# 05: 속성과 리스너

## 학습 목표

- 초기화 파라미터 외 파라미터에 대해 이해한다.
- 리스너를 생성해보고 사용에 대해 이해한다.
- 속성의 생명범위(Scope)에 대해 살펴보고, 속성 사용에 대해 이해한다.

1.파라미터

2.리스터

3.속성

# 파라미터 >> 초기화 파라미터

DD(web.xml) 파일에서:

```
<servlet>
    <servlet-name>BeerParamTests</servlet-name>
    <servlet-class>TestInitParams</servlet-class>

    <init-param>
      <param-name>adminEmail</param-name>
      <param-value>likewecare@wickedlysmart.com</param-value>
    </init-param>

</servlet>
```

DD의 <servlet> 항목 안에 <param-name>과 <param-value>를 작성하면 됩니다.

서블릿 코드에서:

```
out.println(getServletConfig().getInitParameter("adminEmail"));
```

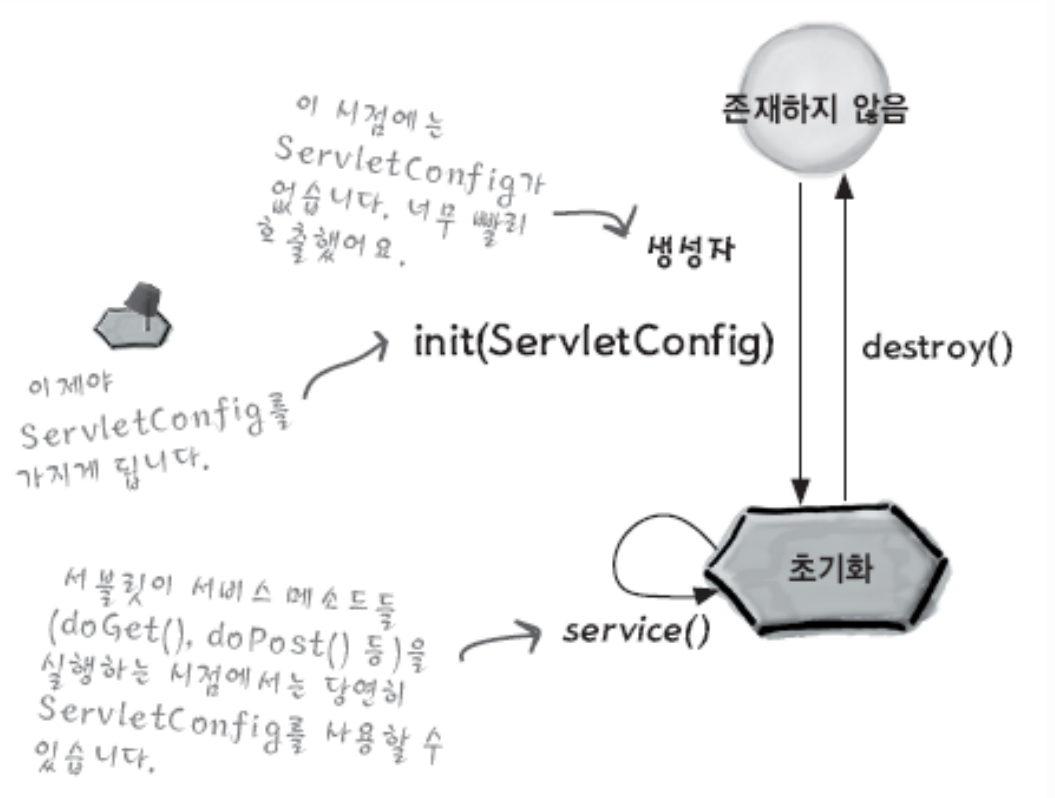
모든 서블릿에는 상속받은 getServletConfig()가 있습니다.

getServletConfig() 메소드의 리턴값은 ...음...뭐였더라... 잠시만요... 커닝 좀 하고... 아 예... ServletConfig입니다. ServletConfig에는 getInitParameter() 메소드가 있습니다.

# 파라미터 >> 초기화 파라미터

## ■ 서블릿 초기화가 된 다음에 초기화 파라미터를 사용할 수 있다

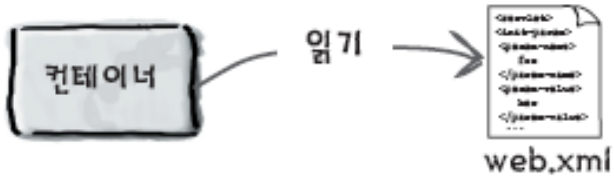
- 컨테이너는 DD에서 서블릿 초기화 파라미터를 읽어, 이 정보를 ServletConfi 넘겨준다.
  그 다음 ServletConfig를 서블릿의 init() 메소드에 제공



# 파라미터 >> 초기화 파라미터 로딩

■ 서블릿 초기화할 때 단 한번만 서블릿 초기화 파라미터를 읽는다

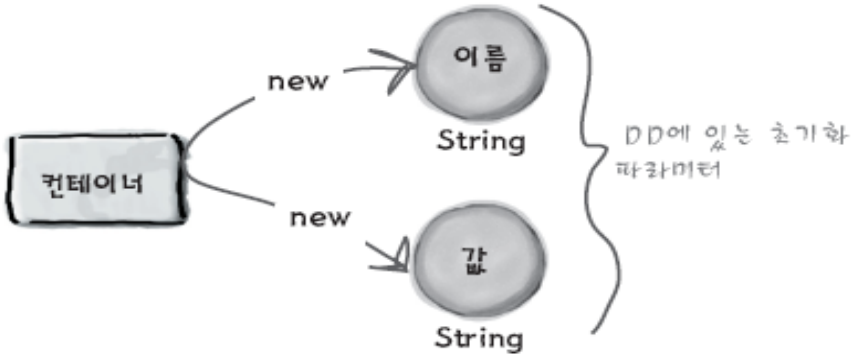
① 컨테이너는 배포 서술자(DD)를 읽습니다. 물론 초기화 파라미터((init-param))도 읽겠지요.



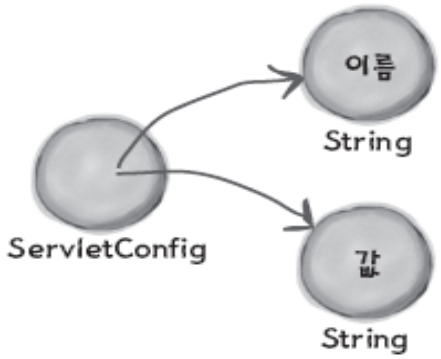

② 컨테이너는 새로운 ServletConfig 인스턴스를 만듭니다(서블릿당 하나씩 만듭니다).

# 파라미터 >> 초기화 파라미터 로딩

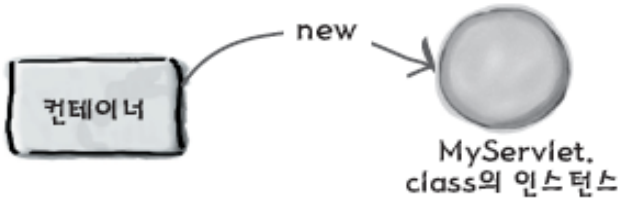
③ 컨테이너는 초기화 파라미터에 있는 값들을 이름/값의 쌍의 형식으로 읽어들입니다. 여기서는 하나의 쌍만 있다고 가정해봅시다.



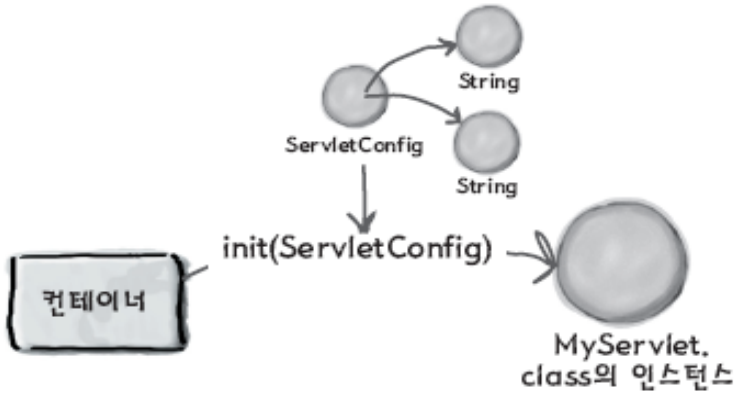

④ 컨테이너는 ServletConfig 객체에 이름/값으로 된 초기화 파라미터를 설정합니다.




⑤ 컨테이너는 서블릿 클래스 인스턴스를 생성합니다.




⑥ 컨테이너는 ServletConfig의 참조를 인자로 서블릿의 init() 메소드를 호출합니다.



# 파라미터 >> ServletConfig 사용하기

## ServletConfig

- 초기화 파라미터 사용
- 컨텍스트 정보 (서블릿 설정 정보) 사용

# 파라미터 >> ServletConfig 사용하기

DD 파일(web.xml):

```
<web-app xmlns="http://java.sun.com/xml/ns/j2ee"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee/web-app_2_4.xsd"
    version="2.4">
  <servlet>
    <servlet-name>BeerParamTests</servlet-name>
    <servlet-class>com.example.TestInitParams</servlet-class>
    <init-param>
      <param-name>adminEmail</param-name>
      <param-value>likewecare@wickedlysmart.com</param-value>
    </init-param>
    <init-param>
      <param-name>mainEmail</param-name>
      <param-value>blooper@wickedlysmart.com</param-value>
    </init-param>
  </servlet>
  <servlet-mapping>
    <servlet-name>BeerParamTests</servlet-name>
    <url-pattern>/Tester.do</url-pattern>
  </servlet-mapping>
</web-app>
```

# 파라미터 >> ServletConfig 사용하기

**서블릿 코드:**

```
package com.example;
import javax.servlet.*;
import javax.servlet.http.*;
import java.io.*;

public class TestInitParams extends HttpServlet {

  public void doGet(HttpServletRequest request, HttpServletResponse response)
                                                    throws IOException, ServletException {
        response.setContentType("text/html");
        PrintWriter out = response.getWriter();
        out.println("test init parameters<br>");

        Enumeration e = getServletConfig().getInitParameterNames();
        while(e.hasMoreElements()) {
           out.println("<br>param name = " + e.nextElement() + "<br>");
        }
        out.println("main email is " + getServletConfig().getInitParameter("mainEmail"));
        out.println("<br>");
        out.println("admin email is " + getServletConfig().getInitParameter("adminEmail"));
    }
}
```

# 파라미터 >> 컨텍스트 초기화 파라미터

■ 모든 웹 애플리케이션에서 사용 가능

DD 파일(web.xml):

```
<servlet>
    <servlet-name>BeerParamTests</servlet-name>
    <servlet-class>TestInitParams</servlet-class>
</servlet>

<context-param>
    <param-name>adminEmail</param-name>
    <param-value>clientheaderror@wickedlysmart.com</param-value>
</context-param>
```

<servlet> 항목 안에 있던 <init-param>은 모두 들어 냈습니다.

<param-name>과 <param-value>는 서블릿 초기화 파라미터와 마찬가지로 이름/값의 쌍입니다. 단 이 항목은 <init-param> 항목이 아니라 <context-param>에 포함됩니다.

<context-param>은 전체 애플리케이션을 위한 항목입니다. 따라서 <servlet> 항목 안에 들어가지 않습니다. <context-param>을 <web-app> 항목에 포함시키되 <servlet> 항목 안에다 두지는 마세요.

# 파라미터 >> 컨텍스트 초기화 파라미터

**서블릿 코드:**

```
out.println(getServletContext().getInitParameter("adminEmail"));
```

모든 서블릿에는 상속받은 getServletContext() 메소드가 있습니다 (JSP도 마찬가지로 컨텍스트에 접근하는 특별한 방법이 있습니다).

getServletContext() 메소드의 리턴 값은 당연히 ServletContext입니다. ServletContext의 메소드 중 하나가 getInitParameter()입니다.

**또는:**

```
ServletContext context = getServletContext();
out.println(context.getInitParameter("adminEmail"));
```

한 행으로 작성해도 되는 코딩을 두 행으로 풀어 썼습니다. ServletContext 참조를 가지고 와서 getInitParameter() 메소드를 호출합니다.

# 파라미터 >> 초기화 파라미터와 컨텍스트 초기화 파라미터

## 차이점 1

| 컨텍스트 초기화 파라미터 | 서블릿 초기화 파라미터 |
|---|---|
| *배포 서술자* | |
| 〈web-app〉 항목 내에 작성해야지, 〈servlet〉 항목 안에 작성하면 안 됩니다. | 〈servlet〉 항목 안에 작성합니다. |

```
<web-app ...>
  <context-param>
    <param-name>foo</param-name>
    <param-value>bar</param-value>
  </context-param>

  <!-- 서블릿 선언을 포함해서
    기타 등등 -->
</web-app>
```

컨텍스트 초기화 파라미터는 DD 안에 어디에서도 초기화(init)라는 말은 없다는 것을 기억하세요.

```
<servlet>
  <servlet-name>
    BeerParamTests
  <servlet-name>
  <servlet-class>
    TestInitParams
  </servlet-class>
  <init-param>
    <param-name>foo</param-name>
    <param-value>bar</param-value>
  </init-param>

  <!-- 기타 등등 -->
</servlet>
```

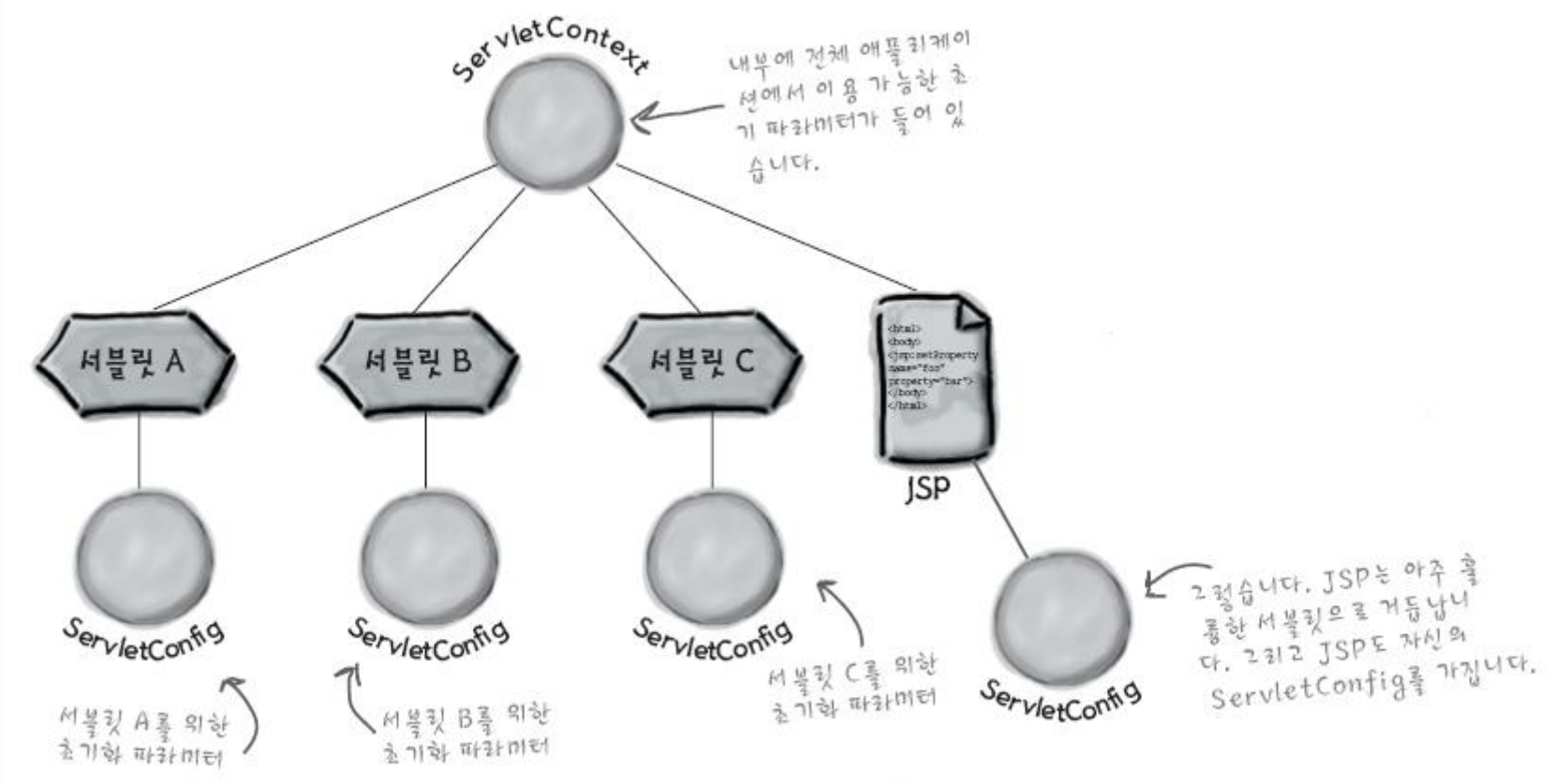
# 파라미터 >> 초기화 파라미터와 컨텍스트 초기화 파라미터

## ■ 차이점 2

| 컨텍스트 초기화 파라미터 | 서블릿 초기화 파라미터 |
|---|---|
| *서블릿 코드* | |
| `getServletContext().getInitParameter("foo");` | `getServletConfig().getInitParameter("foo");` |
| 둘 다 메소드 이름은 같습니다. | |
| *범위* | |
| 웹 애플리케이션에 존재하는 어떤 서블릿이나 JSP 모두 | <init-param> 항목을 포함하고 있는 서블릿만<br>(서블릿 코드 안에서 이 값을 속성(Attribute)에 저장하여 다른 컴포넌트에서도 이용 가능하게 만들 수 있습니다) |

# 파라미터 >> 초기화 파라미터와 컨텍스트 초기화 파라미터

## ■ 정리



# 리스너 >> 컨텍스트 리스너 (ServletContextListener)

## ■ 컨텍스트 초기화 시(애플리케이션 배포) 사용

- **ServletContext**로부터 컨텍스트 초기화 파라미터 로딩
- **DB** 연결을 위해 초기화 파라미터 검색명( **lookup name** ) 사용
- **DB Connection** 객체를 속성(**Attribute**)에 저장

## ■ 컨텍스트 종료 시(애플리케이션 서비스 종료) 사용

- **DB** 연결 종료

**ServletContextListener 클래스:**

ServletContextListener는 javax.servlet 패키지에 있습니다.

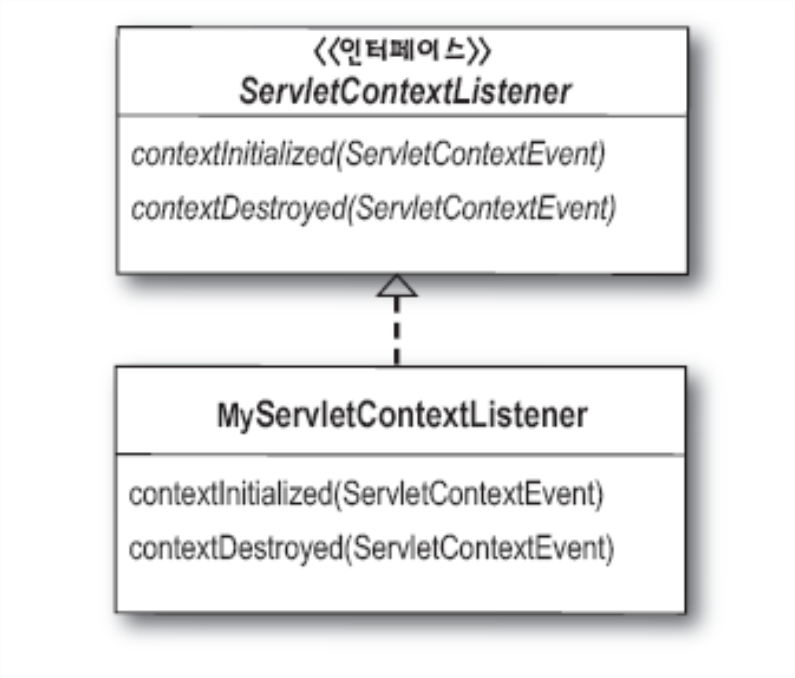
컨텍스트 리스너는 단순합니다: ServletContextListener만 구현하면 되니까.

```
import javax.servlet.*;

public class MyServletContextListener implements ServletContextListener {

    public void contextInitialized(ServletContextEvent event) {
      //여기에 데이터베이스 연결을 초기화하는 코딩을 합니다.
      //그리고 이를 컨텍스트 속성에 저장합니다.
    }

    public void contextDestroyed(ServletContextEvent event) {
      //여기에 데이터베이스 연결을 닫는 코딩을 합니다.
    }
}
```

두 가지 사실을 통보 받습니다. 두 경우 모두 ServletContextEvent를 인자로 넘겨줍니다.

# 리스너 >> ServletContextListener 만들기

■ 리스너 클래스 만들기 : **MyServletContextListener**



# 리스너 >> ServletContextListener 만들기

## ■ 클래스 배포


(클래스가 들어갈 수 있는 곳은 여기뿐만은 아닙니다. WEB-INF/classes는 컨테이너가 클래스를 찾는 곳 중 하나일 뿐이죠. 배포 장에서 위치에 대해 자세히 다루겠습니다.)

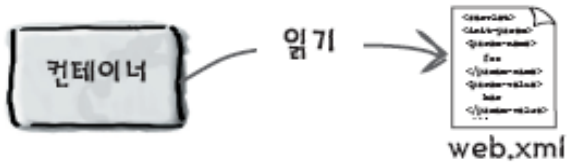
## ■ 배포 서술자 수정 : web.xml의 <web-app> 항목에 <listener> 항목 추가

```
<listener>
  <listener-class>
     com.example.MyServletContextListener
  </listener-class>
</listener>
```

# 리스너 >> ServletContextListener 실행순서

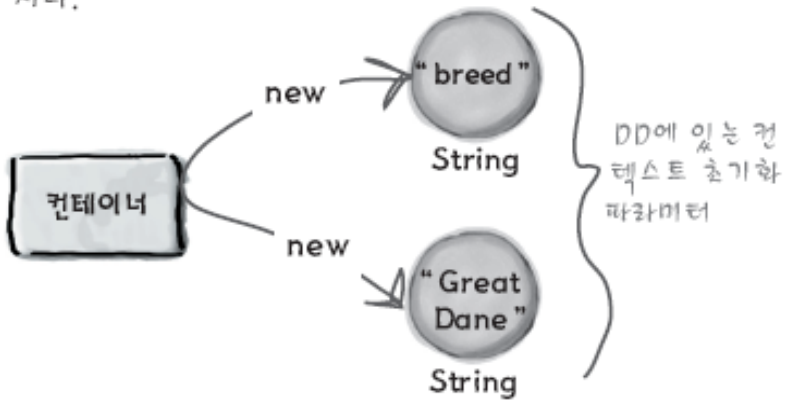
## ServletContextListener 실행 순서 (1)

① 컨테이너는 애플리케이션 배포 서술자를 읽습니다. 물론 〈listener〉 요소와 〈context-param〉 요소도 읽겠죠.

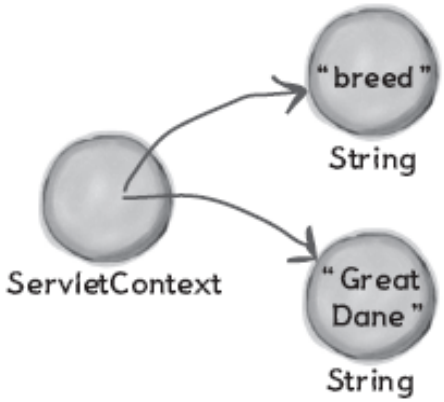



② 컨테이너는 ServletContext 객체를 생성합니다. 애플리케이션에서 이 객체를 공유하겠죠.


③ 컨테이너는 컨텍스트 초기화 파라미터의 이름/값 쌍(String의 쌍)을 만듭니다. 개수만큼 만들겠지만, 쉽게 설명하기 위해 여기서는 하나만 있다고 가정합시다.



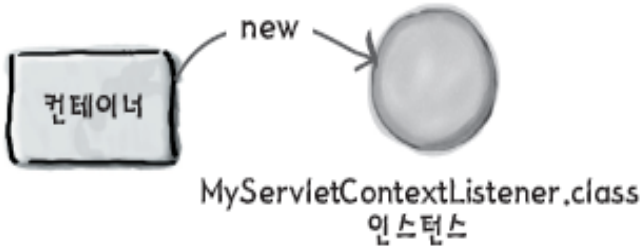

④ 컨테이너는 생성한 컨텍스트 초기화 파라미터의 String 쌍을 ServletContext 객체에 설정합니다.



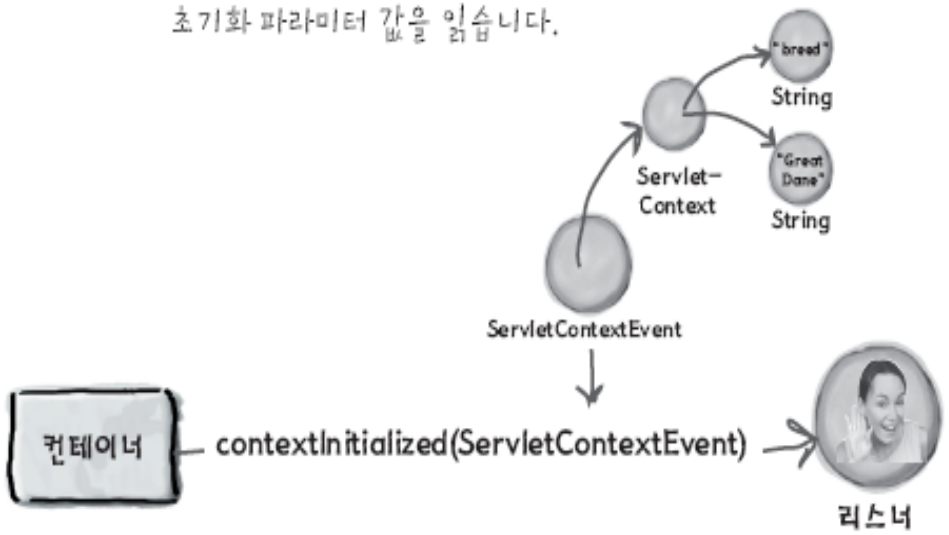
# 리스너 >> ServletContextListener 실행순서

## ServletContextListener 실행 순서 (2)

⑤ 컨테이너는 MyServletContextListener 클래스 인스턴스를 만듭니다.

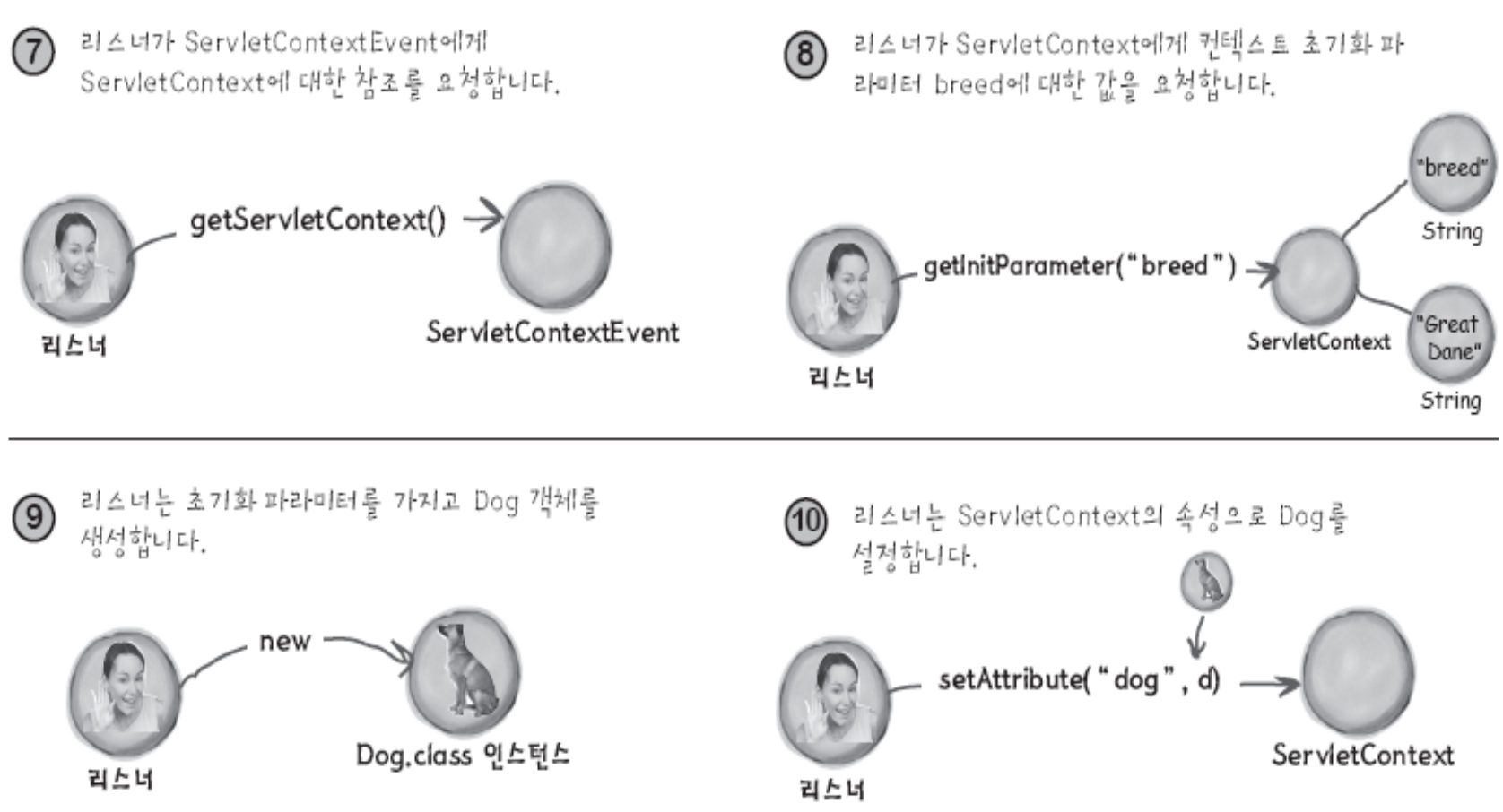



⑥ 컨테이너는 리스너의 contextInitialized() 메소드를 호출합니다. 인자로 ServletContextEvent를 넘깁니다. 이 이벤트 객체를 가지고 ServletContext에 접근합니다. 코드에서는 ServletContextEvent로 접근한 ServletContext로 컨텍스트 초기화 파라미터 값을 읽습니다.



# 리스너 >> ServletContextListener 실행순서

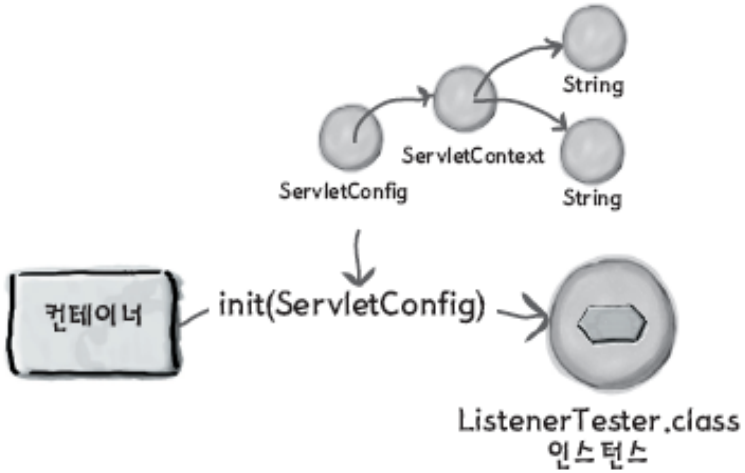
## ServletContextListener 실행 순서 (3)



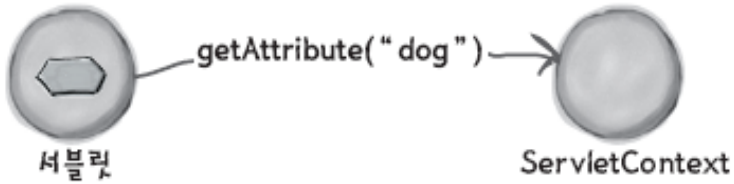
# 리스너 >> ServletContextListener 실행순서

## ServletContextListener 실행 순서 (4)

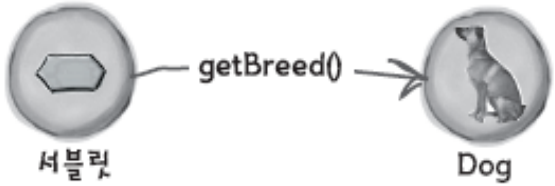
11 컨테이너는 새로운 서블릿을 생성합니다(풀어서 말하자면... 초기화 파라미터로 ServletConfig를 생성하고, 여기에 ServletContext에 대한 참조를 설정한 다음, 서블릿 init() 메소드를 호출합니다).




12 서블릿은 요청을 받고는, ServletContext에게 dog 속성에 매핑된 객체 인스턴스를 요청합니다.




13 서블릿은 Dog 객체의 getBreed() 메소드를 호출합니다(그 다음 HttpResponse에 이 내용을 쓰겠죠).



# 리스너 >> 그외 다른 리스너

| 시나리오 | 리스너 인터페이스 | 이벤트 타입 |
|---|---|---|
| 웹 애플리케이션 컨텍스에 속성(Attribute)을 추가했는지, 제거했는지, 수정했는지 알고 싶습니다. | javax.servlet.**ServletContextAttributeListener**<br>attributeAdded<br>attributeRemoved<br>attributeReplaced | ServletContextAttributeEvent |
| 얼마나 많은 동시 사용자가 지금 물려 있는지 알고 싶습니다. 즉 현재 활성화된(Active) 세션 정보를 알고 싶습니다. | javax.servlet.http.**HttpSessionListener**<br>sessionCreated<br>sessionDestroyed | HttpSessionEvent |
| 요청(Request)이 들어올 때마다 이 사실을 알 수 있을까요? 로그를 남기고 싶어서 그럽니다. | javax.servlet.**ServletRequestListener**<br>requestInitialized<br>requestDestroyed | ServletRequestEvent |
| Request 속성(Attribute)이 추가됐는지, 제거됐는지, 수정됐는지 알고 싶습니다. | javax.servlet.**ServletRequestAttributeListener**<br>attributeAdded<br>attributeRemoved<br>attributeReplaced | ServletRequestAttributeEvent |

# 리스너 >> 그외 다른 리스너

| 시나리오 | 리스너 인터페이스 | 이벤트 타입 |
|---|---|---|
| 속성 객체(속성에 집어 넣을 객체)가 하나 있는데, 이 타입의 객체가 세션에 바인딩 되었는지 아니면 제거됐는지를 알고 싶습니다. | javax.servlet.http.**HttpSessionBindingListener**<br>valueBound<br>valueUnbound | HttpSessionBindingEvent |
| 세션 속성이 추가됐는지, 제거됐는지, 수정됐는지 알고 싶습니다. | javax.servlet.http.**HttpSessionAttributeListener**<br>attributeAdded<br>attributeRemoved<br>attributeReplaced | HttpSessionBindingEvent<br>이름이 좀 이상한 게 느껴지나요? HttpSessionAttribute Listener에 대한 이벤트 이름이 예상한 것과는 다르죠. 아마도 HttpSessionAttributeEvent를 예상했지 않나 싶은데... |
| 컨텍스트가 생성되었는지 소멸되었는지 알고 싶습니다. | javax.servlet.**ServletContextListener**<br>contextInitialized<br>contextDestroyed | ServletContextEvent |
| 속성 객체가 하나 있는데, 세션에 바인딩한 이 타입의 객체가 다른 JVM으로 옮겨 갔는지(migrate), 아니면 옮겨 왔는지를 알고 싶습니다. | javax.servlet.http.**HttpSessionActivationListener**<br>sessionDidActivate<br>sessionWillPassivate | HttpSessionEvent<br>HttpSessionActivationEvent라고 생각했겠죠. |

# 속성 >>속성 (Attribute)

## ■ 속성(Attribute) 란?

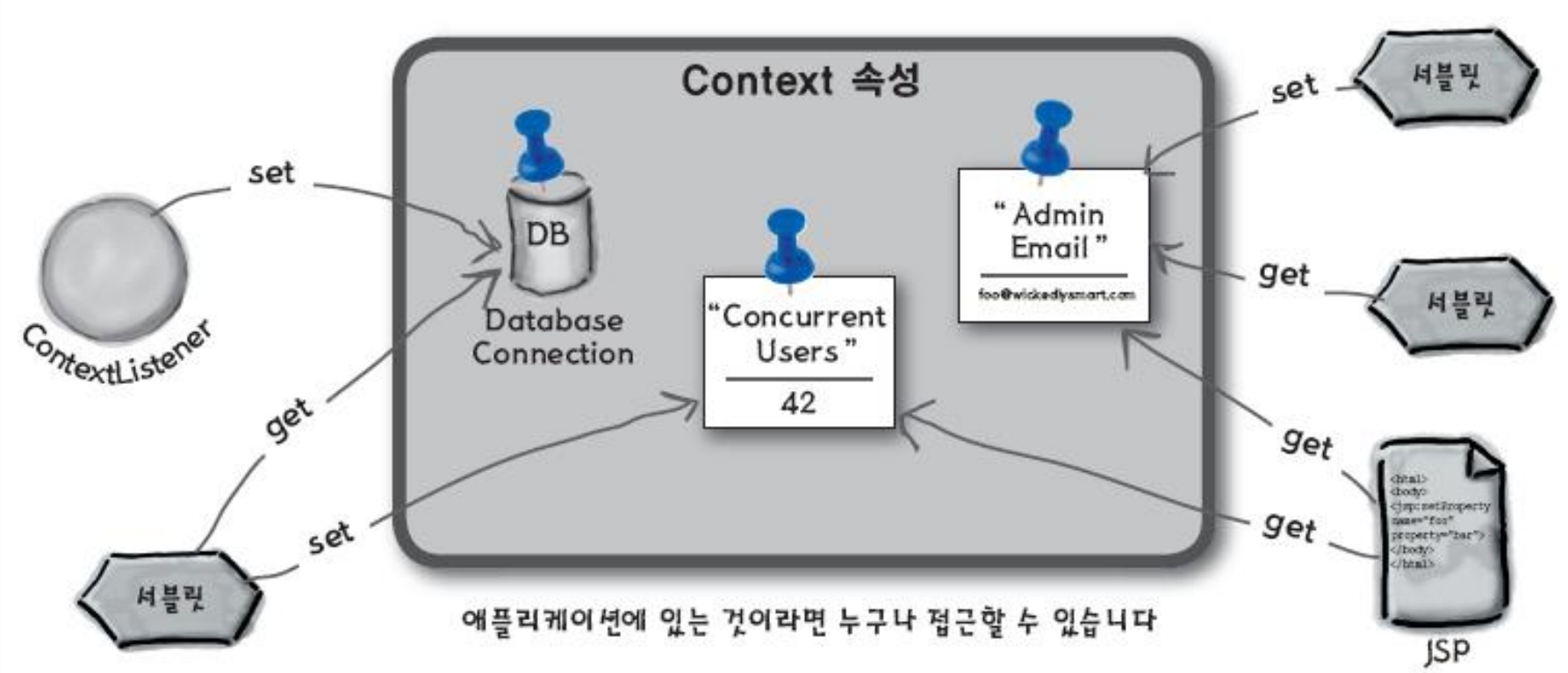
● **ServletContext, HttpServletRequest, HttpServletResponse, HttpSession** 객체 중 하나에 설정해 놓는 객체**(Object)** 이다.

## ■ 속성과 파라미터의 차이점

| | 속성 | 파라미터 |
|---|---|---|
| 타입 | Application/context<br>Request<br>Session<br>서블릿에는 속성이 없습니다(인스턴스 변수만 사용합니다) | Application/context 초기화 파라미터<br>Request 파라미터<br>Servlet 초기화 파라미터<br>세션 파라미터 이런 것은 없습니다. |
| 설정 메소드 | setAttribute(String name, Object value) | 애플리케이션과 서블릿의 초기화 파라미터 값은 런타임 시 설정할 수 없습니다. 오로지 DD에서만 가능합니다. 기억나죠?(Request 파라미터를 가지고, 좀 어렵긴 하지만 쿼리 스트링(Query String)을 설정할 수 있습니다) |
| 리턴 타입 | Object | String ← 가장 큰 차이점 |
| 참조 메소드 | getAttribute(String name)<br>속성은 리턴될 때 Object이므로, 캐스트(cast)해야 합니다. | getInitParameter(String name) |

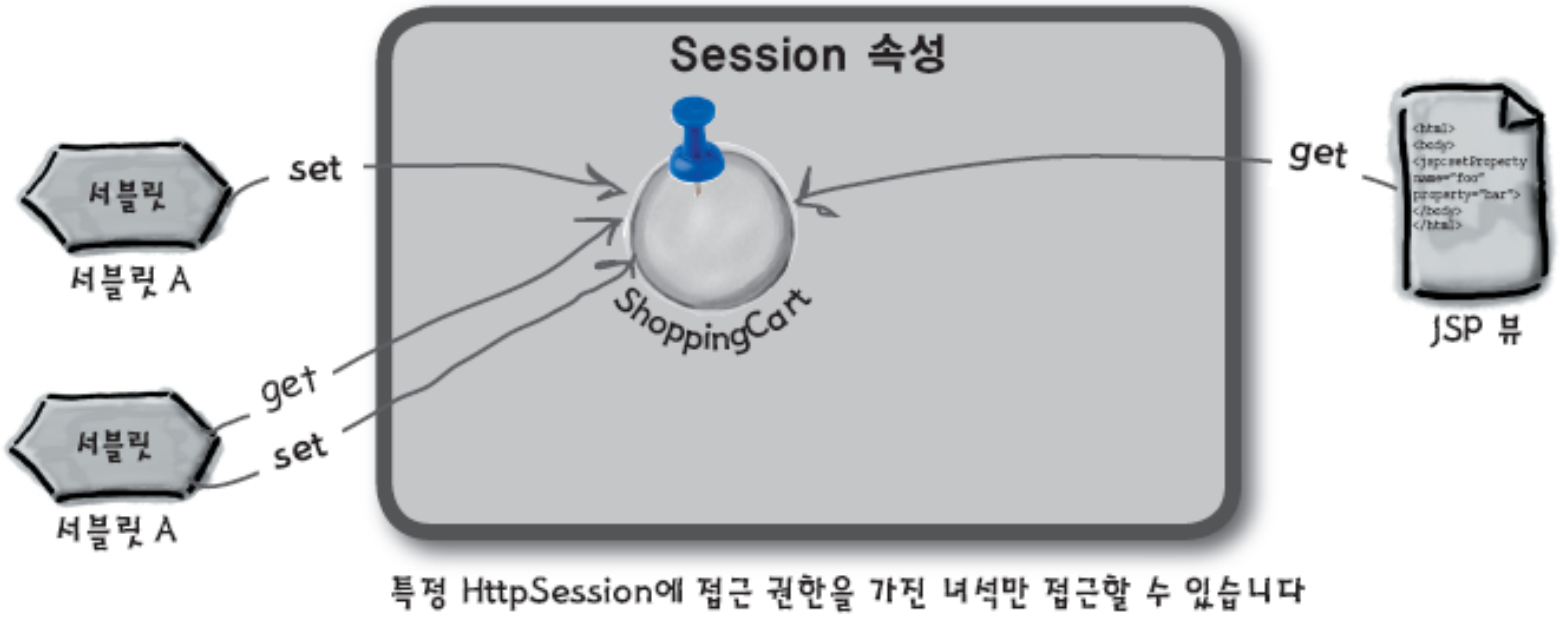
# 속성 >> 속성의 생명범위 (Scope)

## Context Scope



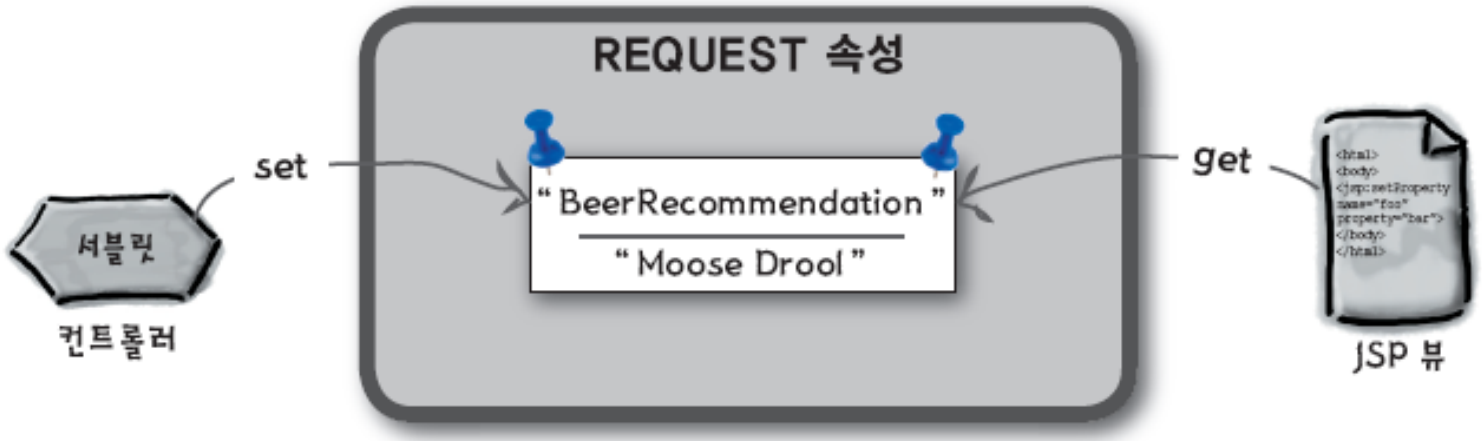
# 속성 >> 속성의 생명범위 (Scope)

## Session Scope



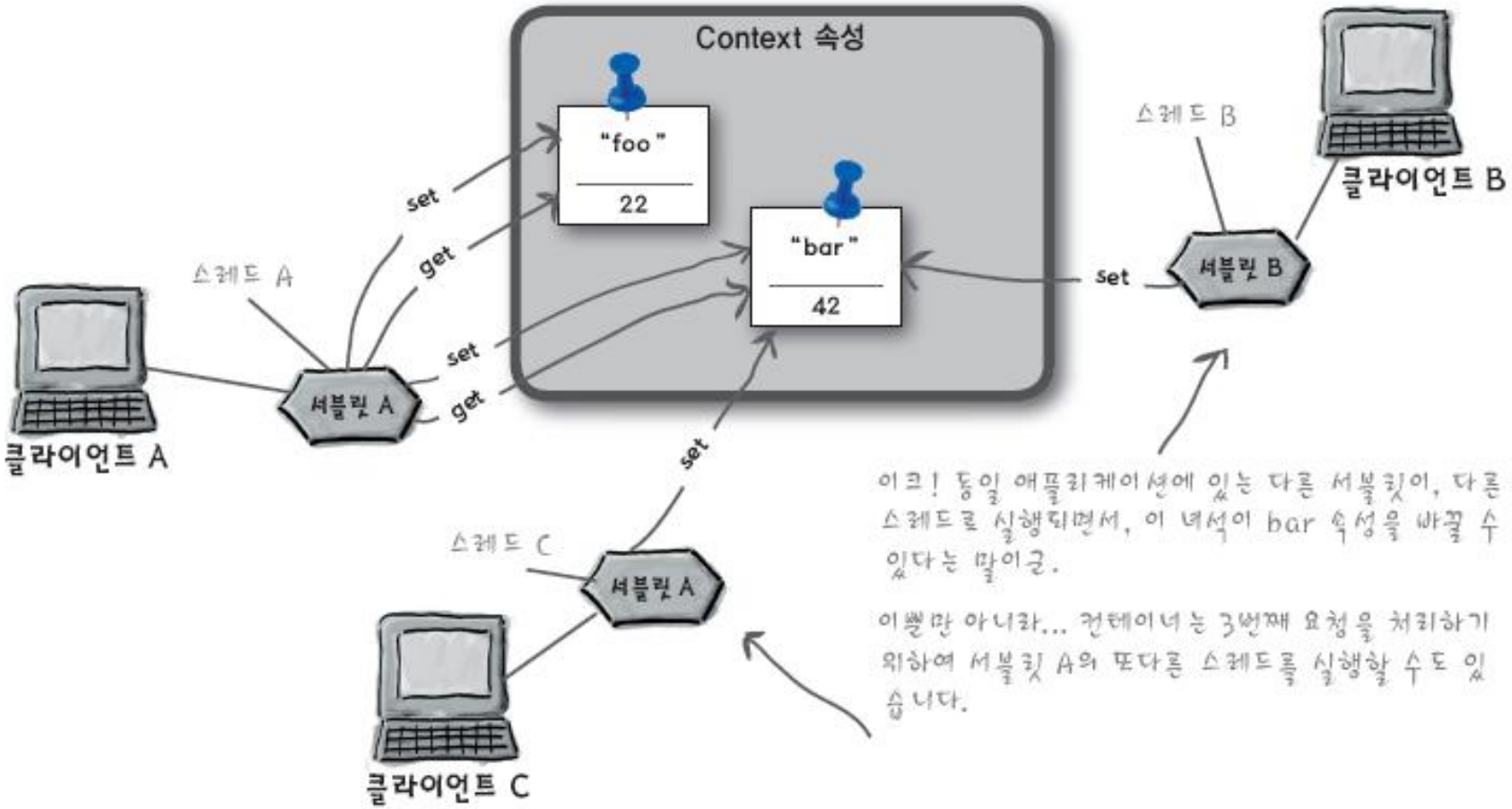
# 속성 >> 속성의 생명범위 (Scope)

## Request Scope



# 속성 >> Context Scope

## ■ Context Scope 문제점 : 쓰레드에 안전하지 않다



# 속성 >> Context Scope

■ 컨텍스트 속성 보호 : 컨텍스트에 락(lock) 을 걸면 된다

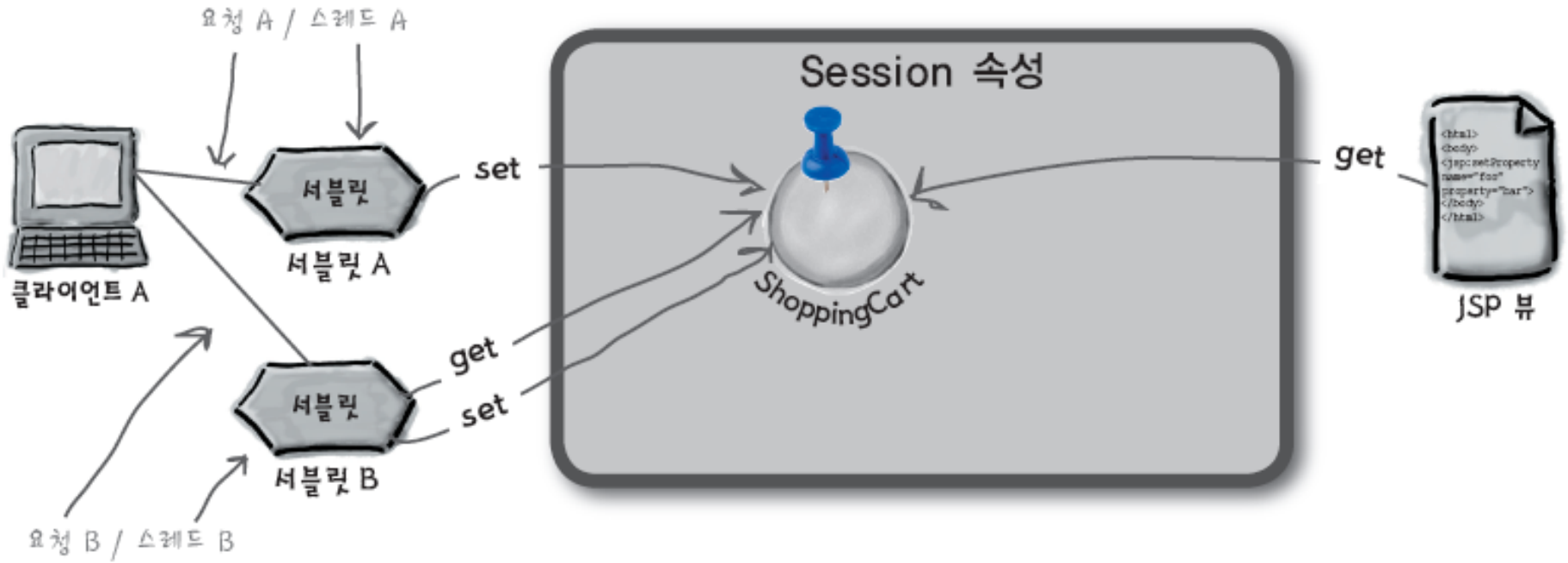
```
public void doGet(HttpServletRequest request, HttpServletResponse response)
                                    throws IOException, ServletException {

   response.setContentType("text/html");
   PrintWriter out = response.getWriter();

   out.println("test context attributes<br>");

   synchronized(getServletContext()) {
      getServletContext().setAttribute("foo", "22");
      getServletContext().setAttribute("bar", "42");

      out.println(getServletContext().getAttribute("foo"));
      out.println(getServletContext().getAttribute("bar"));
   }
}
```

컨텍스트 자체에다가 락을 걸었습니다!! 컨텍스트 속성을 보호하기 위함이죠 (synchronized(this) 이렇게 쓰면 안됩니다).

# 속성 >> Session Scope

## ■ Session Scope 문제점

하나의 클라이언트(HttpSession)에는 안전하나 동일 클라이언트의 다중 쓰레드에 대해서는 안전하지 않다.



# 속성 >> Session Scope

## ■ 세션 속성 보호 : HttpSession 을 동기화 한다

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

   response.setContentType("text/html");
   PrintWriter out = response.getWriter();

   out.println("test context attributes<br>");
   HttpSession session = request.getSession();

   synchronized(session) {
      session.setAttribute("foo", "22");
      session.setAttribute("bar", "42");

      out.println(session.getAttribute("foo"));
      out.println(session.getAttribute("bar"));
   }
}
```

여기서 세션 속성을 보호하기 위해 HttpSession 객체에 동기화를 겁니다.

# 속성 >> Request Scope

■ **Request Scope 만이 쓰레드에 안전하다**

■ **애플리케이션의 다른 컴포넌트가 Request 또는 Request의 일부를 넘겨 받기 위해 사용, RequestDispatcher를 이용**

```
// doGet()의 일부
BeerExpert be = new BeerExpert();
ArrayList result = be.getBrands(c);

request.setAttribute("styles", result);


RequestDispatcher view =
  request.getRequestDispatcher("result.jsp");

view.forward(request, response);
```

모델 데이터를 Request에 넣어둔다.

JSP 뷰를 위한 디스패쳐를 리턴받는다.

JSP에게 "Request를 넘겨 받아라"고 말하는 것입니다. 여기에도 Request, Response 객체가 넘어가는군요.

# 속성 >> RequestDispatcher

## RequestDispatcher 를 얻는 방법

### ServletRequest로부터 RequestDispatcher를 리턴받는 경우

```
RequestDispatcher view = request.getRequestDispatcher("result.jsp");
```

ServletRequest의 getRequestDispatcher() 메소드는 Request를 넘길(forward) 자원에 대한 경로(String)를 인자로 합니다. 경로가 /로 시작하는 경우, 컨테이너는 "웹 애플리케이션의 루트에서 시작하는군"이라고 생각하며, /로 시작하지 않으면, "원래 Request의 경로에 상대경로로 시작하는군"이라고 생각합니다. 그렇다고 해서 트릭을 써서 현재 웹 애플리케이션 밖의 경로로 설정할 수는 없습니다. 즉 "../../../" 같이 현재 웹 애플리케이션의 루트 경로보다 상위 경로로 빠져 나가 다른 곳으로 경로를 지정할 수 없다는 말입니다.

이건 상대경로입니다(앞에 /가 없으니까요). 이 경우 컨테이너는 최초 요청이 들어온 논리적인 위치가 어떻게 되는지 찾습니다(배포 장에서 상대경로와 논리적인 위치에 대하여 자세히 다루겠습니다).

### ServletContext로부터 RequestDispatcher를 리턴받는 경우

```
RequestDispatcher view = getServletContext().getRequestDispatcher("/result.jsp");
```

ServletRequest와 마찬가지로, getRequestDispatcher() 메소드도 Request를 넘길(forward) 자원에 대한 경로(String)를 인자로 합니다. 다른 점이 있다면 상대 경로를 사용할 수 없다는 것입니다. 즉 반드시 /를 시작으로 경로를 명시해야 합니다.

ServletContext의 getRequestDispatcher() 메소드에서는 반드시 /가 제일 앞에 와야 합니다.

# 속성 >> RequestDispatcher

## RequestDispatcher 사용


javax.servlet.RequestDispatcher