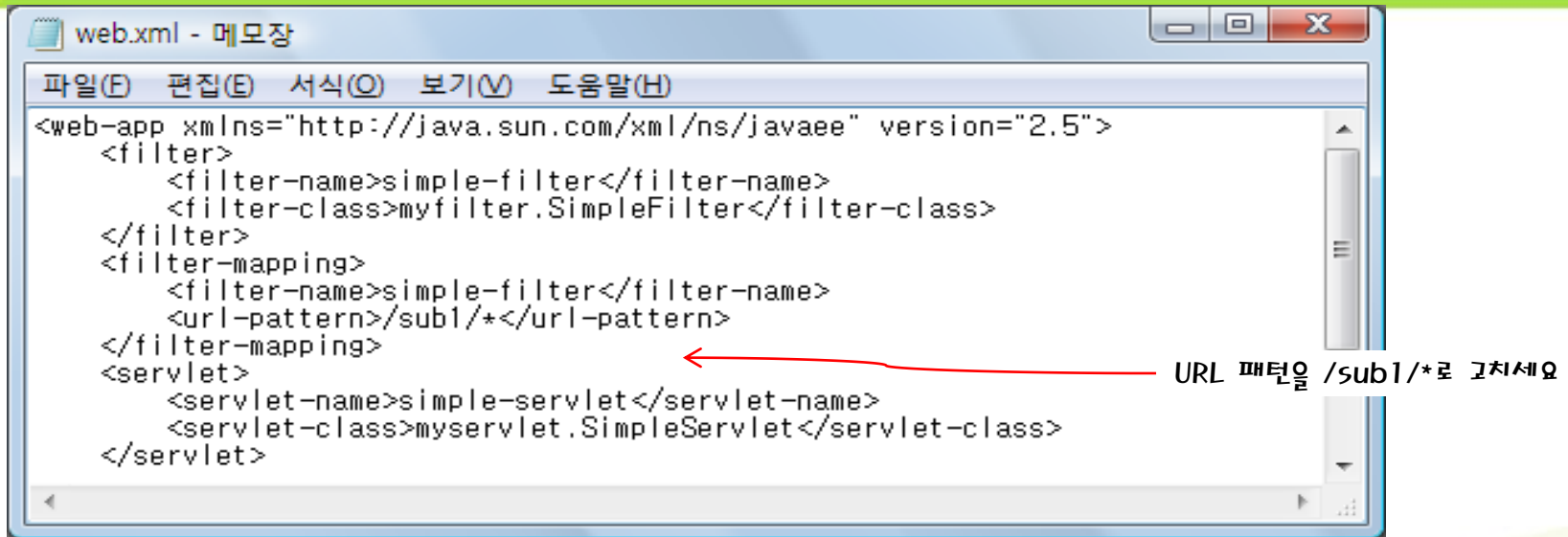


필터에서 출력한 메시지

(예제 11-3)의 서블릿 클래스
에서 출력한 메시지

다시 실행한 후에 기록된 로그 메시지

- web.xml 파일을 열고 <url-pattern> 엘리먼트의 내용을 다음과 같이 /sub1/*로 고쳐보자



/sub1/으로 시작하는 URL 경로명의 웹 컴포넌트에 필터가 적용되도록 만드는 방법

필터를 테스트하기 위한 JSP 페이지 (2)

```
<%@page contentType= "text/html; charset=euc-kr" %>
<% System.out.println( "이것은 sub1 디렉터리에 있는 JSP 페이지에서 출력하는
메시지입니다." ); %>
<HTML>
  <HEAD><TITLE>필터 테스트용 JSP 페이지</TITLE></HEAD>
  <BODY>
    이것은 필터 테스트를 위해 만들어진 JSP 페이지입니다.
  </BODY>
</HTML>
```

필터 클래스의 작성, 설치, 등록

❖ 필터 클래스의 init 메서드와 destroy 메서드

- 필터의 라이프 사이클 동안 단 한 번만 실행하면 되는 코드는 필터 클래스의 init 메서드나 destroy 메서드에 기술하는 것이 좋다.

로그 메시지를 특정 로그 파일로 출력하는 필터 클래스 - 미완성

```
package myfilter;
import javax.servlet.*; import java.io.*;
public class LogMessageFilter implements Filter {
    PrintWriter writer;
    public void init(FilterConfig config) throws ServletException {
        try {
            writer = new PrintWriter(new FileWriter( "C:\\logs\\myfilter.log ", true), true);
        } catch (IOException e) { throw new ServletException( "로그 파일을 열 수 없습니다." ); }
    }
    public void doFilter(ServletRequest request, ServletResponse response, FilterChain chain)
        throws IOException, ServletException {
        writer.println( "이제 곧 웹 컴포넌트가 시작될 것입니다." );
        writer.flush();
        chain.doFilter(request, response);
        writer.println( "이제 막 웹 컴포넌트가 완료되었습니다." );
        writer.flush();
    }
    public void destroy() {
        writer.close();
    }
}
```

❖ 필터 클래스의 init 메서드와 destroy 메서드

- 필터의 초기화 파라미터란 필터 클래스에서 사용할 데이터를 web.xml 파일 안에 이름-값 쌍으로 지정해 놓은 것을 말한다.
- 필터의 초기화 파라미터는 <filter> 엘리먼트 안에 <init-param> 서브엘리먼트를 추가하고, 그 안에 다시 <param-name>과 <param-value> 서브엘리먼트를 추가해서 등록할 수 있다.

```
<filter>
  <filter-name>log-filter</filter-name>
  <filter-class>myfilter.LogMessageFilter</filter-class>
  <init-param>
    <param-name>FILE_NAME</param-name>
    <param-value>C:\\logs\\myfilter.log<
  </init-param>
</filter>
```

초기화 파라미터의 이름

초기화 파라미터의 값

- 필터 클래스의 init 메서드 안에서 필터의 초기화 파라미터를 읽어오려면 FilterConfig 파라미터에 대해 getInitParameter 메서드를 호출하면 된다.

```
String filename = config.getInitParameter( "FILE_NAME " );
```

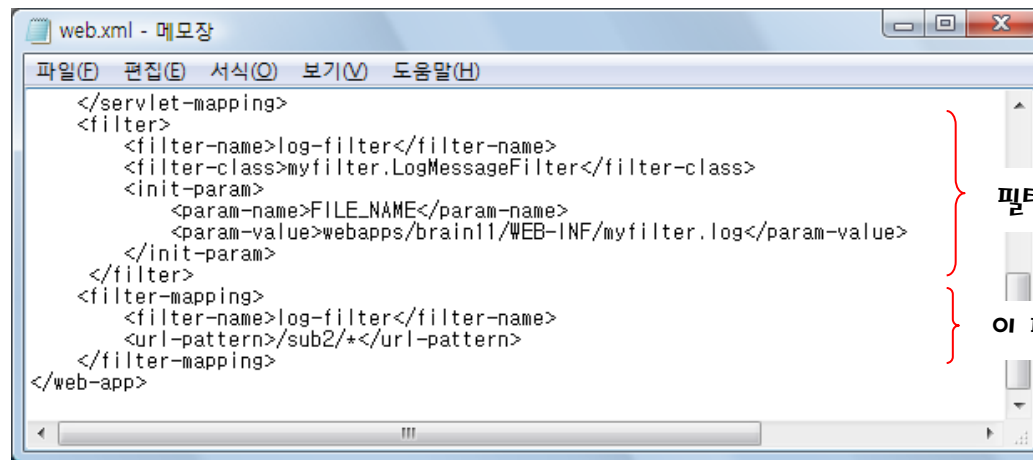
초기화 파라미터의 값을 가져오는 메서드

❖ 필터 클래스의 init 메서드와 destroy 메서드

로그 메시지를 별도의 파일로 출력하는 필터 클래스 - 완성

```
package myfilter;
import javax.servlet.*;import java.io.*;
public class LogMessageFilter implements Filter {
    PrintWriter writer;
    public void init(FilterConfig config) throws ServletException {
        String filename = config.getInitParameter( "FILE_NAME " );
        if (filename == null)
            throw new ServletException( "로그 파일의 이름을 찾을 수 없습니다." );
        try {
            writer = new PrintWriter(new FileWriter(filename, true), true);
        } catch (IOException e) { throw new ServletException( "로그 파일을 열 수 없습니다." ); }
    }
    public void doFilter(ServletRequest request, ServletResponse response, FilterChain chain) throws IOException,
ServletException {
        writer.println( "웹 컴포넌트가 시작될 것입니다." );
        writer.flush();
        chain.doFilter(request, response);
        writer.println( "웹 컴포넌트가 완료되었습니다." );
        writer.flush();
    }
    public void destroy() {
        writer.close();
    }
}
```

❖ 필터 클래스의 init 메서드와 destroy 메서드



```
web.xml - 메모장
파일(F) 편집(E) 서식(O) 보기(V) 도움말(H)
</servlet-mapping>
<filter>
  <filter-name>log-filter</filter-name>
  <filter-class>myfilter.LogMessageFilter</filter-class>
  <init-param>
    <param-name>FILE_NAME</param-name>
    <param-value>webapps/brain11/WEB-INF/myfilter.log</param-value>
  </init-param>
</filter>
<filter-mapping>
  <filter-name>log-filter</filter-name>
  <url-pattern>/sub2/*</url-pattern>
</filter-mapping>
</web-app>
```

} 필터를 등록하는 엘리먼트

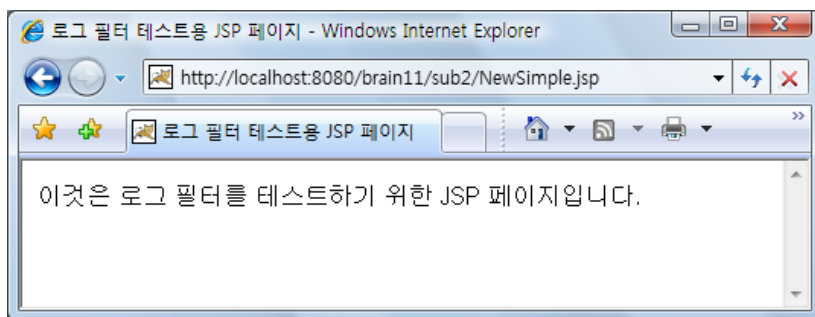
} 이 필터를 적용할 웹 프로그램을 지정하는 엘리먼트

등록하는 방법

❖ 필터 클래스의 init 메서드와 destroy 메서드

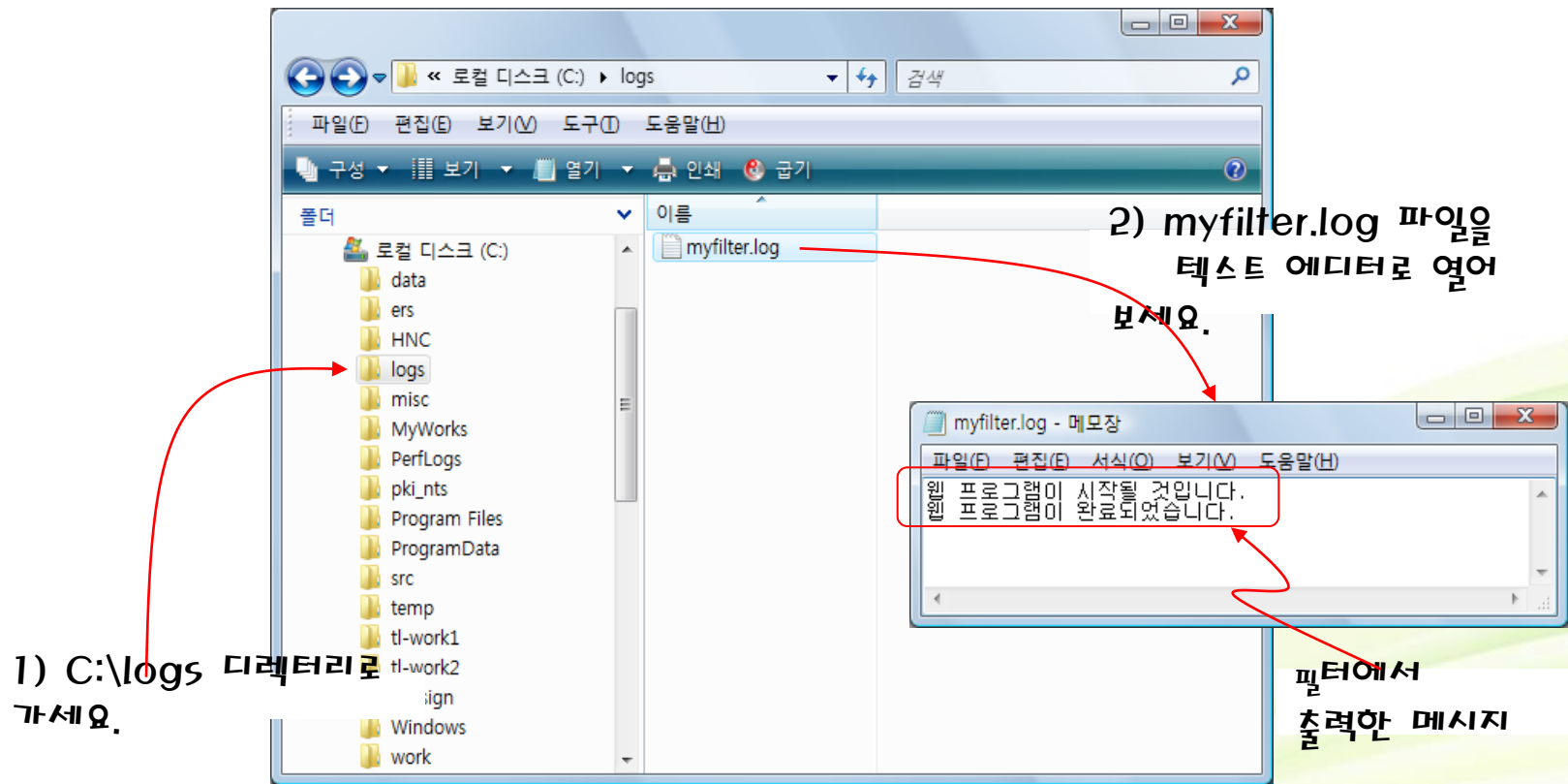
필터를 테스트하기 위한 JSP 페이지

```
<%@page contentType= "text/html; charset=euc-kr" %>
<HTML>
  <HEAD><TITLE>로그 필터 테스트용 JSP
페이지</TITLE></HEAD>
  <BODY>
    이것은 로그 필터를 테스트하기 위한 JSP 페이지입니다.
  </BODY>
</HTML>
```



실행 결과

❖ 필터 클래스의 init 메서드와 destroy 메서드



필터가 실행되었음을 확인하는 방법

❖ 요청 메시지와 응답 메시지에 포함된 정보 조회하기

- 웹 브라우저로부터 요청이 올 때마다 웹 브라우저의 IP 주소와 웹 자원의 콘텐츠 타입을 로그 파일에 기록하는 필터 클래스를 작성해보자.
- 필터 클래스의 doFilter 메서드 안에서 웹 브라우저의 IP 주소를 가져오기 위해서는 ServletRequest 파라미터에 대해 getRemoteAddr 메서드를 호출하면 된다.

```
String clientAddr = request.getRemoteAddr();
```

클라이언트의 IP 주소를 가져오는 메서드

- 필터 클래스의 doFilter 메서드 안에서 웹 자원의 콘텐츠 타입을 가져오기 위해서는 ServletResponse 파라미터에 대해 getContentType 메서드를 호출하면 된다.

```
String contentType = response.getContentType();
```

응답 메시지에 포함된 콘텐츠 타입을
가져오는 메서드

❖ 요청 메시지와 응답 메시지에 포함된 정보 조회하기

유용한 정보를 포함한 로그 메시지를 기록하는 필터 클래스

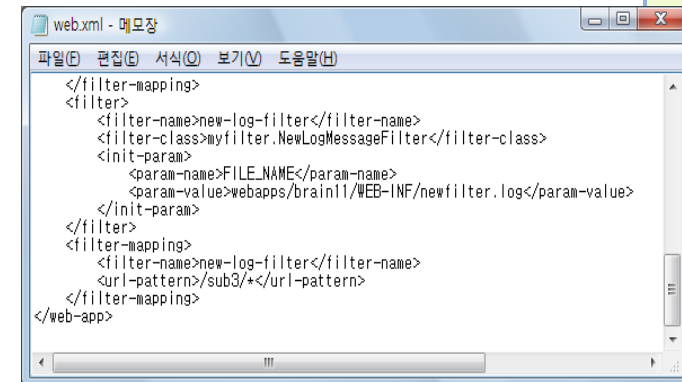
```
package myfilter; import javax.servlet.*; import java.io.*; import java.util.*;

public class NewLogMessageFilter implements Filter {
    PrintWriter writer;

    public void init(FilterConfig config) throws ServletException {
        String filename = config.getInitParameter( "FILE_NAME" );
        if (filename == null)    throw new ServletException( "로그 파일의 이름을 찾을 수 없습니다." );
        try {
            writer = new PrintWriter(new FileWriter(filename, true), true);
        } catch (IOException e) { throw new ServletException( "로그 파일을 열 수 없습니다." ); }
    }

    public void doFilter(ServletRequest request, ServletResponse response, FilterChain chain)
        throws IOException, ServletException {
        GregorianCalendar now = new GregorianCalendar();
        writer.printf( "현재일시: %TF %TT %n", now, now);
        String clientAddr = request.getRemoteAddr();
        writer.printf( "클라이언트 주소: %s %n", clientAddr);
        chain.doFilter(request, response);
        String contentType = response.getContentType();
        writer.printf( "문서의 콘텐츠 타입: %s %n", contentType);
        writer.println( "-----" );
    }

    public void destroy() {
        writer.close();
    }
}
```

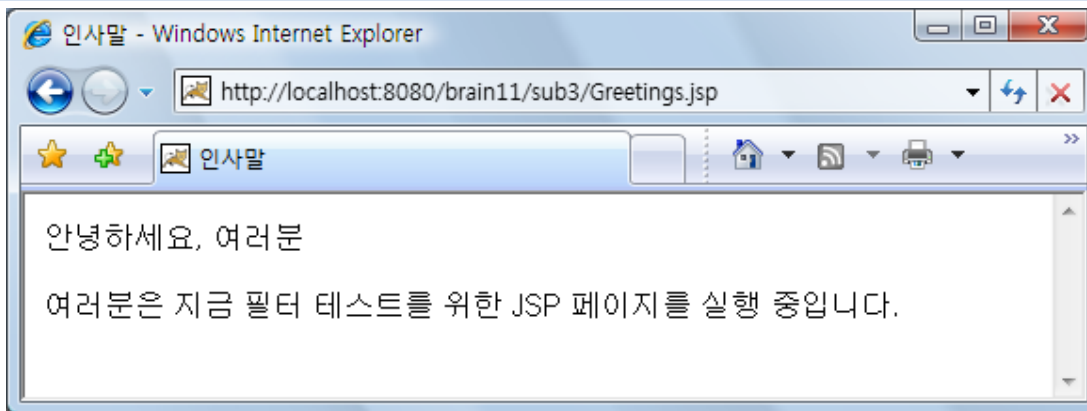


등록하는 방법

❖ 요청 메시지와 응답 메시지에 포함된 정보 조회하기

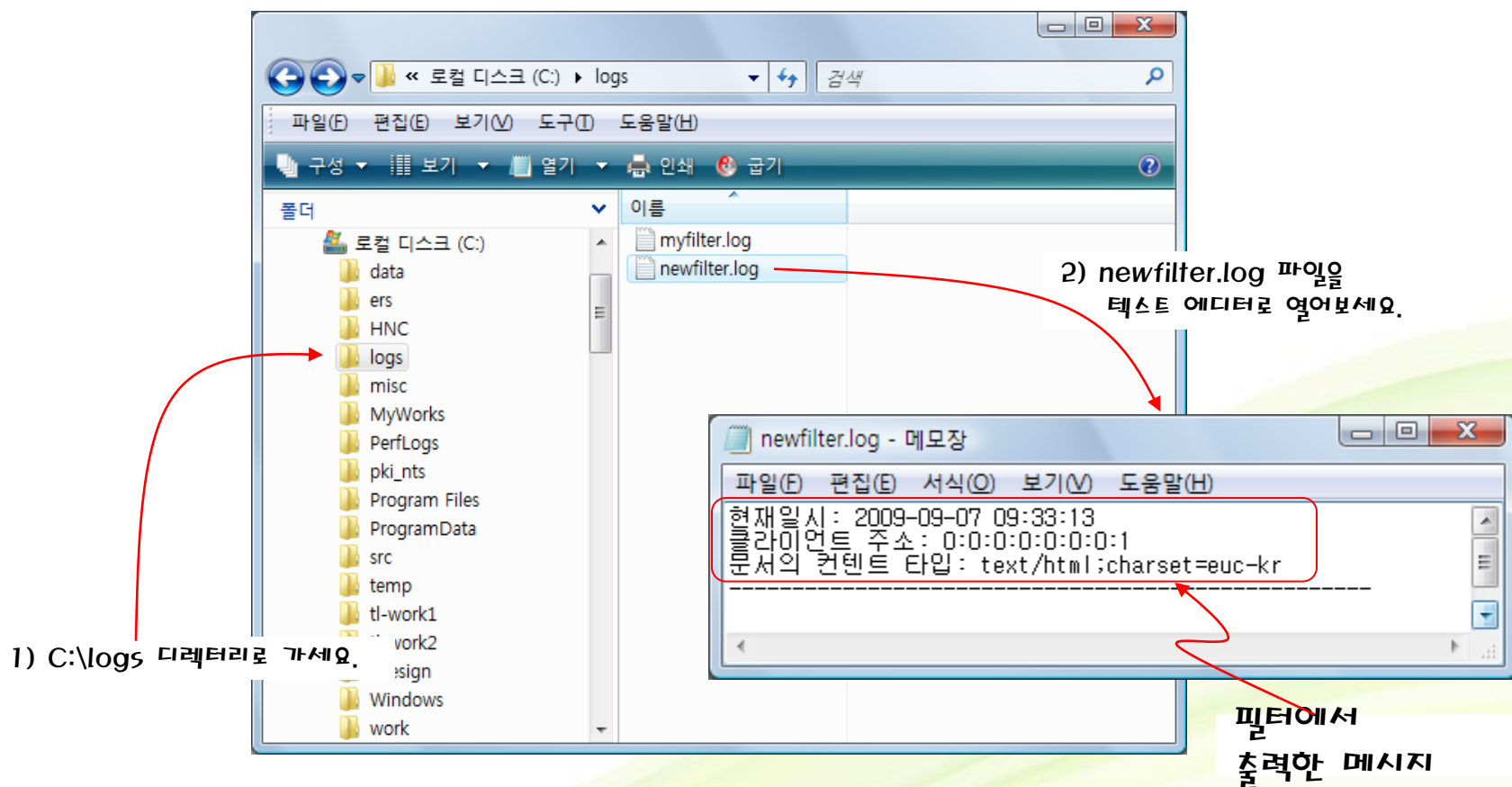
필터 클래스를 테스트하기 위한 JSP 페이지

```
<%@page contentType= "text/html; charset=euc-kr "%>
<HTML>
  <HEAD><TITLE>인사말</TITLE></HEAD>
  <BODY>
    안녕하세요, 여러분 <BR><BR>
    여러분은 지금 필터 테스트를 위한 JSP 페이지를 실행
중입니다.
  </BODY>
</HTML>
```



실행 결과

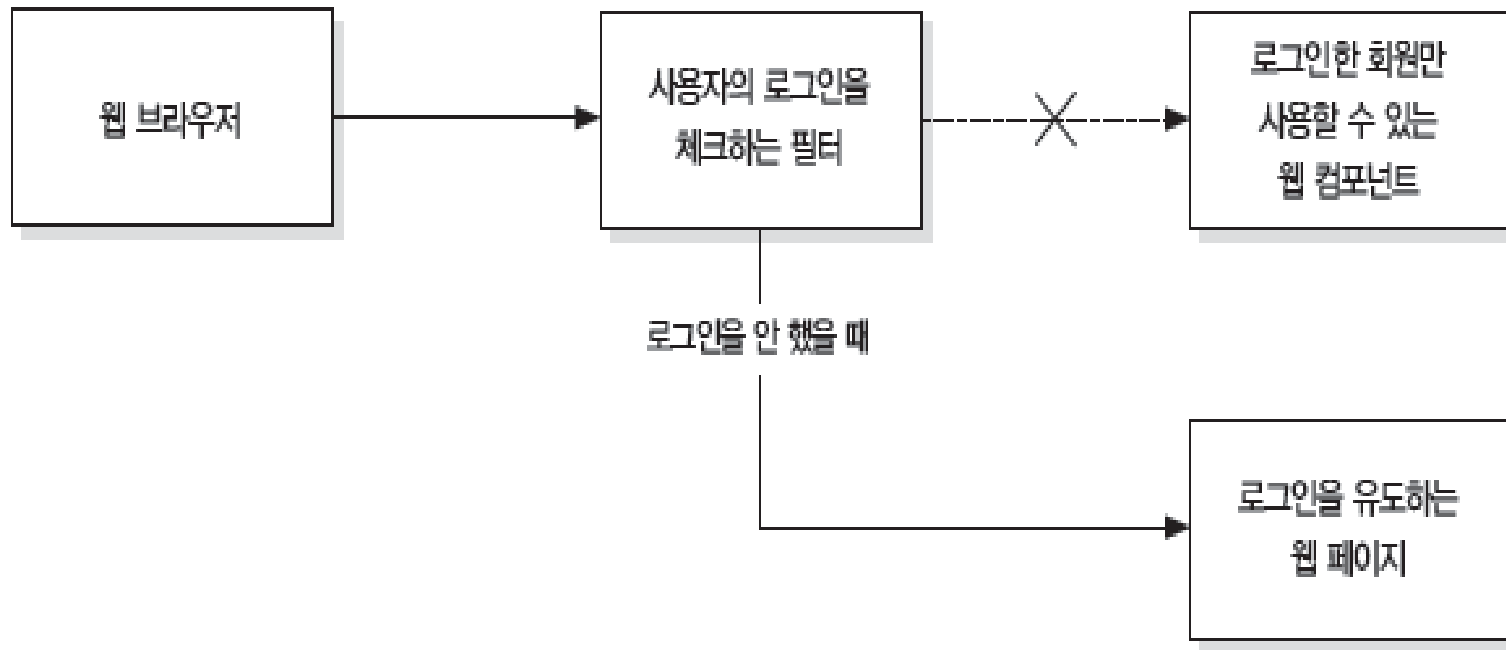
❖ 요청 메시지와 응답 메시지에 포함된 정보 조회하기



필터가 실행되었음을 확인하는 방법

❖ 필터 체인의 방향 바꾸기

- 필터를 이용하면 필터 체인이 향하는 방향을 바꿀 수도 있다



필터 체인의 방향을 바꾸는 필터

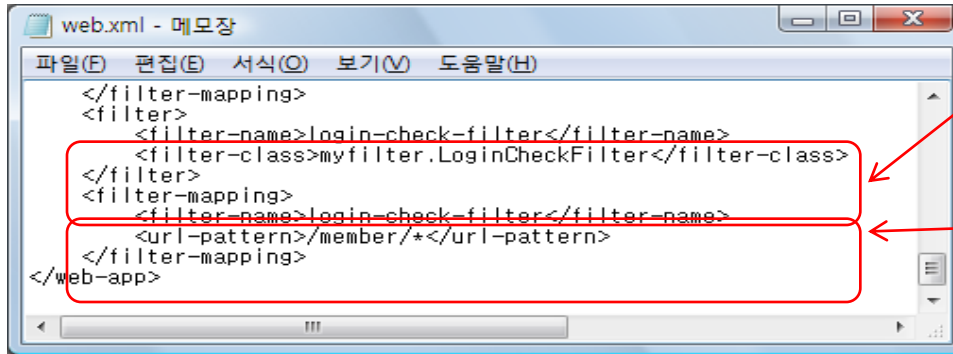
❖ 필터 체인의 방향 바꾸기

로그인하지 않은 사용자를 걸러내는 필터 클래스

```
package myfilter;
import javax.servlet.http.*; import javax.servlet.*; import java.io.*;
public class LoginCheckFilter implements Filter {
    public void init(FilterConfig config) throws ServletException {
    }
    public void doFilter(ServletRequest request, ServletResponse response, FilterChain chain)
                                throws IOException, ServletException {
        HttpServletRequest httpRequest = (HttpServletRequest) request;
        HttpServletResponse httpResponse = (HttpServletResponse) response;
        HttpSession session = httpRequest.getSession();
        if (session == null) {
            httpResponse.sendRedirect("../LoginForm.html ");
        }
        String id = (String) session.getAttribute( "ID ");
        if (id == null) {
            httpResponse.sendRedirect("../LoginForm.html ");
        }
        chain.doFilter(request, response);
    }
    public void destroy() {
    }
}
```


❖ 필터 체인의 방향 바꾸기

- 앞의 필터 클래스를 web.xml 파일에 등록하는 방법은 다음과 같다.



필터 클래스를 등록하는 엘리먼트

이 필터를 적용할 웹 컴포넌트를 지정하는 엘리먼트

로그인 입력 양식을 제공하는 HTML 문서

```
<HTML>   <HEAD>
  <META http-equiv= "Content-Type" content= "text/html; charset=euc-kr" >
  <TITLE>로그인 화면</TITLE>
</HEAD>
<BODY>
  로그인을 하세요.<BR><BR>
  <FORM ACTION=/Login.jsp METHOD=POST>
    아이디 <INPUT TYPE=TEXT NAME=ID SIZE=8> <BR>
    패스워드 <INPUT TYPE=TEXT NAME=PASSWORD SIZE=8> <BR>
    <INPUT TYPE=SUBMIT VALUE= '로그인' >
  </FORM>
</BODY></HTML>
```

❖ 필터 체인의 방향 바꾸기

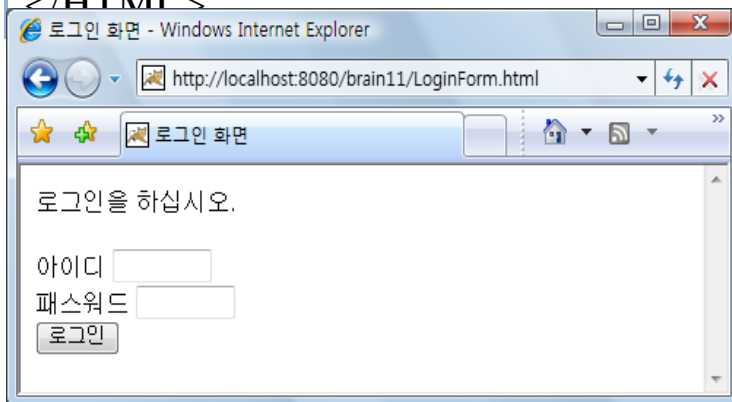
아이디와 패스워드를 받아서 로그인 처리를 하는 JSP 페이지

```
<%@page contentType= "text/html; charset=euc-kr"%>
<%
    String id = request.getParameter( "ID " );
    String password = request.getParameter( "PASSWORD " );
    String message;
    if (id.equals( "duke " ) && password.equals( "1234 " )) {
        session.setAttribute( "ID ", id );
        message = "로그인되었습니다. " ;
    }
    else {
        message = "등록되지 않은 아이디 또는 패스워드입니다. " ;
    }
%>
<HTML>
    <HEAD><TITLE>로그인 결과</TITLE></HEAD>
    <BODY>
        <%= message %>
    </BODY>
</HTML>
```

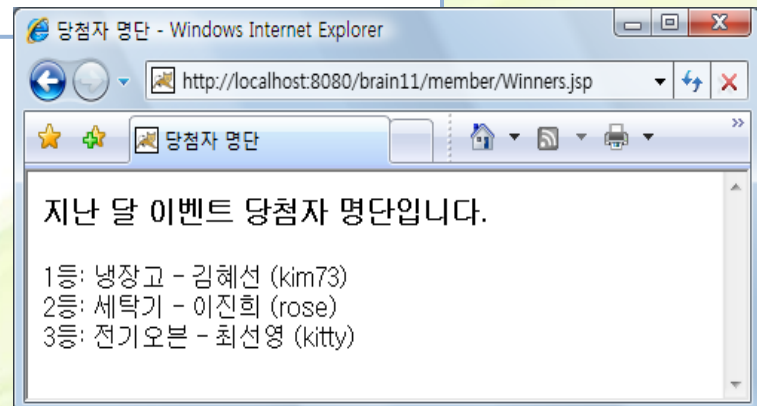
❖ 필터 체인의 방향 바꾸기

필터 클래스를 테스트하기 위한 JSP 페이지

```
<%@page contentType= "text/html; charset=euc-kr" %>
<HTML>
  <HEAD><TITLE>당첨자 명단</TITLE></HEAD>
  <BODY>
    <H3>지난 달 이벤트 당첨자 명단입니다.</H3>
    1등: 냉장고 - 김혜선 (kim73) <BR>
    2등: 세탁기 - 이진희 (rose) <BR>
    3등: 전기오븐 - 최선영 (kitty) <BR>
  <% session.invalidate() %>
  </BODY>
</HTML>
```



실행 결과 - 로그인하기 전



실행 결과 - 로그인한 다음

래퍼 클래스 작성, 설치, 사용

- 웹 브라우저와 웹 컴포넌트 사이를 오가는 데이터에 변형을 가하려면 필터 클래스와 더불어 래퍼 클래스를 작성해야 한다.
- 래퍼 클래스(wrapper class)란 글자 그대로 포장하는 역할을 하는 클래스이다.
- 래퍼 클래스의 종류는 다음 두 가지이다.
 - 요청 객체를 포장하는 요청 래퍼 클래스
 - 응답 객체를 포장하는 응답 래퍼 클래스
- 이 두 종류의 래퍼 클래스를 작성할 때는 각각 다음의 두 클래스를 상속하도록 만들어야 한다.
 - javax.servlet.http.HttpServletRequestWrapper 클래스
 - javax.servlet.http.HttpServletResponseWrapper 클래스

❖ 요청 래퍼 클래스를 작성하는 방법

- 요청 래퍼 클래스는 `HttpServletRequestWrapper` 클래스를 상속받아야 하므로 다음과 같은 골격을 만드는 것으로 클래스 작성을 시작해야 한다.

```
public class MyRequestWrapper extends HttpServletRequestWrapper {  
}
```

↑
프로그래머가 정하는
요청 래퍼 클래스의 이름

↑
요청 래퍼 클래스가
상속해야 하는 클래스

- 요청 래퍼 클래스의 가장 기본적인 역할인 요청 객체를 포장하는 일은 생성자를 통해서 할 수 있다. 즉, 요청 객체를 파라미터로 받는 생성자를 선언해놓고, 그 안에서 파라미터 값을 필드(field, 클래스의 멤버 변수)에 저장해 놓으면 된다.

```
public class MyRequestWrapper extends HttpServletRequestWrapper {  
    private HttpServletRequest request;  
    public MyRequestWrapper(HttpServletRequest request) {  
        this.request = request;  
    }  
}
```

생성자 안에서 파라미터로 받은 요청 객체를
필드에 저장합니다

❖ 요청 래퍼 클래스를 작성하는 방법

- 요청 래퍼 클래스의 생성자 안에서는 반드시 `HttpServletRequestWrapper` 클래스의 생성자를 호출해야 하고, 요청 객체를 파라미터로 넘겨줘야 한다.

```
public class MyRequestWrapper extends HttpServletRequestWrapper {  
    HttpServletRequest request;  
    public ParamUpperCaseRequestWrapper(HttpServletRequest request) {  
        super(request);  
        this.request = request;  
    }  
}
```

← 슈퍼 클래스의 생성자에 요청 객체를 넘겨줍니다.

- 요청 데이터를 변형하는 일은 웹 컴포넌트가 입력 데이터를 가져올 때 호출하는 메서드와 똑같은 시그니처의 메서드를 만들어서 그 안에서 해야 한다.

```
public class MyRequestWrapper extends HttpServletRequestWrapper { ...  
    public String getParameter(String name) {  
    }  
}
```

← HTML 문서의 <FORM> 엘리먼트를 통해 입력된 데이터를 변형하는 코드는 여기에 써넣어야 합니다.

- 나중에 웹 컴포넌트는 입력 데이터를 가져오기 위해 이 메서드를 호출하고, 그 결과 반영된 데이터를 받게 될 것이다.

❖ 요청 래퍼 클래스를 작성하는 방법

- 요청 데이터를 변형하는 메서드는 웹 컴포넌트의 기대에 맞추어서 만들어야 한다. 예를 들어 `getParameter` 메서드는 입력 데이터 이름을 파라미터로 받고, 그에 해당하는 데이터 값을 리턴해야 한다.

```
public class MyRequestWrapper extends HttpServletRequestWrapper {  
    HttpServletRequest request;  
  
    ...  
    public String getParameter(String name) {  
        String value = request.getParameter(name);  
        String newValue = value.toUpperCase();  
        return newValue;  
    }  
}
```

요청 객체(request)를 이용해서
입력 데이터를 가져옵니다

가져온 입력 데이터를
변형합니다.

변형된 결과를 리턴합니다

❖ 요청 래퍼 클래스를 작성하는 방법

- 필터 클래스의 doFilter 메서드 안에서는 요청 객체를 요청 래퍼 객체 바꾸는 일과, 그 요청 래퍼 객체를 가지고 chain.doFilter 메서드를 호출하는 일을 해야 한다.

```
public class MyFilter implements Filter {  
    ...  
    public void doFilter(ServletRequest request, ServletResponse response, FilterChain chain) ... {  
        MyRequestWrapper requestWrapper = new MyRequestWrapper((HttpServletRequest)  
request);  
        chain.doFilter(requestWrapper, response);  
    }  
}
```

요청 래퍼 객체

요청 객체를 이용해서
요청 래퍼 객체를
생성합니다.

❖ 요청 래퍼 클래스를 작성하는 방법

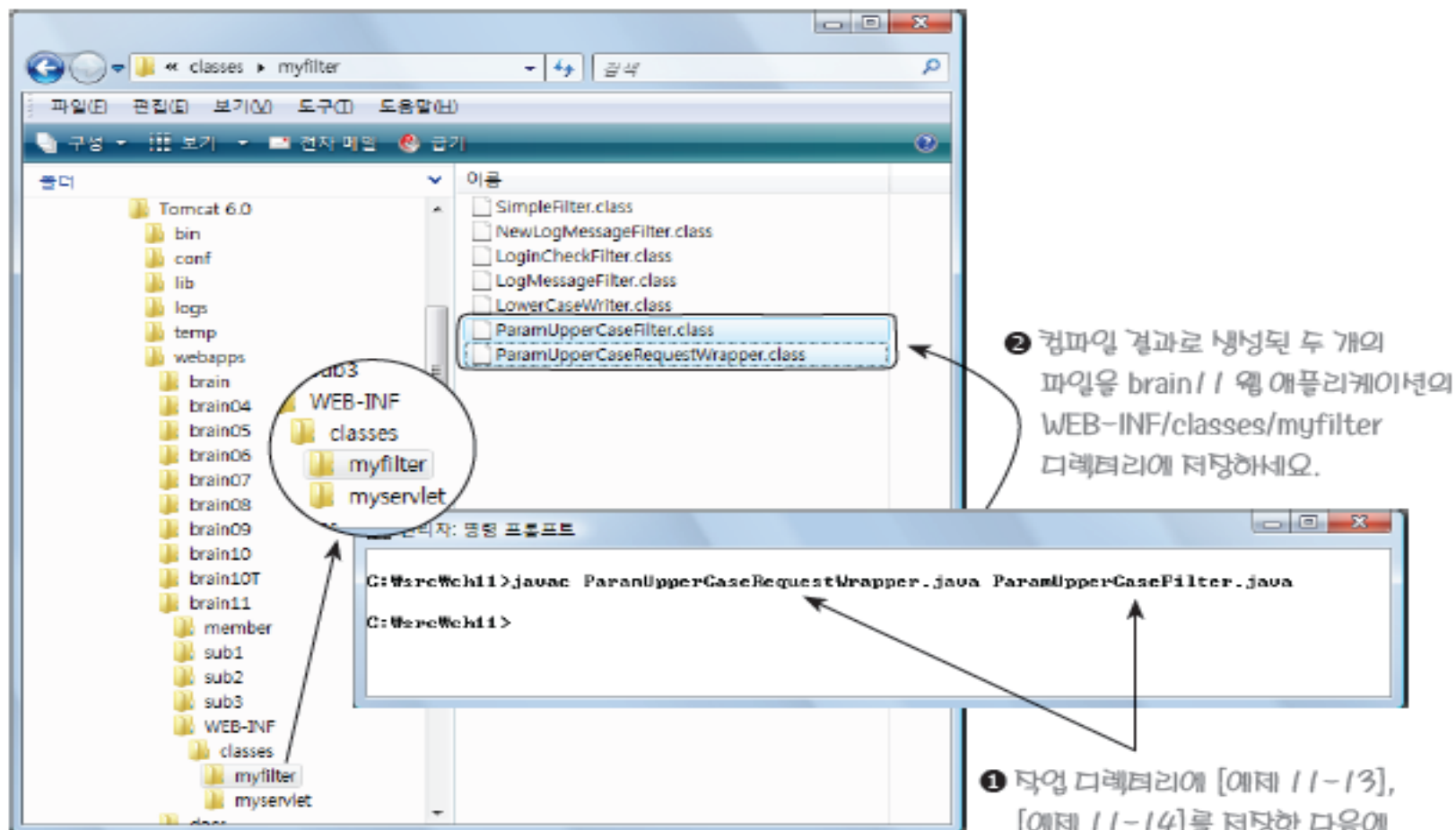
FORM 데이터의 모든 소문자를 대문자로 바꾸는 요청 래퍼 클래스 - 미완성

```
package myfilter; import javax.servlet.http.*; import java.io.*;
public class ParamUpperCaseRequestWrapper extends HttpServletRequestWrapper {
    HttpServletRequest request;
    public ParamUpperCaseRequestWrapper(HttpServletRequest request) {
        super(request);        this.request = request;
    }
    public String getParameter(String name) {
        String str = request.getParameter(name);
        if (str == null)        return null;
        return str.toUpperCase();
    }
}
```

요청 래퍼 클래스를 사용하는 필터 클래스

```
package myfilter; import javax.servlet.http.*; import javax.servlet.*; import java.io.*;
public class ParamUpperCaseFilter implements Filter {
    public void init(FilterConfig config) throws ServletException {
    }
    public void doFilter(ServletRequest request, ServletResponse response, FilterChain chain) throws
IOException, ServletException {
        ParamUpperCaseRequestWrapper requestWrapper = new
ParamUpperCaseRequestWrapper((HttpServletRequest) request);
        chain.doFilter(requestWrapper, response);
    }
    public void destroy() {    }
}
```

❖ 요청 래퍼 클래스를 작성하는 방법

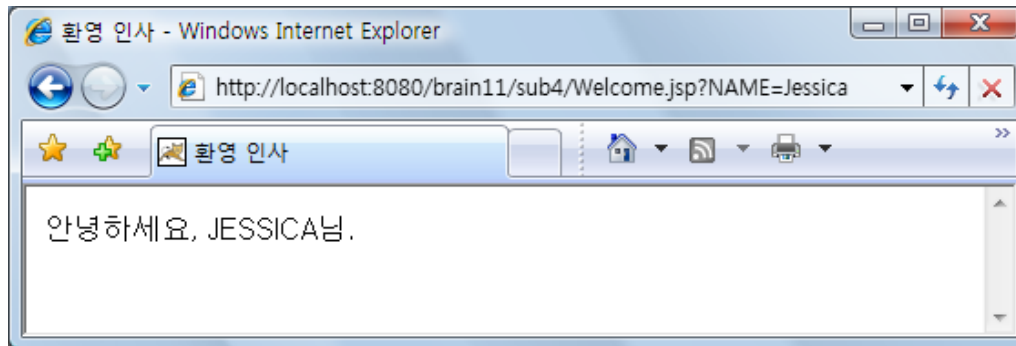


[그림 11-29] 예제 11-14, 예제 11-15를 컴파일하고 설치하는 방법

❖ 요청 래퍼 클래스를 작성하는 방법

테스트하기 위한 JSP 페이지 (1)

```
<%@page contentType= "text/html; charset=euc-kr" %>
<% String name = request.getParameter( "NAME" ); %>
<HTML>
  <HEAD><TITLE>환영 인사</TITLE></HEAD>
  <BODY>
    안녕하세요, <%= name %>님.
  </BODY>
</HTML>
```



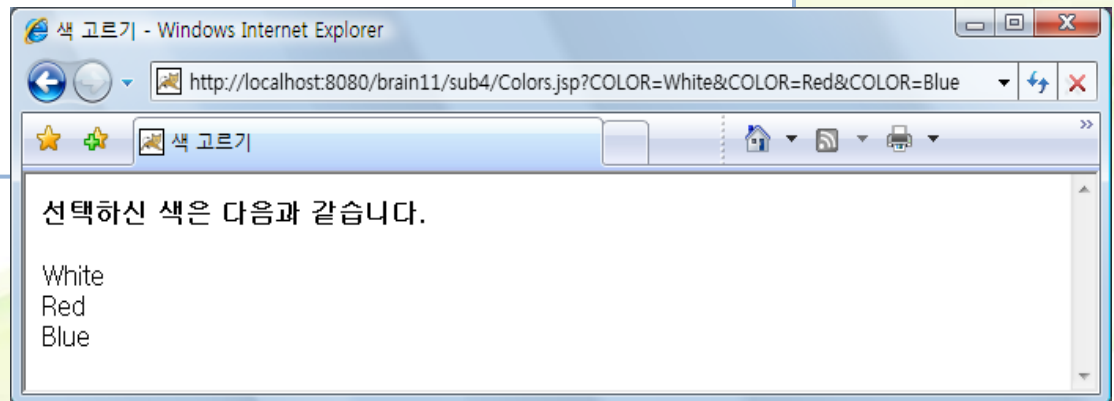
실행 결과

❖ 요청 래퍼 클래스를 작성하는 방법

- 문제점 - 서블릿 클래스나 JSP 페이지에서 <FORM> 데이터를 가져올 때에는 getParmeter 메서드뿐만 아니라 getParameterValues나 getParameterMap 메서드를 사용할 수도 있으므로 다음과 같은 예제는 변형된 입력 데이터를 받지 못한다.

테스트하기 위한 JSP 페이지 (2)

```
<%@page contentType= "text/html; charset=euc-kr "%>
<% String color[] = request.getParameterValues( "COLOR " ); %>
<HTML>
  <HEAD><TITLE>색 고르기</TITLE></HEAD>
  <BODY>
    <H4>선택하신 색은 다음과 같습니다.</H4>
    <%
      if (color != null) {
        for (int cnt = 0; cnt < color.length; cnt++)
          out.println(color[cnt] + "<BR> ");
      }
    %>
  </BODY>
</HTML>
```

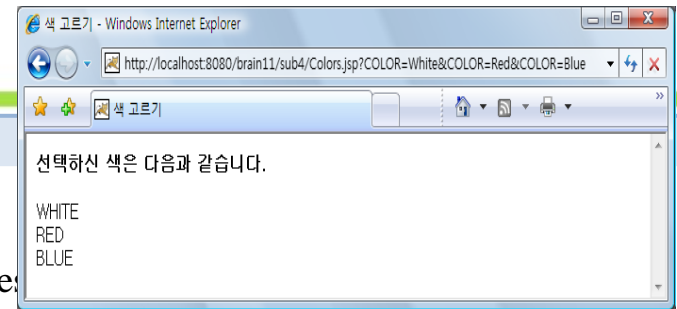


실행 결과 (1)

❖ 요청 래퍼 클래스를 작성하는 방법

<FORM> 데이터의 모든 소문자를 대문자로 바꾸는 요청 래퍼 클래스 - 완성

```
package myfilter; import javax.servlet.http.*;
import java.io.*; import java.util.*;
public class ParamUpperCaseRequestWrapper extends HttpServletRequestWrapper {
    HttpServletRequest request;
    public ParamUpperCaseRequestWrapper(HttpServletRequest request) {
        super(request);        this.request = request;
    }
    public String getParameter(String name) {
        String str = request.getParameter(name);
        if (str == null)        return null;
        return str.toUpperCase();
    }
    public String[] getParameterValues(String name) {
        String str[] = request.getParameterValues(name);
        if (str == null)        return null;
        for (int cnt = 0; cnt < str.length; cnt++)        str[cnt] = str[cnt].toUpperCase();
        return str;
    }
    public Map getParameterMap() {
        Map map = request.getParameterMap();
        HashMap<String, String[]> newMap = new HashMap<String, String[]>();
        Object name[] = map.keySet().toArray();
        for (int cnt = 0; cnt < name.length; cnt++) {
            String value[] = (String[]) map.get(name[cnt]);
            for (int index = 0; index < value.length; index++) value[index] = value[index].toUpperCase();
            newMap.put((String) name[cnt], value);
        }
        return newMap;
    }
}
```



[실행 결과 (2)]

❖ 응답 래퍼 클래스를 작성하는 방법

- 응답 래퍼 클래스는 다음과 같이 HttpServletResponseWrapper 클래스를 상속하도록 만들어야 한다.

```
public class MyResponseWrapper extends HttpServletResponseWrapper {  
}
```

프로그래머가 정하는
응답 래퍼 클래스의 이름

응답 래퍼 클래스가
상속해야 하는 클래스

- 응답 래퍼 클래스도 요청 래퍼 클래스와 마찬가지로 생성자 안에서 슈퍼클래스의 생성자를 호출하고, 응답 객체를 포장하는 일을 해야 한다

```
public class MyResponseWrapper extends HttpServletResponseWrapper {  
    HttpServletResponse response;  
    public ParamUpperCaseResponseWrapper(HttpServletResponse response) {  
        super(response);  
        this.response = response;  
    }  
}
```

생성자가 파라미터로
받은 응답 객체를
필드에 저장합니다

슈퍼클래스의 생성자를 호출하면서
응답 객체를 넘겨줍니다

❖ 응답 래퍼 클래스를 작성하는 방법

- 응답 데이터를 변형하는 코드는 웹 컴포넌트에서 해당 데이터를 출력할 때 사용하는 것과 동일한 시그니처의 메서드 안에 써넣어야 한다.
- 예를 들어 쿠키 데이터를 변형하는 코드는 다음과 같이 addCookie 메서드 안에 써넣어야 한다.

```
public class MyRequestWrapper extends HttpServletRequestWrapper {  
    HttpServletRequest request;  
    ...  
    public void addCookie(Cookie cookie) {  
        String name = cookie.getName();  
        String value = cookie.getValue();  
        String new Value = value.toLowerCase();  
        Cookie newCookie = new Cookie(name, new Value);  
        response.addCookie(newCookie);  
    }  
}
```

파라미터로 주어진 쿠키 객체로부터
쿠키 이름과 쿠키 값을 가져옵니다

쿠키 값을 변형합니다

변형된 쿠키 값을 가지고 새로운 쿠키 객체를 생성합니다

응답 객체를 통해 새로운 쿠키 객체를 전송합니다

- 이렇게 해놓으면 웹 컴포넌트는 이 메서드를 응답 객체의 addCookie 메서드인 줄로 알고 호출할 것이고, 그 결과 쿠키에 포함된 데이터가 변형된 후에 웹 브라우저로 전달될 것이다

❖ 응답 래퍼 클래스를 작성하는 방법

- 필터 클래스의 doFilter 메서드 안에서는 응답 객체를 응답 래퍼 객체 바꾸는 일과, 그 응답 래퍼 객체를 가지고 chain.doFilter 메서드를 호출하는 일을 해야 한다.

```
public class MyFilter implements Filter {  
    ...  
    public void doFilter(ServletRequest request, ServletResponse response, FilterChain chain) ... {  
        MyResponseWrapper responseWrapper = new  
        MyResponseWrapper((HttpServletResponse) response);  
        chain.doFilter(request, responseWrapper);  
    }  
}
```

응답 래퍼 객체

응답 객체를 이용해서
응답 래퍼 객체를 생성합니다

❖ 응답 래퍼 클래스를 작성하는 방법

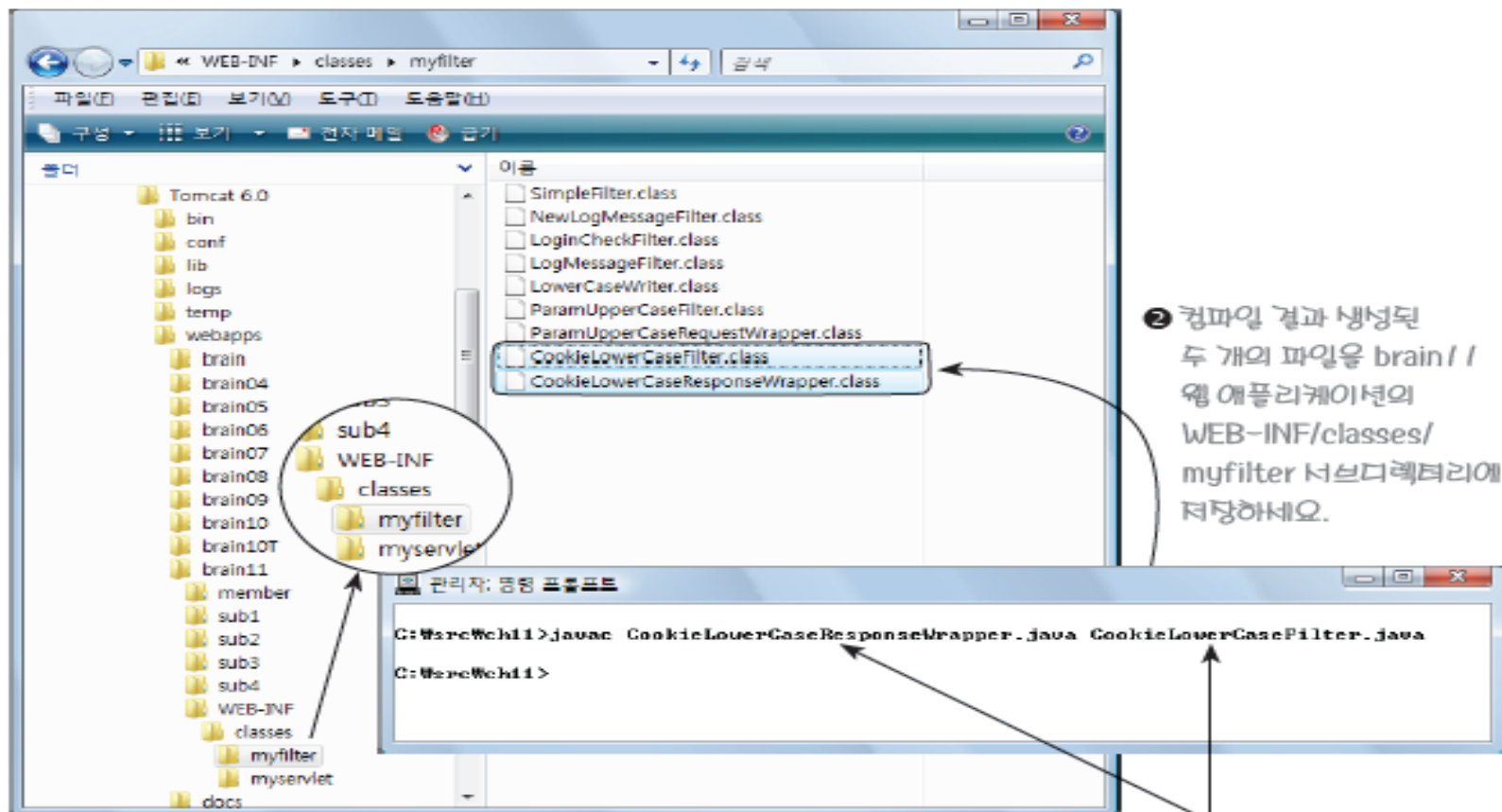
쿠키 데이터에 포함된 모든 대문자를 소문자로 바꾸는 응답 래퍼 클래스

```
package myfilter; import javax.servlet.http.*; import javax.servlet.*; import java.io.*;
public class CookieLowerCaseResponseWrapper extends HttpServletResponseWrapper {
    private HttpServletResponse response;
    public CookieLowerCaseResponseWrapper(HttpServletResponse response) {
        super(response);        this.response = response;
    }
    public void addCookie(Cookie cookie) {
        String value = cookie.getValue();
        String newValue = value.toLowerCase();
        cookie.setValue(newValue);
        response.addCookie(cookie);
    }
}
```

응답 래퍼 클래스를 사용하는 필터 클래스

```
package myfilter; import javax.servlet.http.*; import javax.servlet.*; import java.io.*;
public class CookieLowerCaseFilter implements Filter {
    public void init(FilterConfig config) throws ServletException {
    }
    public void doFilter(ServletRequest request, ServletResponse response, FilterChain chain) throws
IOException, ServletException {
        CookieLowerCaseResponseWrapper responseWrapper
=CookieLowerCaseResponseWrapper((HttpServletResponse) response);
        chain.doFilter(request, responseWrapper);
    }
    public void destroy() {    }
}
```

❖ 응답 래퍼 클래스를 작성하는 방법

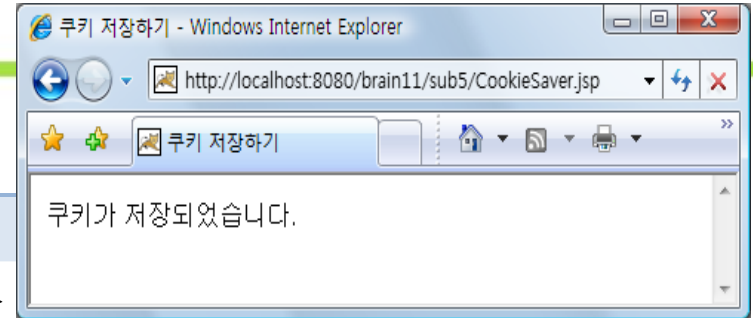


[그림 11-34] 예제 11-19, 예제 11-20을 컴파일하고 설치하는 방법

❖ 응답 래퍼 클래스를 작성하는 방법

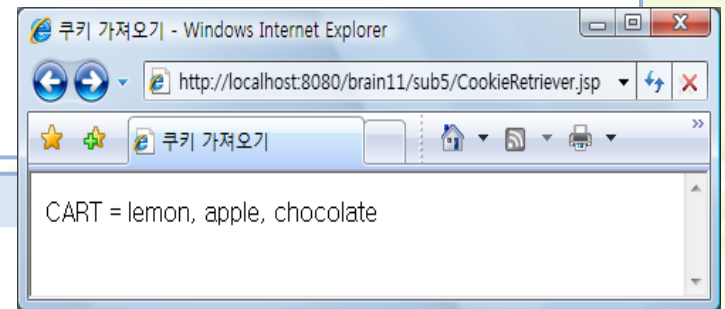
테스트하기 위한 JSP 페이지 (1)

```
<%@page contentType= "text/html; charset=euc-kr"%>
<%
    Cookie cookie = new Cookie( "CART ", "Lemon, Apple, Chocolate ");
    response.addCookie(cookie);
%>
<HTML>
    <HEAD><TITLE>쿠키 저장하기</TITLE></HEAD>
    <BODY>
        쿠키가 저장되었습니다.<br>
        <a href= "CookieRetriever.jsp">확인</a>
    </BODY>
</HTML>
```



테스트하기 위한 JSP 페이지 (2)

```
<%@page contentType= "text/html; charset=euc-kr"%>
<HTML>
    <HEAD><TITLE>쿠키 가져오기</TITLE></HEAD>
    <BODY>
        CART = ${cookie.CART.value}
    </BODY>
</HTML>
```



❖ 응답 메시지의 본체 내용을 변형하는 래퍼 클래스

- 이번에는 HTTP 응답 메시지의 본체(body) 내용, 즉 HTML 코드 부분을 변형하는 응답 래퍼 클래스를 만들어 보자
- 응답 메시지의 본체 내용을 출력하는 메커니즘은 쿠키 데이터를 출력하는 메커니즘보다 복잡하기 때문에 이런 래퍼 클래스의 작성 방법도 복잡하다. 예를 들어 서블릿 클래스에서 HTML 코드를 출력하기 위해서는 다음과 같은 코드를 작성해야 한다.

```
public class SimpleServlet extends HttpServlet {  
    public void doGet(HttpServletRequest request, HttpServletResponse response) ... {  
        response.setContentType( "text/html;charset=euc-kr ");  
        PrintWriter out = response.getWriter();  
        out.println( "<HTML> ");  
        out.println( "<HEAD><TITLE>1부터 100까지의 합</TITLE></HEAD> ");  
        out.println( "<BODY> ");  
        out.printf( "1부터 100까지의 합 = %d " , total);  
        out.println( "</BODY> ");  
        out.println( "</HTML> ");  
    }  
}
```

PrintWriter 객체를 얻습니다.

PrintWriter 객체를 이용해서 HTML을 출력합니다.

❖ 응답 메시지의 본체 내용을 변형하는 래퍼 클래스

- `StringWriter` 클래스는 `java.io` 패키지에 속하며, 텍스트를 이 객체 내부에 있는 버퍼로 출력하는 기능을 갖는다. 버퍼의 내용은 `toString` 메서드를 이용해서 가져올 수 있다.

```
StringWriter writer = new StringWriter();  
writer.write( "<BODY> ");  
writer.write( "헬로 " );  
writer.write( "</BODY> ");  
String str = writer.toString();
```

StringWriter 객체를 생성한 다음에
그 객체를 이용해서 출력한 데이터는
toString 메서드를 이용해서 가져올 수 있습니다

- `PrintWriter` 객체로 출력되는 내용이 `StringWriter` 객체의 버퍼에 저장되도록 만들려면 `PrintWriter` 클래스의 생성자에 `StringWriter` 객체를 파라미터로 넘겨주면 된다.

```
StringWriter strWriter = new StringWriter();  
PrintWriter writer = new PrintWriter(strWriter);  
writer.println( "<BODY> ");  
writer.println( "1+...+100 = %d ", total);  
writer.println( "</BODY> ");  
String str = strWriter.toString();
```

StringWriter 객체를 이용해서
PrintWriter 객체를 생성한 다음에
PrintWriter 객체를 이용해서 출력한
데이터는
StringWriter 객체에 대해 toString 메서드를
호출해서 가져올 수 있습니다

❖ 응답 메시지의 본체 내용을 변형하는 래퍼 클래스

- 응답 메시지의 본체 내용을 변형하려면 `ServletResponse` 인터페이스의 `getWriter` 메서드와 동일한 시그니처를 갖는 메서드를 응답 래퍼 클래스 안에 선언한다. 그리고 그 메서드가 앞에서 설명한 방법대로 `PrintWriter` 객체를 만들어서 리턴하도록 만든다.

```
public class MyResponseWrapper extends HttpServletResponseWrapper {  
    StringWriter strWriter;  
    ...  
    public PrintWriter getWriter() throws IOException {  
        strWriter = new StringWriter();  
        PrintWriter newWriter = new PrintWriter(strWriter);  
        return newWriter;  
    }  
}
```

StringWriter 객체를 이용해서
PrintWriter 객체를 생성한 후
리턴합니다

- 이런 메서드를 선언해 놓으면 웹 컴포넌트는 `ServletResponse` 인터페이스의 `getWriter` 메서드인줄 알고 이 메서드를 호출할 것이고, 그 결과 웹 컴포넌트에서 출력하는 데이터는 위 코드에서 생성한 `StringWriter` 객체의 버퍼 안에 저장될 것이다.
- 하지만 위 코드는 `getWriter` 메서드를 호출할 때마다 새로운 `PrintWriter` 객체를 생성해서 리턴한다는 문제점이 있다.

❖ 응답 메시지의 본체 내용을 변형하는 래퍼 클래스

- 앞 페이지 코드의 문제점을 해결하기 위해서는 `getWriter` 메서드가 두 번 이상 호출될 때 전에 만들어 두었던 `PrintWriter` 객체를 리턴하도록 수정해야 한다.

```
public class MyResponseWrapper extends HttpServletResponseWrapper {  
    StringWriter strWriter;  
    PrintWriter newWriter;  
    ...  
    public PrintWriter getWriter() throws IOException {  
        if (newWriter == null) {  
            strWriter = new StringWriter();  
            newWriter = new PrintWriter(strWriter);  
        }  
        return newWriter;  
    }  
}
```

PrintWriter 객체가 없을
경우에만 새로 생성합니다

❖ 응답 메시지의 본체 내용을 변형하는 래퍼 클래스

- HTML 코드를 변형하는 코드는 필터 클래스의 다음 위치에서 실행되도록 만들어야 한다.

```
public class MyFilter implements Filter {  
    ...  
    public void doFilter(ServletRequest request, ServletResponse response,  
FilterChain chain) ... {  
        MyResponseWrapper responseWrapper = new  
MyResponseWrapper(response);  
        chain.doFilter(request, responseWrapper);  
        ...  
    }  
    ...  
}
```

HTML 코드를 변형하는 코드는
이 위치에서 실행되도록 만들어야 합니다.

❖ 응답 메시지의 본체 내용을 변형하는 래퍼 클래스

- 예를 들어 HTML 코드 중에 있는 “강아지”라는 문자열을 “멍멍이”로 교체하는 데이터 변형 코드는 다음과 같이 작성하면 된다.

필터 클래스

```
public class MyFilter implements Filter {  
    ...  
    public void doFilter(ServletRequest request, ServletResponse response, FilterChain chain) ... {  
        MyResponseWrapper responseWrapper = new MyResponseWrapper(response);  
        chain.doFilter(request, responseWrapper);  
        responseWrapper.modifyAndSend();  
    }    ...  
}
```

호출

응답 래퍼 클래스

```
public class MyResponseWrapper extends HttpServletResponseWrapper {  
    HttpServletResponse response;    StringWriter strWriter;  
    ...  
    public void modifyAndSend() throws IOException {  
        String content = strWriter.toString();  
        String newContent = content.replaceAll(“강아지”, “멍멍이”);  
        PrintWriter writer = response.getWriter();  
        writer.print(newContent);  
    }  
    ...  
}
```

버퍼에 있는 HTML 코드를 가져옵니다

HTML 코드를 변형합니다.

변형된 HTML 코드를 응답 객체를 이용해서
웹 브라우저로 전송합니다

❖ 응답 메시지의 본체 내용을 변형하는 래퍼 클래스

응답 메시지에 포함된 특정 단어를 치환하는 응답 래퍼 클래스

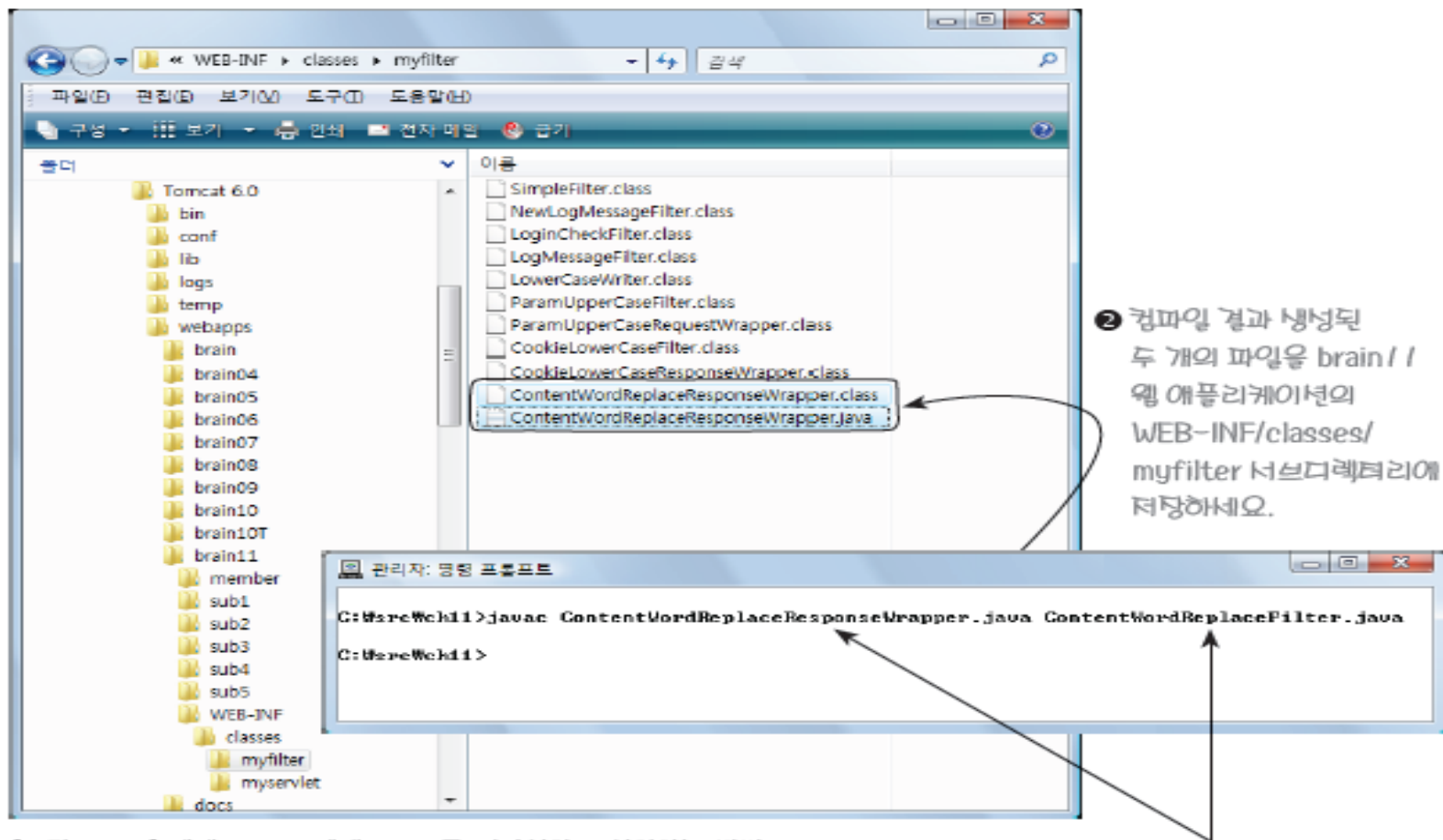
```
package myfilter; import javax.servlet.http.*; import javax.servlet.*; import java.io.*;
public class ContentWordReplaceResponseWrapper extends HttpServletResponseWrapper {
    HttpServletResponse response;
    PrintWriter newWriter;
    StringWriter strWriter;
    public ContentWordReplaceResponseWrapper(HttpServletResponse response) {
        super(response);
        this.response = response;
    }
    public PrintWriter getWriter() throws IOException {
        if (newWriter == null) {
            strWriter = new StringWriter();
            newWriter = new PrintWriter(strWriter);
        }
        return newWriter;
    }
    public void modifyAndSend() throws IOException {
        String content = strWriter.toString();
        String newContent = content.replaceAll( “강아지 ” , “멍멍이 ” );
        PrintWriter writer = response.getWriter();
        writer.print(newContent);
    }
}
```

❖ 응답 메시지의 본체 내용을 변형하는 래퍼 클래스

응답 래퍼 클래스를 사용하는 필터 클래스

```
package myfilter;
import javax.servlet.http.*;
import javax.servlet.*;
import java.io.*;
public class ContentWordReplaceFilter implements Filter {
    public void init(FilterConfig config) throws ServletException {
    }
    public void doFilter(ServletRequest request, ServletResponse response, FilterChain chain)
throws IOException, ServletException {
        ContentWordReplaceResponseWrapper responseWrapper =
            new
ContentWordReplaceResponseWrapper( (HttpServletResponse) response);
        chain.doFilter(request, responseWrapper);
        responseWrapper.modifyAndSend();
    }
    public void destroy() {
    }
}
```

❖ 응답 메시지의 본체 내용을 변형하는 래퍼 클래스



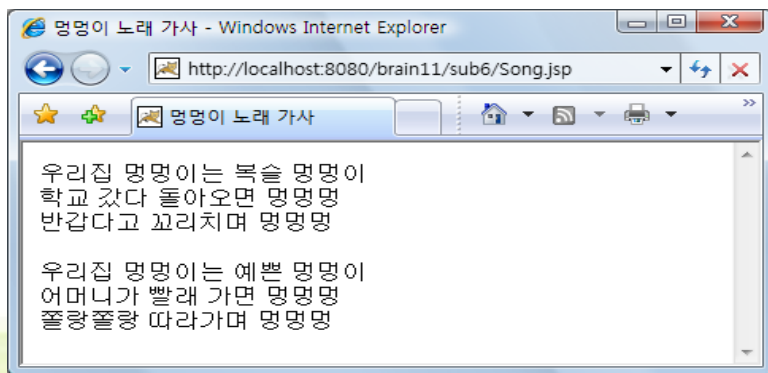
[그림 11-38] 예제 11-23, 예제 11-24를 컴파일하고 설치하는 방법

① 작업 디렉터리에 [예제 11-23], [예제 11-24]을 저장한 다음에 이렇게 한꺼번에 컴파일하세요.

❖ 응답 메시지의 본체 내용을 변형하는 래퍼 클래스

테스트하기 위한 JSP 페이지

```
<%@page contentType= "text/html; charset=euc-kr" %>
<HTML>
  <HEAD><TITLE>강아지 노래 가사</TITLE></HEAD>
  <BODY>
    우리집 강아지는 복슬 강아지 <BR>
    학교 갔다 돌아오면 멍멍멍 <BR>
    반갑다고 꼬리치며 멍멍멍 <BR>
    <P>
    우리집 강아지는 예쁜 강아지 <BR>
    어머니가 빨래 가면 멍멍멍 <BR>
    짹짹 따라가며 멍멍멍 <BR>
  </BODY>
</HTML>
```



실행 결과