

# Atributos e *Listeners*

Parâmetros  
Atributos & *Listeners*  
Repasse de requisições



# Parâmetros de inicialização



# Parâmetros de inicialização

- Eis o que o nosso programador não quer:

```
PrintWriter out = response.getWriter();  
out.println("ze@comp.com");
```

- Solução?

- Ponha este e outros **parâmetros** no **DD - Deployment Descriptor (web.xml)**.
- Faça o servlet ler os parâmetros definidos

# Parâmetros de inicialização

## Como funciona?

### ■ No DD:

```
<servlet>
  <servlet-name>InitParam</servlet-name>
  <servlet-class>InitParam</servlet-class>
  <init-param>
    <param-name>email</param-name>
    <param-value>ze@comp.com</param-value>
  </init-param>
</servlet>
```

# Parâmetros de inicialização

## Como funciona?

### ■ No servlet:

Herdado de *GenericServlet*



```
ServletConfig cf = getServletConfig();  
out.println(cf.getInitParameter("email"));
```

# Servlets 3.0

```
@WebServlet(  
    name = "MyServlet",  
    urlPatterns = {"/foo"},
```

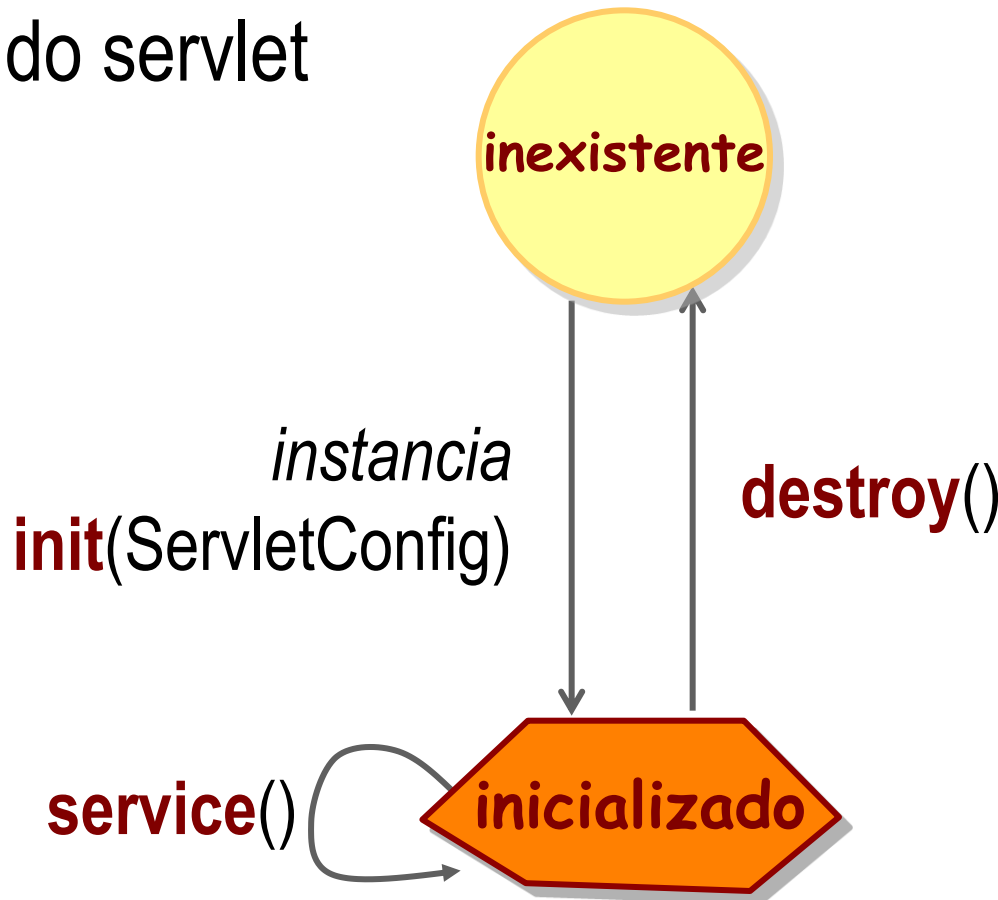
```
<servlet>  
    <servlet-name>MyServlet</servlet-name>  
    <servlet-class>CalculatorServlet</servlet-class>  
    <init-param>  
        <param-name>param1</param-name>  
        <param-value>value1</param-value>  
    </init-param>  
    <init-param>  
        <param-name>param2</param-name>  
        <param-value>value2</param-value>  
    </init-param>  
</servlet>
```

# Servlets 3.0

```
@WebServlet(  
    name = "MyServlet",  
    urlPatterns = {"/foo"},  
    initParams = {  
        @WebInitParam(name="param1", value="value1"),  
        @WebInitParam(name="param2", value="value2")  
    }  
)  
    <param-name>param1</param-name>  
    <param-value>value1</param-value>  
</init-param>  
<init-param>  
    <param-name>param2</param-name>  
    <param-value>value2</param-value>  
</init-param>  
</servlet>
```

# Parâmetros de inicialização

- Parâmetros só podem ser usados após a inicialização do servlet



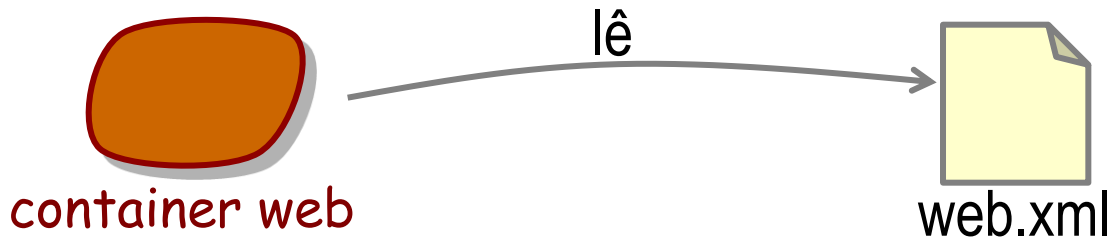
Os parâmetros de inicialização são lidos apenas uma única vez, quando o container cria o `ServletConfig` e inicializa o servlet.



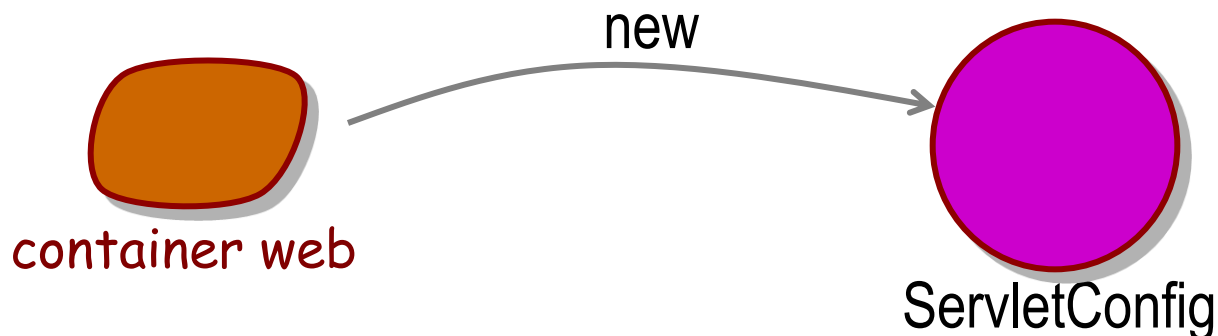
# Parâmetros de inicialização

## Seqüência de criação dos parâmetros

- Container lê o DD



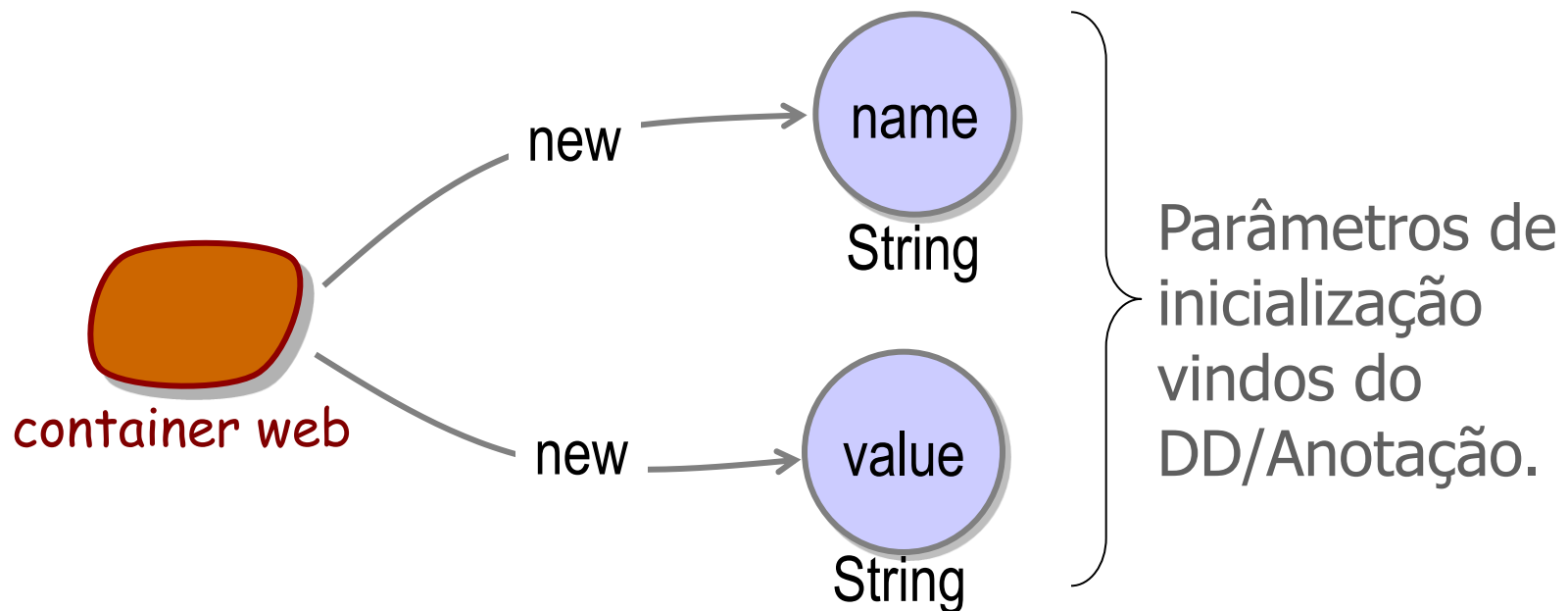
- Container cria uma nova instância de ServletConfig



# Parâmetros de inicialização

## Seqüência de criação dos parâmetros

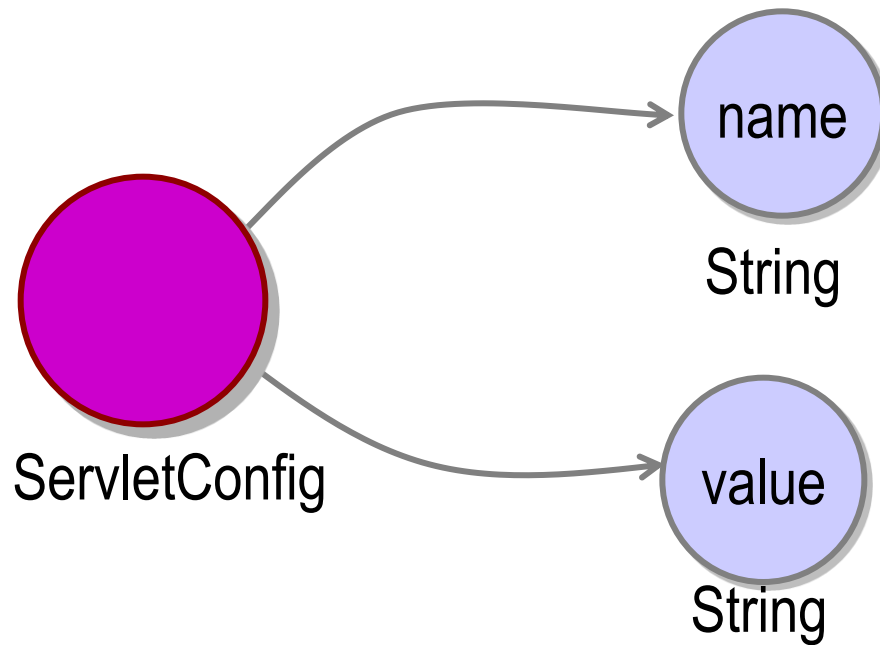
- Container cria um par de Strings para cada parâmetro (um para o nome outro para o valor)



# Parâmetros de inicialização

## Seqüência de criação dos parâmetros

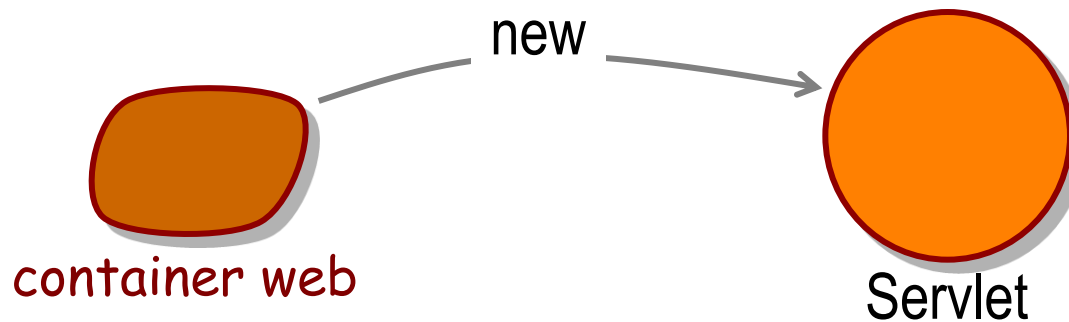
- Container faz o ServletConfig referenciá-los



# Parâmetros de inicialização

## Seqüência de criação dos parâmetros

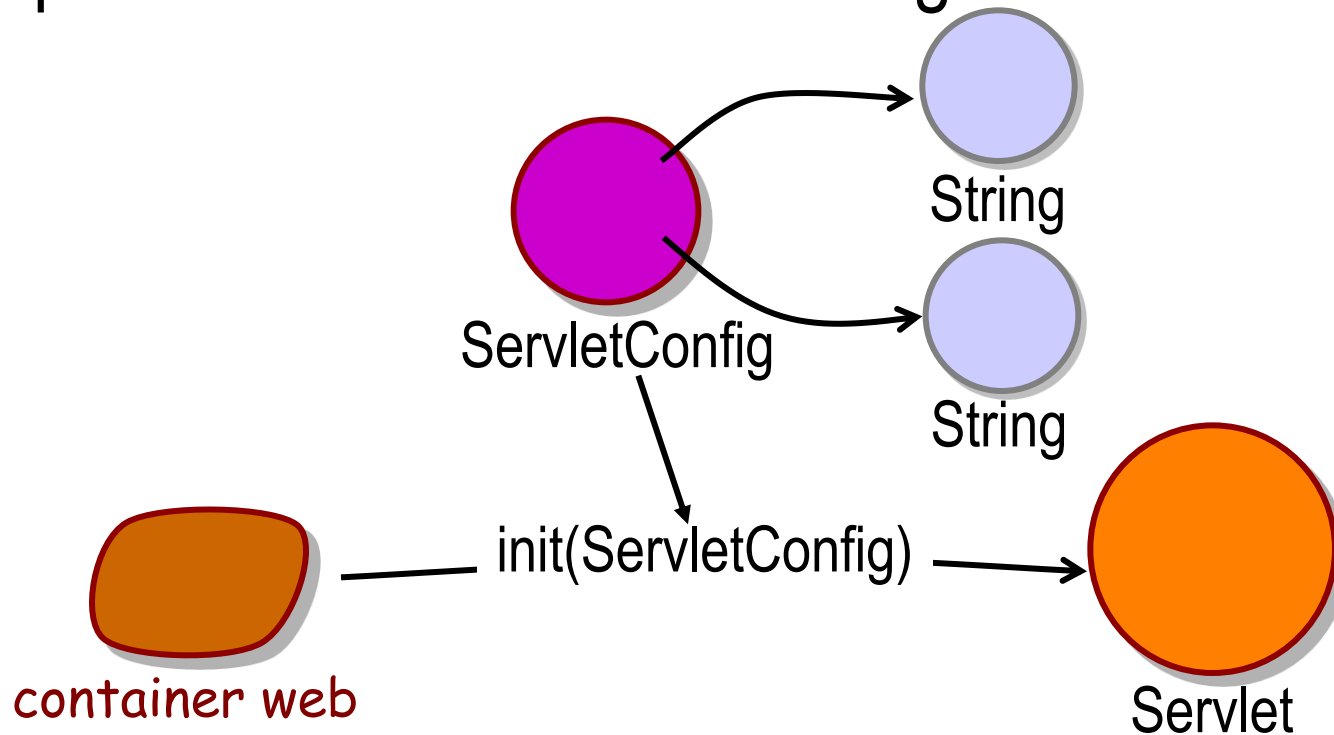
- Container cria instância do servlet



# Parâmetros de inicialização

## Seqüência de criação dos parâmetros

- Container chama método `init()` do servlet, passando-lhe o `ServletConfig` criado



# Parâmetros de inicialização

Se o container só lê os parâmetros uma vez no ciclo de vida do servlet, de que adianta mudar o email no DD?



Resposta: Após alterar o DD, deve-se fazer o redeploy da aplicação no Tomcat. Ou você desliga-liga o Tomcat ou faz o redeploy com o *Manager*.

# Parâmetros do contexto

E se os parâmetros tiverem que ser compartilhados por **vários servlets**? Ou ainda, como faço para pegá-los num **documento JSP**?



# Parâmetros do contexto

- Parâmetros idênticos aos dos servlets, só que pertencem a **toda** aplicação web

## Como funciona?

- No DD:

Fora de qualquer  
marcador <servlet>

```
<servlet>
  <servlet-name>ParamInit</servlet-name>
  <servlet-class>ParamInit</servlet-class>
</servlet>
<context-param>
  <param-name>email</param-name>
  <param-value>ze@comp.com</param-value>
</context-param>
```



# Parâmetros do contexto

## Como funciona?

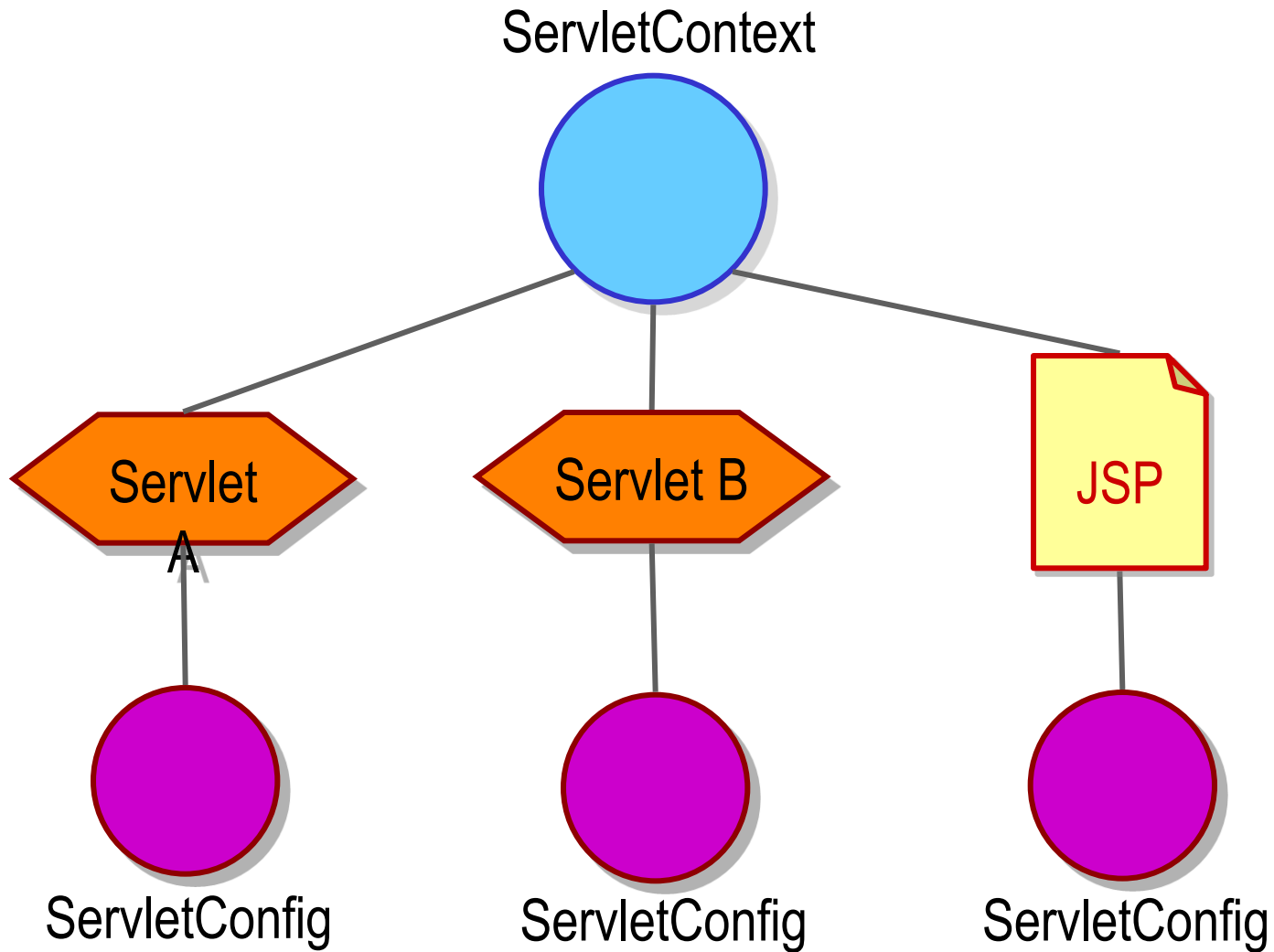
### ■ No servlet:

Herdado de Servlet

```
ServletContext ctx = getServletContext() ;  
out.println(ctx.getInitParameter("email")) ;
```

Cuidado: Parâmetros de servlets são lidos com o **ServletConfig** e os da aplicação web (contexto) são lidos com o **ServletContext**.

# Parâmetros do contexto



# Interface ServletContext

<<interface>>

## **ServletContext**

***getInitParameter(String)***

***getInitParameterNames()***

***getAttribute(String)***

***getAttributeNames()***

***setAttribute(String, Object)***

***removeAttribute(String)***

-----  
***getMajorVersion()***

***getServerInfo()***


-----  
***getRealPath()***

***getResourceAsStream(String)***

***getRequestDispatcher(String)***

...

Métodos para ler  
parâmetros e manipular  
atributos



Facilita a portabilidade  
de arquivos dentro da  
aplicação web



Transfere requisições



# Listeners

Se eu precisar executar  
alguma computação em  
eventos específicos do ciclo  
de vida da minha aplicação  
web?



# Listeners

Você precisa é de um **Listener**! Um objeto que fique atento a um determinado evento e execute um código quando ele ocorrer!



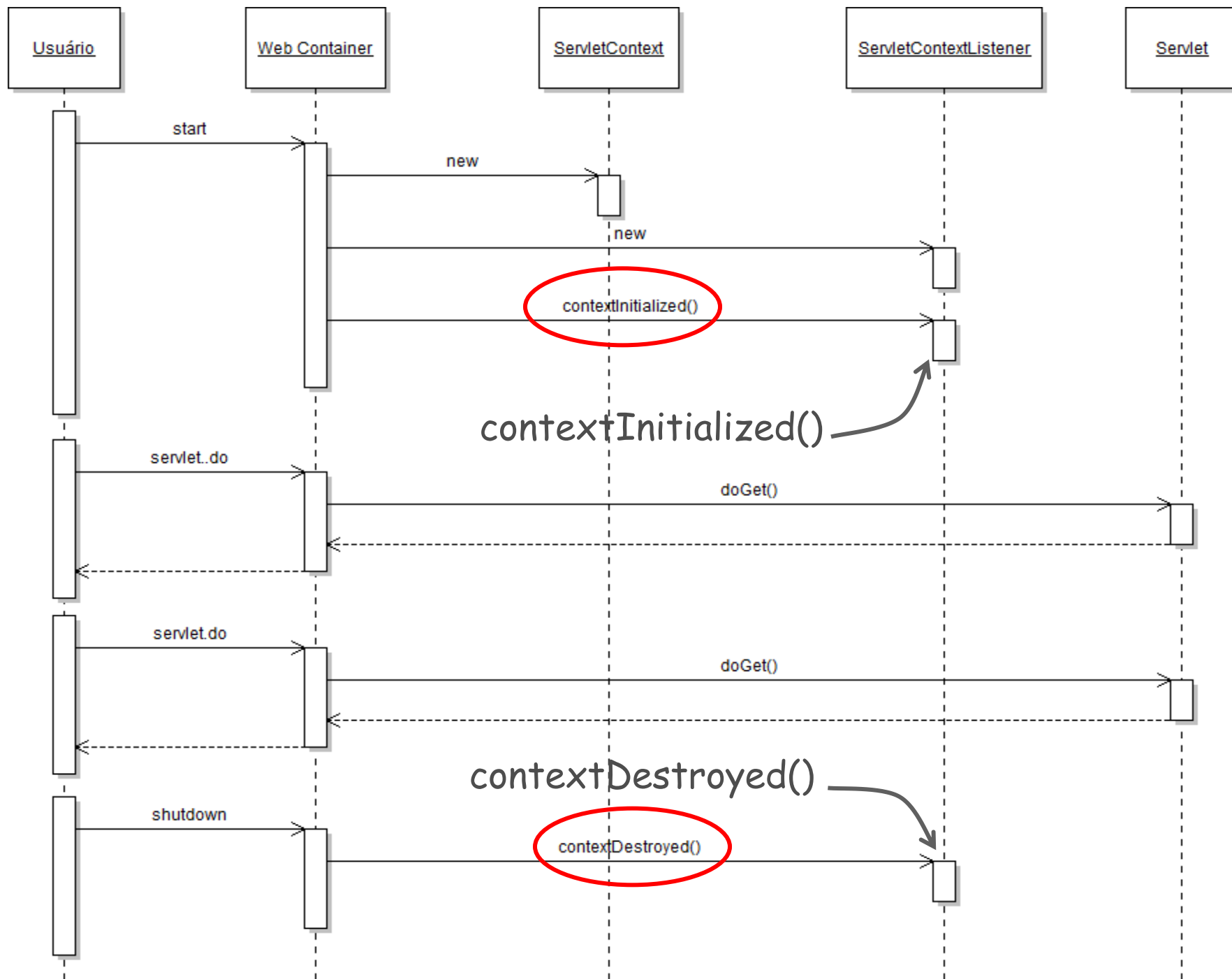
# ServletContextListener

## O que é?

- Uma interface que define "**observadores**" para eventos de inicialização e destruição de um ServletContext.

## Como funciona?

- Um *Listener* é registrado no DD.
- Quando o evento ao qual ele escuta acontece na aplicação web, por exemplo, a aplicação é inicializada, o *listener* executa um método em resposta.
- Sempre que o evento ocorrer, o *listener* executa automaticamente seu respectivo método.



# ServletContextListener

## ■ Exemplo de um ServletContextListener:

```
import javax.servlet.*;
public class MeuListener
    implements ServletContextListener {

    public void
    contextInitialized(ServletContextEvent e) {
        ...
    }

    public void
    contextDestroyed(ServletContextEvent e) {
        ...
    }
}
```



# ServletContextListener

## Detalhes do funcionamento

- Onde esta classe do listener deve ficar?
  - No diretório **WEB-INF\classes** da aplicação
- Como eu instancio um objeto desta classe?
  - Você **não** instancia, apenas registra a classe no DD ou a anota (Servlet 3.0). É o container quem instancia.
- Quando ele executa o listener?
  - Durante o ciclo de vida do ServletContext os métodos de inicialização e destruição do listener são enviados para ele pelo container.

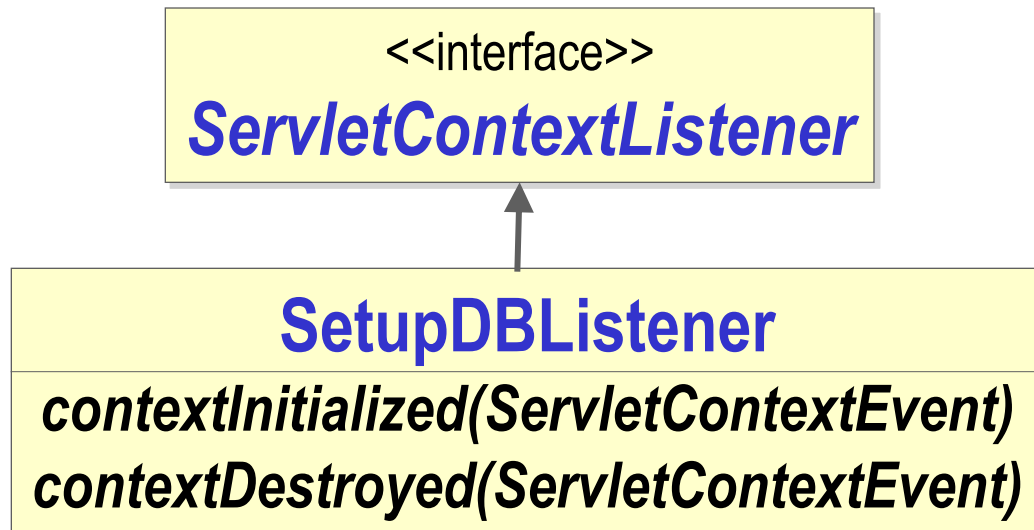
# ServletContextListener

## ■ Exemplo:

- Um ServletContextListener poderia acessar um banco de dados e povoar as tabelas com dados de teste
- Assim que a aplicação fosse carregada, o listener executaria a operação de povoar as tabelas com dados de teste no método *contextInitialized()*.
- Quando a aplicação fosse finalizada, o método *contextDestroyed()* limparia o banco de dados.

# ServletContextListener

- Precisaremos de um listener:



- Povoia o BD no *contextInitialized()*
- Limpa o BD no *contextDestroyed()*

# ServletContextListener

- Suponha a existência de uma classe DBSetup (faz os inserts e deletes necessários nas tabelas do BD):

<b>DBSetup</b>
<i>insertAlunos()</i>
<i>insertTurmas()</i>
<i>deleteAlunos()</i>
<i>deleteTurmas()</i>
...

# Escrevendo o listener

```
package com.exemplo;
import javax.servlet.*;
public class SetupDBListener implements
    ServletContextListener {

    public void contextInitialized(ServletContextEvent e)
    {
        DBSetup setup = new DBSetup();
        setup.insertTurmas();
        setup.insertAlunos();
        //demais inserts de teste do banco
    }
    ...
}
```

# Escrevendo o listener

```
public void contextDestroyed(ServletContextEvent e)
{
    DBSetup setup = new DBSetup();
    setup.removeFKs();
    setup.deleteAlunos();
    setup.deleteTurmas();
    //demais deletes
}
}
```

# Escrevendo o web.xml

```
<web-app>
  <servlet>
    <servlet-name>AlgumServlet</servlet-name>
    <servlet-class>com.exemplo.AlgumServlet
    </servlet-class>
  </servlet>
  ...
  <listener>
    <listener-class>com.exemplo.SetupDBListener
    </listener-class>
  </listener>
  ...
</web-app>
```

# Servlets 3.0

## ■ Anotação @WebListener

### **@WebListener**

```
public class SetupDBListener implements
    ServletContextListener {

    public void contextInitialized(ServletContextEvent e) {
        ...
    }
}
```



# Outros Listeners

- Regra geral: para cada ciclo de vida de um objeto há um listener para ele
    - ServletContextAttributeListener
    - HttpSessionListener
    - ServletRequestListener
    - ServletRequestAttributeListener
    - HttpSessionBindingListener
    - HttpSessionAttributeListener
    - ServletContextListener
    - HttpSessionActivationListener
- Listeners para todos os gostos e usos!

# ServletContextAttributeListener

## ■ Cenário de uso

- Você deseja saber quando um atributo é adicionado, removido ou alterado

## ■ Métodos

- `attributeAdded()`
- `attributeRemoved()`
- `attributeReplaced()`

## ■ Tipo do evento:

- `ServletContextAttributeEvent`

# HttpSessionListener

- Cenário de uso
  - Você deseja saber quantos usuários concorrentes estão acessando a sua aplicação web
- Métodos
  - `sessionCreated()`
  - `sessionDestroyed()`
- Tipo do evento:
  - `HttpSessionEvent`

# ServletRequestListener

- Cenário de uso
  - Você deseja saber quando um ServletRequest é criado
- Métodos
  - requestInitialized()
  - requestDestroyed()
- Tipo do evento:
  - ServletRequestAttributeEvent

# HttpSessionBindingListener

## ■ Cenário de uso

- Você tem objetos que são atributos de uma sessão e quer que eles (não a sessão) sejam avisados quando forem atribuídos ou retirados da sessão

## ■ Métodos

- valueBound()
- valueUnbound()

## ■ Tipo do evento:

- HttpSessionBindingEvent

# HttpSessionBindingListener

## ■ Exemplo de uso:

```
public class Aluno
    implements HttpSessionBindingListener {
    //código da classe

    public void valueBound(HttpSessionBindingEvent e)
    {
        ...
    }

    public void valueUnBound(HttpSessionBindingEvent e)
    {
        ...
    }
}
```

# HttpSessionAttributeListener

## ■ Cenário de uso

- Você quer saber quando objetos são adicionados, removidos ou alterados em uma sessão

## ■ Métodos

- `attributeAdded()`
- `attributeRemoved()`
- `attributeReplaced()`

## ■ Tipo do evento:

- `HttpSessionBindingEvent`

O mesmo evento de  
`HttpSessionBindingListener`



# ServletContextListener

## ■ Cenário de uso

- Você deseja saber quando um contexto é criado ou destruído

## ■ Métodos

- `contextInitialized()`
- `contextDestroyed()`

## ■ Tipo do evento:

- `ServletContextEvent`





# Atributos

Parâmetros? Atributos? Qual a diferença entre eles? Como defini-los? Quem pode acessá-los?

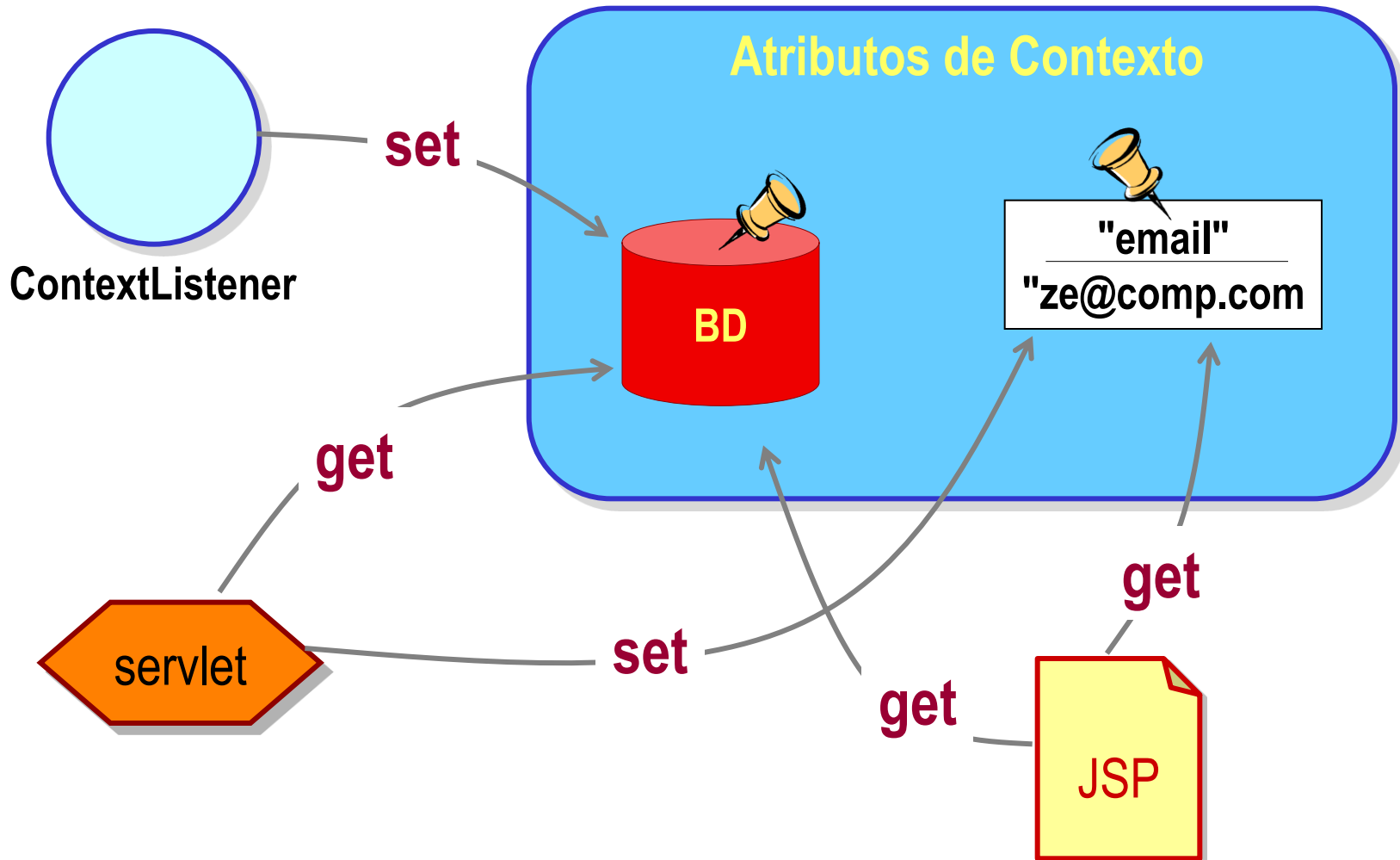
Escopo!



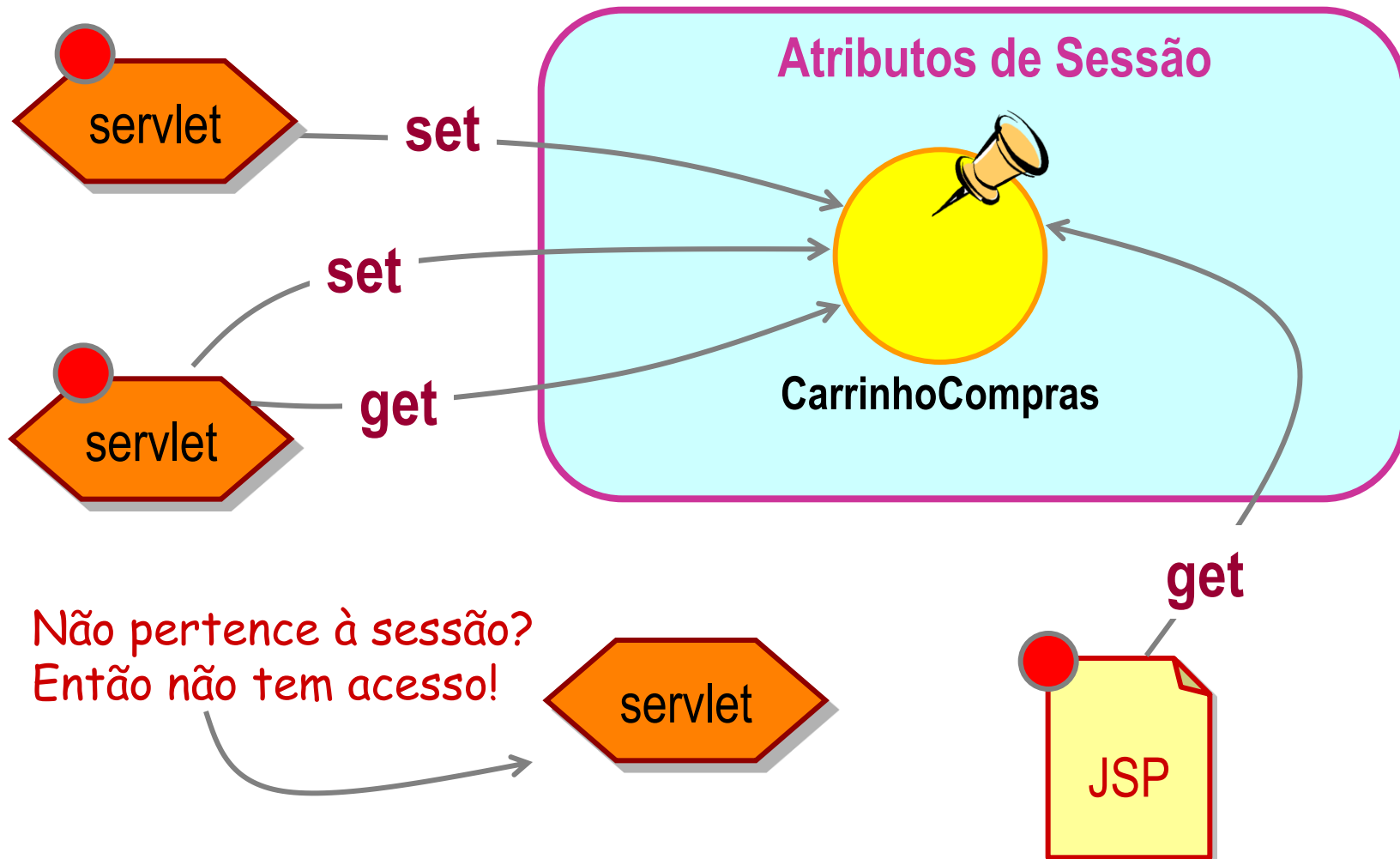
# Atributos vs Parâmetros

	<b>Atributos</b>	<b>Parâmetros</b>
Tipos	Contexto Request Session	Contexto init Request Servlet init
Método para "colar"	setAttribute(String, Object)	Contexto e Servlet são definidos no DD e Request não tem
Retornam um	Object	String
Método para "pegar"	getAttribute(String)	getInitParameter(String) getParameter(String) [R]

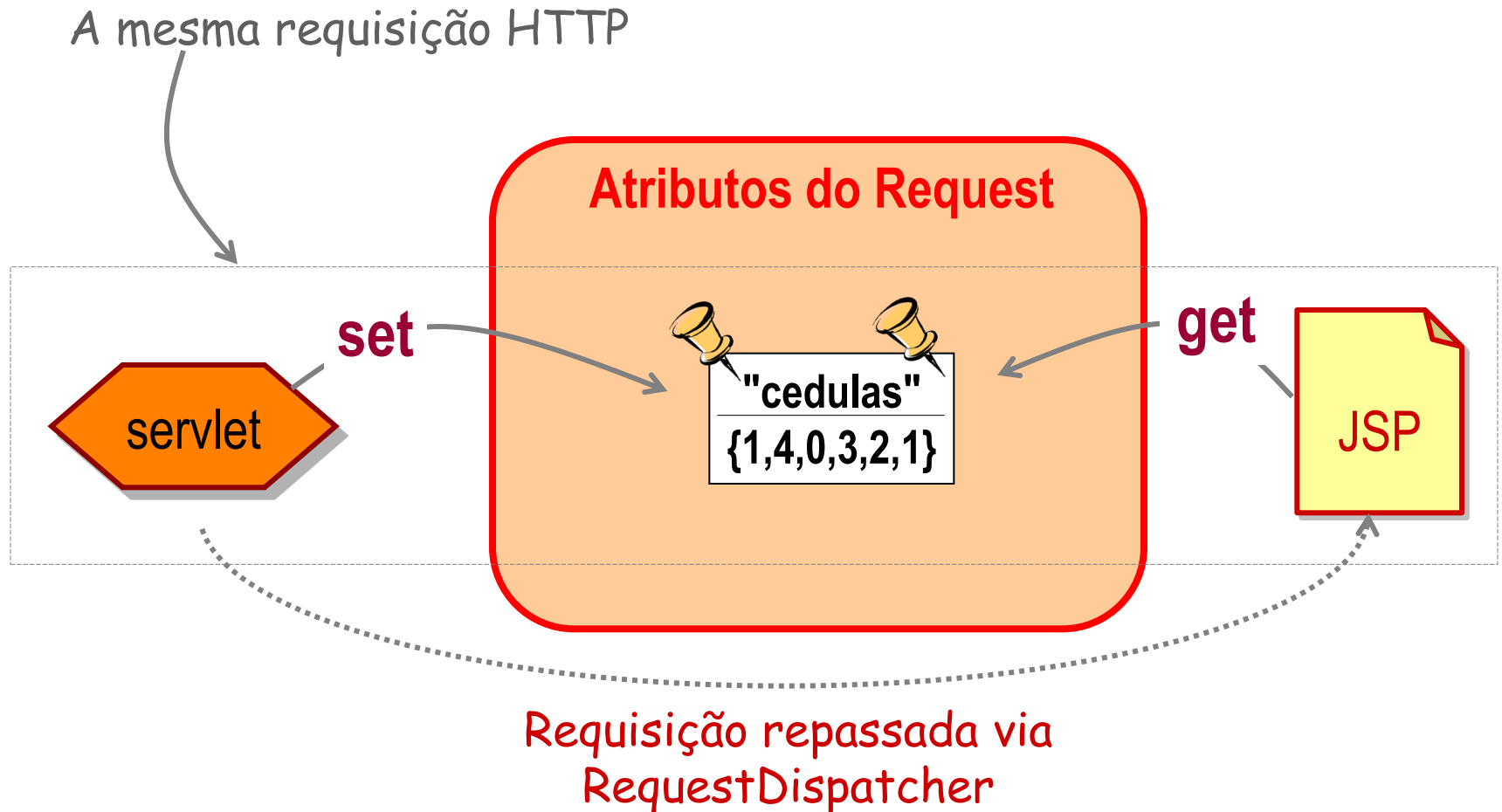
# Escopo de um atributo



# Escopo de um atributo



# Escopo de um atributo



# API de atributos

**Object** *getAttribute(String)*  
*setAttribute(String, Object)*  
*removeAttribute(String)*  
**enumeration** *getAttributeNames()*

<<interface>>

**ServletRequest**

<<interface>>

**ServletContext**

<<interface>>

**HttpSession**

# Atributos e concorrência

- Atributos podem trazer "sensacionais" revelações

```
public void doGet(HttpServletRequestContext req,
    HttpServletResponse resp) {

    resp.setContentType("text/html");
    PrintWriter out = response.getWriter();
    out.println("Valores dos atributos:<br>");
    getServletContext().setAttribute("A", "10");
    getServletContext().setAttribute("B", "20");
    out.println(getServletContext().getAttribute("A"));
    out.println(getServletContext().getAttribute("B"));
}
```



# Atributos e concorrência

- O que você diria se aparecesse o resultado abaixo?

Valores dos atributos:

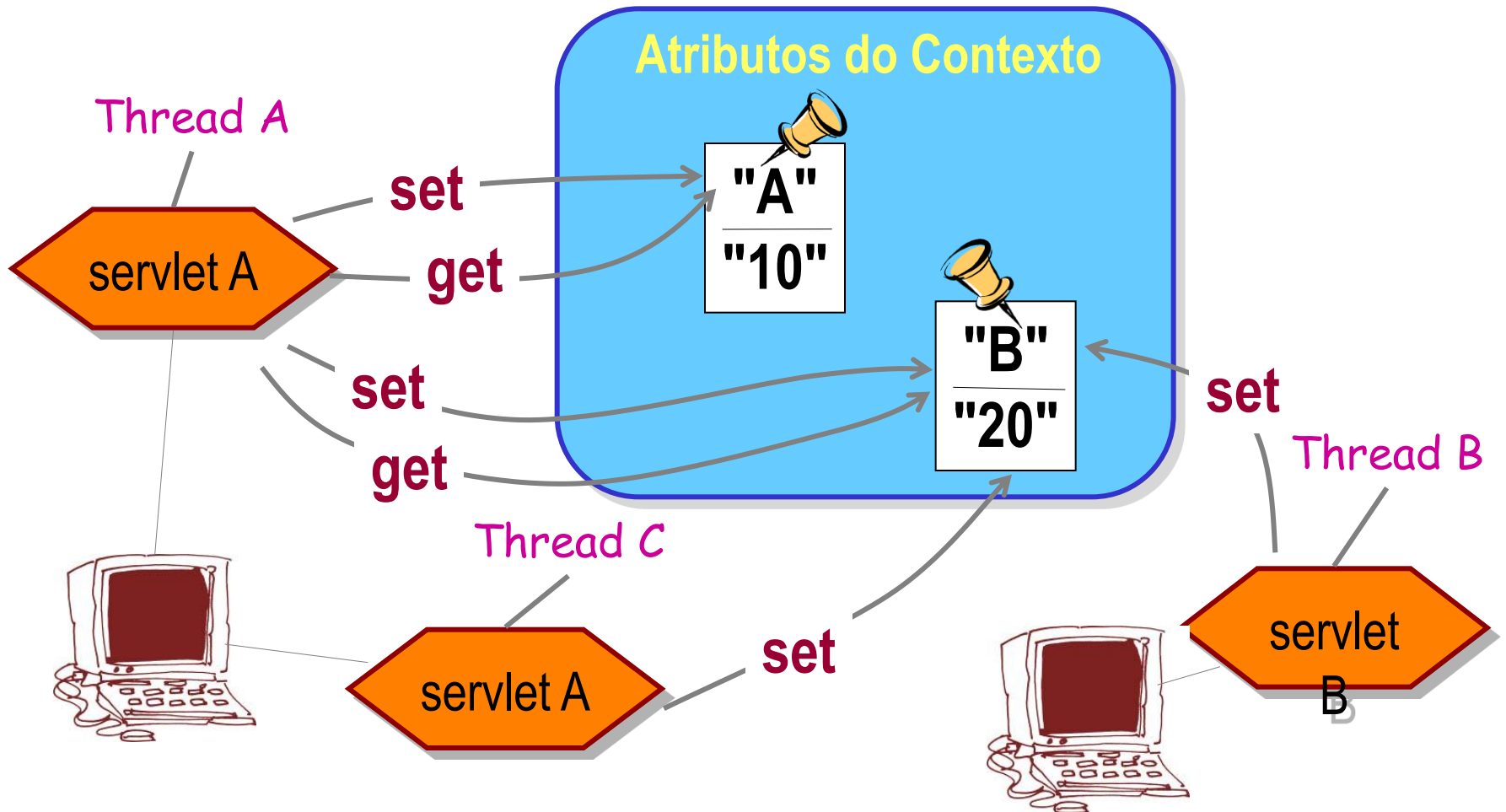
10 33

Eu deveria  
ter feito  
agronomia...



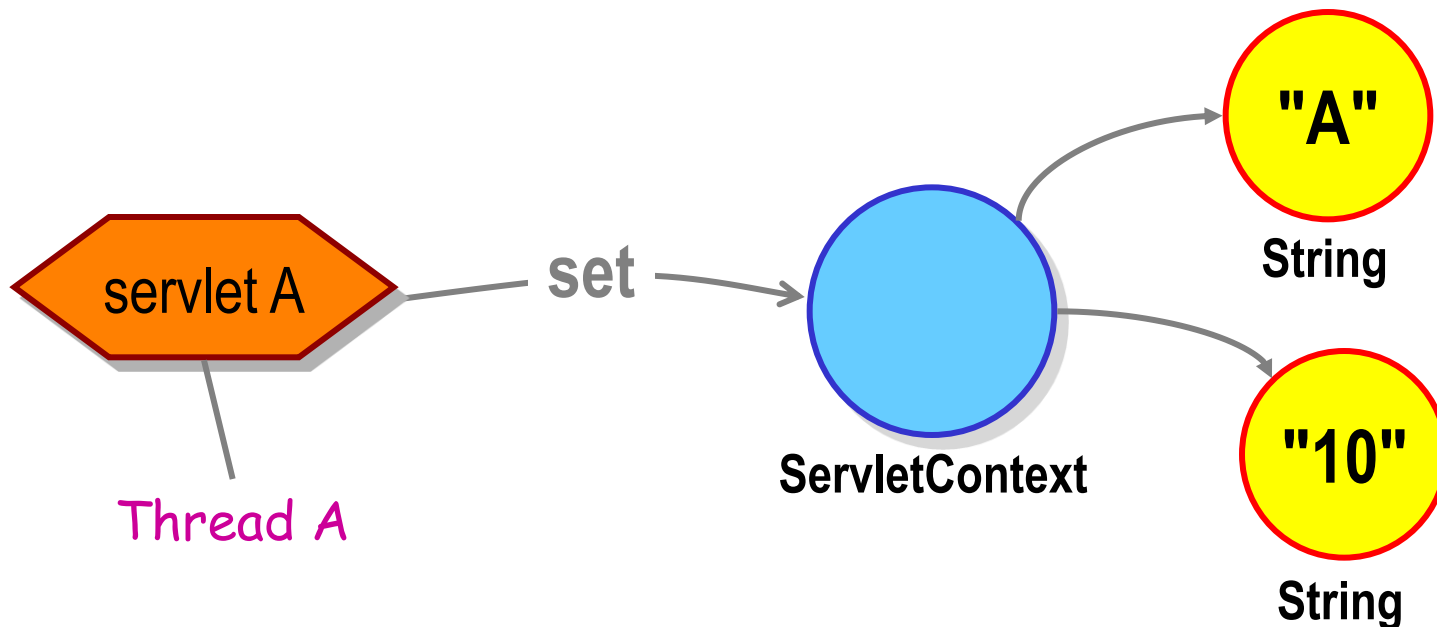
# Atributos e concorrência

- O escopo de contexto não é *thread safe*



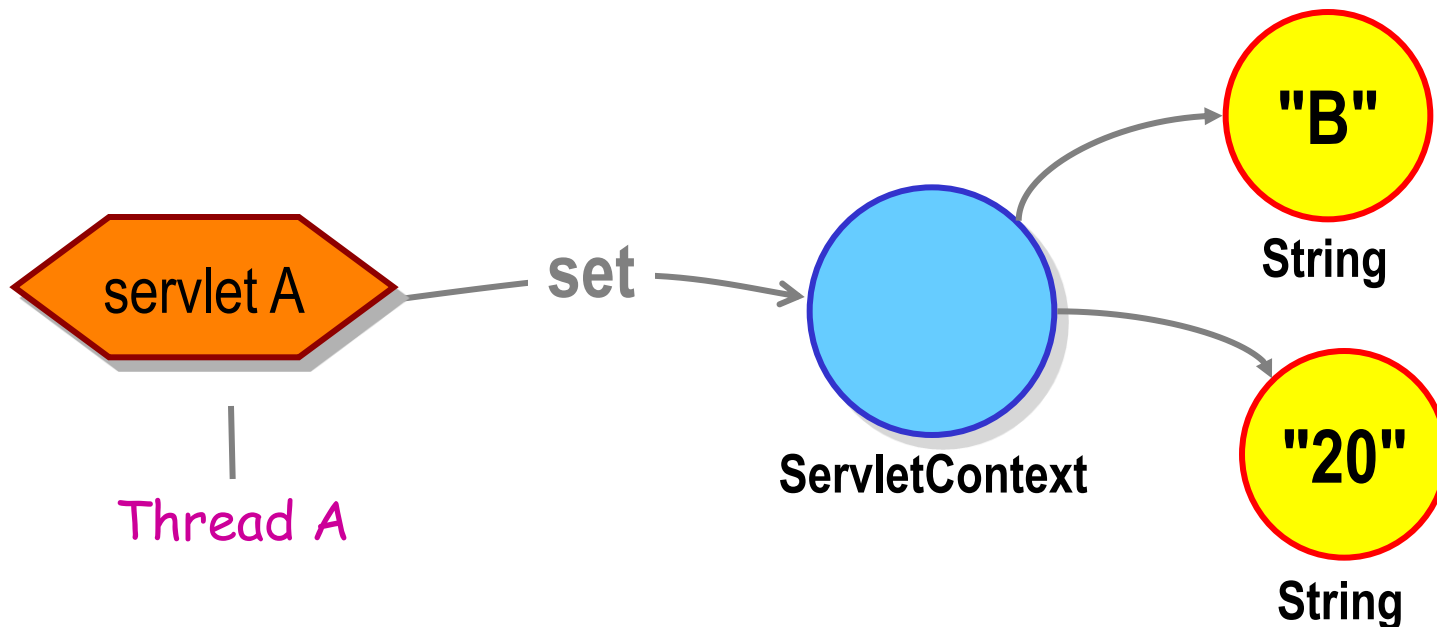
# Atributos e concorrência

- O **thread A** do servlet **A** define o atributo "A" com o valor "10"



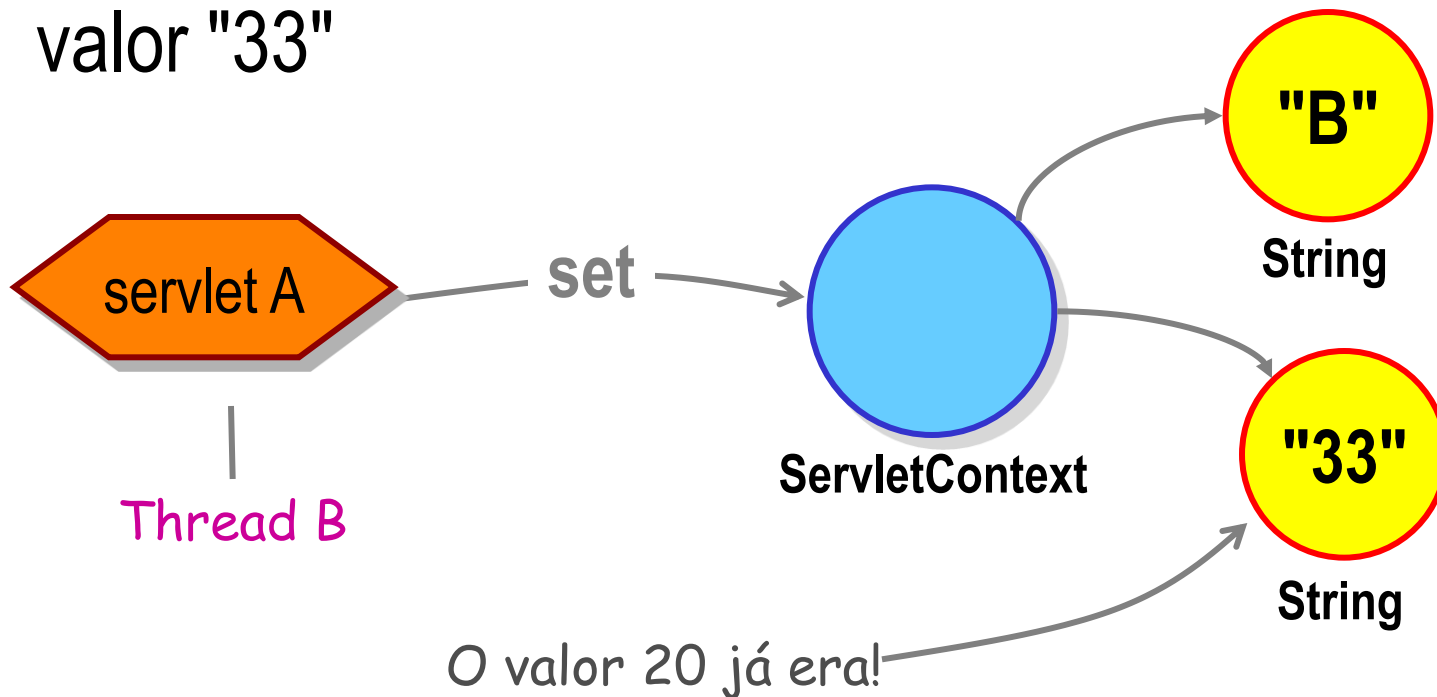
# Atributos e concorrência

- O **thread A** do servlet **A** define o atributo "B" com o valor "20"



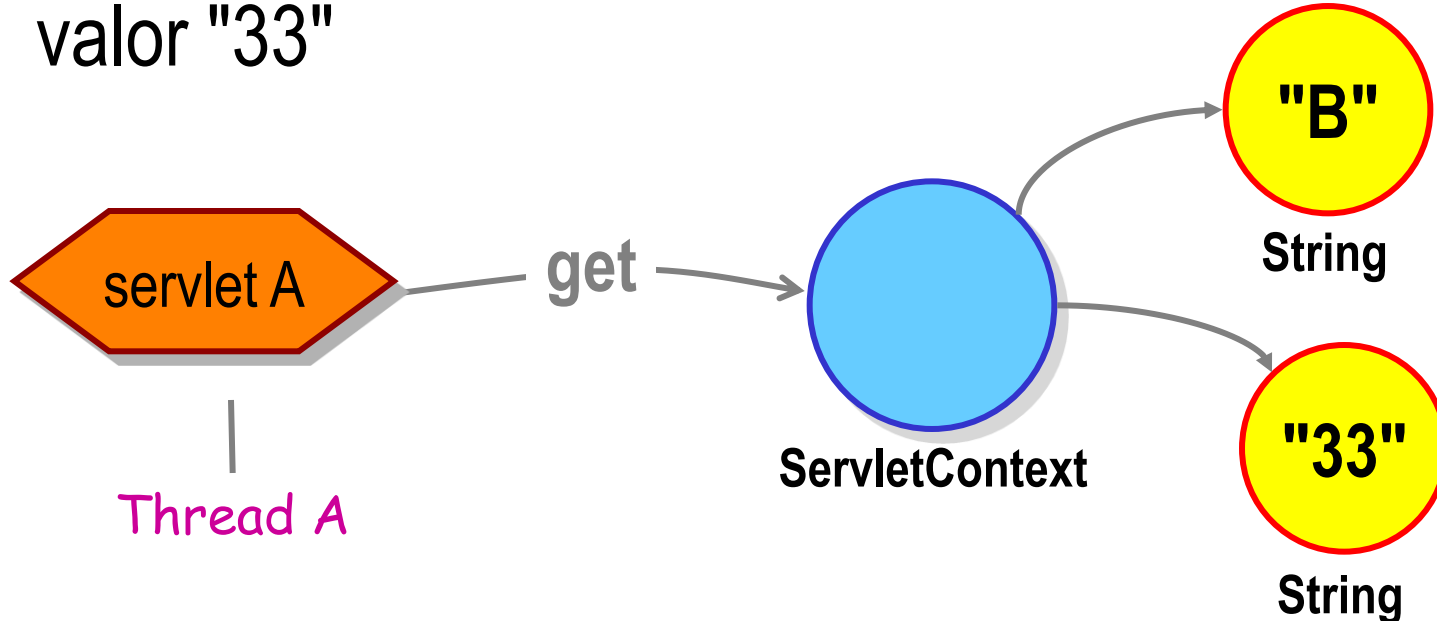
# Atributos e concorrência

- O **thread B** do servlet **A** é escolhido para executar
- O **thread B** do servlet **A** define o atributo "B" com o valor "33"



# Atributos e concorrência

- O **thread A** do servlet **A** é escolhido para executar
- O **thread A** do servlet **A** pega o atributo "B" com o valor "33"



# Atributos e concorrência

## ■ Voltando para o código...

```
public void doGet(HttpServletRequestContext req,  
    HttpServletResponse
```

```
    resp.setContentType  
    PrintWriter out  
    out.println("Vá
```

```
    getServletContext.setAttribute("A", "10");
```

```
    getServletContext.setAttribute("B", "20");
```

```
    out.println(getServletContext.getAttribute("A"));
```

```
    out.println(getServletContext.getAttribute("B"));
```

```
}
```

Entre estas duas linhas, outro thread de outro servlet foi escolhido para executar e mudou o valor do atributo "B"

# Atributos e concorrência

- Alguma idéia para resolver o problema?

Humm, eu acho que poderíamos **sincronizar** o método doGet(). Será que é uma boa idéia?

Modificador **synchronized**  
de Java

```
public synchronized void doGet(...)
```





# Atributos e concorrência



Se você sincronizar o `doGet()` vai **jogar fora a capacidade do servlet de tratar requisições de forma concorrente.** O servlet só poderá tratar um cliente de cada vez. Não é uma boa idéia.

# Atributos e concorrência

Bom, eles poderiam então ter definido os métodos get e set para atributos do ServletContext já sincronizados.



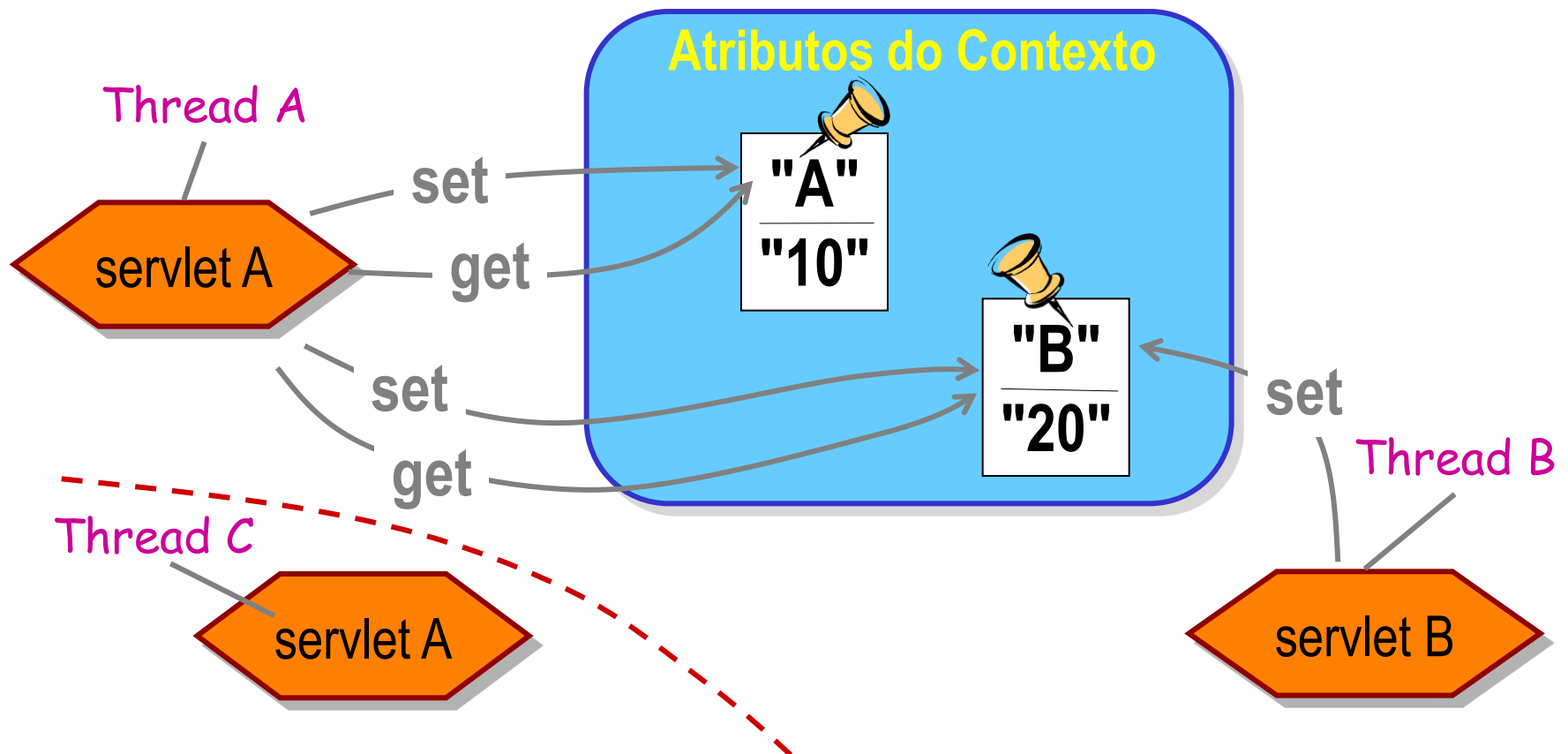
# Atributos e concorrência



Se eles fizessem isso, então TODAS as aplicações que manipulassem atributos do ServletContext teriam a sobrecarga da sincronização. Eles preferiram definir a API sem o **synchronized**. Você que sincronize sua aplicação!

# Atributos e concorrência

- Sincronizar o método de serviço, além de ser estúpido pode ser inócuo. Por quê?




# Atributos e concorrência


- Sobre qual objeto devemos sincronizar?
  - Sobre o próprio ServletContext

```
...  
resp.setContentType("text/html");  
PrintWriter out = response.getWriter();  
out.println("Valores dos atributos:<br>");  
synchronized(getServletContext()) {  
    getServletContext().setAttribute("A", "10");  
    getServletContext().setAttribute("B", "20");  
    out.println(getServletContext().getAttribute("A"));  
    out.println(getServletContext().getAttribute("B"));  
}  
}
```

# Atributos e concorrência




E os atributos da **sessão**? São protegidos contra threads concorrentes?




Na maioria dos casos sim, pois cada sessão é acessível por um único request de um único cliente.

# Atributos e concorrência



Sim, mas...e se o mesmo cliente abrir **dois browsers** para a mesma aplicação?



Bem, neste caso, você precisa sincronizar o acesso aos atributos usando a sessão.

# Atributos e concorrência

- Apenas **variáveis locais ao método de serviço** e **atributos do Request** são protegidas de *threads*
- **Variáveis de instância não são** protegidas e não devem ser usadas para guardar estado

Use os diversos  
atributos para  
isso.  
Sincronizados  
quando for o  
caso, é claro!





# Repasse de requisições

- Um componente pode atender uma requisição e não devolver a resposta!
  - Ele **repassa** a requisição para outro componente (servlet ou página JSP)
- O repasse de requisições é realizado por um objeto chamado **RequestDispatcher**
  - Use **atributos do HttpServletRequest** para passar dados para o outro componente!

# Repasse de requisições

- Veja como é simples repassar uma requisição:

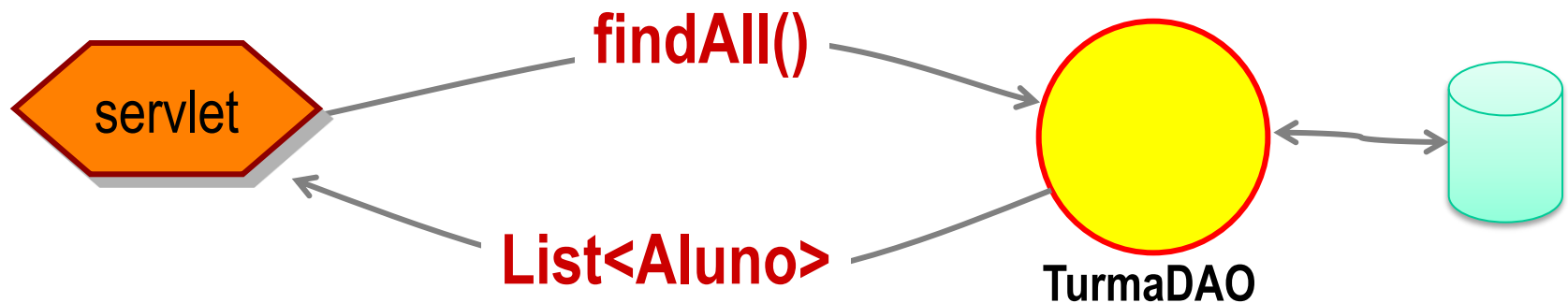
```
//código de um doGet() de um Servlet
EntityManager em =
    PersistenceUtil.getEntityManager();
TurmasDAO dao = new TurmasDAO(em);
List<Turma> turmas = dao.findAll(); //pega todas

request.setAttribute("turmas", turmas);

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

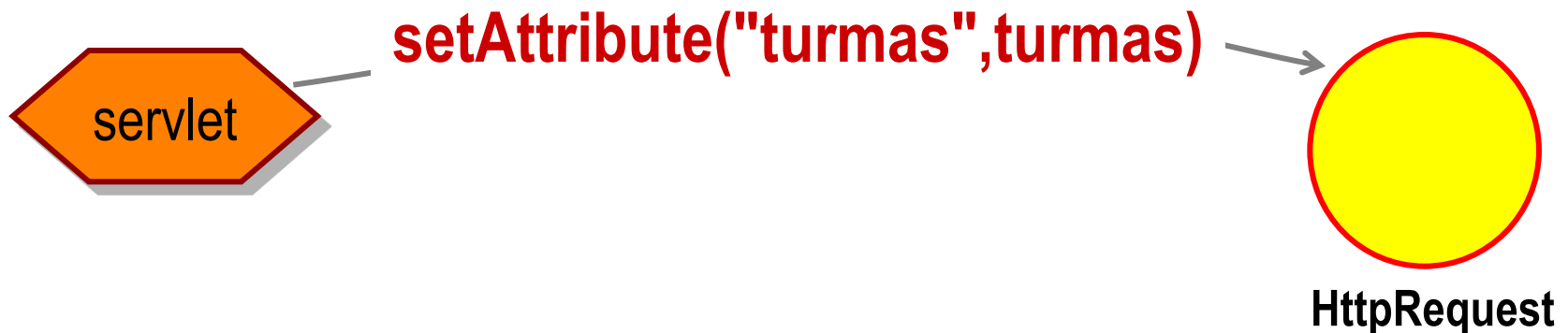
# Repasse de requisições

- O servlet busca os registros de turmas no BD através de um DAO



# Repasse de requisições

- O servlet guarda a lista de turmas como atributo do request



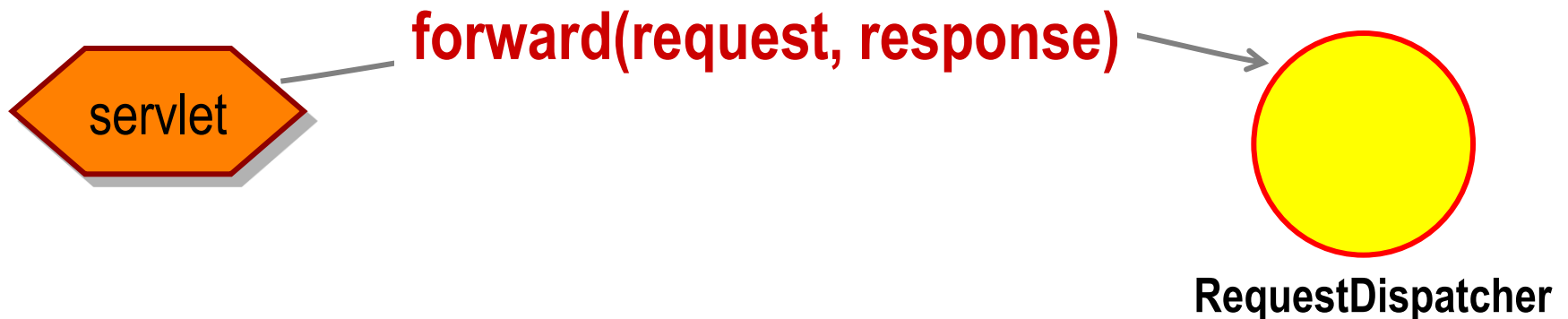
# Repasse de requisições

- O servlet pede um "despachante" de requisições para o request



# Repasse de requisições

- O servlet pede para o despachante repassar a requisição para o outro recurso



# Repasse de requisições

- A interface RequestDispatcher

<<interface>>

***RequestDispatcher***

***forward(ServletRequest, ServletResponse)***  
***include(ServletRequest, ServletResponse)***

# Repasse de requisições

## ■ Obtendo um *dispatcher* do request

```
request.getRequestDispatcher("docm.jsp");
```

- O caminho para o componente pode ser relativo ao componente que foi chamado. Se começar com "/" será tomado a partir do diretório raiz do contexto.

## ■ Obtendo um *dispatcher* do ServletContext

```
getServletContext().getRequestDispatcher("/docm.jsp");
```

- Você deve começar o caminho **sempre** com "/" (diretório raiz do contexto).



# Bibliografia

- **Bashan, B., Sierra, K. e Bates, B.** "Head First Servlets & JSP". Capítulo 5. 2005.