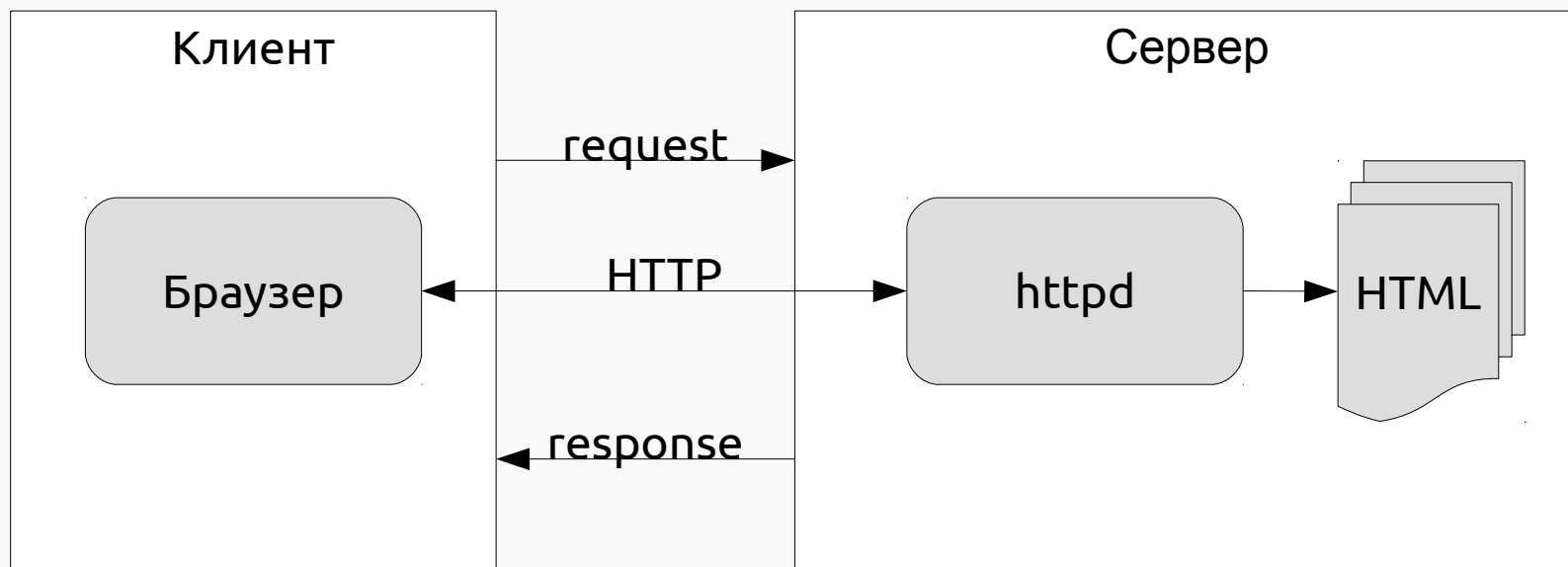


# 1. Введение

# Интернет-приложения



# Стандарты и протоколы сети Интернет

- Hypertext Transfer Protocol (HTTP) — предназначен для передачи гипертекста между клиентом и сервером.
- Hypertext Markup Language (HTML) — язык разметки гипертекста

## 2. Протокол HTTP

# Протокол HTTP

- Протокол прикладного уровня
- Основа — технология «клиент-сервер»
- Может быть использован в качестве «транспорта» для других протоколов прикладного уровня
- Основной объект манипуляции — *ресурс*, на который указывает *URI*
- Обмен сообщениями идёт по схеме «запрос-ответ»
- Stateless-протокол (один запрос — одно соединение). Для реализации сессий используются cookies.

# URI, URL и URN

- URI (Uniform Resource Identifier) — уникальный идентификатор ресурса — символьная строка, позволяющая идентифицировать ресурс.
- URL (Uniform Resource Locator) — URI, позволяющий определить местонахождение ресурса.
- URN (Uniform Resource Name) — URI, содержащий единообразное имя ресурса (не указывает на его местонахождение).

# URI, URL и URN (продолжение)

- URI:  
<схема> : <идентификатор - в - зависимости - от - схемы>
- URL:  
<http://cs.ifmo.ru/spip.html>  
[../task.shtml](#)  
<mailto:Joe.Bloggs@somedomain.com>
- URN:  
<urn:isbn:5170224575>  
<urn:sha1:YNCKHTQCWBTRNJIV4WNAE52SJUQ CZ05C>

# REST

- *Representational State Transfer* (передача состояния представления) - подход к архитектуре сетевых протоколов, обеспечивающих доступ к информационным ресурсам.
- Основные концепции:
  - Данные должны передаваться в виде небольшого числа стандартных форматов (HTML, XML, JSON).
  - Сетевой протокол должен поддерживать кэширование, не должен зависеть от сетевого слоя, не должен сохранять информацию о состоянии между парами «запрос-ответ».
- Антипод REST — подход, основанный на вызове удаленных процедур (*Remote Procedure Call — RPC*).



# Структура запроса HTTP

- Стартовая строка:  
Метод URI HTTP/Версия  
GET /spip.html HTTP/1.1
- Заголовки:  
Host: cs.ifmo.ru  
User-Agent: Mozilla/5.0 (X11; U; Linux i686; ru; rv:1.9b5)  
Gecko/2008050509 Firefox/3.6  
Accept: text/html  
Connection: close
- Тело сообщения

# Структура ответа HTTP

- **Стартовая строка:**  
HTTP/Версия КодСостояния Пояснение  
HTTP/1.1 200 Ok
- **Заголовки:**  
Server: Apache/2.2.11 (Win32) PHP/5.3.0  
Last-Modified: Sat, 16 Jan 2010 21:16:42 GMT  
Content-Type: text/plain; charset=windows-1251  
Content-Language: ru
- **Тело сообщения**

# Методы HTTP

- OPTIONS — определение возможностей сервера.
- GET — запрос содержимого ресурса.
- HEAD — аналог GET, но в ответе отсутствует тело.
- POST — передача данных ресурсу.
- PUT — загрузка содержимого запроса на указанный URI.

# Коды состояния

- Состоят из 3-х цифр.
- Первая цифра — *класс состояния*:
  - «1» — Informational — информационный;
  - «2» — Success — успешно;
  - «3» — Redirection — перенаправление;
  - «4» — Client error — ошибка клиента;
  - «5» — Server error — ошибка сервера.
- Примеры:
  - 201 Webpage Created
  - 403 Access allowed only for registered users
  - 507 Insufficient Storage

# Заголовки HTTP

- Формат:  
ключ:значение
- 4 группы:
  - General Headers — могут включаться в любое сообщение клиента и сервера. Пример — Cache-Control.
  - Request Headers — используются только в запросах клиента. Пример — Referer.
  - Response Headers — используются только в запросах сервера. Пример — Allow.
  - Entity Headers — сопровождают любую сущность сообщения. Пример — Content-Language.

# Примеры сообщений HTTP

- Запрос клиента:  
GET /iaps/labs HTTP/1.1  
Host: cs.ifmo.ru  
User-Agent: Mozilla/5.0 (X11; U; Linux i686; ru; rv:1.9b5)  
Gecko/2008050509 Firefox/3.6.14  
Accept: text/html  
Connection: close
- Ответ сервера:  
HTTP/1.0 200 OK  
Date: Wed, 02 Mar 2011 11:11:11 GMT  
Server: Apache  
X-Powered-By: PHP/5.2.4-2ubuntu5wm1  
Last-Modified: Wed, 02 Mar 2011 11:11:11 GMT  
Content-Language: ru  
Content-Type: text/html; charset=utf-8  
Content-Length: 1234  
Connection: close  
...HTML-код запрашиваемой страницы...

# 3. Основы HTML

# Что такое HTML

- Стандартный язык разметки документов в Интернете.
- Интерпретируется *браузером* и отображается в виде документа.
- Разработан в 1989-91 годах *Тимом Бернерсом-Ли*.
- Является частным случаем *SGML* (стандартного обобщённого языка разметки).
- Существует нотация *XHTML*, являющаяся частным случаем языка *XML*.



- *Браузер* — программа, отображающая HTML-документ в его отформатированном виде.
- Популярные браузеры:
  - Google Chrome
  - Mozilla Firefox
  - Internet Explorer
  - Apple Safari
  - Opera

# Структура HTML-документа

- Документ состоит из *элементов*.
- Начало и конец элемента обозначаются *тегами*:  
`<b>текст</b>`
- Теги могут быть пустыми:  
`<br>`
- Теги могут иметь *атрибуты*:  
`<a href="http://www.example.com">Здесь элемент содержит атрибут href.</a>`
- Элементы могут быть вложенными:  
`<b>  
    Этот текст будет полужирным,  
    <i>а этот - ещё и курсивным</i>  
</b>`

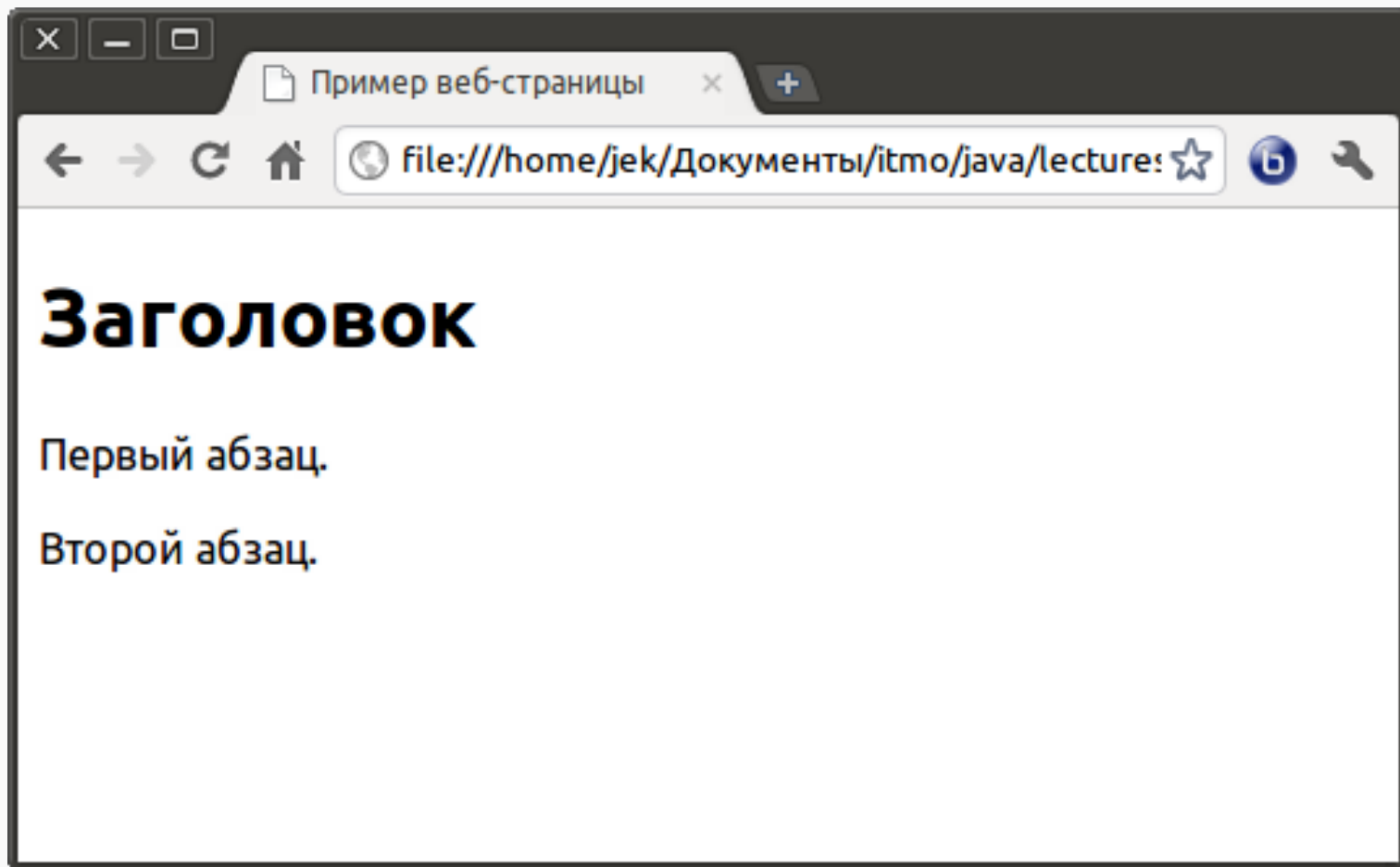
# Структура HTML-документа (продолжение)

- Документ должен начинаться со строки объявления версии HTML:  
`<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN"  
"http://www.w3.org/TR/html4/strict.dtd">`
- Начало и конец документа обозначаются тегами `<html>` и `</html>`.
- Внутри этих тегов должны находиться заголовок (`<head>...</head>`) и тело документа (`<body>...</body>`).

# Пример HTML-документа

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN"
"http://www.w3.org/TR/html4/strict.dtd">
<html>
  <head>
    <meta http-equiv="Content-Type" content="text/html;
      charset=utf-8">
    <title>Пример веб-страницы</title>
  </head>
  <body>
    <h1>Заголовок</h1>
    <!-- Комментарий -->
    <p>Первый абзац.</p>
    <p>Второй абзац.</p>
  </body>
</html>
```

# Пример HTML-документа (продолжение)



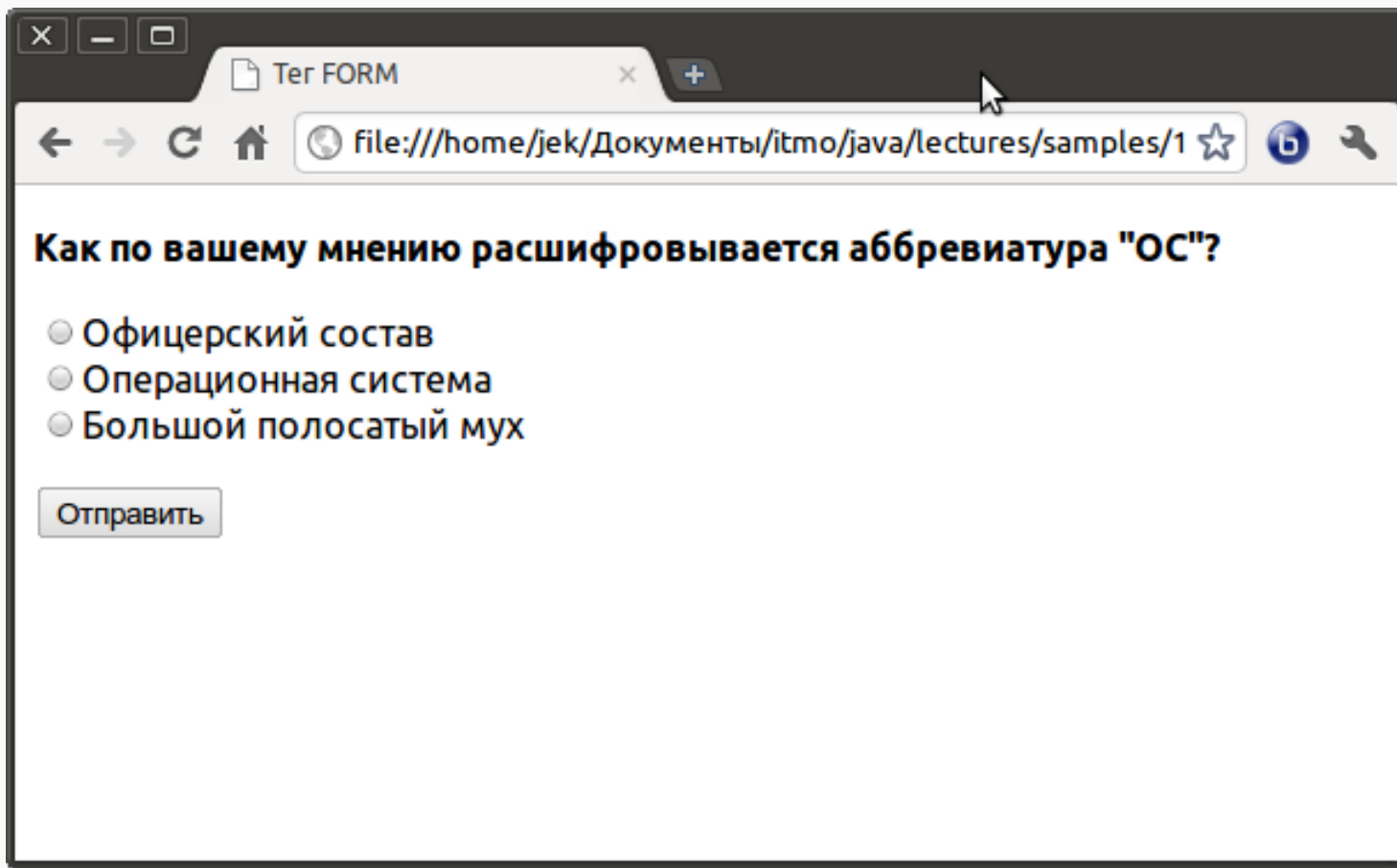
# HTML-формы

- Предназначены для обмена данными между пользователем и сервером.
- Документ может содержать любое число форм, но одновременно на сервер может быть отправлена только одна из них.
- Вложенные формы запрещены.
- Границы формы задаются тегами `<form>...</form>`.
- Метод HTTP задаётся атрибутом `method` тега `<form>`:  
`<form method="GET" action="URL">...</form>`

# Пример HTML-формы

```
<form method="POST" action="handler.php">
  <p><b>Как по вашему мнению расшифровывается
    аббревиатура "ОС"?</b></p>
  <p><input type="radio" name="answer"
    value="a1">Офицерский состав<br>
  <input type="radio" name="answer"
    value="a2">Операционная система<br>
  <input type="radio" name="answer"
    value="a3">Большой полосатый мух</p>
  <p><input type="submit"></p>
</form>
```

# Пример HTML-формы (продолжение)



Ter FORM

file:///home/jek/Документы/itmo/java/lectures/samples/1

**Как по вашему мнению расшифровывается аббревиатура "ОС"?**

- ☐ Офицерский состав
- ☐ Операционная система
- ☐ Большой полосатый мух

Отправить



# Объектная модель документа (DOM)

- DOM — это платформо-независимый интерфейс, позволяющий программам и скриптам получить доступ к содержимому HTML-документов.
- Стандартизирована W3C.
- Документ в DOM представляет собой *дерево узлов*.
- Узлы связаны между собой отношением «родитель-потомок».
- Используется для динамического изменения страниц HTML.

# Объектная модель документа (DOM, продолжение)

специализация 220111

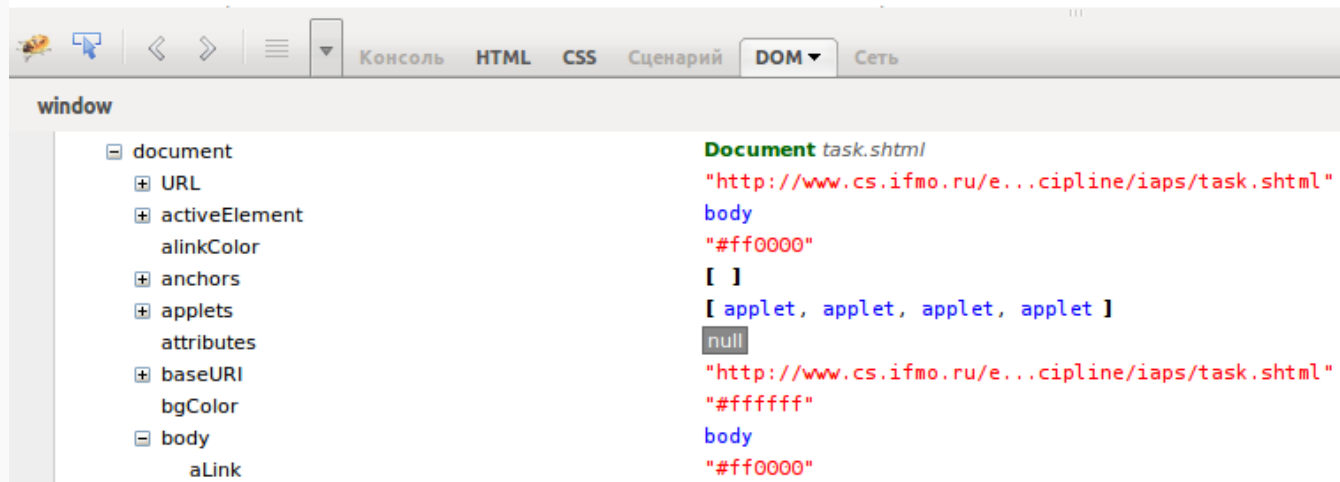
дисциплины документация студенты

кафедра  
карта  
спец-и

В рамках лабораторных работ по дисциплине "Системы программирования И" следующие задания:

## Лабораторная работа #1

На языке Java написать консольную программу, которая определяет, какие элементы массива A входят в заданную область S. Программа должна запрашивать данные у пользователя и выводить на экран координаты точек, входящих в область. Для хранения координат и параметра R использовать типы данных с плавающей точкой. Для ввода использовать стандартный поток ввода System.in.



The screenshot shows a web browser's developer console with the DOM tab selected. The left pane displays the DOM tree structure, and the right pane shows the details of the selected 'document' object.

**DOM Tree Structure:**

- document
  - URL
  - activeElement
    - alinkColor
  - anchors
  - applets
  - attributes
  - baseURI
  - bgColor
  - body
    - aLink

**Document Object Details (task.shtml):**

- URL: "http://www.cs.ifmo.ru/e...cipline/iaps/task.shtml"
- body
  - alinkColor: "#ff0000"
  - anchors: [ ]
  - applets: [ applet, applet, applet, applet ]
  - attributes: null
  - baseURI: "http://www.cs.ifmo.ru/e...cipline/iaps/task.shtml"
  - bgColor: "#ffffff"
  - body
    - alink: "#ff0000"

# 4. Основы CSS

# Что такое CSS

- CSS — технология описания внешнего вида документа, написанного языком разметки.
- Используется для задания цветов, шрифтов и других аспектов представления документа.
- Основная цель — разделение содержимого документа и его представления.
- Позволяет представлять один и тот же документ в различных методах вывода (например, обычная версия и версия для печати).

# Источники CSS

- Авторские стили (информация стилей, предоставляемая автором страницы) в виде:
  - Inline-стилей — стиль элемента указывается в его атрибуте style.
  - Встроенных стилей — блоков CSS внутри самого HTML-документа.
  - Внешних таблиц стилей — отдельного файла .css.
- Пользовательские стили:
  - Локальный CSS-файл, указанный пользователем в настройках браузера, переопределяющий авторские стили.
- Стиль браузера:
  - Стандартный стиль, используемый браузером по умолчанию для представления элементов.

# Структура CSS

- Таблица стилей состоит из набора *правил*.
- Каждое правило состоит из набора *селекторов* и *блока определений*:  
селектор, селектор {  
    свойство: значение;  
    свойство: значение;  
    свойство: значение;  
}
- Пример:  
div, td {  
    background-color: red;  
}

# Приоритеты стилей

- Если к одному элементу «подходит» сразу несколько стилей, применён будет наиболее приоритетный.
- Приоритеты рассчитываются таким образом (от большего к меньшему):
  1. свойство задано при помощи `!important`;
  2. стиль прописан напрямую в теге;
  3. наличие идентификаторов (`#id`) в селекторе;
  4. количество классов (`.class`) и псевдоклассов (`:pseudoclass`) в селекторе;
  5. количество имён тегов в селекторе.
- Имеет значение относительный порядок расположения свойств — свойство, указанное позже, имеет приоритет.

# Пример CSS

```
p {  
  font-family: "Garamond", serif;  
}
```

```
h2 {  
  font-size: 110 %;  
  color: red;  
  background: white;  
}
```

```
.note {  
  color: red;  
  background: yellow;  
  font-weight: bold;  
}
```

```
p#paragraph1 {  
  margin: 0;  
}
```

```
a:hover {  
  text-decoration: none;  
}
```

```
#news p {  
  color: blue;  
}
```



# Пример страницы с CSS

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN"
"http://www.w3.org/TR/html4/strict.dtd">
<html>
  <head>
    <meta http-equiv="Content-Type" content="text/html;
charset=utf-8">
    <title>Заголовки</title>
    <style type="text/css">
      h1 { color: #a6780a; font-weight: normal; }
      h2 {
        color: olive;
        border-bottom: 2px solid black;
      }
    </style>
  </head>
  <body>
    <h1>Заголовок 1</h1>
    <h2>Заголовок 2</h2>
  </body>
</html>
```

# SASS / LESS / SCSS

Языки стилей, позволяющие повысить уровень абстракции CSS-кода и упростить структуру таблиц стилей.

По сравнению с «обычным» CSS, имеются следующие особенности:

- Можно использовать переменные (константы и примеси).
- Можно использовать вложенные правила.

Браузеры могут не поддерживать SASS / LESS / SCSS-таблицы стилей — нужен специальный транслятор, который преобразует эти правила в «обычный» CSS.

# Пример SCSS

Код на SCSS:

```
#header {  
  background: #FFFFFF;  
  .error {  
    color: #FF0000;  
  }  
  a {  
    text-decoration:  
      none;  
    &:hover {  
      text-decoration:  
        underline;  
    }  
  }  
}
```

Будет преобразован в  
«обычный» CSS:

```
#header {  
  background: #FFFFFF;  
}  
#header .error {  
  color: #FF0000;  
}  
#header a {  
  text-decoration: none;  
}  
#header a:hover {  
  text-decoration:  
    underline;  
}
```

# Ещё один пример SCSS

## Код на SCSS:

```
@mixin border-radius($radius,$border,
$color) {
  -webkit-border-radius: $radius;
  -moz-border-radius: $radius;
  -ms-border-radius: $radius;
  border-radius: $radius;
  border:$border solid $color
}

.box {
  @include border-radius(10px,1px,red);
}
```

## «Обычный» CSS:

```
.box {
  -webkit-border-radius: 10px;
  -moz-border-radius: 10px;
  -ms-border-radius: 10px;
  border-radius: 10px;
  border: 1px solid red;
}
```

# 5. Клиентские сценарии на языке JavaScript

# JavaScript и клиентские сценарии

- JavaScript — объектно-ориентированный скриптовый язык программирования.
- Используется для придания интерактивности веб-страницам.
- Основные архитектурные черты:
  - динамическая типизация;
  - слабая типизация;
  - автоматическое управление памятью;
  - прототипное программирование;
  - функции как объекты первого класса.

# Особенности синтаксиса

- Все идентификаторы регистрозависимы.
- В названиях переменных можно использовать буквы, подчёркивание, символ доллара, арабские цифры.
- Названия переменных не могут начинаться с цифры,
- Для оформления однострочных комментариев используются //, многострочные и внутристрочные комментарии начинаются с /\* и заканчиваются \*/.

# Структура языка

- Ядро (ECMAScript);
- Объектная модель браузера (Browser Object Model);
- Объектная модель документа (Document Object Model).



# Особенности ECMAScript

- Встраиваемый расширяемый не имеющий средств ввода/вывода язык программирования.
- 5 примитивных типов данных — Number, String, Boolean, Null и Undefined.
- Объектный тип данных — Object.
- 15 различных видов инструкций.

# Особенности ECMAScript (продолжение)

Блок не ограничивает область видимости функции:

```
function foo() {  
    var sum = 0;  
    for (var i = 0; i < 42; i += 2) {  
        var tmp = i + 2;  
        sum += i * tmp;  
    }  
    for (var i = 1; i < 42; i += 2) {  
        sum += i*i;  
    }  
    alert(tmp);  
    return sum;  
}  
  
foo();
```

# Особенности ECMAScript (продолжение)

Если переменная объявляется вне функции, то она попадает в глобальную область видимости:

```
var a = 42;  
  
function foo() {  
    alert(a);  
}  
  
foo();
```

# Особенности ECMAScript (продолжение)

Функция — это тоже объект:

```
// объявление функции
function sum(arg1, arg2) {
    return arg1 + arg2;
}
```

```
// задание функции с помощью инструкции
var sum2 = function(arg1, arg2) {
    return arg1 + arg2;
};
```

```
// задание функции с использованием
// объектной формы записи
var sum3 = new Function("arg1", "arg2",
    "return arg1 + arg2;");
```

# Объектная модель браузера

- ВОМ - прослойка между ядром и DOM.
- Основное предназначение — управление окнами браузера и обеспечение их взаимодействия.
- Специфична для каждого браузера.
- Каждое из окон браузера представляется объектом `window`:  

```
var contentsWindow;  
contentsWindow =  
    window.open("http://cs.ifmo.ru", "contents");
```

# Объектная модель браузера (продолжение)

- Возможности BOM:
  - управление фреймами,
  - поддержка задержки в исполнении кода и зацикливания с задержкой,
  - системные диалоги,
  - управление адресом открытой страницы,
  - управление информацией о браузере,
  - управление информацией о параметрах монитора,
  - ограниченное управление историей просмотра страниц,
  - поддержка работы с HTTP cookie.

# Объектная модель документа

- С помощью JavaScript можно производить следующие манипуляции:
  - получение узлов:  
`document.all("image1").outerHTML;`
  - изменение узлов;
  - изменение связей между узлами;
  - удаление узлов.

# Встраивание в веб-страницы

- Внутри страницы:  

```
<script type="text/javascript">  
    alert('Hello, World!');  
</script>
```
- Внутри тега:  

```
<a href="delete.php" onclick="return confirm('Вы  
уверены? ');">Удалить</a>
```
- Отделение от разметки (используется DOM):  

```
window.onload = function() {  
    var linkWithAlert = document.getElementById("alertLink");  
    linkWithAlert.onclick = function() {  
        return confirm('Вы уверены?');  
    };  
};  
...  
<a href="delete.php" id="alertLink">Удалить</a>
```
- В отдельном файле:  

```
<script type="text/javascript"  
    src="http://Путь_к_файлу_со_скриптом"></script>
```



# 6. DHTML и AJAX

- Dynamic HTML — способ создания интерактивного веб-сайта, использующий сочетание:
  - статичного языка разметки HTML;
  - выполняемого на стороне клиента скриптового языка JavaScript;
  - CSS (каскадных таблиц стилей);
  - DOM (объектной модели документа).

# Пример страницы DHTML

```
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01//EN"
  "http://www.w3.org/TR/html4/strict.dtd">
<html>

<head>
  <title>Заголовок страницы</title>
  <script type="text/javascript">
    window.onload= function () {
      myObj = document.getElementById("navigation");
      // .... какой-то код
    }
  </script>
</head>

<body>
  <div id="navigation">
  </div>
</body>
</html>
```

# Что такое AJAX

- *AJAX (Asynchronous Javascript and XML)* — подход к построению интерактивных пользовательских интерфейсов веб-приложений.
- Основан на «фоновом» обмене данными браузера с веб-сервером.
- При обмене данными между клиентом и сервером веб-страница не перезагружается полностью.

# Основные принципы AJAX

- Использование технологии динамического обращения к серверу «на лету», без перезагрузки всей страницы полностью, например:
  - с использованием XMLHttpRequest;
  - через динамическое создание дочерних фреймов;
  - через динамическое создание тега `<script>`.
- Использование DHTML для динамического изменения содержания страницы.

# XMLHttpRequest

- XMLHttpRequest (XMLHttpRequest, XHR) — набор API, позволяющий осуществлять HTTP-запросы к серверу без необходимости перезагружать страницу.
- Данные можно пересылать в виде XML, JSON, HTML или просто неструктурированным текстом.
- При пересылке используется текстовый протокол HTTP и потому данные должны передаваться в виде текста.

# XMLHttpRequest (пример)

```
var req;

function loadXMLDoc(url) {
    req = null;
    if (window.XMLHttpRequest) {
        try {
            req = new XMLHttpRequest();
        } catch (e){}
    } else if (window.ActiveXObject) {
        try {
            req = new ActiveXObject('Msxml2.XMLHTTP');
        } catch (e){
            try {
                req = new ActiveXObject('Microsoft.XMLHTTP');
            } catch (e){}
        }
    }
    if (req) {
        req.open("GET", url, true);
        req.onreadystatechange = processReqChange;
        req.send(null);
    }
}
```

# XMLHttpRequest (пример, продолжение)

```
function processReqChange() {  
    try { // Важно!  
        // только при состоянии "complete"  
        if (req.readyState == 4) {  
            // для статуса "OK"  
            if (req.status == 200) {  
                // обработка ответа  
            } else {  
                alert("Не удалось получить данные:\n" +  
                    req.statusText);  
            }  
        }  
    }  
    catch( e ) {  
        // alert('Ошибка: ' + e.description);  
        // В связи с багом XMLHttpRequest в Firefox  
        // приходится отлавливать ошибку  
    }  
}
```



# Преимущества и недостатки AJAX

- Преимущества:
  - экономия трафика;
  - уменьшение нагрузки на сервер;
  - ускорение реакции интерфейса;
- Недостатки:
  - отсутствие интеграции со стандартными инструментами браузера;
  - динамически загружаемое содержимое недоступно поисковикам;
  - старые методы учёта статистики сайтов становятся неактуальными;
  - усложнение проекта;
  - требуется включенный JavaScript в браузере.

# Протокол WebSocket

- *WebSocket* — протокол полнодуплексной связи поверх TCP-соединения, предназначенный для обмена сообщениями между браузером и веб-сервером в режиме реального времени.
- Позволяет серверу отправлять данные браузеру без дополнительного запроса со стороны клиента.
- Обмен данными ведётся через отдельное TCP-соединение.
- Поддерживается всеми современными браузерами (даже IE).
- Альтернатива — AJAX + Long Polling.

# Протокол WebSocket (продолжение)

```
<script>
```

```
var websocket = new WebSocket('ws://localhost/echo');
```

```
websocket.onopen = function(event) {  
    alert('onopen');  
    websocket.send("Hello Web Socket!");  
};
```

```
websocket.onmessage = function(event) {  
    alert('onmessage, ' + event.data);  
    websocket.close();  
};
```

```
websocket.onclose = function(event) {  
    alert('onclose');  
};
```

```
</script>
```

# 7. Серверные сценарии

# Веб-сайты и веб-приложения

- Веб-сайт — это набор статических файлов, HTML-страниц, графики и других ресурсов.
- Веб-приложение — это веб-сайт с той или иной динамической функциональностью на стороне сервера.
- Веб-приложение осуществляет вызов программ на стороне сервера, к примеру:
  - Браузер отправляет на веб-сервер запрос на получение HTML-формы.
  - Веб-сервер формирует HTML-форму и возвращает её браузеру.
  - Браузер отправляет на сервер новый запрос с данными из HTML-формы.
  - Веб-сервер делегирует обработку данных из формы какой-либо программе на стороне сервера.

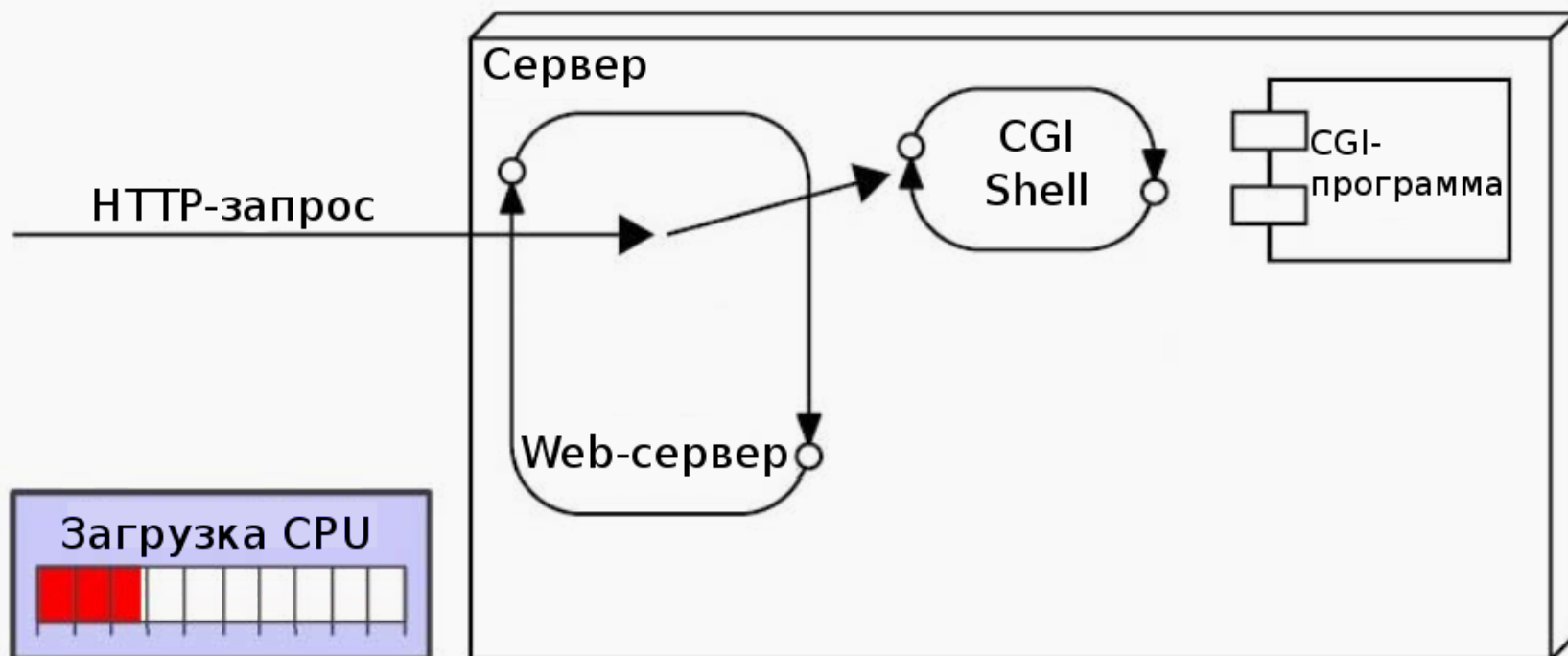
- HTML over HTTP
- Common Gateway Interface (CGI)
- FastCGI
- PHP
- Servlets
- JavaServer Pages (JSP)
- JSP Standard Tag Library (JSTL)
- XML
- Struts
- JavaServer Faces

# CGI-сценарии

- CGI — механизм вызова пользователем программ на стороне сервера.
- Данные отправляются программе посредством HTTP-запроса, формируемого веб-браузером.
- То, какая именно программа будет вызвана, обычно определяется URL запроса.
- Каждый запрос обрабатывается отдельным процессом CGI-программы.
- Взаимодействие программы с веб-сервером осуществляется через `stdin` и `stdout`.

# Выполнение CGI-сценариев

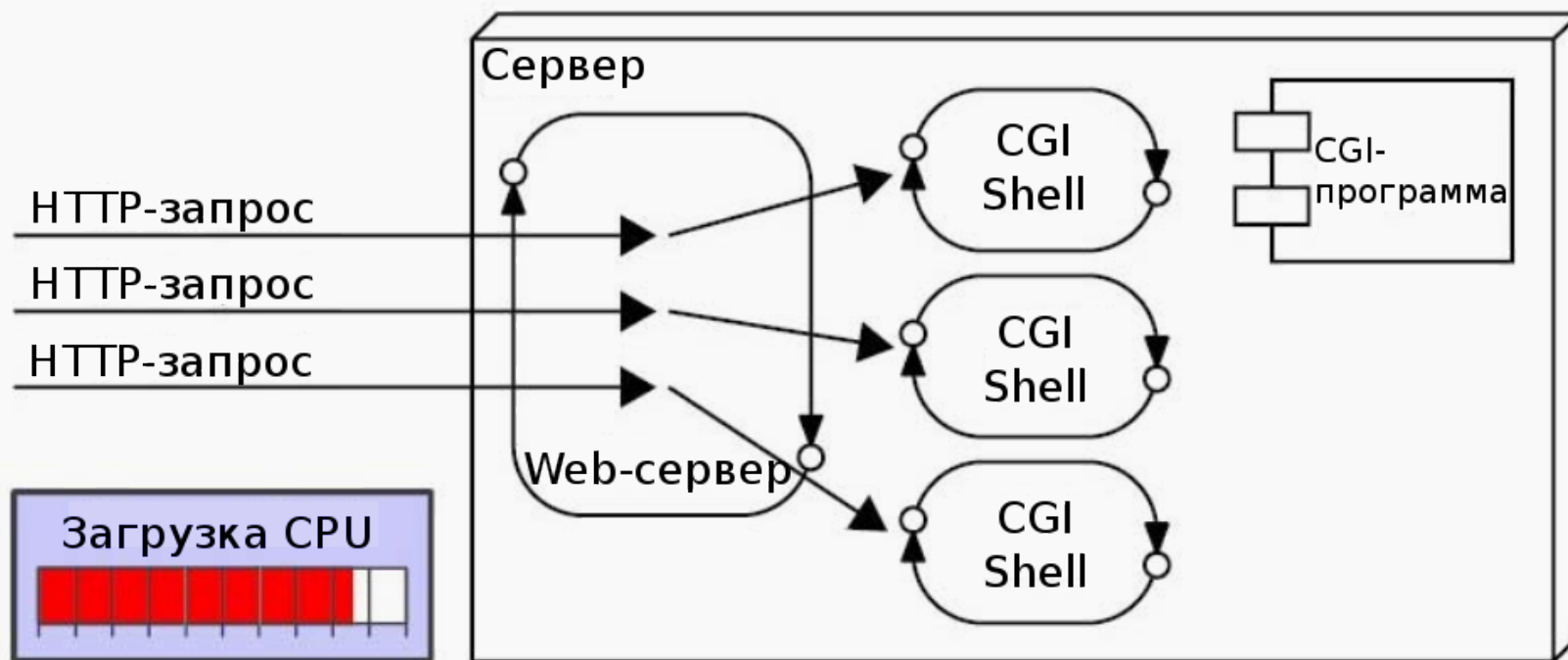
Один запрос:





# Выполнение CGI-сценариев (продолжение)

Параллельная обработка нескольких запросов:



# Пример реализации CGI-сценария

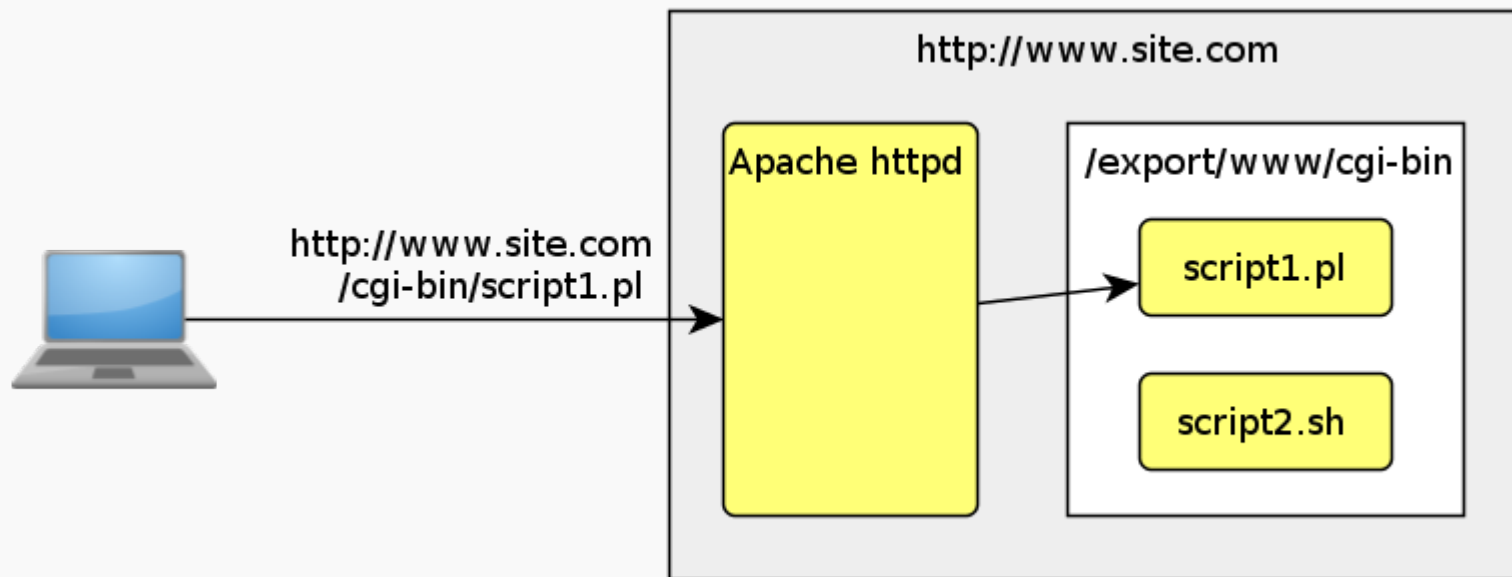
```
#include <stdio.h>
```

```
int main(void) {  
    printf("Content-Type:  
        text/html;charset=UTF-8\n\n");  
  
    printf("<HTML>\n");  
    printf("<HEAD>\n");  
    printf("<TITLE>Hello, World!</TITLE>\n");  
    printf("</HEAD>\n");  
    printf("<BODY>\n");  
    printf("<H1>Hello, World</H1>\n");  
    printf("</BODY>\n");  
    printf("</HTML>\n");  
  
    return 0;  
}
```

# Конфигурация веб-сервера

Apache (httpd.conf):

```
ScriptAlias /cgi-bin/ "/opt/www/cgi-bin/"
```



# Достоинства и недостатки CGI-сценариев

- Достоинства:
  - Программы могут быть написаны на множестве языков программирования.
  - «Падение» CGI-сценария не приводит к «падению» всего сервера.
  - Исключены конфликты при параллельной обработке нескольких запросов.
  - Хорошая поддержка веб-серверами.
- Недостатки:
  - Высокие накладные расходы на создание нового процесса.
  - Плохая масштабируемость.
  - Слабое разделение уровня представления и бизнес-логики.
  - Могут быть платформо-зависимыми.

- Развитие технологии CGI.
- Все запросы могут обрабатываться одним процессом CGI-программы (фактическая реализация определяется программистом).
- Веб-сервер взаимодействует с процессом через UNIX Domain Sockets или TCP/IP (а не через stdin и stdout).

# 8. Серверные сценарии на РНР

- PHP (*PHP: Hypertext Preprocessor*) — скриптовый язык, часто используемый для написания веб-приложений.
- Первая версия разработана в 1994 г. Расмусом Лердорфом (Rasmus Lerdorf).
- Распространяется по лицензии с открытым исходным кодом.

# Синтаксис PHP

- Интерпретатор выполняет код, находящийся внутри ограничителей:  

```
<?php  
    echo 'Hello, world!';  
?>
```
- Имена переменных начинаются с символа «\$»:  

```
$hello = 'Hello, world!';
```
- Инструкции разделяются символом «;»:  

```
$a = 'Hello '; $b = 'world!';  
$c = $b + $a;
```



# Синтаксис PHP (продолжение)

- Весь текст вне ограничителей оставляется интерпретатором без изменений:

```
<html>
  <head>
    <title>Тестируем PHP</title>
  </head>
  <body>
    <?php echo 'Hello, world!'; ?>
  </body>
</html>
```

# Типы данных

- PHP — язык с динамической типизацией; при объявлении переменных их тип не указывается.
- 6 скалярных типов данных — integer, float, double, boolean, string и NULL. Диапазоны числовых типов зависят от платформы.
- 3 не скалярных типа — ресурс (например, дескриптор файла), массив и объект.
- 4 псевдотипа — mixed, number, callback и void.

# Суперглобальные массивы (Superglobal Arrays)

Предопределённые массивы, имеющие глобальную область видимости:

- `$_GLOBALS` — массив всех глобальных переменных.
- `$_SERVER` — параметры, которые ОС передаёт серверу при его запуске.
- `$_ENV` — переменные среды ОС.

# Суперглобальные массивы (продолжение)

- `$_GET`, `$_POST` — параметры GET- и POST-запроса:

```
<?php
    echo 'Привет, '
        .htmlspecialchars($_GET["name"])
        . '!';
?>
```

- `$_FILES` — сведения об отправленных методом POST файлах.
- `$_COOKIE` — массив cookies.
- `$_REQUEST` — содержит элементы из массивов `$_GET`, `$_POST`, `$_COOKIE` и `$_FILES`.
- `$_SESSION` — данные HTTP-сессии.

# Поддержка ООП

- Полная поддержка появилась в PHP5.
- Реализованы все основные механизмы ООП — инкапсуляция, полиморфизм и наследование.
- Поля и методы могут быть приватными (private), публичными (public) и защищёнными (protected).
- Можно объявлять финальные и абстрактные методы и классы (аналогично Java).
- Множественное наследование не поддерживается, но есть интерфейсы и механизм особенностей (traits).
- Объекты передаются по ссылке.
- Обращение к константам, статическим свойствам и методам класса осуществляется с помощью конструкции «::».

# Пример PHP-класса

```
class C1 extends C2 implements I1, I2
{
    private $a;
    protected $b;

    function __construct($a, $b)
    {
        parent::__construct($a, $b);
        $this->a = $a;
        $this->b = $b;
    }

    public function plus()
    {
        return $this->a + $this->b;
    }
    /* ..... */
}

$d = new C1(1, 2);
echo $d->plus(); // 3
```

# Конфигурация и варианты использования интерпретатора PHP

- Конфигурационные параметры хранятся в файле `php.ini`.
- Можно подключать дополнительные *модули*, расширяющие возможности языка (например, добавляющие поддержку взаимодействия с СУБД).
- Способы использования интерпретатора PHP:
  - С помощью SAPI / ISAPI (например, `mod_php` для Apache).
  - С помощью CGI / FastCGI.
  - Через интерфейс командной строки.

# 9. Сервлеты

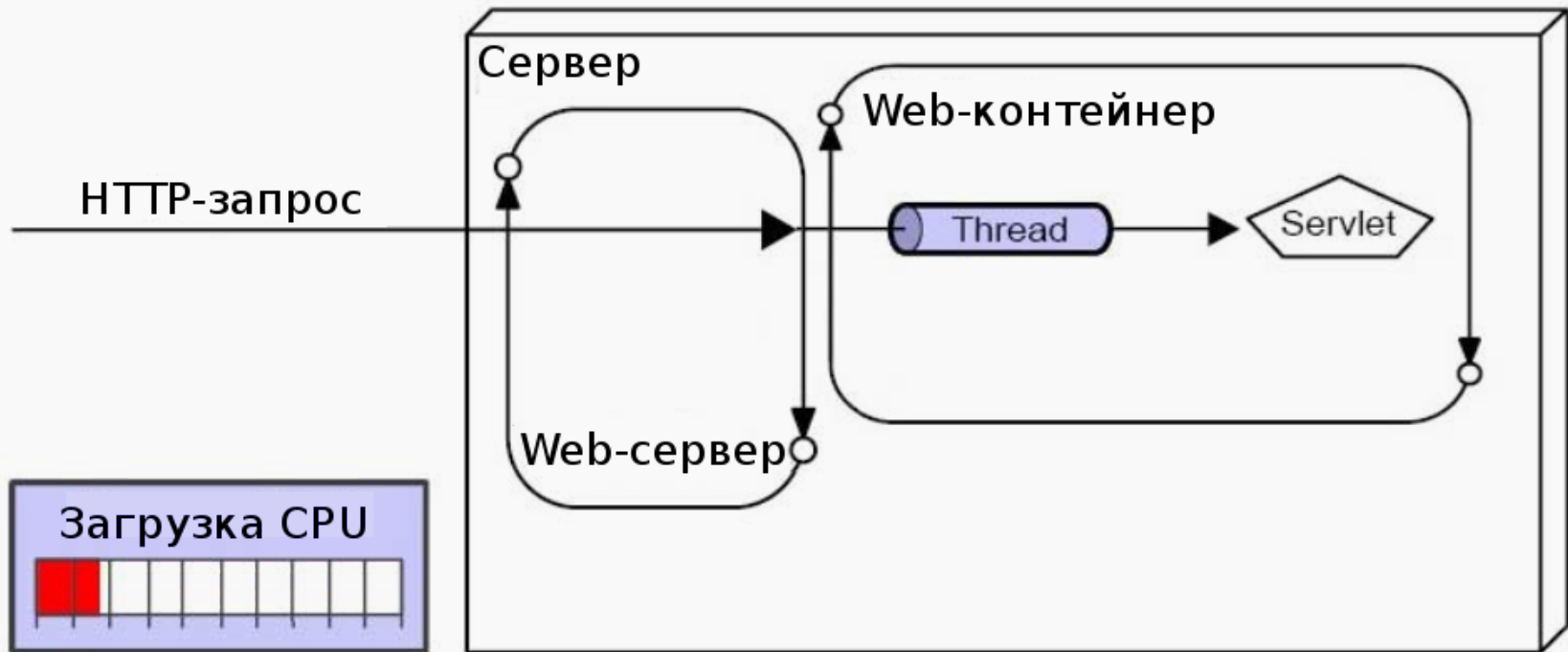


- Набор стандартов и спецификаций для создания корпоративных приложений на Java.
- Спецификации Java EE реализуются серверами приложений:
  - Apache Tomcat
  - Sun / Oracle GlassFish
  - BEA / Oracle WebLogic
  - IBM WebSphere
  - RedHat JBoss

- Сервлеты — это серверные сценарии, написанные на Java.
- Жизненным циклом сервлетов управляет веб-контейнер (он же контейнер сервлетов).
- В отличие от CGI, запросы обрабатываются в отдельных потоках (а не процессах) на веб-контейнере.

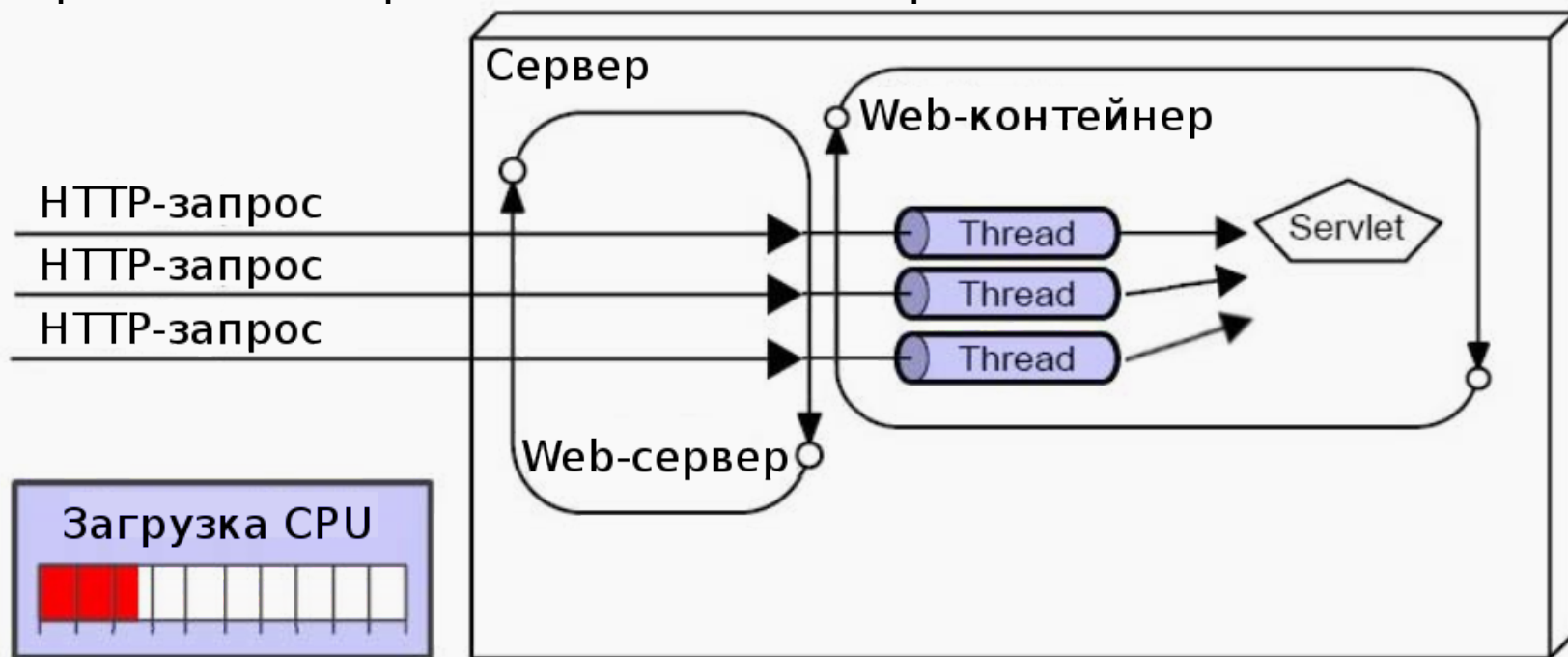
# Обработка HTTP-запросов сервлетом

Один запрос:



# Обработка HTTP-запросов сервлетом (продолжение)

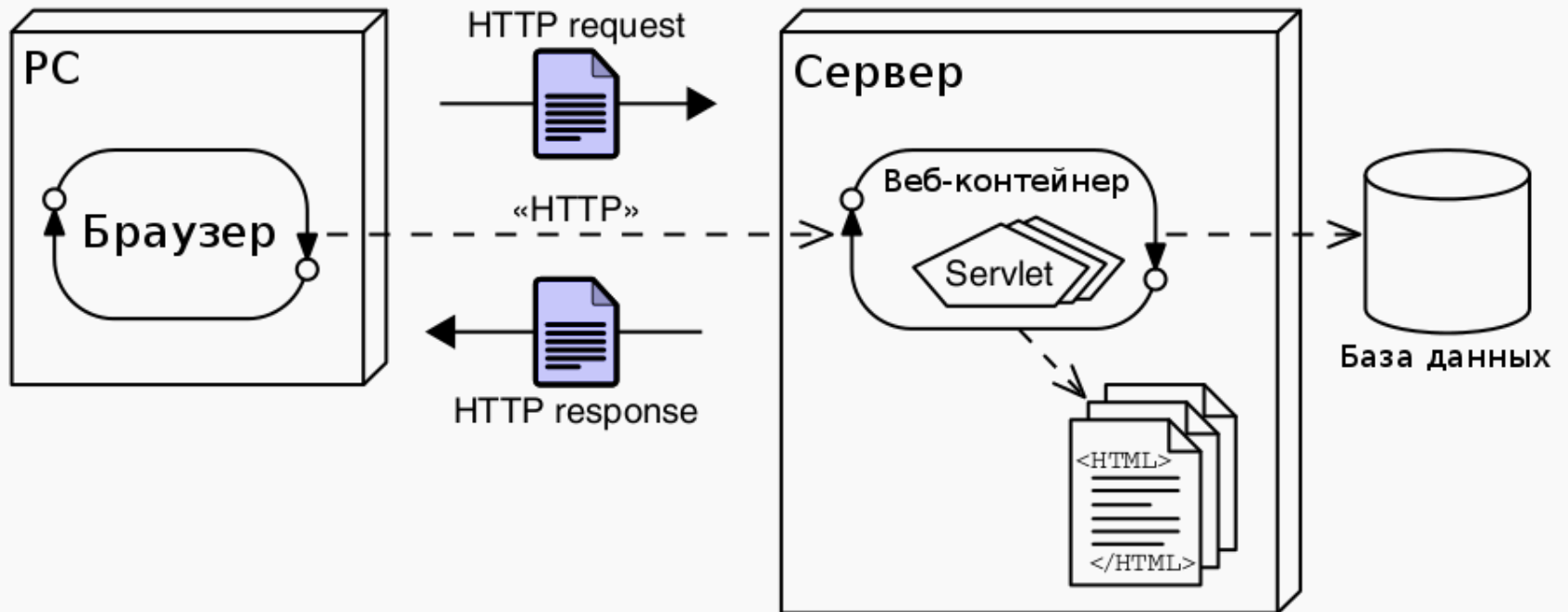
Параллельная обработка нескольких запросов:



- Преимущества сервлетов:
  - Выполняются быстрее, чем CGI-сценарии.
  - Хорошая масштабируемость.
  - Надёжность и безопасность (реализованы на Java).
  - Платформенно-независимы.
  - Множество инструментов мониторинга и отладки.
- Недостатки сервлетов:
  - Слабое разделение уровня представления и бизнес-логики.
  - Возможны конфликты при параллельной обработке запросов.

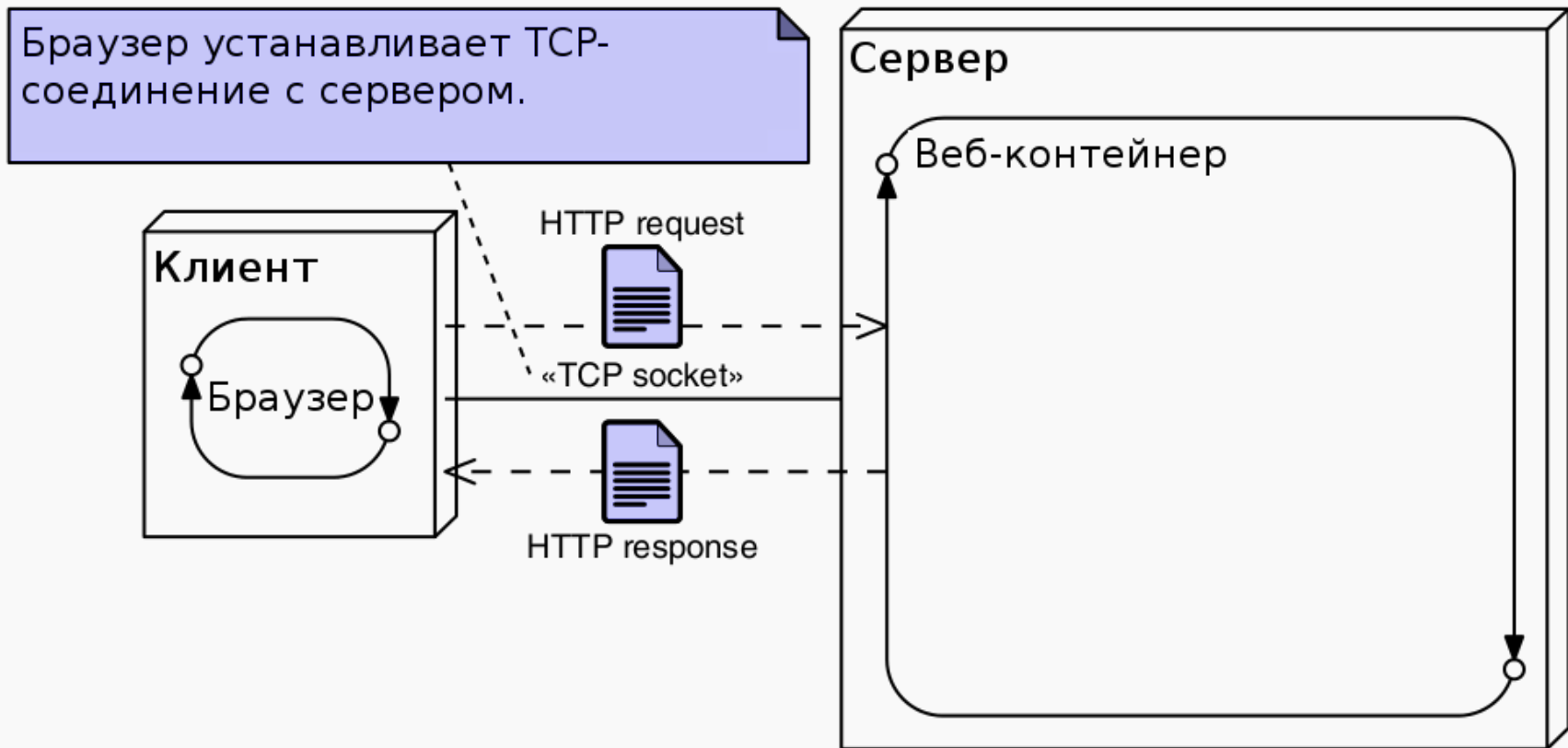
# 10. Разработка сервлетов

# Архитектура веб-контейнера



# Обработка HTTP-запроса

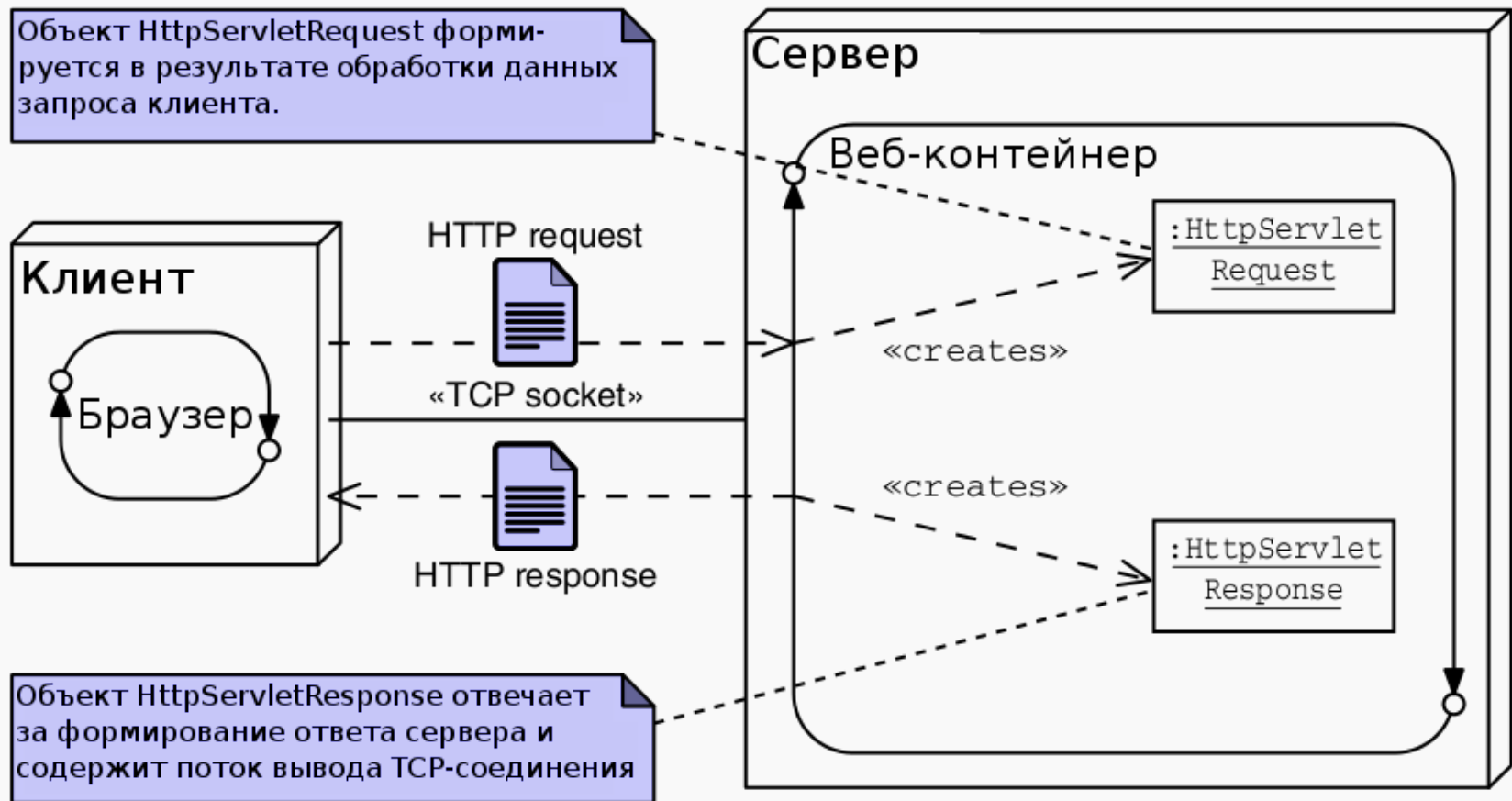
1. Браузер формирует HTTP-запрос и отправляет его на сервер.





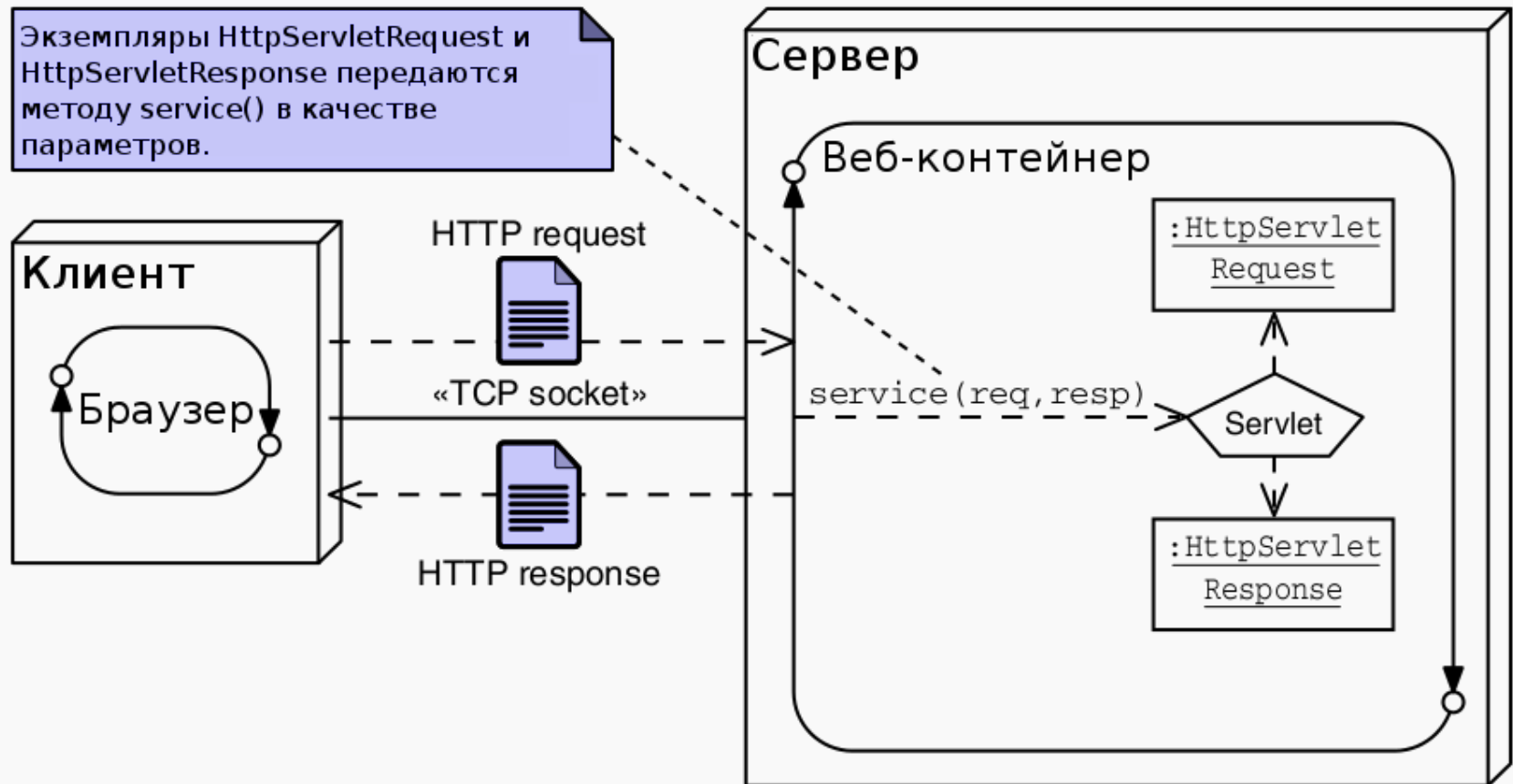
# Обработка HTTP-запроса (продолжение)

2. Веб-контейнер создаёт объекты `HttpServletRequest` и `HttpServletResponse`.



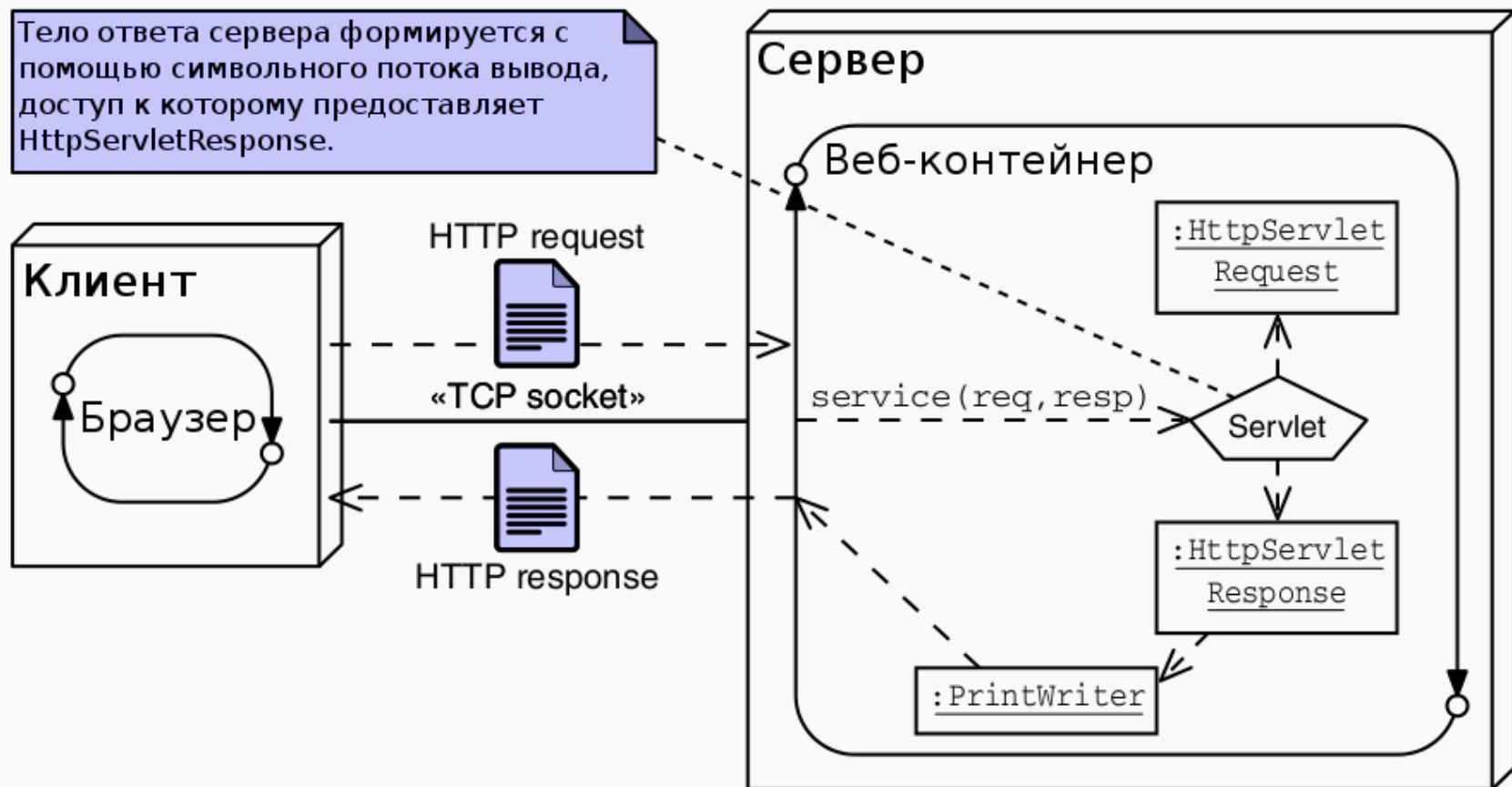
# Обработка HTTP-запроса (продолжение)

3. Веб-контейнер вызывает метод `service` сервлета.

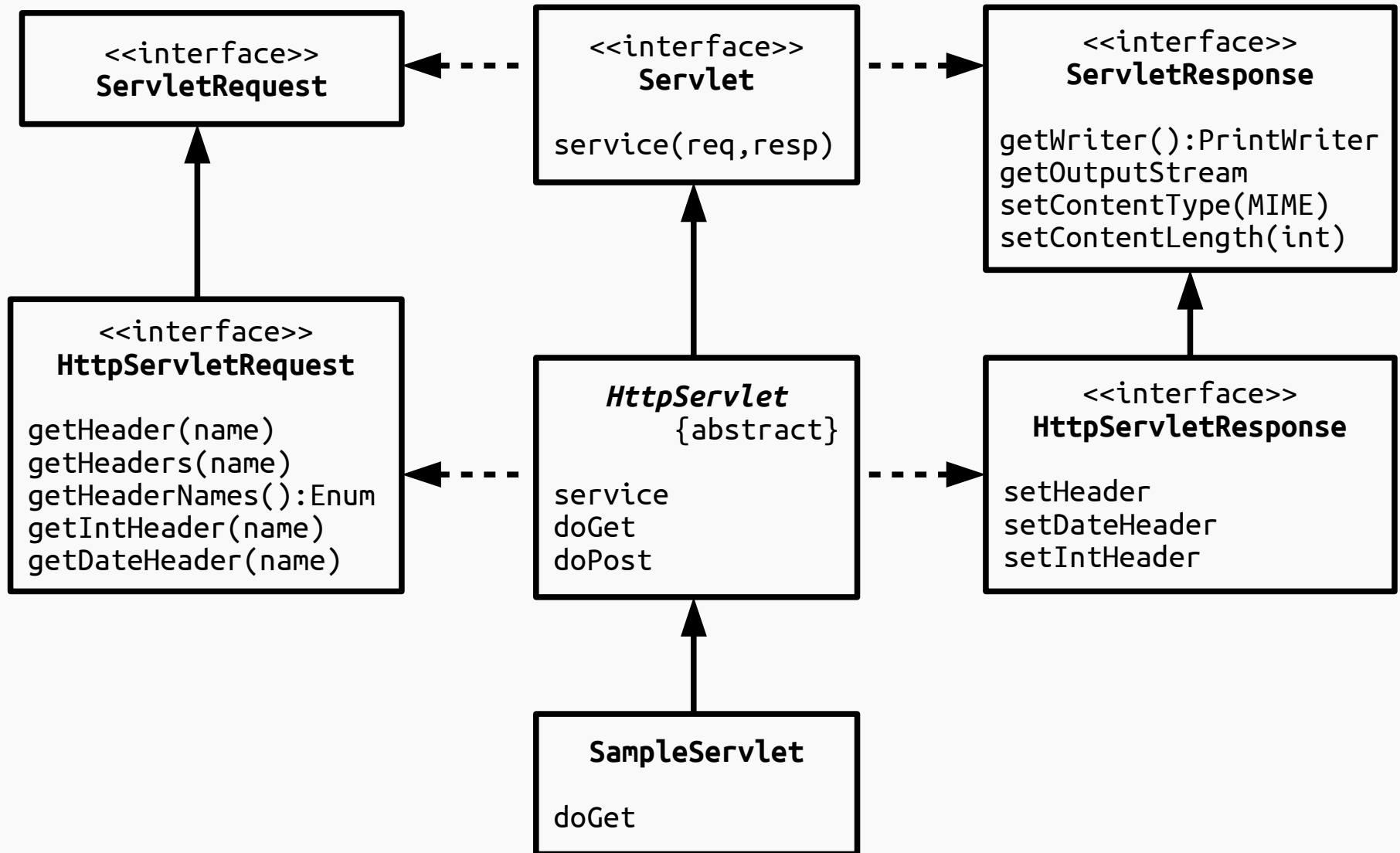


# Обработка HTTP-запроса (продолжение)

4. Сервлет формирует ответ и записывает его в поток вывода  
`HttpServletResponse`.



# HttpServlet API



# Пример сервлета

```
package sample.servlet;

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

// Support classes
import java.io.IOException;
import java.io.PrintWriter;

public class SampleServlet extends HttpServlet {

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

        // Заголовок страницы
        String pageTitle = "Пример сервлета";
```

# Пример сервлета (продолжение)

```
// Content Type
response.setContentType("text/html");

PrintWriter out = response.getWriter();

// Формируем HTML
out.println("<html>");
out.println("<head>");
out.println("<title>" + pageTitle + "</title>");
out.println("</head>");
out.println("<body bgcolor='white'>");
out.println("<h3>" + pageTitle + "</h3>");
out.println("<p>");
out.println("Hello, world!");
out.println("</p>");
out.println("</body>");
out.println("</html>");
```

```
}
```

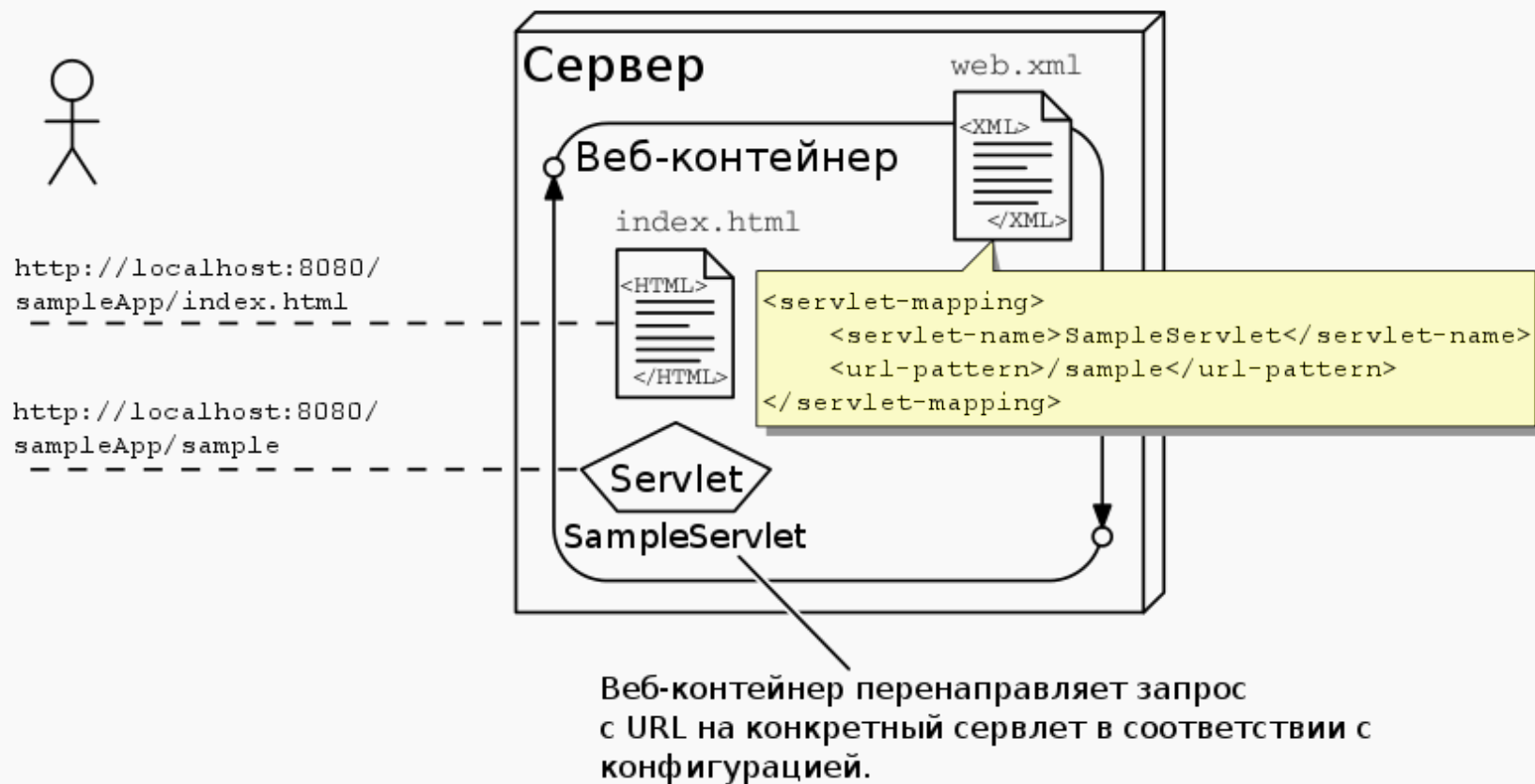
```
}
```

# Конфигурация сервлета



Веб-контейнер создаёт один (строго) экземпляр сервлета на каждую запись в дескрипторе.

# Конфигурация сервлета (продолжение)





# Жизненный цикл сервлета

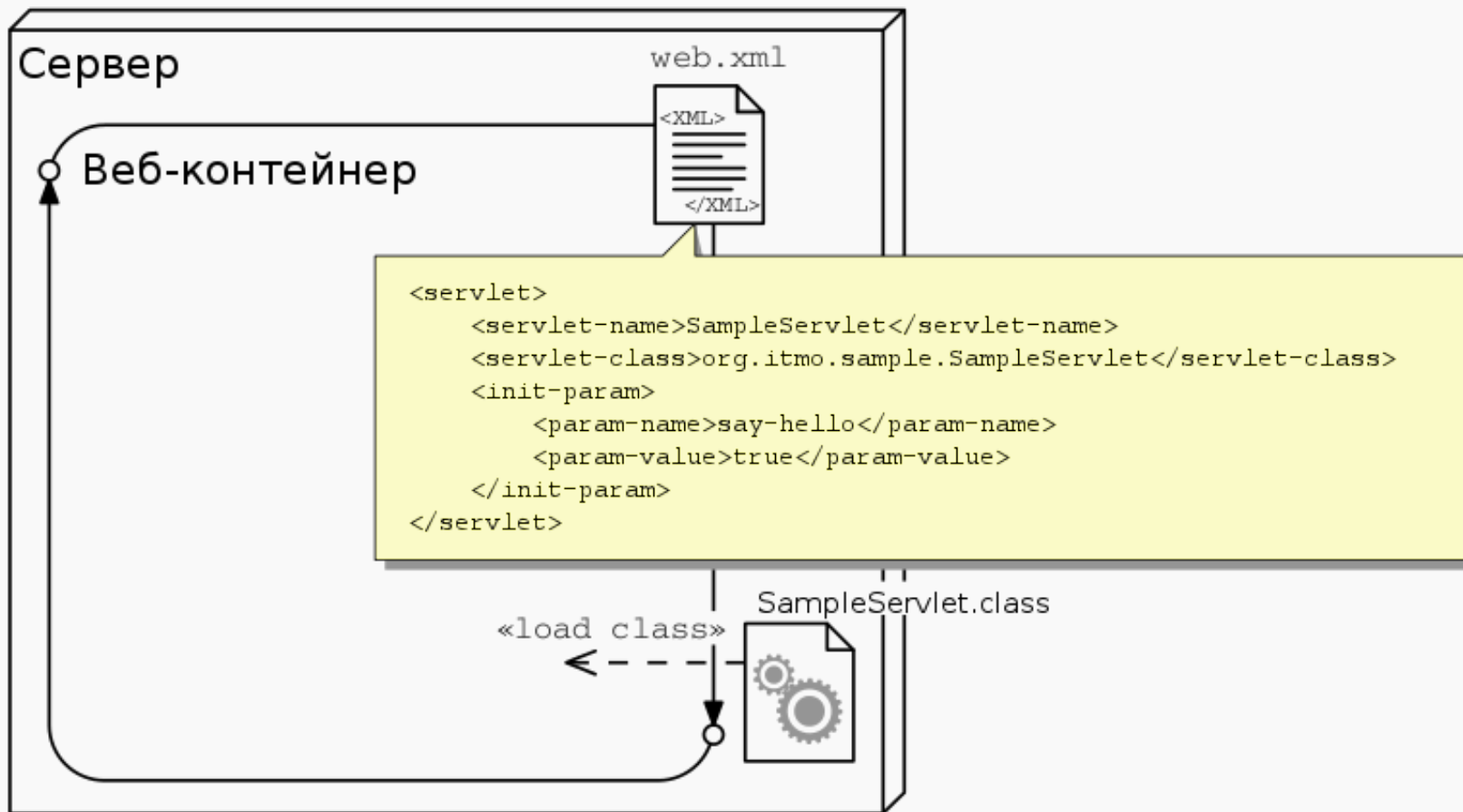
- Жизненным циклом сервлета управляет веб-контейнер.
- Методы, управляющие жизненным циклом, должен вызывать *только* веб-контейнер.



# Жизненный цикл сервлета (продолжение)

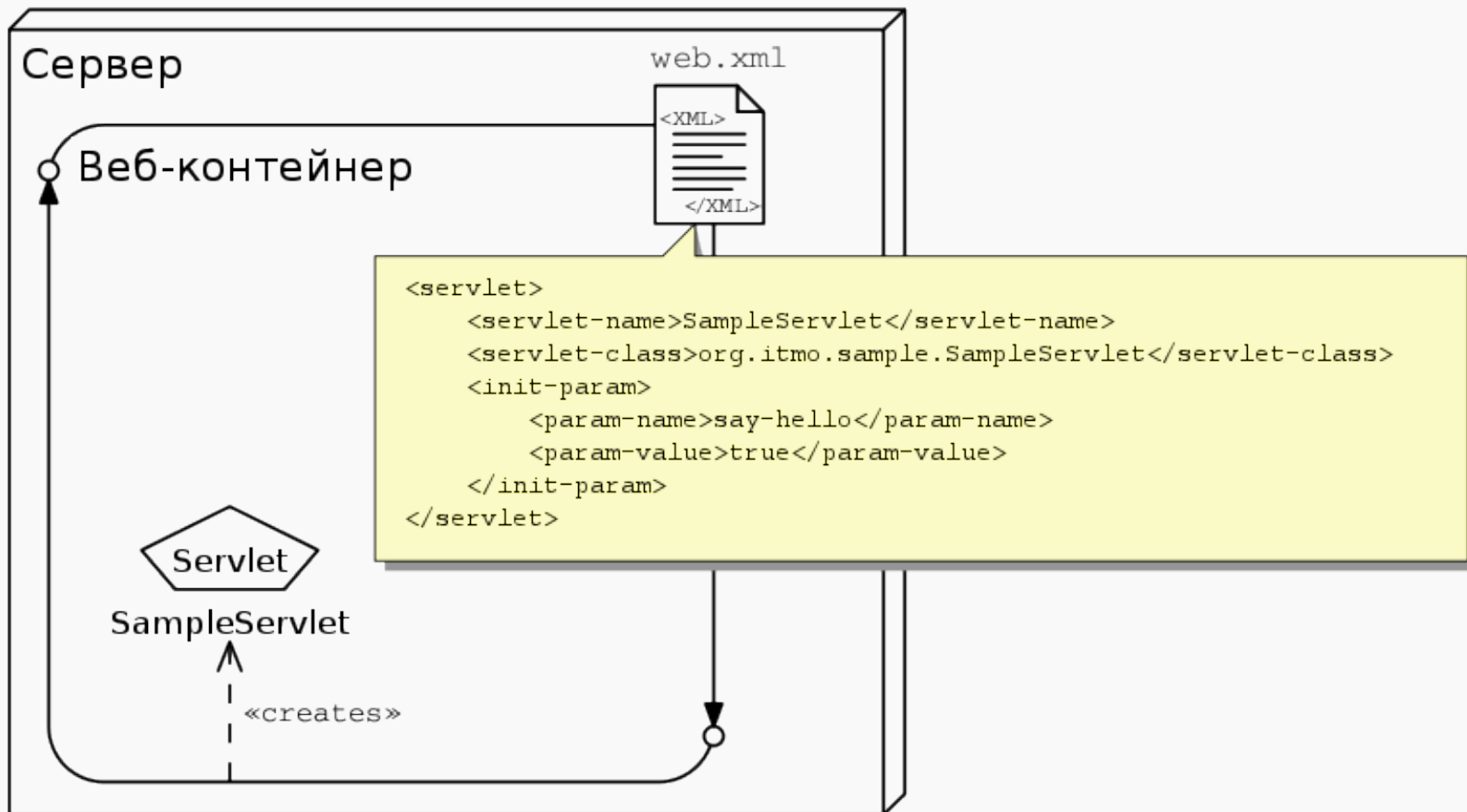
## 1. Загрузка класса сервлета.

Проверяются пути: /WEB-INF/classes/, WEB-INF/lib/\*.jar, стандартные классы Java SE и классы веб-контейнера.



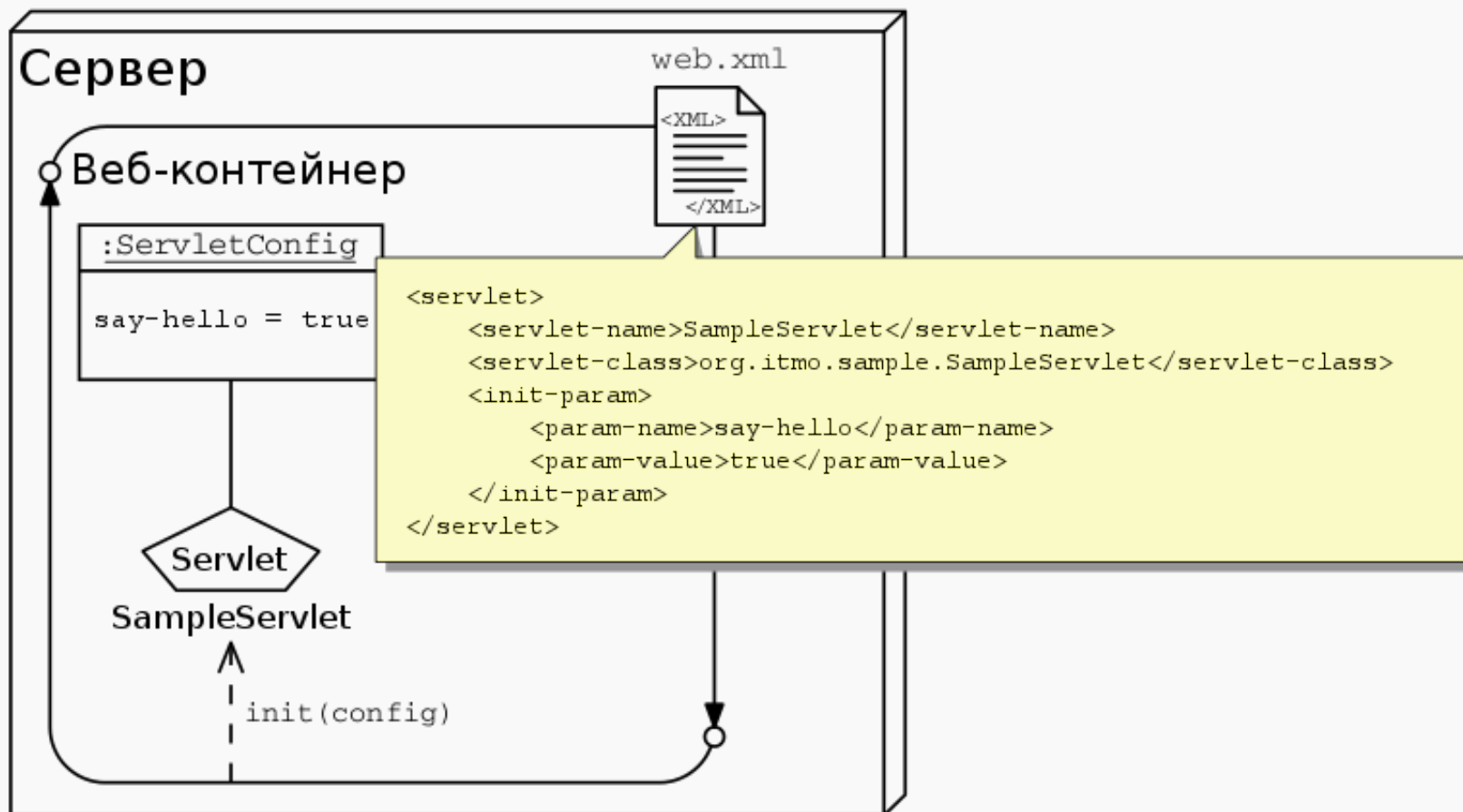
# Жизненный цикл сервлета (продолжение)

## 2. Создание экземпляра сервлета.



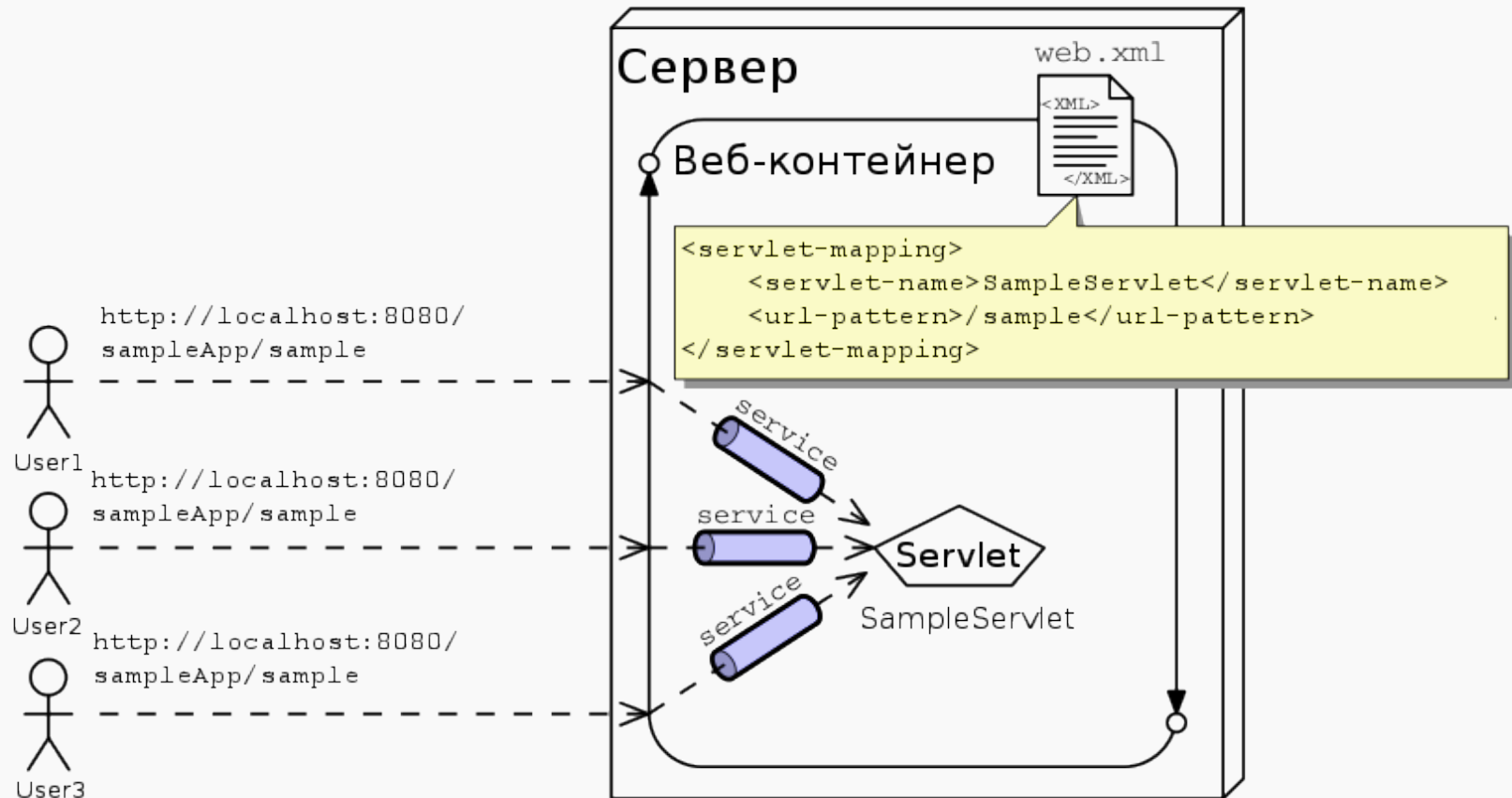
# Жизненный цикл сервлета (продолжение)

## 3. Вызов метода init.



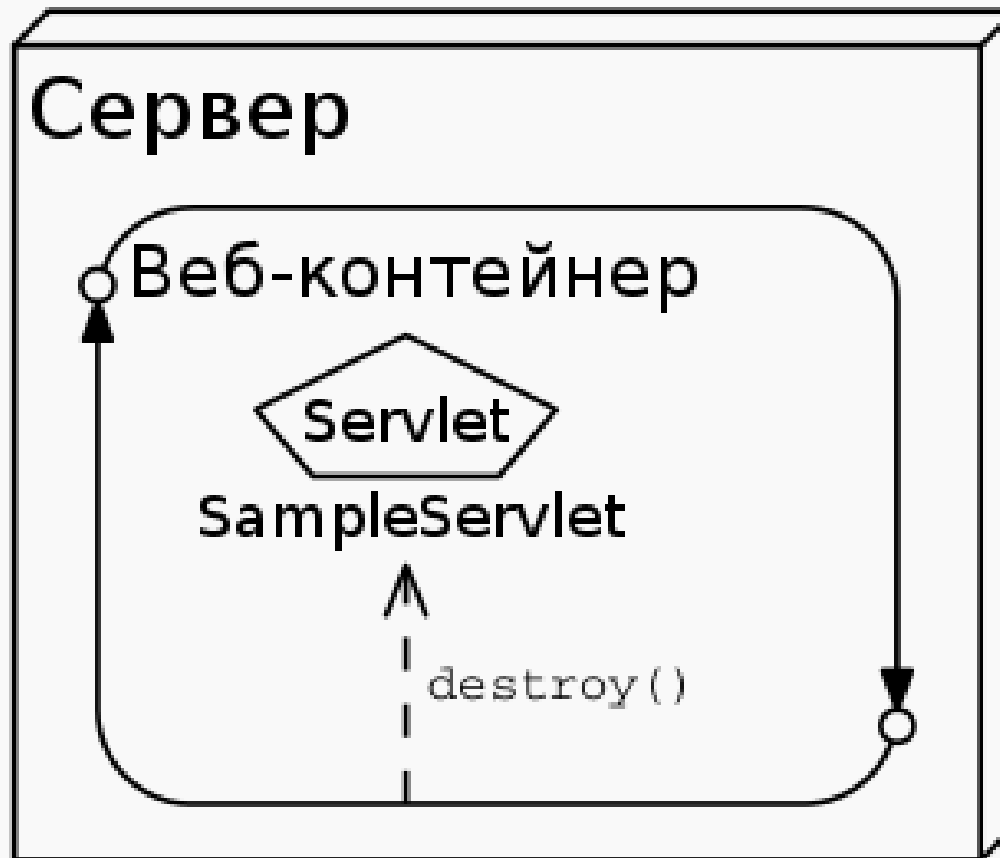
# Жизненный цикл сервлета (продолжение)

## 4. Обработка HTTP-запросов.



# Жизненный цикл сервлета (продолжение)

## 5. Вызов метода destroy.



# Контекст сервлетов

- API, с помощью которого сервлет может взаимодействовать со своим контейнером.
- Доступ к методам осуществляется через интерфейс `javax.servlet.ServletContext`.
- У всех сервлетов внутри приложения общий контекст.
- В контекст можно помещать общую для всех сервлетов информацию (методы `getAttribute` и `setAttribute`).
- Если приложение — распределённое, то на каждом экземпляре JVM контейнером создаётся свой контекст.

# Контекст сервлетов (продолжение)

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

public class DemoServlet extends HttpServlet{
    public void doGet(HttpServletRequest req,HttpServletResponse res)
        throws ServletException,IOException {

        res.setContentType("text/html");
        PrintWriter pw=res.getWriter();

        //creating ServletContext object
        ServletContext context=getServletContext();

        //Getting the value of the initialization parameter
        // and printing it
        String driverName=context.getInitParameter("dname");
        pw.println("driver name is="+driverName);

        pw.close();
    }
}
```



- HTTP — stateless-протокол.
- `javax.servlet.HttpSession` — интерфейс, позволяющий идентифицировать конкретного клиента (браузер) при обработке множества HTTP-запросов от него.
- Экземпляр `HttpSession` создаётся при первом обращении клиента к приложению и сохраняется некоторое (настраиваемое) время после последнего обращения.
- Идентификатор сессии либо помещается в cookie, либо добавляется к URL. Если удалить этот идентификатор, то сервер не сможет идентифицировать клиента и создаст новую сессию.
- В экземпляр `HttpSession` можно помещать общую для этой сессии информацию (методы `getAttribute` и `setAttribute`).
- Сессия «привязана» к конкретному приложению; у разных приложений — разные сессии.
- В распределённом окружении обеспечивается сохранение целостности данных в HTTP-сессии (независимо от количества экземпляров JVM).

# HTTP-сессии (продолжение)

```
public class SimpleSession extends HttpServlet {

    public void doGet(HttpServletRequest request, HttpServletResponse response)
        throws ServletException, java.io.IOException {

        response.setContentType("text/html");
        java.io.PrintWriter out = response.getWriter();
        HttpSession session = request.getSession();

        out.println("<html>");
        out.println("<head>");
        out.println("<title>Simple Session Tracker</title>");
        out.println("</head>");
        out.println("<body>");

        out.println("<h2>Session Info</h2>");
        out.println("session Id: " + session.getId() + "<br><br>");
        out.println("The SESSION TIMEOUT period is "
            + session.getMaxInactiveInterval() + " seconds.<br><br>");
        out.println("Now changing it to 20 minutes.<br><br>");
        session.setMaxInactiveInterval(20 * 60);
        out.println("The SESSION TIMEOUT period is now "
            + session.getMaxInactiveInterval() + " seconds.");

        out.println("</body>");
        out.println("</html>");
    }
}
```

# Диспетчеризация запросов сервлетами

- Сервлеты могут делегировать обработку запросов другим ресурсам (сервлетам, JSP и HTML-страницам).
- Диспетчеризация осуществляется с помощью реализаций интерфейса `javax.servlet.RequestDispatcher`.
- Два способа получения `RequestDispatcher` — через `ServletRequest` (абсолютный или относительный URL) и `ServletContext` (только абсолютный URL).
- Два способа делегирования обработки запроса — `forward` и `include`.

# Диспетчеризация запросов сервлетами (продолжение)

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

public class MyServlet extends HttpServlet{
    public void doGet(HttpServletRequest req,
                      HttpServletResponse res)
        throws ServletException,IOException {

        RequestDispatcher dispatcher =
            request.getRequestDispatcher("index.jsp");
        dispatcher.forward( request, response );
    }
}
```

# Фильтры запросов

- Фильтры позволяют осуществлять пред- и постобработку запросов до и после передачи их ресурсу (сервлету, JSP или HTML-странице).
- Пример предобработки — допуск к странице только авторизованных пользователей.
- Пример постобработки — запись в лог времени обработки запроса.
- Реализуют интерфейс `javax.servlet.Filter`.
- Ключевой метод — `doFilter`.
- Метод `doFilter` класса `FilterChain` передаёт управление следующему фильтру или целевому ресурсу; таким образом, возможна реализация последовательностей фильтров, обрабатывающих один и тот же запрос.

# Фильтры запросов (продолжение)

```
import javax.servlet.*;

public class MyFilter implements Filter{

    public void init(FilterConfig arg0) throws ServletException {}

    public void doFilter(ServletRequest req, ServletResponse resp,
        FilterChain chain) throws IOException, ServletException {

        PrintWriter out=resp.getWriter();
        out.print("filter is invoked before");

        chain.doFilter(req, resp);//sends request to next resource

        out.print("filter is invoked after");
    }

    public void destroy() {}
}
```

# Конфигурация фильтров

```
<web-app>
```

```
  <servlet>
```

```
    <servlet-name>s1</servlet-name>
```

```
    <servlet-class>HelloServlet</servlet-class>
```

```
  </servlet>
```

```
  <servlet-mapping>
```

```
    <servlet-name>s1</servlet-name>
```

```
    <url-pattern>/servlet1</url-pattern>
```

```
  </servlet-mapping>
```

```
  <filter>
```

```
    <filter-name>f1</filter-name>
```

```
    <filter-class>MyFilter</filter-class>
```

```
  </filter>
```

```
  <filter-mapping>
```

```
    <filter-name>f1</filter-name>
```

```
    <url-pattern>/servlet1</url-pattern>
```

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

```
</web-app>
```

# 11. JavaServer Pages



- Страницы JSP — это текстовые файлы, содержащие статический HTML и JSP-элементы.
- JSP-элементы позволяют формировать динамическое содержимое.
- При загрузке в веб-контейнер страницы JSP транслируются компилятором (jasper) в сервлеты.
- Позволяют отделить бизнес-логику от уровня представления (если их комбинировать с сервлетами).

# Преимущества и недостатки JSP

- Преимущества:
  - Высокая производительность — транслируются в сервлеты.
  - Не зависят от используемой платформы — код пишется на Java.
  - Позволяют использовать Java API.
  - Простые для понимания — структура похожа на обычный HTML.
- Недостатки:
  - Трудно отлаживать, если приложение целиком основано на JSP.
  - Возможны конфликты при параллельной обработке нескольких запросов.

# Сервлеты и JSP

```
public class HelloServlet extends HttpServlet {  
    private static final String DEFAULT_NAME = "World";  
    public void doGet(HttpServletRequest request,  
        HttpServletResponse response)  
        throws IOException {  
        generateResponse(request, response);  
    }  
  
    public void doPost(HttpServletRequest request,  
        HttpServletResponse response)  
        throws IOException {  
        generateResponse(request, response);  
    }  
  
    public void generateResponse(HttpServletRequest  
request,  
        HttpServletResponse response) throws IOException {  
        String name = request.getParameter("name");
```

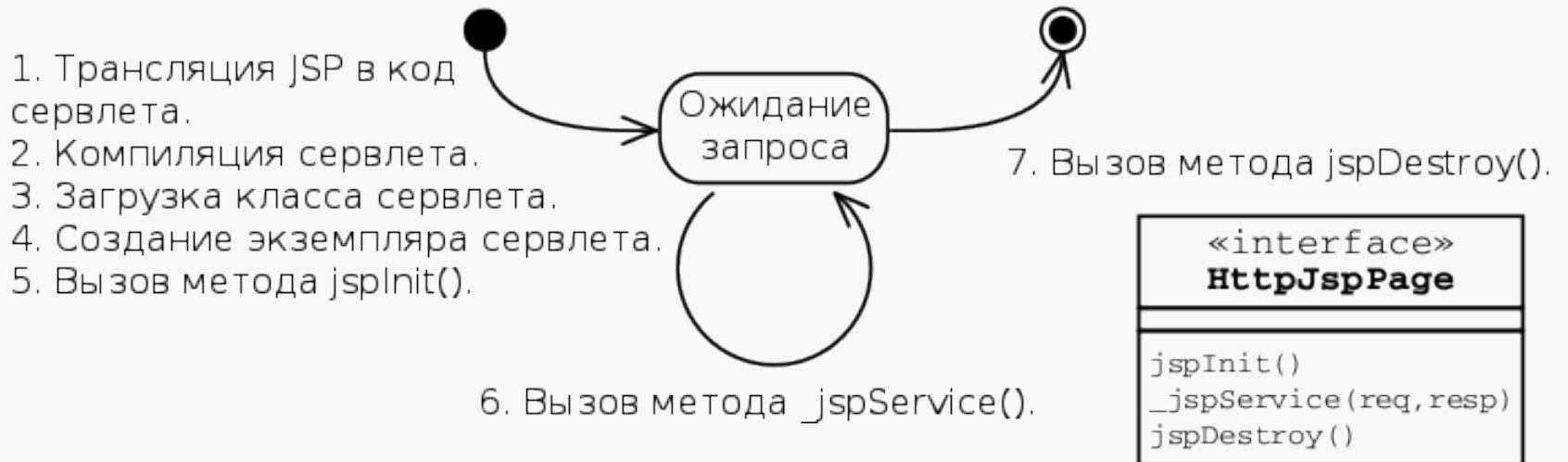
# Сервлеты и JSP (продолжение)

```
if ( (name == null) || (name.length() == 0) ) {  
    name = DEFAULT_NAME;  
}  
response.setContentType("text/html");  
PrintWriter out = response.getWriter();  
out.println("<HTML>");  
out.println("<HEAD>");  
out.println("<TITLE>Hello Servlet</TITLE>");  
out.println("</HEAD>");  
out.println("<BODY BGCOLOR='white'>");  
out.println("<B>Hello, " + name + "</B>");  
out.println("</BODY>");  
out.println("</HTML>");  
out.close();  
}  
}
```

# Сервлеты и JSP (продолжение)

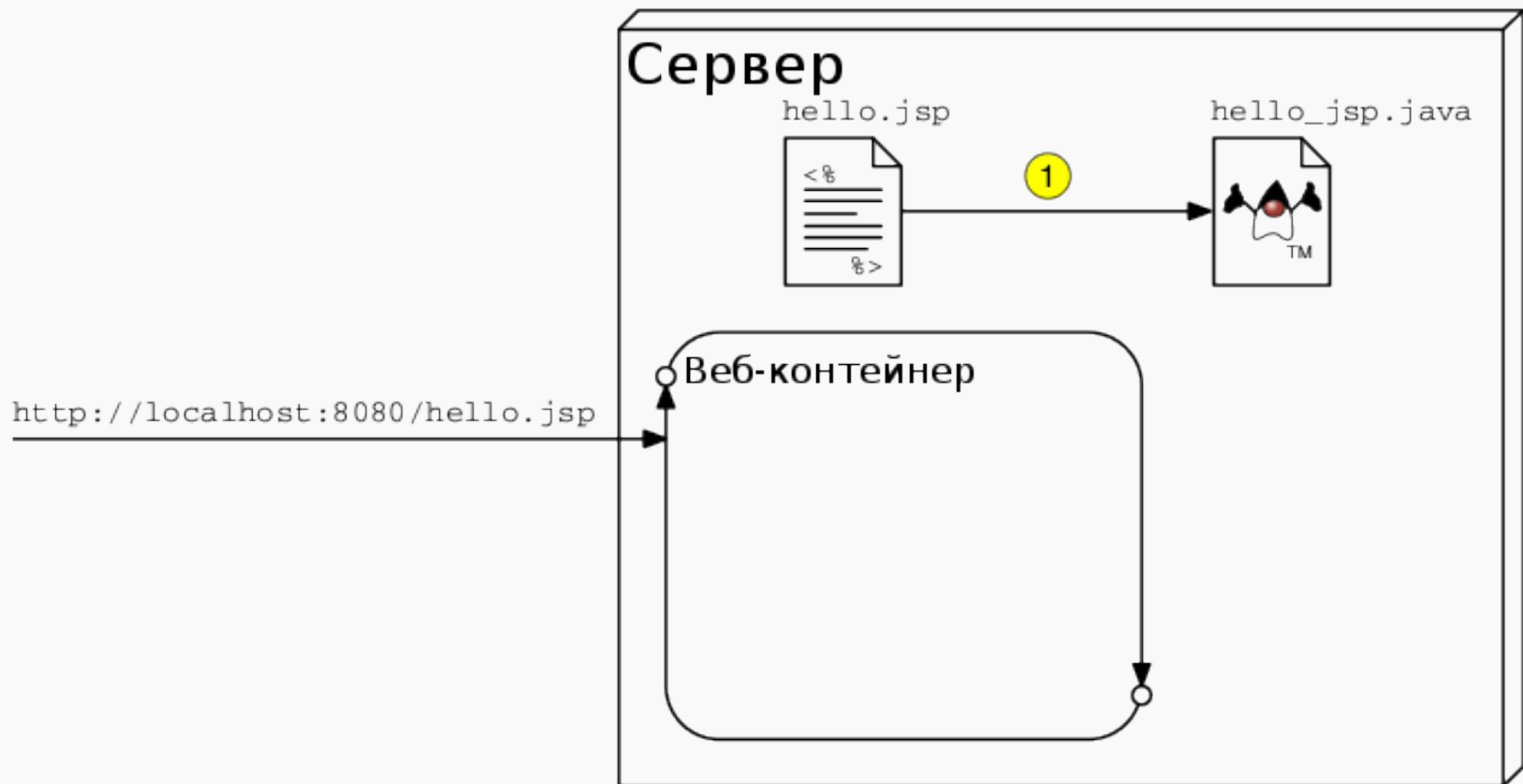
```
<%! private static final String DEFAULT_NAME = "World";
%>
<html>
<head>
<title>Hello JavaServer Page</title>
</head>
<!-- Determine the specified name (or use default) --%>
<%
    String name = request.getParameter("name");
    if ( (name == null) || (name.length() == 0) ) {
        name = DEFAULT_NAME;
    }
%>
<body bgcolor='white'>
<b>Hello, <%= name %></b>
</body>
</html>
```

# Жизненный цикл JSP



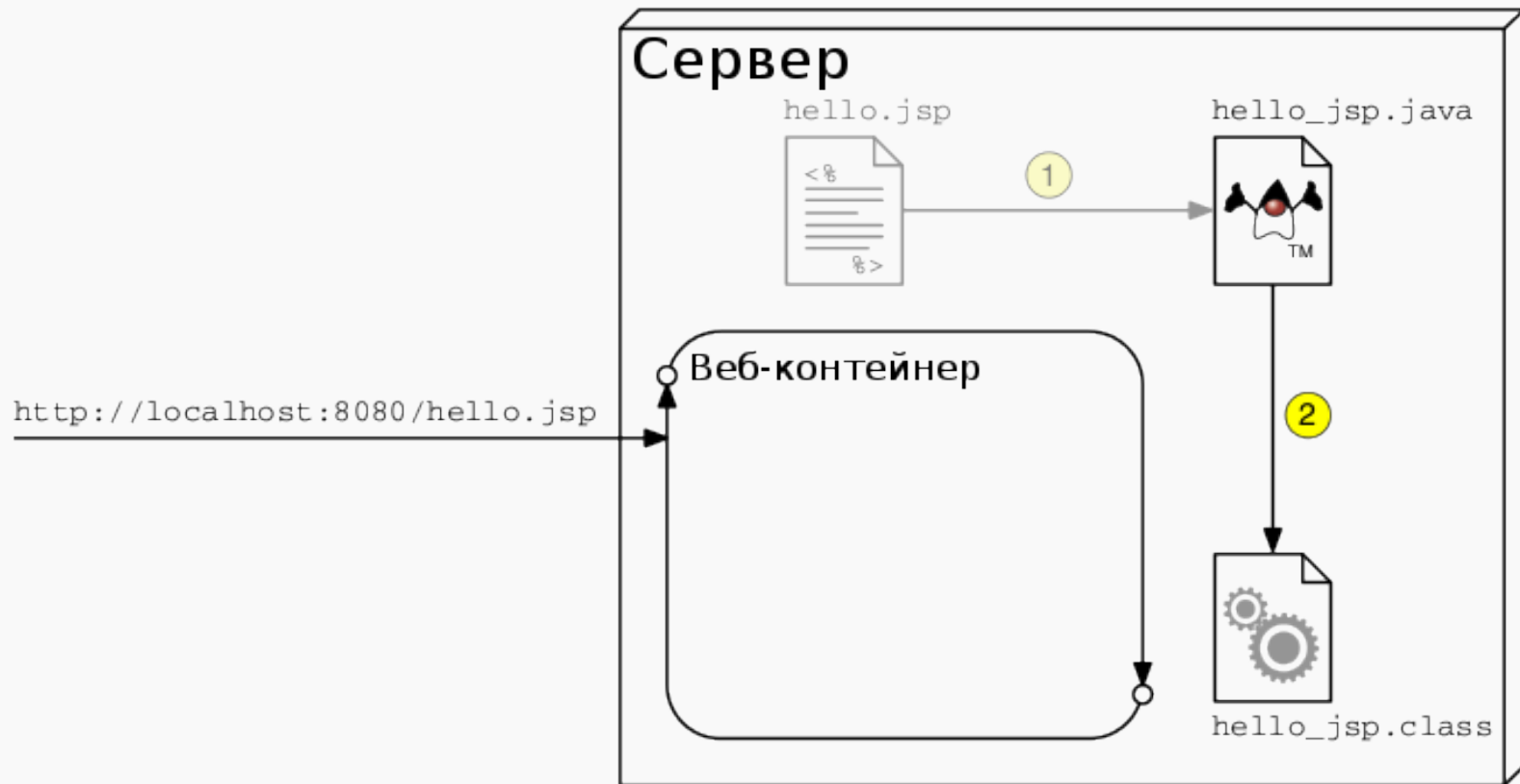
# Жизненный цикл JSP (продолжение)

## 1. Трансляция.



# Жизненный цикл JSP (продолжение)

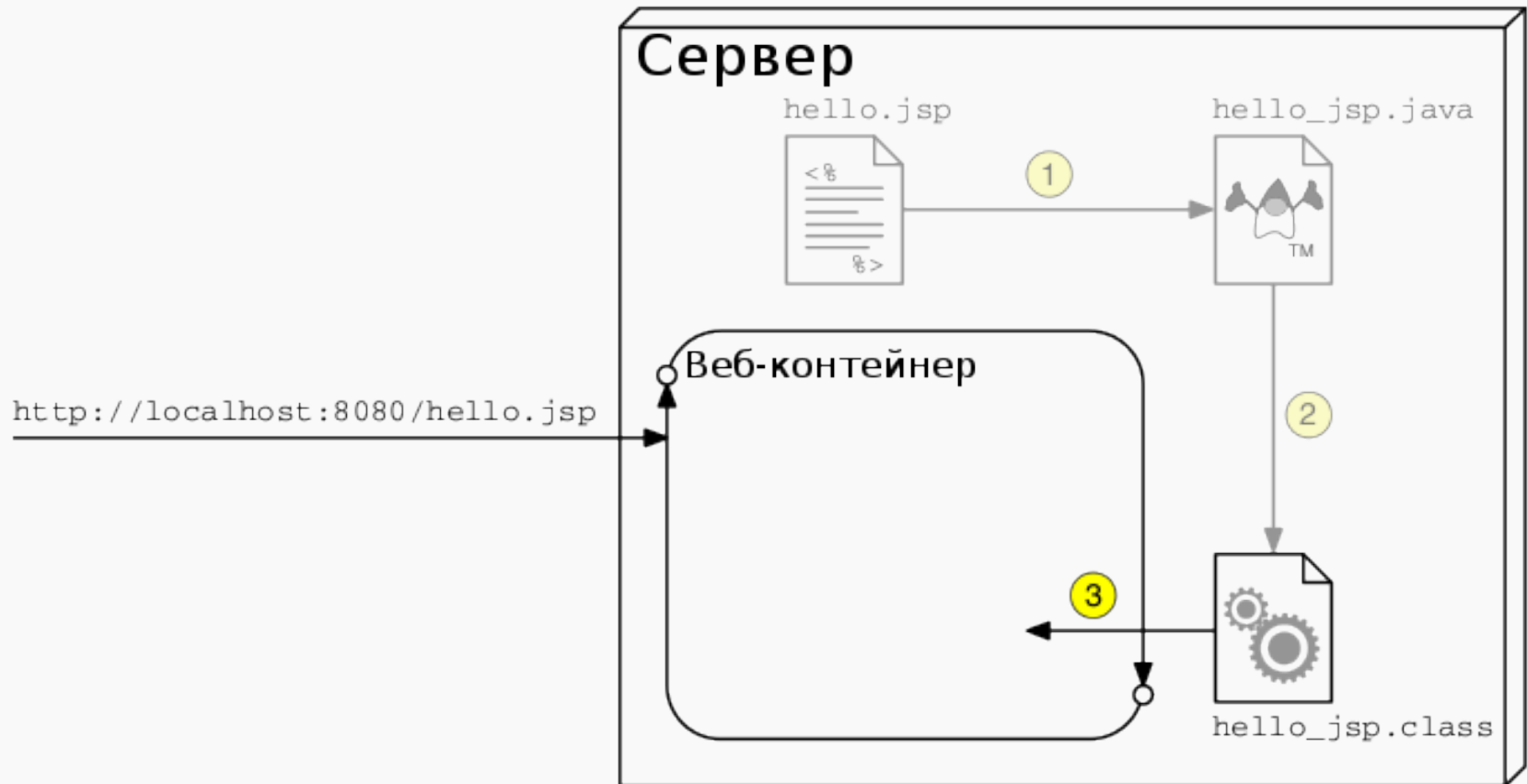
## 2. Компиляция сервлета.





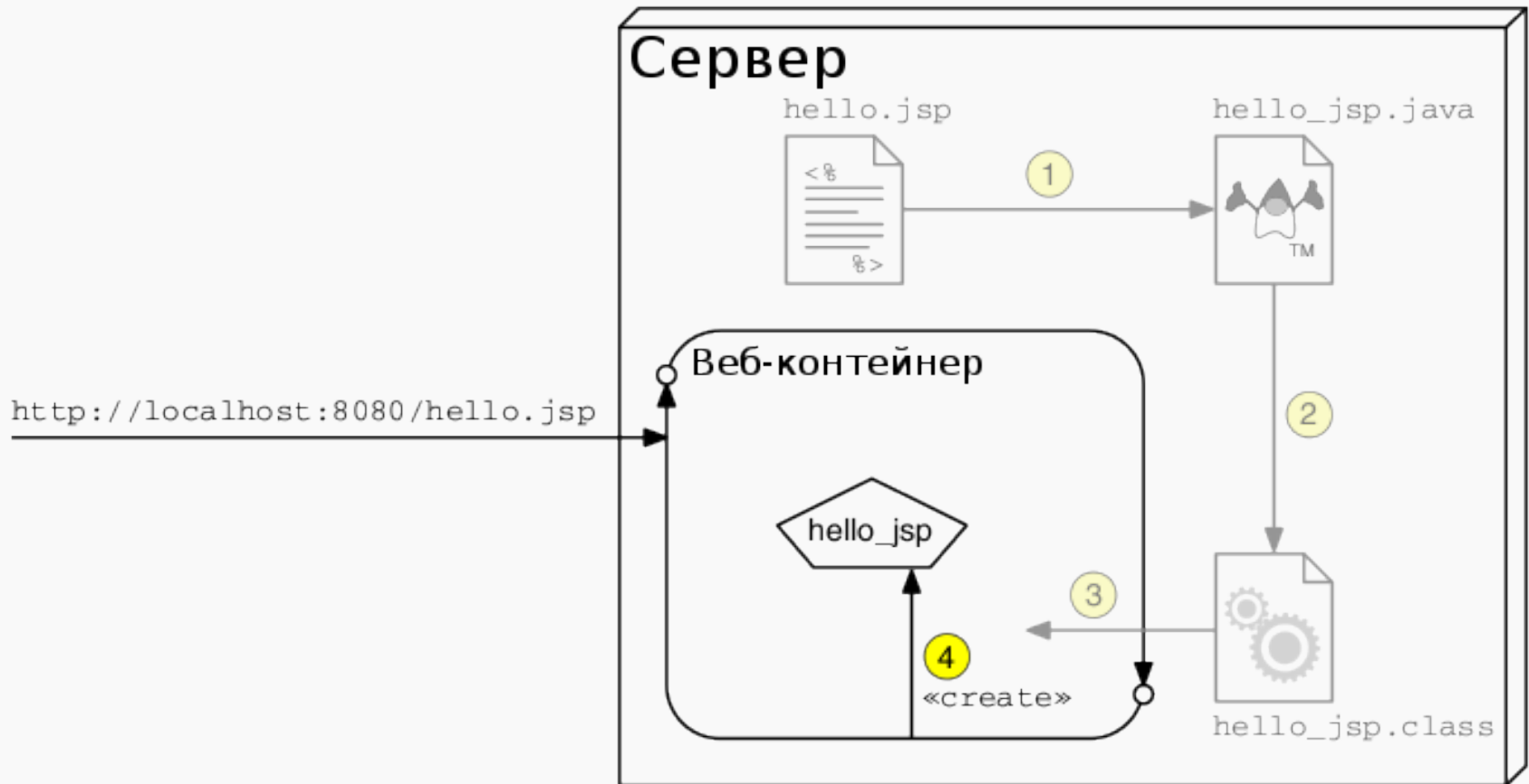
# Жизненный цикл JSP (продолжение)

## 3. Загрузка сервлета веб-контейнером.



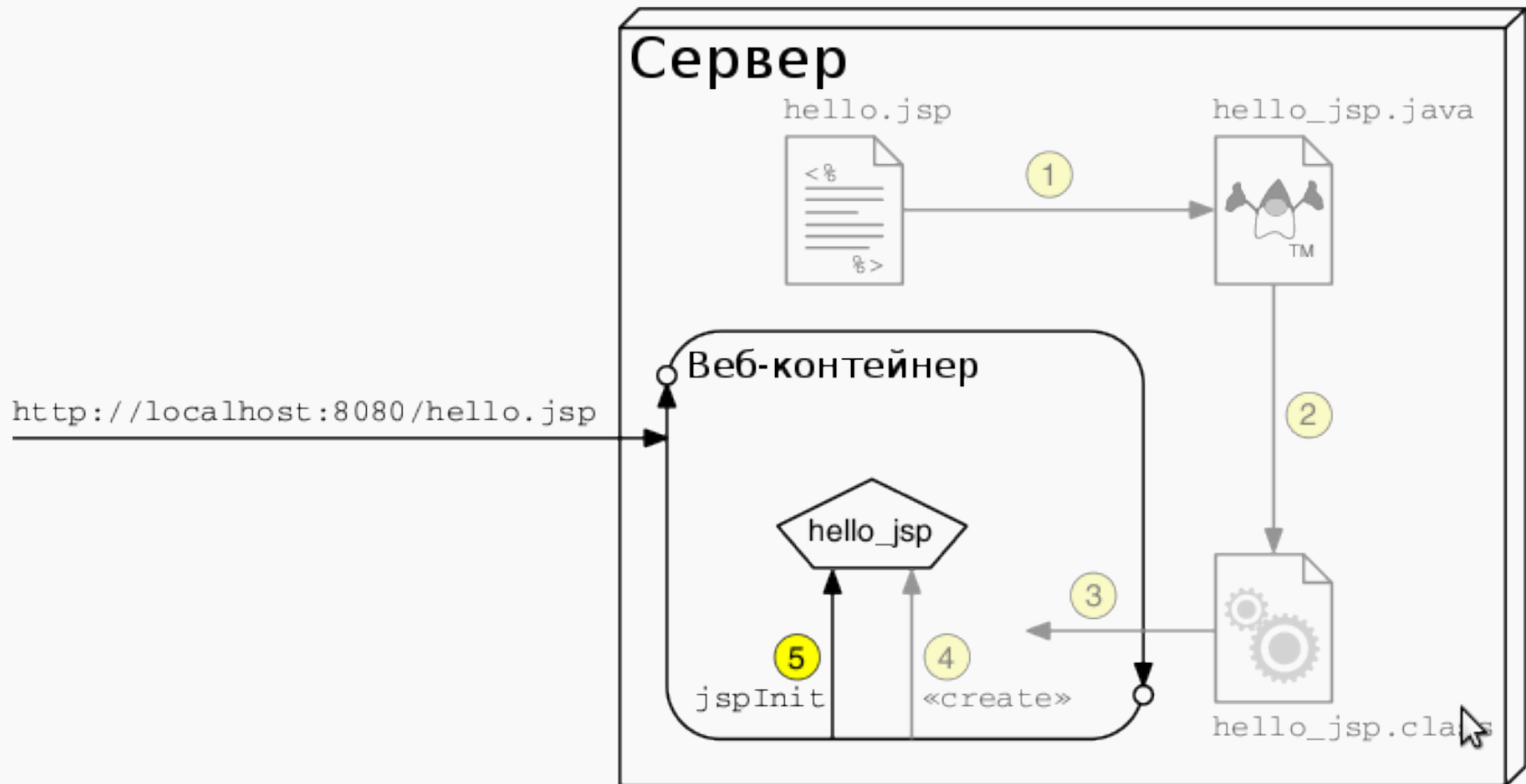
# Жизненный цикл JSP (продолжение)

## 4. Создание веб-контейнером экземпляра сервлета.



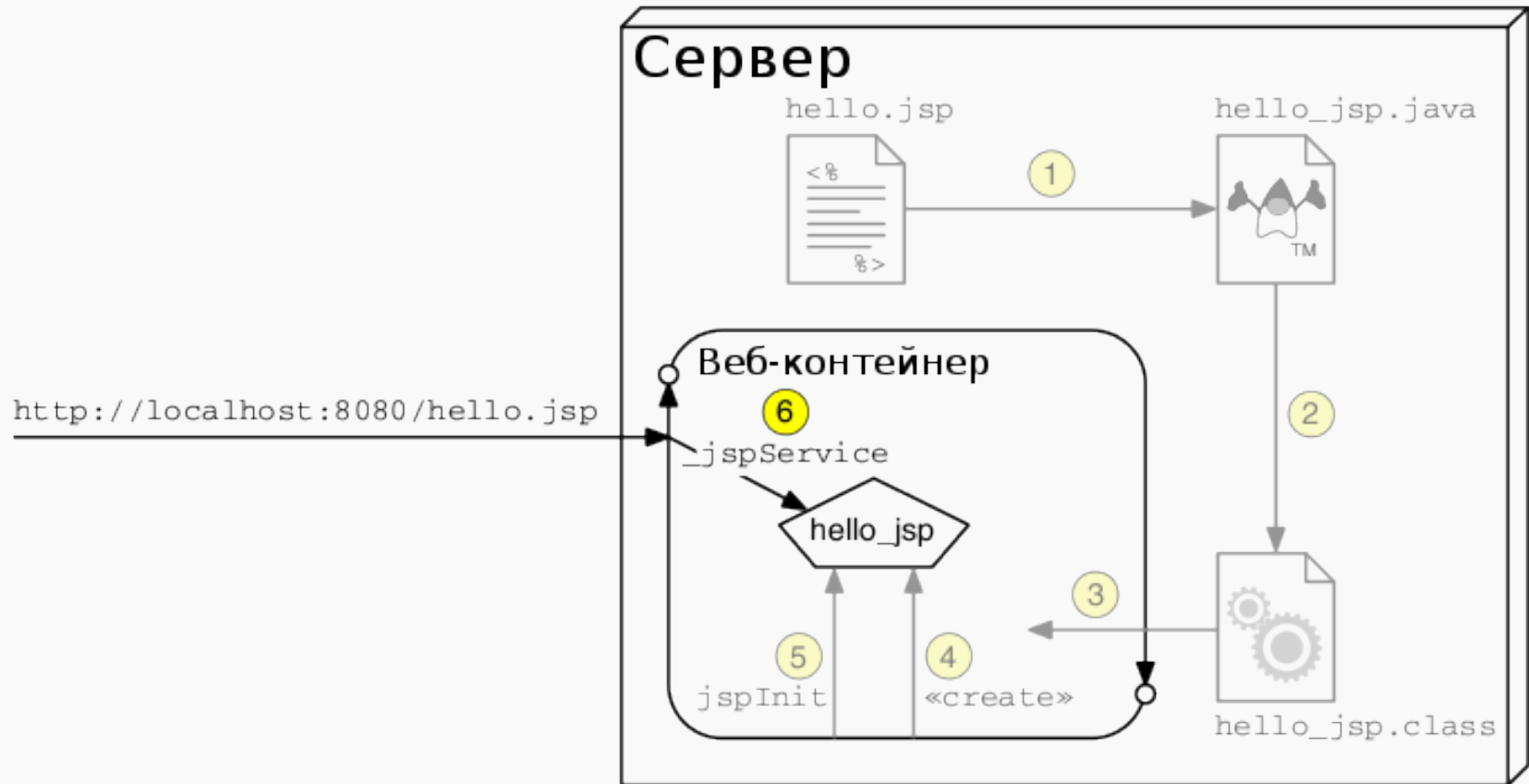
# Жизненный цикл JSP (продолжение)

## 5. Инициализация сервлета.



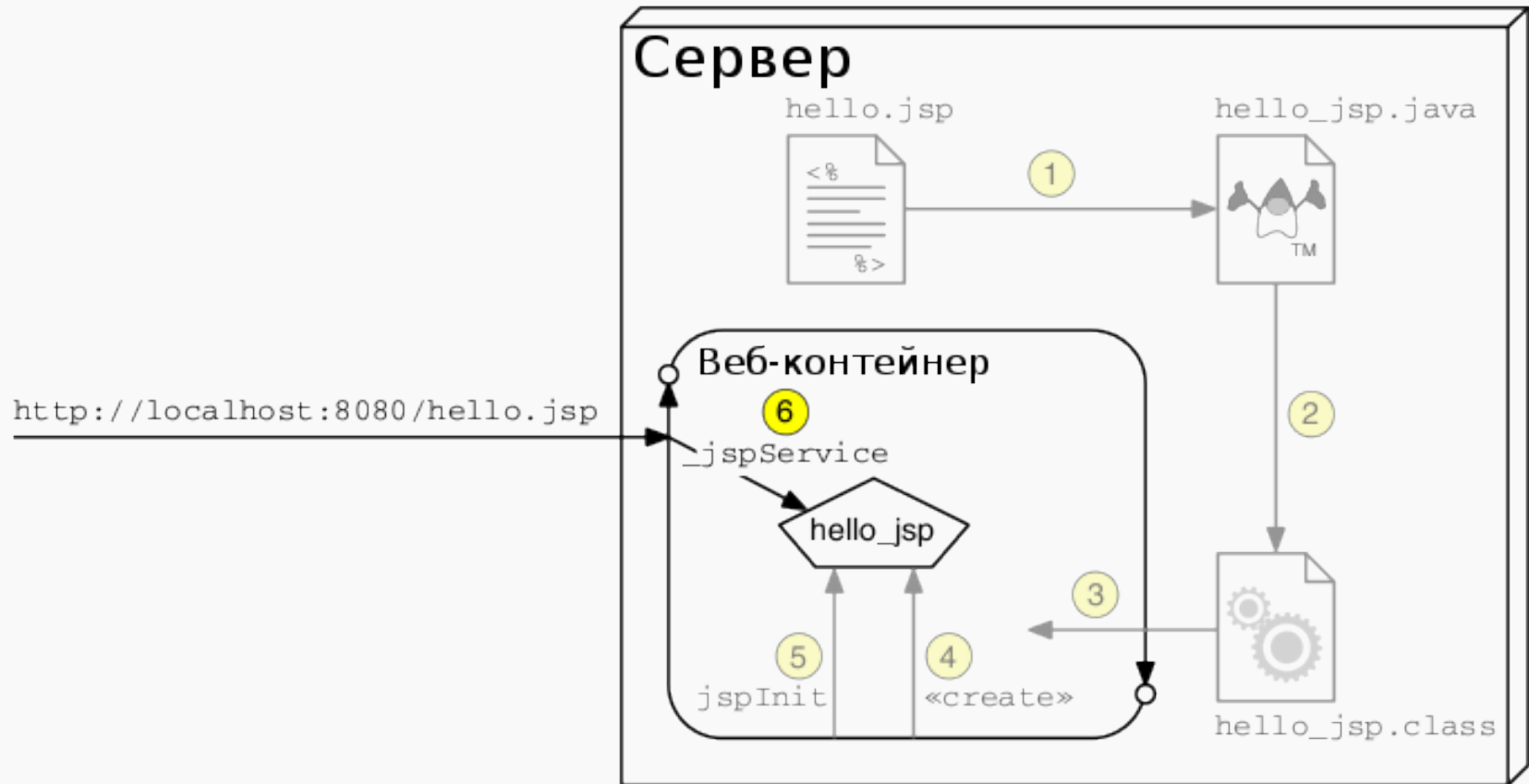
# Жизненный цикл JSP (продолжение)

## 6. Обработка запросов.



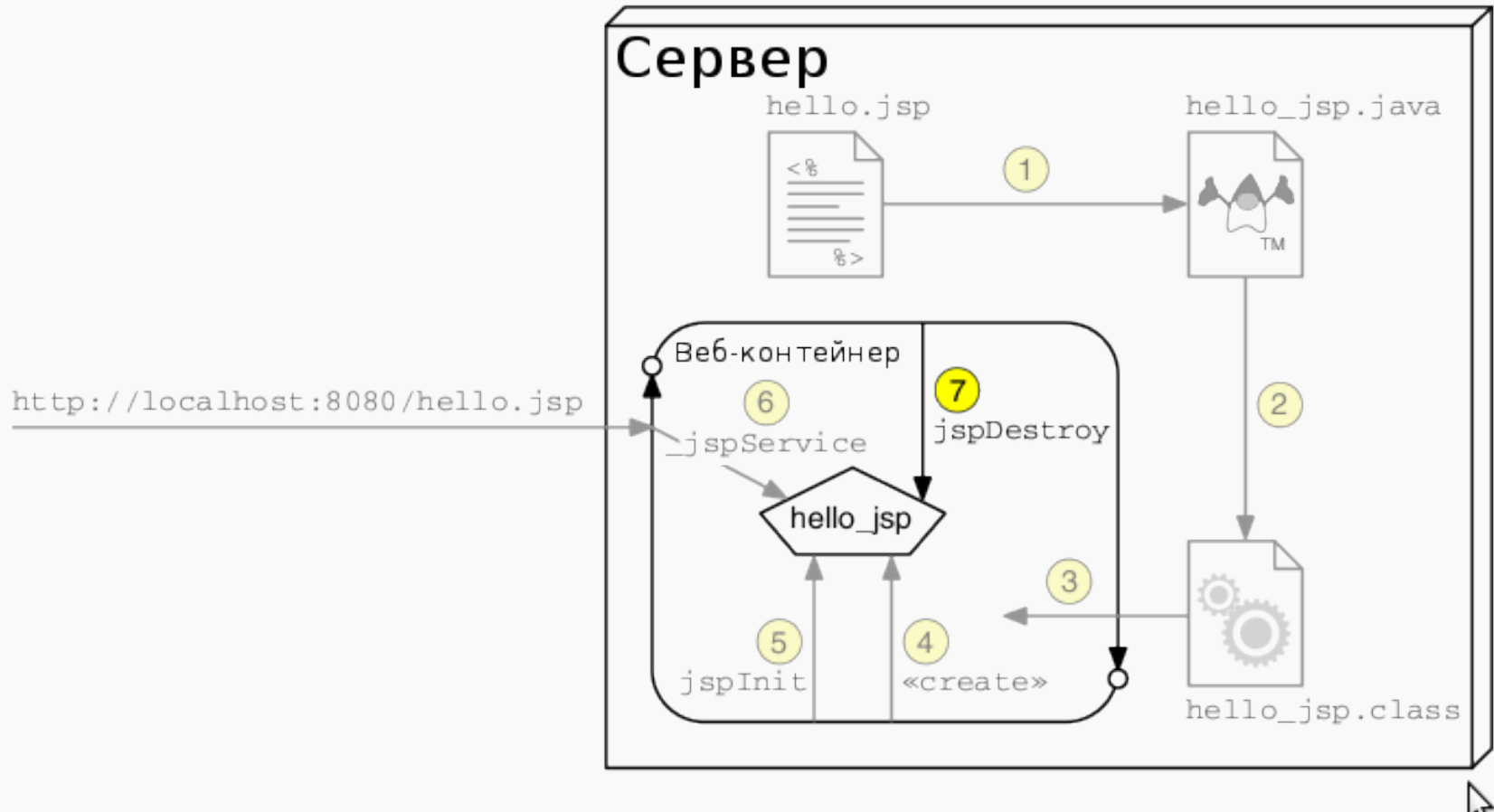
# Жизненный цикл JSP (продолжение)

## 6. Обработка запросов.



# Жизненный цикл JSP (продолжение)

## 7. Вызов метода jspDestroy.



- 2 варианта синтаксиса — на базе HTML и XML.
- Обозначаются тегами `<% %>` (HTML-вариант):  
`<html>`  
`<%- - scripting element - -%>`  
`</html>`
- Существует 5 типов JSP-элементов:
  - Комментарий — `<%- - Comment - -%>;`
  - Директива — `<%@ directive %>;`
  - Объявление — `<%! decl %>;`
  - Скриптлет — `<% code %>;`
  - Выражение — `<%= expr %>.`

Поддерживаются 3 типа комментариев:

- HTML-комментарии:

```
<!-- This is an HTML comment.  
      It will show up in the response. -->
```

- JSP-комментарии:

```
<%-- This is a JSP comment.  
      It will only be seen in the JSP code.  
      It will not show up in either the servlet code  
      or the response.  
--%>
```

- Java-комментарии:

```
<%  
    /* This is a Java comment.  
       It will show up in the servlet code.  
       It will not show up in the response. */  
%>
```



Управляют процессом трансляции страницы в сервлет.

- Синтаксис:

```
<%@ DirectiveName [attr="value"]* %>
```

- Примеры:

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

```
<%@ include file="incl/copyright.html" %>
```

Позволяют объявлять поля и методы:

- Синтаксис:

```
<%! JavaClassDeclaration %>
```

- Примеры:

```
<%!  
public static final String DEFAULT_NAME = "World";  
%>
```

```
<%!  
public String getName(HttpServletRequest request) {  
    return request.getParameter("name");  
}  
%>
```

```
<%! int counter = 0; %>
```

Позволяют задать Java-код, который будет выполняться при обработке запросов (при вызове метода `_jspService`).

- Синтаксис:

```
<% JavaCode %>
```

- Примеры:

```
<% int i = 0; %>
```

```
<% if ( i > 10 ) %>
```

```
    I am a big number
```

```
<% } else { %>
```

```
    I am a small number
```

```
<% } %>
```

Позволяют вывести результат вычисления выражения.

- Синтаксис:

`<%= JavaExpression %>`

- Примеры:

`<B>Ten is <%= (2 * 5) %></B>`

`Thank you, <I><%= name %></I>, for registering  
for the soccer league.`

`The current day and time is: <%= new  
java.util.Date() %>`

# Предопределённые переменные

В процессе трансляции контейнер добавляет в метод `_jspService` ряд объектов, которые можно использовать в скриптелях и выражениях:

Имя переменной	Класс
<code>application</code>	<code>javax.servlet.ServletContext</code>
<code>config</code>	<code>javax.servlet.ServletConfig</code>
<code>exception</code>	<code>java.lang.Throwable</code>
<code>out</code>	<code>javax.servlet.jsp.JspWriter</code>
<code>page</code>	<code>java.lang.Object</code>
<code>PageContext</code>	<code>javax.servlet.jsp.PageContext</code>
<code>request</code>	<code>javax.servlet.ServletRequest</code>
<code>response</code>	<code>javax.servlet.ServletResponse</code>
<code>session</code>	<code>javax.servlet.http.HttpSession</code>

- Exception — используется только на страницах-перенаправлениях с информацией об ошибках (Error Pages).
- Page — API для доступа к экземпляру класса сервлета, в который транслируется JSP.
- PageContext — контекст JSP-страницы.

# Директива *Page*

- Позволяет задавать параметры, используемые контейнером при управлении жизненным циклом страницы.
- Обычно расположена в начале страницы.
- На одной странице может быть задано несколько директив page с разными указаниями контейнеру.
- Синтаксис:  
`<%@ page attribute="value" %>`

# Атрибуты директивы *Page*

Атрибут	Для чего нужен
buffer	Задаёт параметры буферизации и размер буфера для потока вывода сервлета.
autoFlush	Указывает, автоматически ли выгружается содержимое буфера при его переполнении.
contentType	Позволяет задать Content Type и кодировку страницы.
errorPage	Позволяет задать страницу, на которую будет осуществлено перенаправление при возникновении Runtime Exception.
isErrorPage	Указывает, является ли текущая страница Error Page.
extends	Позволяет задать имя родительского класса, от которого будет наследоваться сервлет.



# Атрибуты директивы *Page* (продолжение)

Атрибут	Для чего нужен
<code>import</code>	Импорт классов или пакетов.
<code>info</code>	Задаёт строку, которую будет возвращать метод <code>getServletInfo()</code> .
<code>isThreadSafe</code>	Если <code>isThreadSafe == false</code> , то контейнер блокирует параллельную обработку нескольких запросов страниц.
<code>language</code>	Позволяет задать язык программирования, на котором пишутся скриптовые элементы на странице (по умолчанию — Java).
<code>session</code>	Указывает контейнеру, создавать ли ему предопределённую переменную <code>session</code> .
<code>isELIgnored</code>	Указывает, вычисляются EL-выражения контейнером, или нет.
<code>isScriptingEnabled</code>	Указывает, обрабатываются ли скриптовые элементы.

# JSP Actions

- XML-элементы, позволяющие управлять поведением сервлета.
- Синтаксис:  
`<jsp:action_name attribute="value" />`

JSP Action	Для чего нужен
<code>jsp:include</code>	Включает в страницу внешний файл <i>во время обработки запроса</i> .
<code>jsp:useBean</code>	Добавляет на страницу экземпляр Java Bean с заданным контекстом.
<code>jsp:getProperty</code> <code>jsp:setProperty</code>	Получение и установка свойств Java Bean.
<code>jsp:forward</code>	Перенаправление на другую страницу.

- Задаётся в дескрипторе развёртывания (web.xml).
- Находится внутри элемента `jsp-config`.
- Пример:

```
<jsp-config>  
  <property name="initial-capacity"  
            value="1024" >  
</jsp-config>
```

## 12. Шаблоны проектирования в веб- приложениях

- **Шаблон проектирования** или **паттерн** — повторимая архитектурная конструкция, представляющая собой решение проблемы проектирования в рамках некоторого часто возникающего контекста (© Wikipedia).
- Описывает подход к решению типовой задачи.
- Одну и ту же задачу часто можно решить с использованием разных шаблонов.
- Существует много литературы с описанием различных шаблонов проектирования.

# Зачем нужны паттерны

- Позволяют избежать «типовых» ошибок при разработке типовых решений.
- Позволяют кратко описать подход к решению задачи — программистам, знающим шаблоны, проще обмениваться информацией.
- Легче поддерживать код — его поведение более предсказуемо.

# GoF-паттерны

- Описаны в книге 1994 г. «*Design Patterns: Elements of Reusable Object-Oriented Software*» («Приёмы объектно-ориентированного проектирования. Паттерны проектирования»).
- Авторы — *Эрих Гамма* (Erich Gamma), *Ричард Хелм* (Richard Helm), *Ральф Джонсон* (Ralph Johnson), *Джон Влассидс* (John Vlissides) — Gang of Four (GoF, «Банда Четырёх»).
- В книге описаны 23 классических шаблона проектирования.

# Порождающие GoF-паттерны

- Abstract Factory — Абстрактная фабрика.
- Builder — Строитель.
- Factory Method — Фабричный метод.
- Prototype — Прототип.
- Singleton — Одиночка.



# Пример порождающего GoF-паттерна

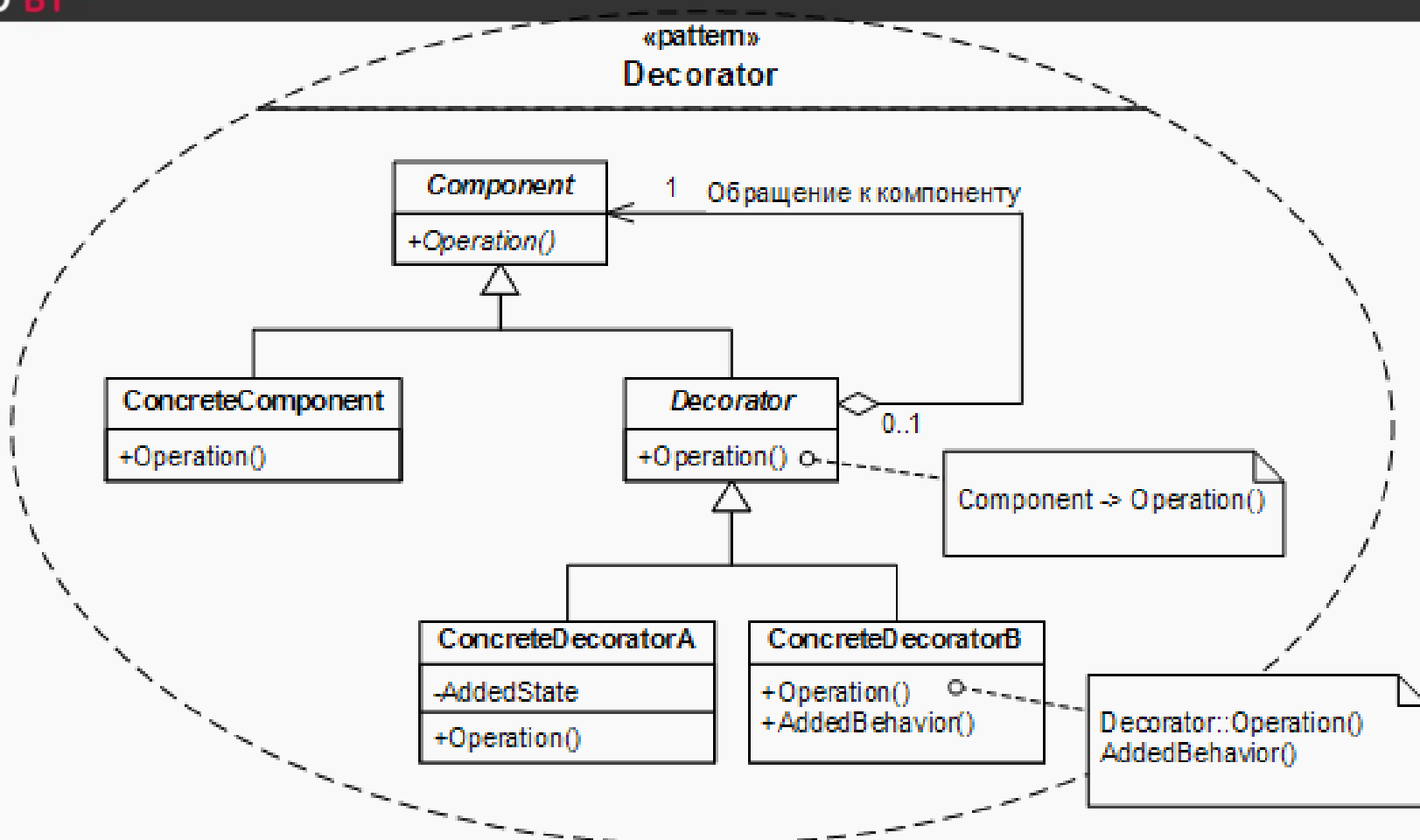
Singleton
+Instance():Singleton
-Singleton():void
-instance:Singleton

## Singleton (одиночка):

- Гарантирует, что у класса есть *только один экземпляр*, и предоставляет к нему глобальную точку доступа.
- Можно пользоваться *экземпляром* класса (в отличие от статических методов).

- Adapter — Адаптер.
- Bridge — Мост.
- Composite — Компоновщик.
- Decorator — Декоратор.
- Facade — Фасад.
- Flyweight — Приспособленец.
- Proxy — Заместитель.

# Пример структурного GoF-паттерна

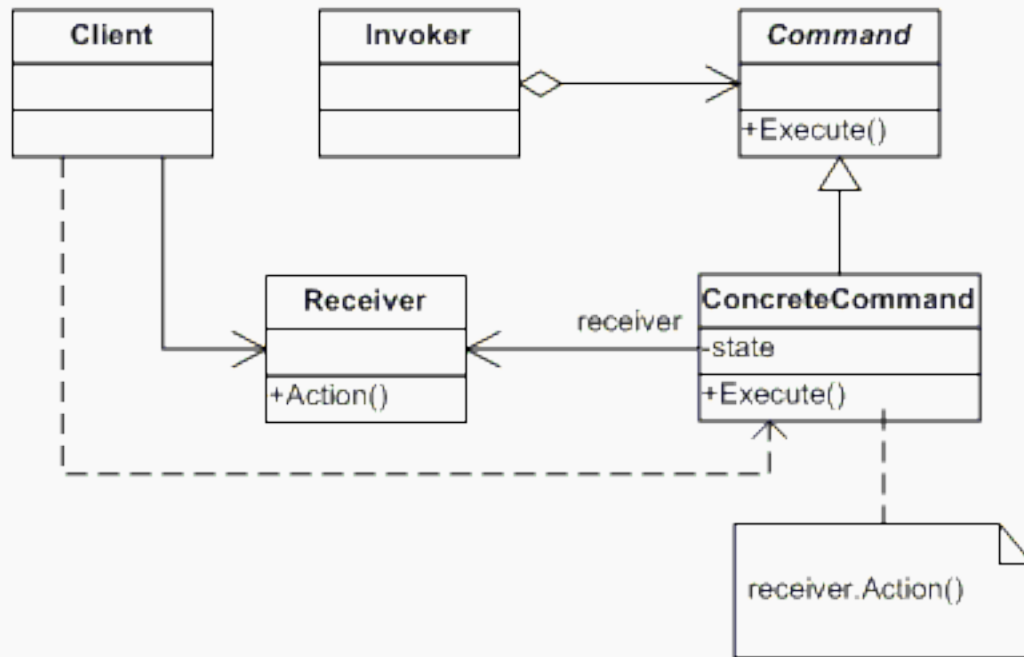


**Decorator (декоратор)** — позволяет динамически подключать дополнительное поведение к объекту без использования наследования.

# Поведенческие GoF-паттерны

- Chain of responsibility — Цепочка обязанностей.
- Command — Команда.
- Interpreter — Интерпретатор.
- Iterator — Итератор.
- Mediator — Посредник.
- Memento — Хранитель.
- Observer — Наблюдатель.
- State — Состояние.
- Strategy — Стратегия.
- Template — Шаблонный метод.
- Visitor — Посетитель.

# Пример поведенческого GoF-паттерна



**Command (команда)** — команда передаётся с помощью специального объекта, который заключает в себе само действие (т. е. логику) и его параметры.

# Архитектурные шаблоны

- Более высокий уровень по сравнению с шаблонами проектирования.
- Описывают архитектуру всей системы или приложения.
- Обычно имеют дело не с отдельными классами, а с целыми компонентами или модулями.
- Компоненты и модули могут быть построены с использованием различных шаблонов проектирования.

# Архитектура веб-приложений

3 уровня архитектуры:



Клиент



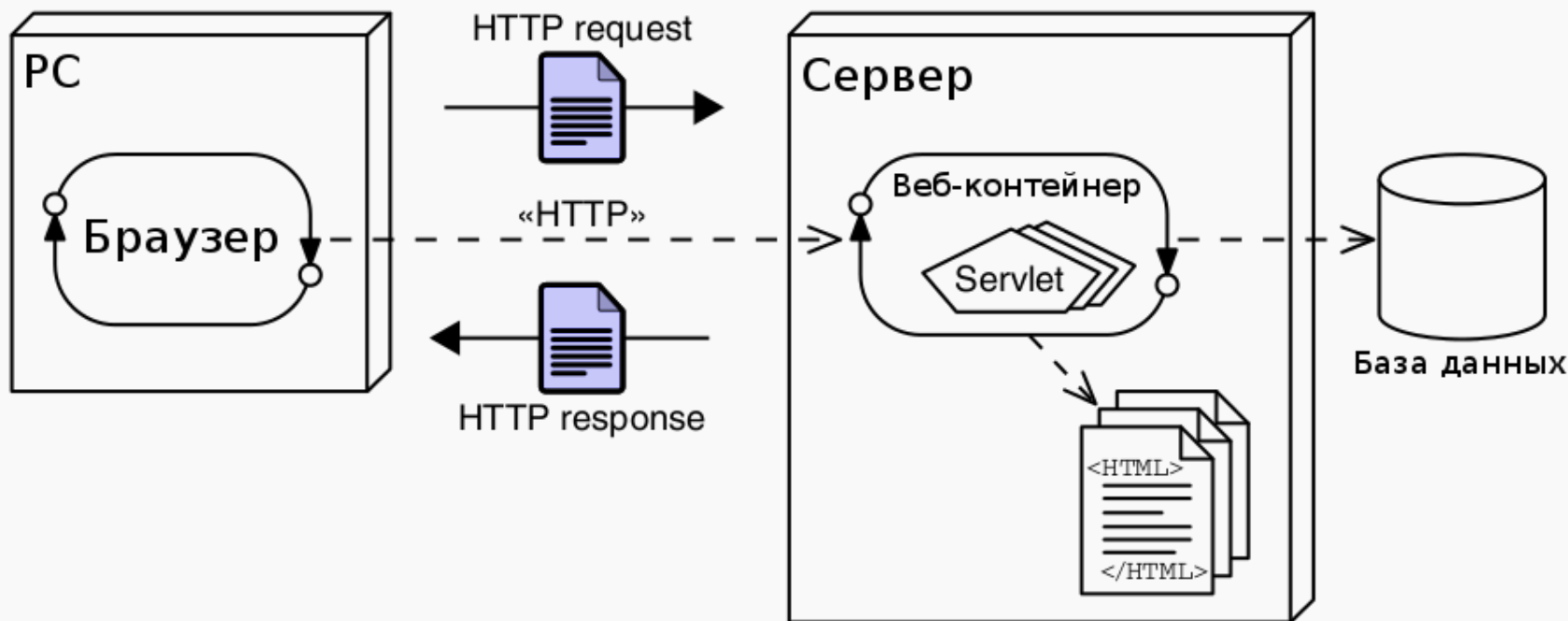
Бизнес-логика



Данные

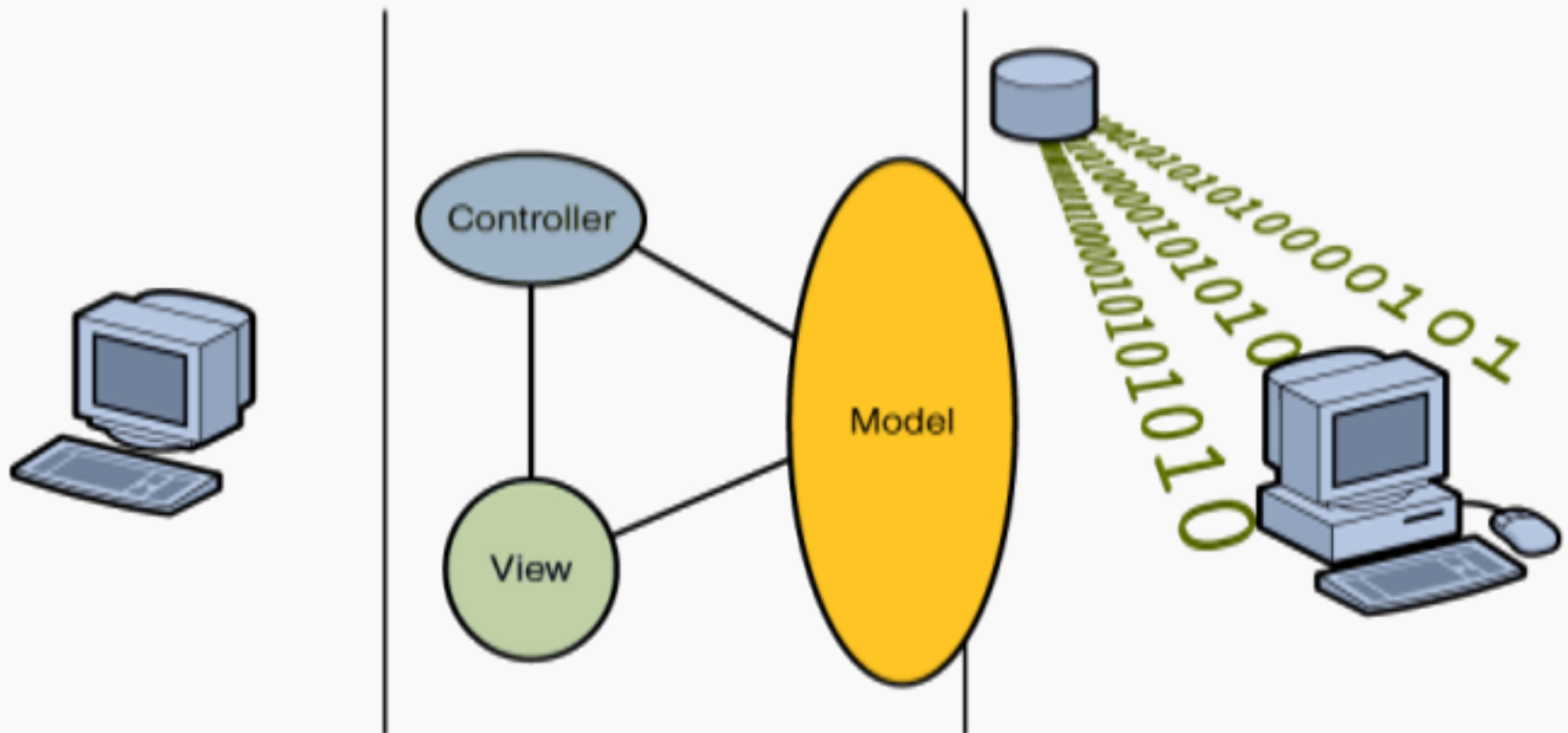
# Архитектура Model 1

- Предназначена для проектирования приложений небольшого масштаба и сложности.
- За обработку данных и представления отвечает *один и тот же* компонент (сервлет или JSP).



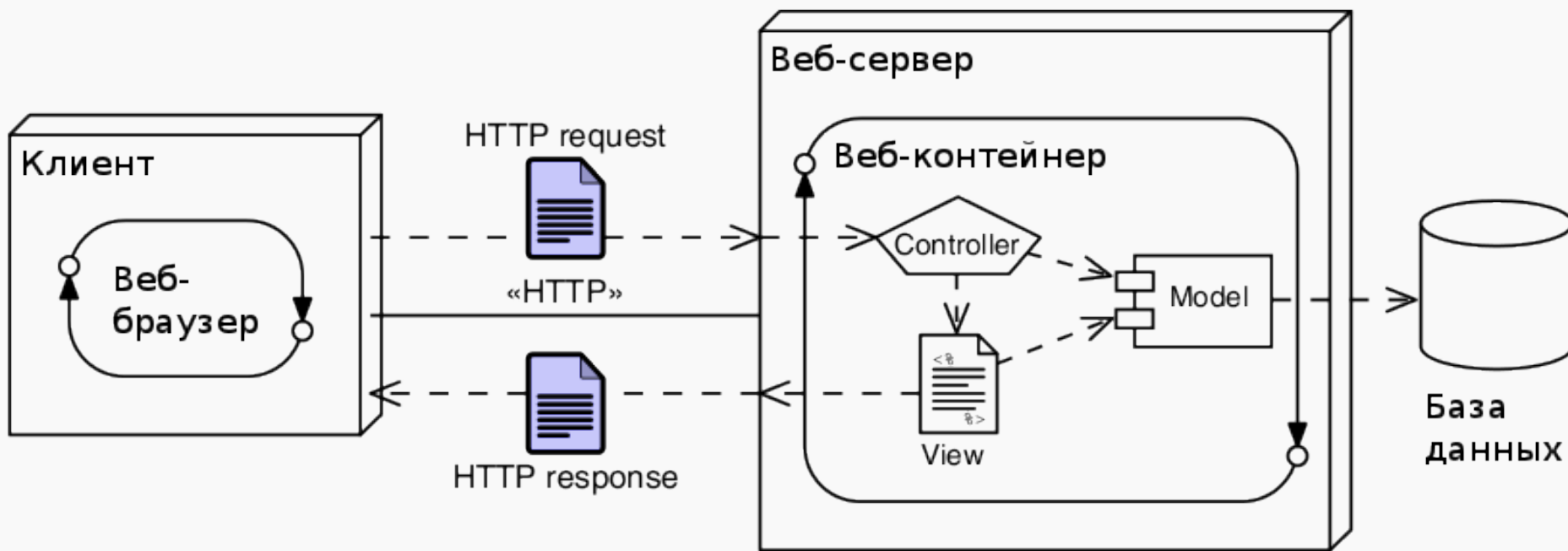


# Шаблон MVC



# Архитектура Model 2

- Предназначена для проектирования достаточно сложных веб-приложений.
- За обработку и представление данных отвечают *разные* компоненты (сервлеты и JSP).



- Apache Struts.
- Apache Velocity.
- JavaServer Faces (в составе Java EE).

# 13. JavaServer Faces

- JSF — фреймворк для разработки веб-приложений.
- Входит в состав платформы Java EE.
- Основан на использовании *компонентов*.
- Для отображения данных используются JSP или XML-шаблоны (*facelets*).

# Достоинства JSF

- Чёткое разделение бизнес-логики и интерфейса (фреймворк реализует шаблон MVC).
- Управление обменом данными на уровне компонент.
- Простая работа с событиями на стороне сервера.
- Доступность нескольких реализаций от различных компаний-разработчиков.
- Расширяемость (можно использовать дополнительные наборы компонентов).
- Широкая поддержка со стороны интегрированных средств разработки (IDE).

# Недостатки JSF

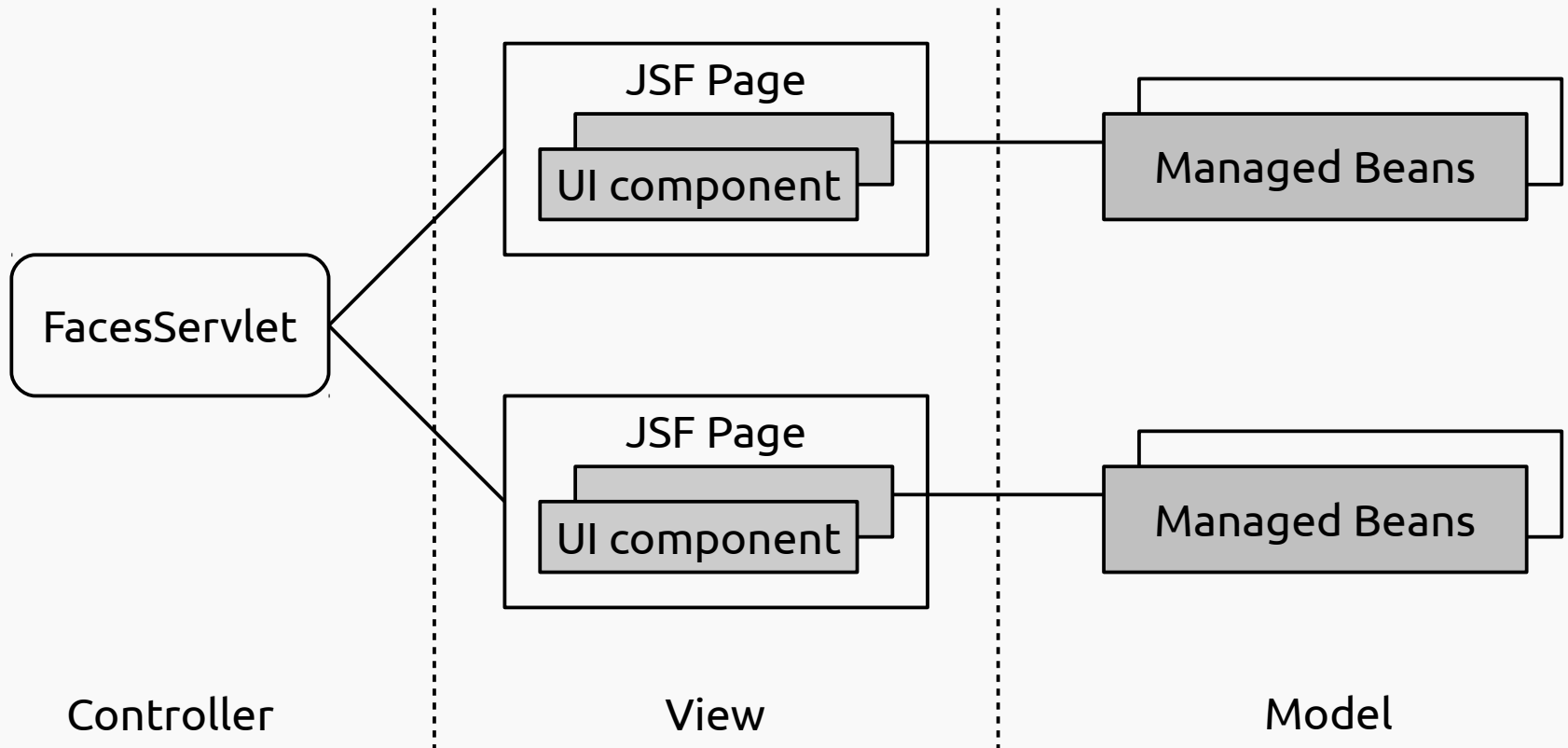
- Высокоуровневый фреймворк — сложно реализовывать не предусмотренную авторами функциональность.
- Сложности с обработкой GET-запросов (устранены в JSF 2.0).
- Сложность разработки собственных компонентов.

# Структура JSF-приложения

- JSP или XHTML-страницы, содержащие компоненты GUI.
- Библиотеки тегов.
- Управляемые бины.
- Дополнительные объекты (компоненты, конвертеры и валидаторы).
- Дополнительные теги.
- Конфигурация — `faces-config.xml` (опционально).
- Дескриптор развёртывания — `web.xml`.

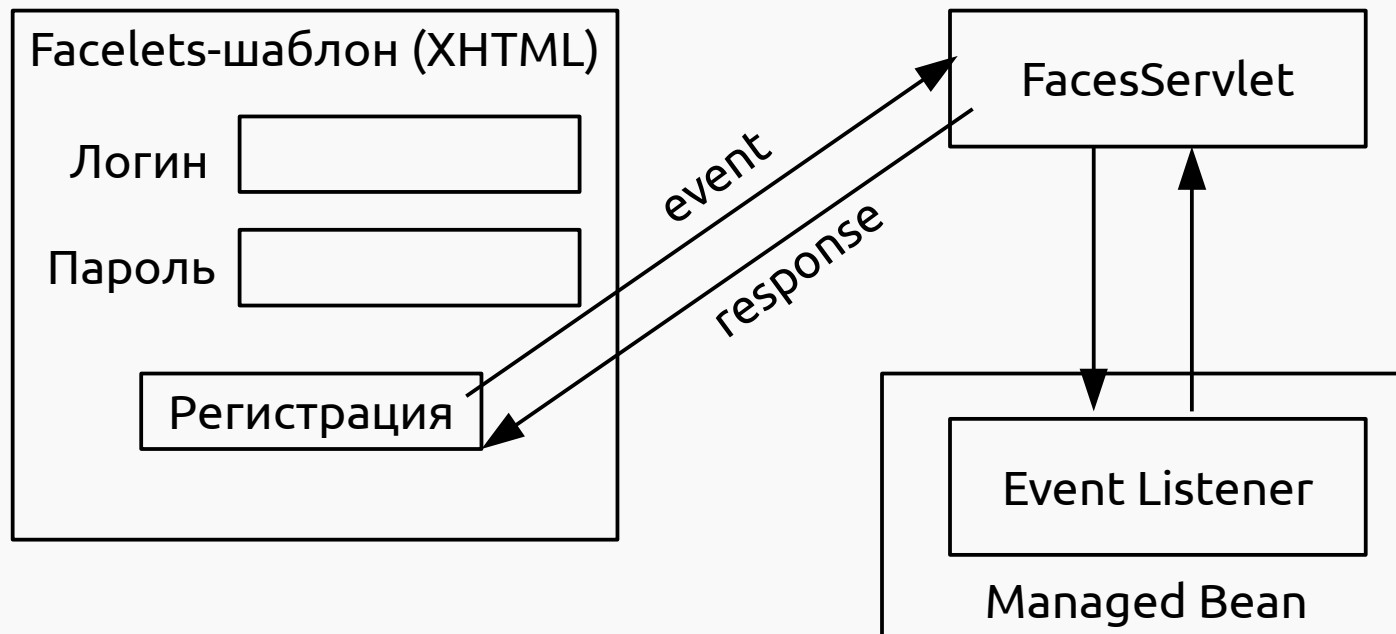


# MVC-модель JSF



# FacesServlet

- Обрабатывает запросы с браузера.
- Формирует объекты-события и вызывает методы-слушатели.



# Конфигурация FacesServlet

Конфигурация задаётся в web.xml:

```
<!-- Faces Servlet -->
<servlet>
  <servlet-name>Faces Servlet</servlet-name>
  <servlet-class>
    javax.faces.webapp.FacesServlet
  </servlet-class>
  <load-on-startup>1</load-on-startup>
</servlet>
<!-- Faces Servlet Mapping -->
<servlet-mapping>
  <servlet-name>Faces Servlet</servlet-name>
  <url-pattern>/faces/*</url-pattern>
</servlet-mapping>
```

# Страницы и компоненты UI

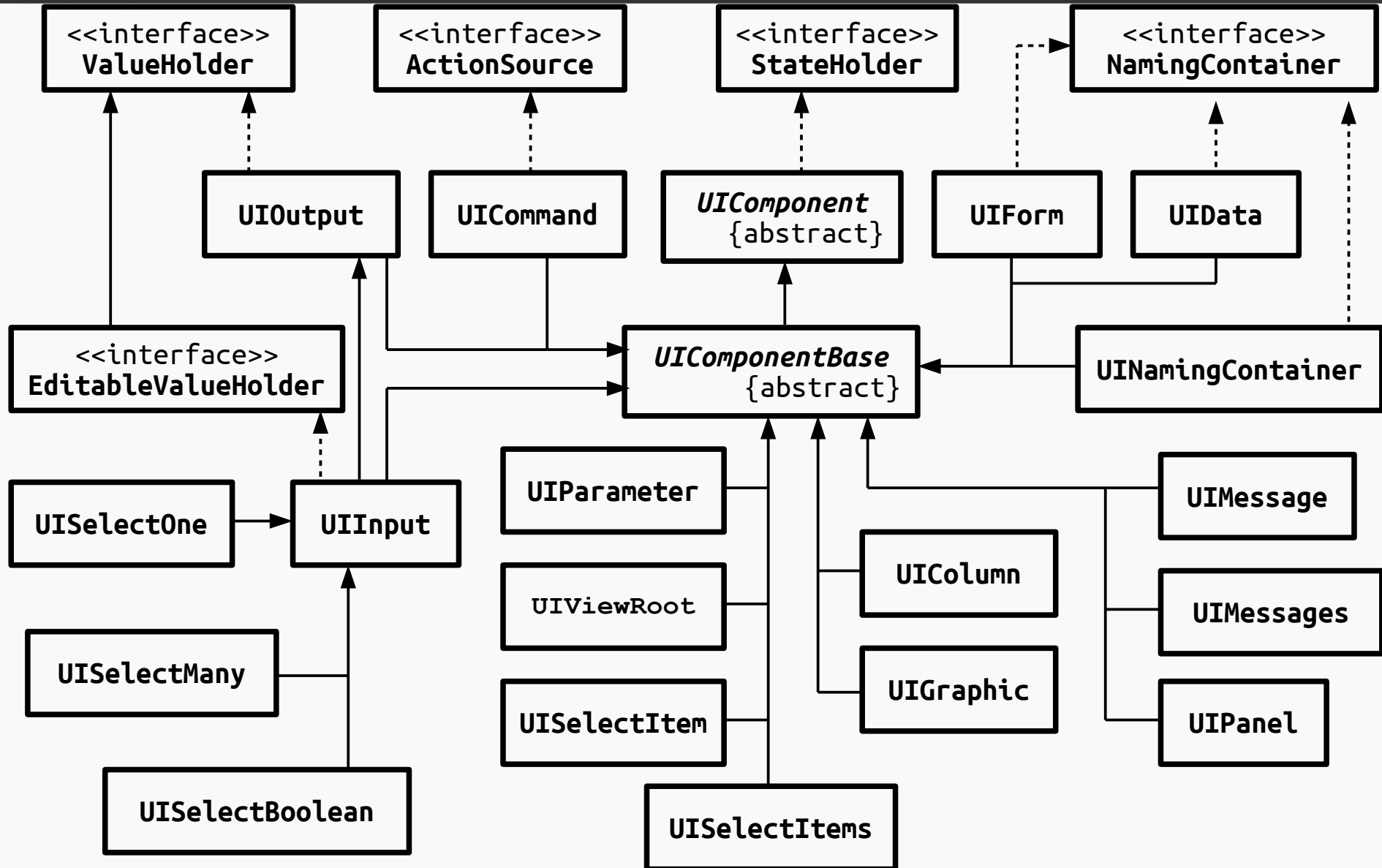
- Интерфейс строится из компонентов.
- Компоненты расположены на Facelets-шаблонах или страницах JSP.
- Компоненты реализуют интерфейс `javax.faces.component.UIComponent`.
- Можно создавать собственные компоненты.
- Компоненты на странице объединены в древовидную структуру — *представление*.
- Корневым элементом представления является экземпляр класса `javax.faces.component.UIViewRoot`.

# Пример страницы JSF (Facelets)

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
    xmlns:f="http://java.sun.com/jsf/core"
    xmlns:h="http://java.sun.com/jsf/html">
    <h:body>
        <h3>JSF 2.0 + Ajax Hello World Example</h3>
        <h:form>
            <h:inputText id="name"
value="#{helloBean.name}"></h:inputText>
            <h:commandButton value="Welcome Me">
                <f:ajax execute="name" render="output" />
            </h:commandButton>
            <h2>
                <h:outputText id="output"
value="#{helloBean.sayWelcome}" />
            </h2>
        </h:form>
    </h:body>
</html>
```



# Иерархия компонентов JSF



# Навигация между страницами JSF

- Реализуется экземплярами класса `NavigationHandler`.
- Правила задаются в файле `faces-config.xml`:

```
<navigation-rule>  
  <from-view-id>/pages/inputname.xhtml</from-view-id>  
  <navigation-case>  
    <from-outcome>sayHello</from-outcome>  
    <to-view-id>/pages/greeting.xhtml</to-view-id>  
  </navigation-case>  
  <navigation-case>  
    <to-view-id>/pages/goodbye.xhtml</to-view-id>  
  </navigation-case>  
</navigation-rule>
```
- Пример перенаправления на другую страницу:

```
<h:commandButton id="submit"  
  action="sayHello" value="Submit" />
```

# Управляемые бины

- Содержат параметры и методы для обработки данных с компонентов.
- Используются для обработки событий UI и валидации данных.
- Жизненным циклом управляет JSF Runtime Environment.
- Доступ из JSF-страниц осуществляется с помощью элементов EL.
- Конфигурация задаётся в `faces-config.xml` (JSF 1.X), либо с помощью аннотаций (JSF 2.0).





ИТМО ВТ

# Пример управляемого бина

```
package org.itmo.sample;

import javax.faces.bean.ManagedBean;
import javax.faces.bean.SessionScoped;
import java.io.Serializable;

@ManagedBean
@SessionScoped
public class HelloBean implements Serializable {

    private static final long serialVersionUID = 1L;
    private String name;

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public String getSayWelcome(){
        if("").equals(name) || name == null){ //check if null?
            return "";
        }else{
            return "Ajax message : Welcome " + name;
        }
    }
}
```

# Контекст (scope) управляемых бинов

- Задаётся через `faces-config.xml` или с помощью аннотаций.
- 6 вариантов конфигурации:
  - `@NoneScoped` — контекст не определён, жизненным циклом управляют другие бины.
  - `@RequestScoped` (применяется по умолчанию) — контекст — запрос.
  - `@ViewScoped` (JSF 2.0) — контекст — страница.
  - `@SessionScoped` — контекст — сессия.
  - `@ApplicationScoped` — контекст — приложение.
  - `@CustomScoped` (JSF 2.0) — бин сохраняется в Map; программист сам управляет его жизненным циклом.

Способ 1 — через faces-config.xml:

```
<managed-bean>
  <managed-bean-name>customer</managed-bean-name>
  <managed-bean-class>CustomerBean</managed-bean-class>
  <managed-bean-scope>request</managed-bean-scope>
  <managed-property>
    <property-name>areaCode</property-name>
    <value>#{initParam.defaultAreaCode}</value>
  </managed-property>
</managed-bean>
```

# Конфигурация управляемых бинов (продолжение)

Способ 2 (JSF 2.0) — с помощью аннотаций:

```
@ManagedBean(name="customer")
@RequestScoped
public class CustomerBean {

    ...

    @ManagedProperty(value="#{initParam.defaultAreaCode}"
        name="areaCode")
    private String areaCode;

    ...
}
```

Осуществляется с помощью EL-выражений:

```
...  
<h:inputText value="#{user.name}"  
              validator="#{user.validate}" />  
  
...  
<h:inputText binding="#{user.nameField}" />  
  
...  
<h:commandButton action="#{user.save}"  
                  value="Save" />  
  
...
```

# Конвертеры данных

- Используются для преобразования данных компонента в заданный формат (дата, число и т. д.).
- Реализуют интерфейс `javax.faces.convert.Converter`.
- Существуют стандартные конвертеры для основных типов данных.
- Можно создавать собственные конвертеры.

# Назначение конвертеров

- Автоматическое (на основании типа данных):  
`<h:inputText value="#{user.age}" />`
- С помощью атрибута `converter`:  
`<h:inputText  
    converter="#{javax.faces.DateTime}" />`
- С помощью вложенного тега:  
`<h:outputText value="#{user.birthday}">  
    <f:converter  
        converterId="#{javax.faces.DateTime}" />  
</h:outputText>`

- Осуществляется перед обновлением значения компонента на уровне модели.
- Класс, осуществляющий валидацию, должен реализовывать интерфейс `javax.faces.validator.Validator`.
- Существуют стандартные валидаторы для основных типов данных.
- Можно создавать собственные валидаторы.



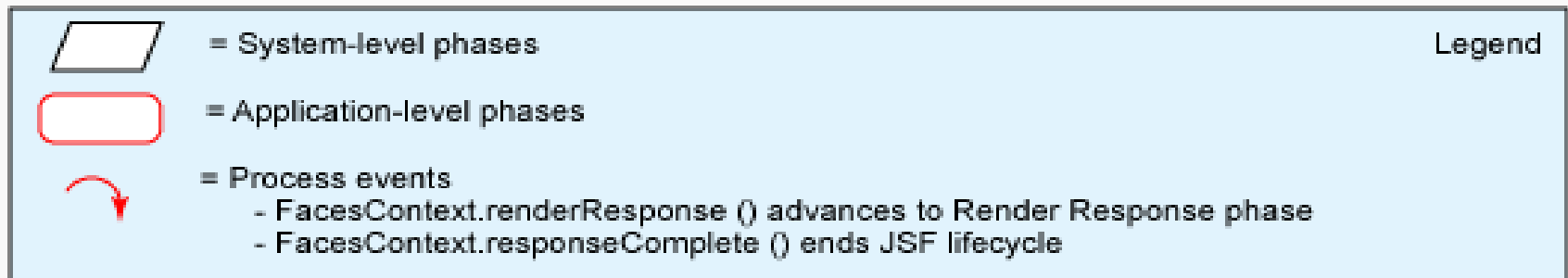
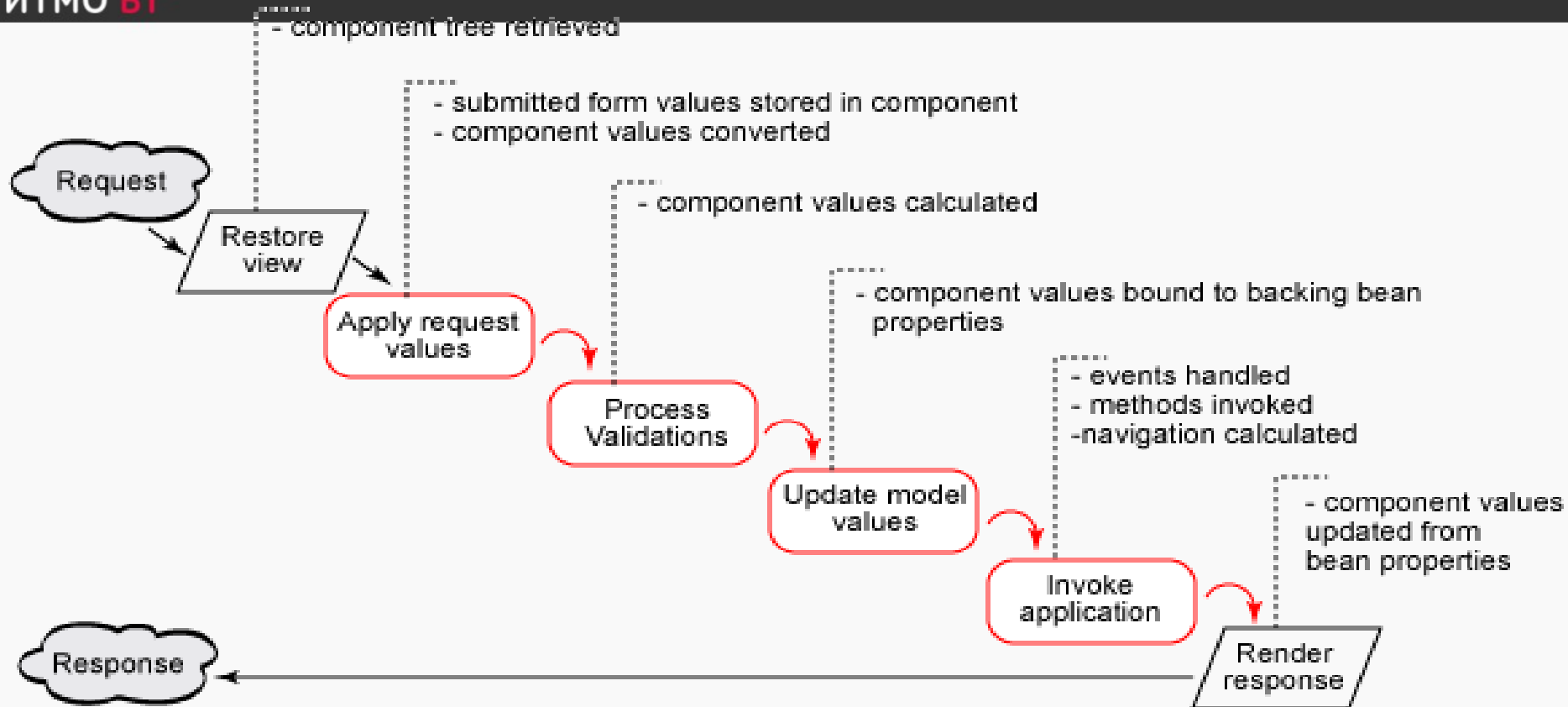
# Способы валидации данных

- С помощью параметров компонента:  

```
<h:inputText id="zip" size="10"  
              value="#{customerBean.zip}"  
              required="true">  
</h:inputText>  
<h:message for="zip"/>
```
- С помощью вложенного тега:  

```
<h:inputText id="quantity" size="4"  
              value="#{item.quantity}">  
  <f:validateLongRange minimum="1"/>  
</h:inputText>  
<h:message for="quantity"/>
```
- С помощью логики на уровне управляемого бина.

# Обработка событий



# Фаза формирования представления (Restore View Phase)

- JSF Runtime формирует представление (начиная с UIViewRoot):
  - Создаются объекты компонентов.
  - Назначаются слушатели событий, конвертеры и валидаторы.
  - Все элементы представления помещаются в FacesContext.
- Если это первый запрос пользователя к странице JSF, то формируется пустое представление.
- Если это запрос к уже существующей странице, то JSF Runtime синхронизирует состояние компонентов представления с клиентом.

# Фаза получения значений компонентов (Apply Request Values Phase)

- На стороне клиента все значения хранятся в строковом формате — нужна проверка их корректности:
  - Вызывается конвертер в соответствии с типом данных значения.
- Если конвертация заканчивается успешно, значение сохраняется в *локальной переменной* компонента.
- Если конвертация заканчивается неудачно, создаётся сообщение об ошибке, которое помещается в FacesContext.

# Фаза валидации значений компонентов (Process Validations Phase)

- Вызываются валидаторы, зарегистрированные для компонентов представления.
- Если значение компонента не проходит валидацию, формируется сообщение об ошибке, которое сохраняется в FacesContext.

# Фаза обновления значений компонентов (Update Model Values Phase)

- Если данные валидны, то значение компонента обновляется.
- Новое значение присваивается *полю* объекта компонента.

# Фаза вызова приложения (Invoke Application Phase)

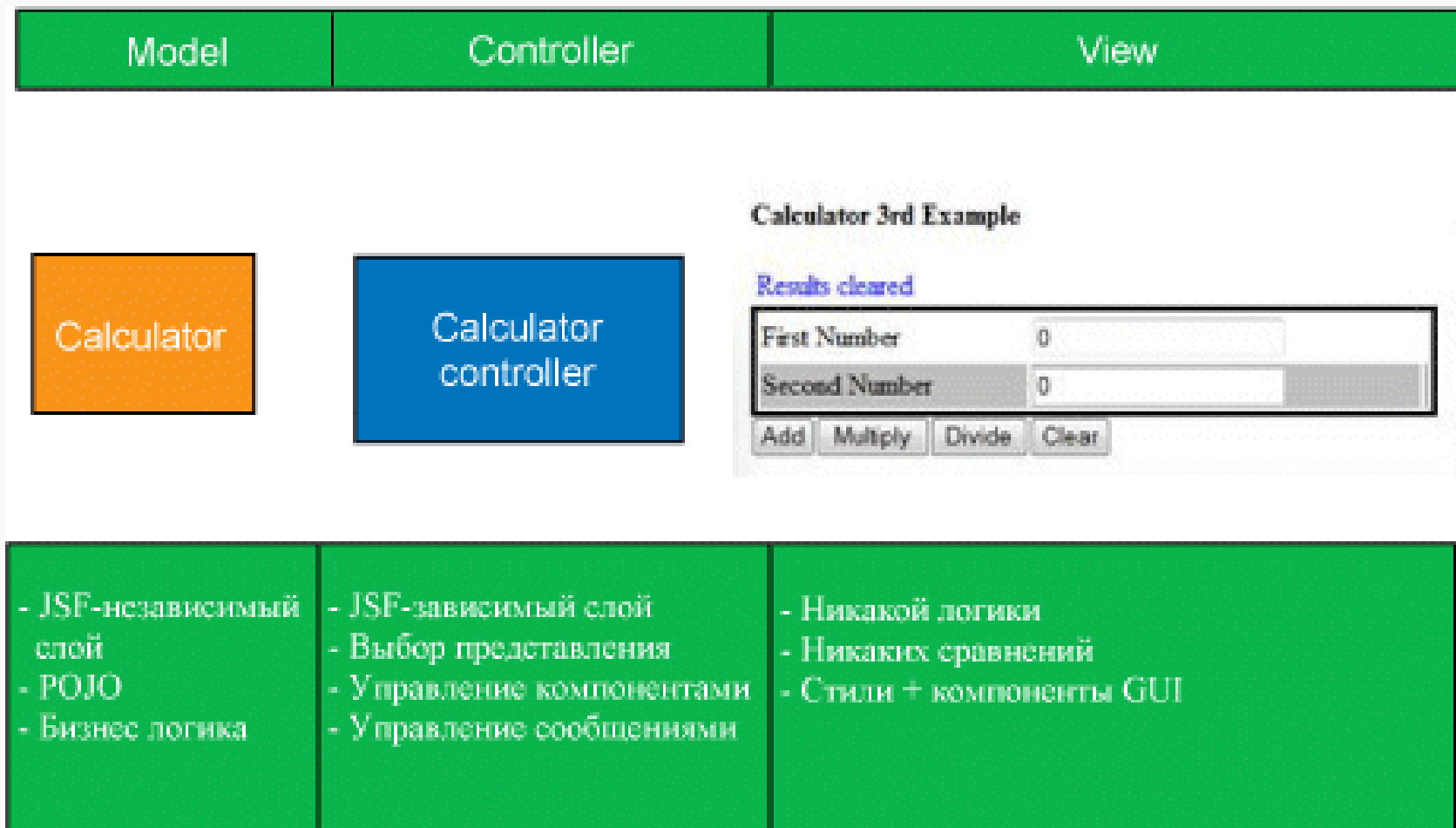
- Управление передаётся слушателям событий.
- Формируются новые значения компонентов.

# Фаза формирования ответа сервера (Render Response Phase)

- JSF Runtime обновляет представление в соответствии с результатами обработки запроса.
- Если это первый запрос к странице, то компоненты помещаются в иерархию представления.
- Формируется ответ сервера на запрос.
- На стороне клиента происходит обновление страницы.



# Пример JSF-приложения





# Пример JSF-приложения (продолжение)

Calculator 3rd Example

First Number	<input type="text" value="aaa"/>	not a number
Second Number	<input type="text"/>	required
<input type="button" value="Add"/> <input type="button" value="Multiply"/> <input type="button" value="Divide"/> <input type="button" value="Clear"/>		

Обрабатывает ошибки  
и валидирует входные данные.  
Выводит сообщения об ошибках  
красным шрифтом рядом с полями ввода

/ by zero

First Number	<input type="text" value="100"/>
Second Number	<input type="text" value="1"/>
<input type="button" value="Add"/> <input type="button" value="Multiply"/> <input type="button" value="Divide"/> <input type="button" value="Clear"/>	

Исправляет ошибку деления на ноль,  
присваивая делителю значение 1  
и выводя соответствующее сообщение.

Calculator 3rd Example

Added successfully

First Number	<input type="text" value="5"/>
Second Number	<input type="text" value="5"/>
<input type="button" value="Add"/> <input type="button" value="Multiply"/> <input type="button" value="Divide"/> <input type="button" value="Clear"/>	

Results

First Number 5
Second Number 5
Result 10

Позволяет производить операции  
сложения, умножения, деления,  
а также сбрасывать результат.  
Результат выводится только после  
выполнения операции

Конфигурация web.xml:

```
...
<servlet>
  <servlet-name>Faces Servlet</servlet-name>
  <servlet-class>javax.faces.webapp.FacesServlet</servlet-class>
  <load-on-startup>1</load-on-startup>
</servlet>
...
<servlet-mapping>
  <servlet-name>Faces Servlet</servlet-name>
  <url-pattern>*.jsf</url-pattern>
</servlet-mapping>
<servlet-mapping>
  <servlet-name>Faces Servlet</servlet-name>
  <url-pattern>/faces/*</url-pattern>
</servlet-mapping>
...
```

```
package org.itmo.sample;

public class Calculator {

    /** Первый операнд */
    private int firstNumber = 0;

    /** Результат операции */
    private int result = 0;

    /** Второй операнд */
    private int secondNumber = 0;

    /** Сложение операндов */
    public void add() {
        result = firstNumber + secondNumber;
    }

    /** Перемножение операндов */
    public void multiply() {
        result = firstNumber * secondNumber;
    }

    /** Сброс результата */
    public void clear() {
        result = 0;
    }
}
```



# Пример JSF-приложения (продолжение)

```
/* ----- СВОЙСТВА ----- */  
  
public int getFirstNumber() {  
    return firstNumber;  
}  
  
public void setFirstNumber(int firstNumber) {  
    this.firstNumber = firstNumber;  
}  
  
public int getResult() {  
    return result;  
}  
  
public void setResult(int result) {  
    this.result = result;  
}  
  
public int getSecondNumber() {  
    return secondNumber;  
}  
  
public void setSecondNumber(int secondNumber) {  
    this.secondNumber = secondNumber;  
}  
}
```

Конфигурация faces-config.xml:

```
<?xml version="1.0" encoding="UTF-8"?>
<faces-config xmlns="http://java.sun.com/xml/ns/javaee"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
    http://java.sun.com/xml/ns/javaee/web-facesconfig_1_2.xsd"
  version="1.2">
  <managed-bean>
    <managed-bean-name>calculator</managed-bean-name>
    <managed-bean-class>
      org.itmo.sample.Calculator
    </managed-bean-class>
    <managed-bean-scope>request</managed-bean-scope>
  </managed-bean>
</faces-config>
```



# Пример JSF-приложения (продолжение)

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<%@ taglib uri="http://java.sun.com/jsf/html" prefix="h"%>
<%@ taglib uri="http://java.sun.com/jsf/core" prefix="f"%>

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">

<head>
  <title>Calculator Application</title>
</head>

<body>
<f:view>
  <h:form id="calcForm">
    <h4>Calculator</h4>
    <table>
      <tr>
        <td><h:outputLabel value="First Number" for="firstNumber" /></td>
        <td><h:inputText id="firstNumber"
          value="#{calculator.firstNumber}" required="true" /></td>
        <td><h:message for="firstNumber" /></td>
      </tr>
```

```
<tr>
  <td><h:outputLabel value="Second Number"
                    for="secondNumber" />
  </td>
  <td><h:inputText id="secondNumber"
                  value="#{calculator.secondNumber}"
                  required="true" /></td>
  <td><h:message for="secondNumber" /></td>
</tr>
</table>

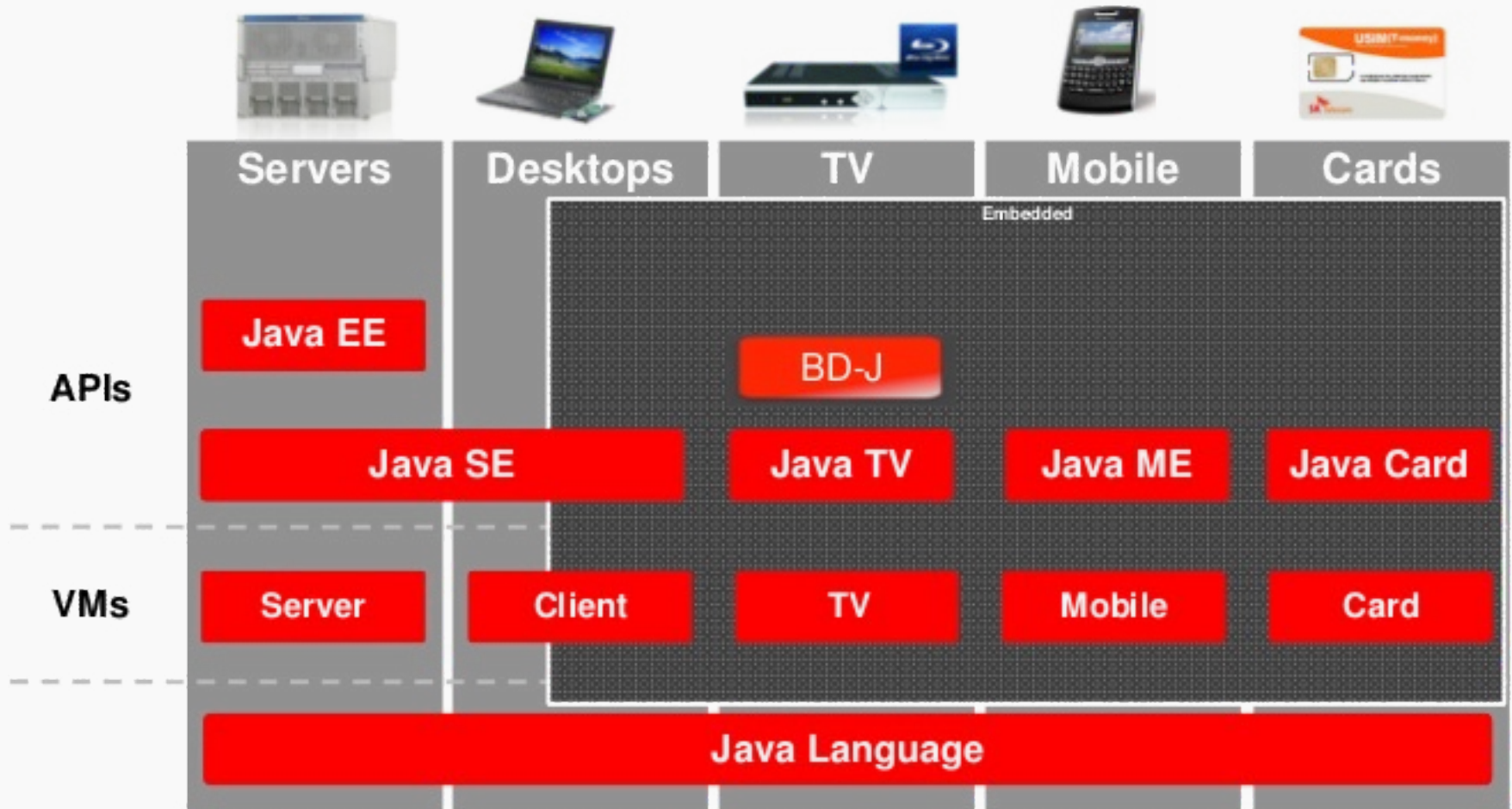
<div>
  <h:commandButton action="#{calculator.add}"
                  value="Add" />
  <h:commandButton action="#{calculator.multiply}"
                  value="Multiply" />
  <h:commandButton action="#{calculator.clear}"
                  value="Clear" immediate="true"/>
</div>
</h:form>
```



```
<h:panelGroup rendered="#{calculator.result != 0}">
  <h4>Results</h4>
  <table>
    <tr><td>
      First Number  ${calculator.firstNumber}
    </td></tr>
    <tr><td>
      Second Number ${calculator.secondNumber}
    </td></tr>
    <tr><td>
      Result ${calculator.result}
    </td></tr>
  </table>
</h:panelGroup>
</f:view>
</body>
</html>
```

# 14. Java EE Full Profile

# Java Platforms



# Основные концепции

- Приложения строятся из компонентов, работающих под управлением контейнеров.
- Используются следующие принципы:
  - Inversion of Control (IoC) + Contexts & Dependency Injection (CDI).
  - Location Transparency.

# Принципы IoC и CDI

- IoC (применительно к Java EE):
  - Жизненным циклом компонента управляет контейнер (а не программист).
  - За взаимодействие между компонентами отвечает тоже контейнер.
- CDI — позволяет снизить (или совсем убрать) зависимость компонента от контейнера:
  - Не требуется реализации каких-либо интерфейсов.
  - Не нужны прямые вызовы API.
  - Реализуется через аннотации.

# Пример CDI (JSF Managed Bean)

```
@ManagedBean(name="message")
@SessionScoped
public class MessageBean implements Serializable {
    //business logic and whatever methods
}
```

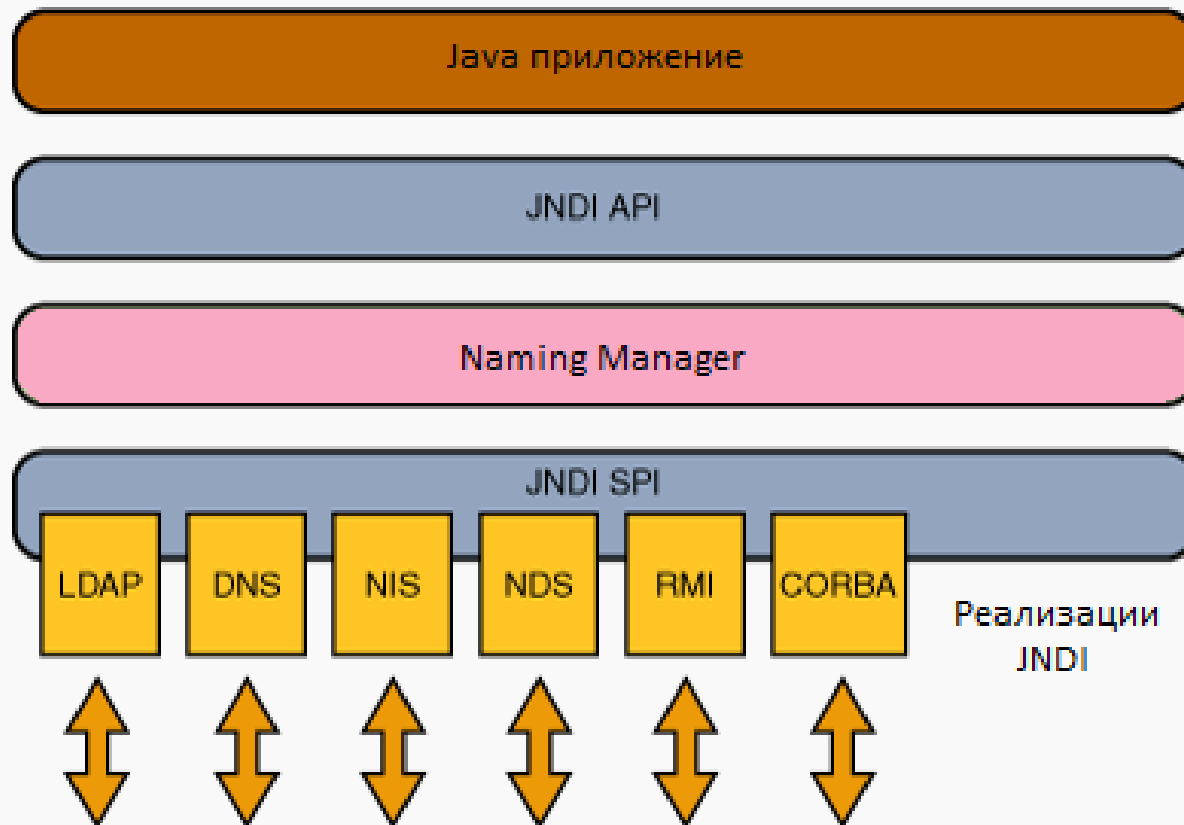
```
@ManagedBean
@SessionScoped
public class HelloBean implements Serializable {

    @ManagedProperty(value="#{message}")
    private MessageBean messageBean;

    //must provide the setter method
    public void setMessageBean(MessageBean messageBean) {
        this.messageBean = messageBean;
    }
}
```

# Java Naming & Directory Interface (JNDI)

JNDI — это набор Java API, организованный в виде службы каталогов, который позволяет Java-клиентам открывать и просматривать данные и объекты по их именам (C) Wikipedia.



Два варианта использования JNDI:

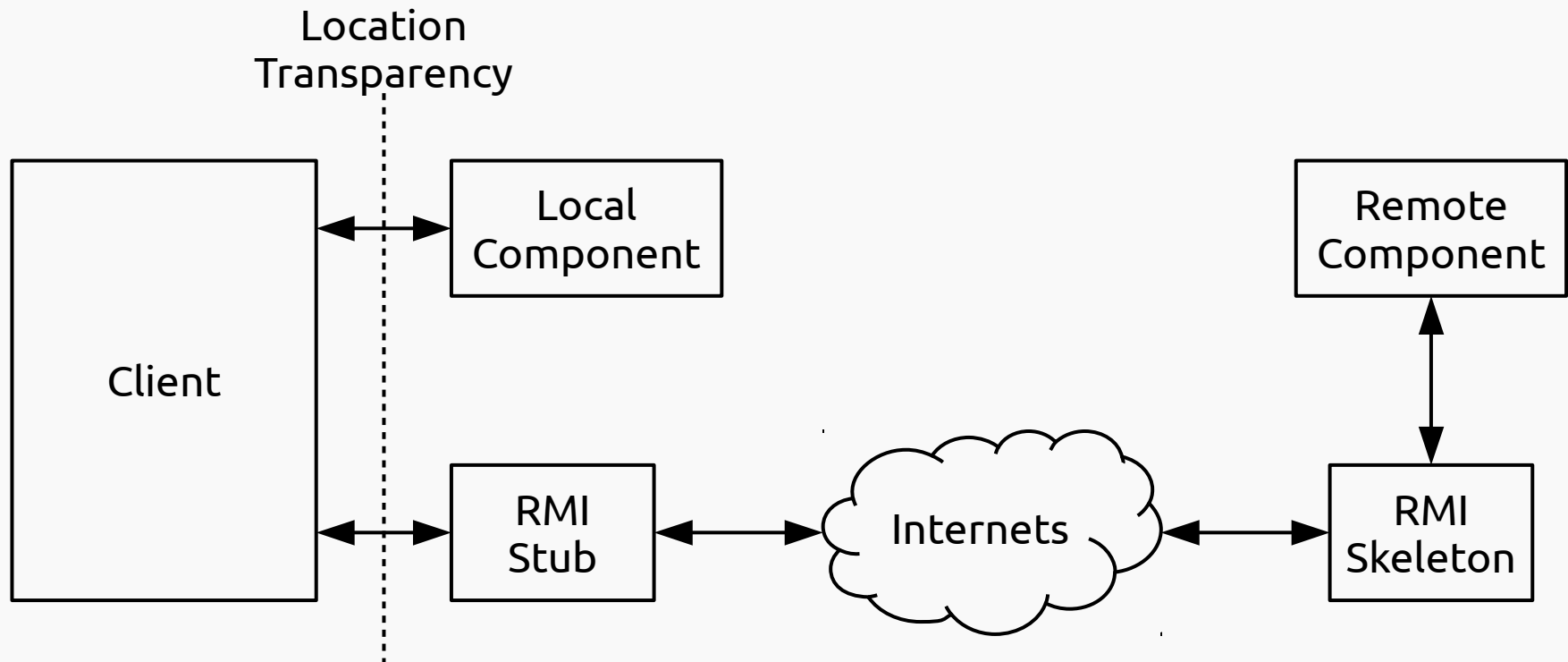
- CDI (аннотации) — работает только в managed компонентах.
- Прямой вызов API — работает везде.

```
// Пример получения ссылки на JDBC datasource.
DataSource dataSource = null;
try {
    // Инициализируем контекст по умолчанию.
    Context context = new InitialContext();
    dataSource =
        (DataSource) context.lookup("Database");
} catch (NamingException e) {
    // Ссылка не найдена
}
```



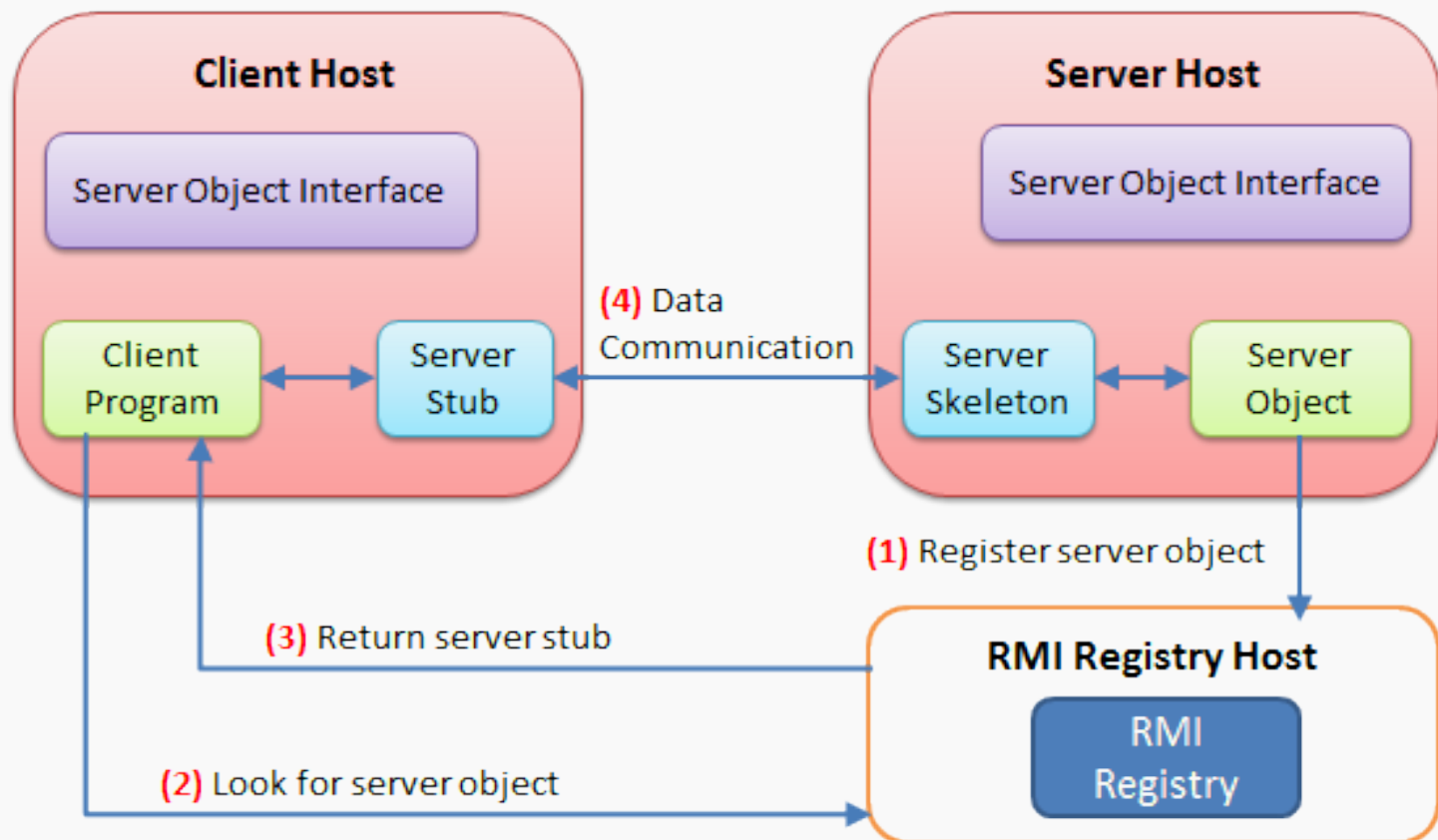
# Принцип Location Transparency

Благодаря CDI не важно, где физически расположен вызываемый компонент — за его вызов отвечает контейнер.



# Remote Method Invocation

RMI — Java API, позволяющий вызывать методы удалённых объектов.



- В общем случае, объекты передаются по значению (копии).
- Передаваемые объекты должны быть Serializable.

# Пример реализации RMI — сервер

```
public class PrimeNumbersSearchServer implements ClientRegister {  
  
    ...  
  
    public static void main(String[] args) {  
        PrimeNumbersSearchServer server =  
            new PrimeNumbersSearchServer();  
        try {  
            ClientRegister stub = (ClientRegister)  
                UnicastRemoteObject.exportObject(server, 0);  
            Registry registry = LocateRegistry.createRegistry(12345);  
            registry.bind("ClientRegister", stub);  
            server.startSearch();  
        } catch (Exception e) {  
            System.out.println ("Error occurred: " + e.getMessage());  
            System.exit (1);  
        }  
    }  
}
```

# Пример реализации RMI — клиент

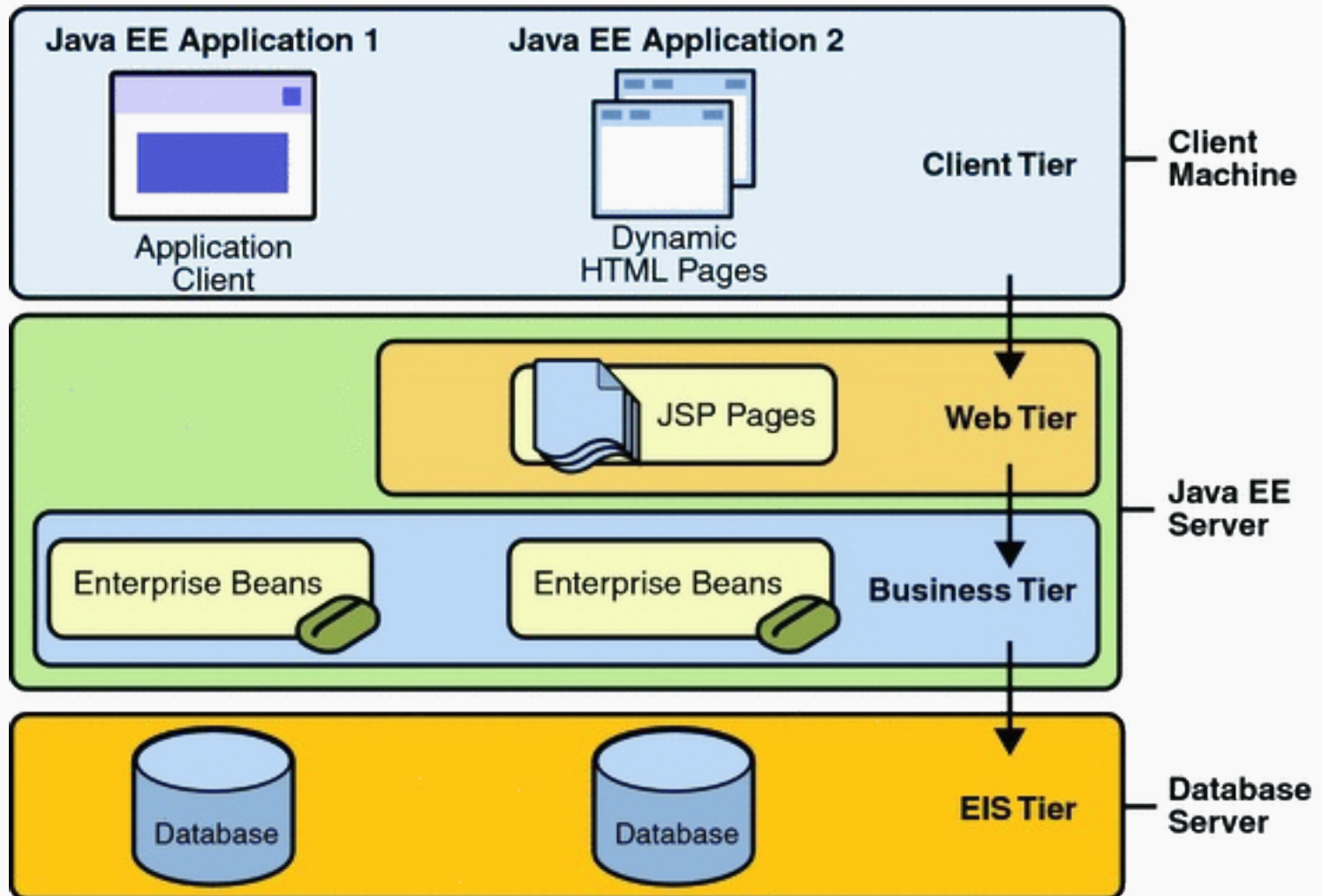
```
public class PrimeNumbersSearchClient implements PrimeChecker {

    ...

    public static void main(String[] args) {
        PrimeNumbersSearchClient client =
            new PrimeNumbersSearchClient();

        try {
            Registry registry = LocateRegistry.getRegistry(null, 12345);
            ClientRegister server =
                (ClientRegister) registry.lookup("ClientRegister");
            PrimeChecker stub = (PrimeChecker)
                UnicastRemoteObject.exportObject(client, 0);
            server.register(stub);
        } catch (Exception e) {
            System.out.println ("Error occurred: " + e.getMessage());
            System.exit (1);
        }
    }
}
```

# Java EE Application Tiers

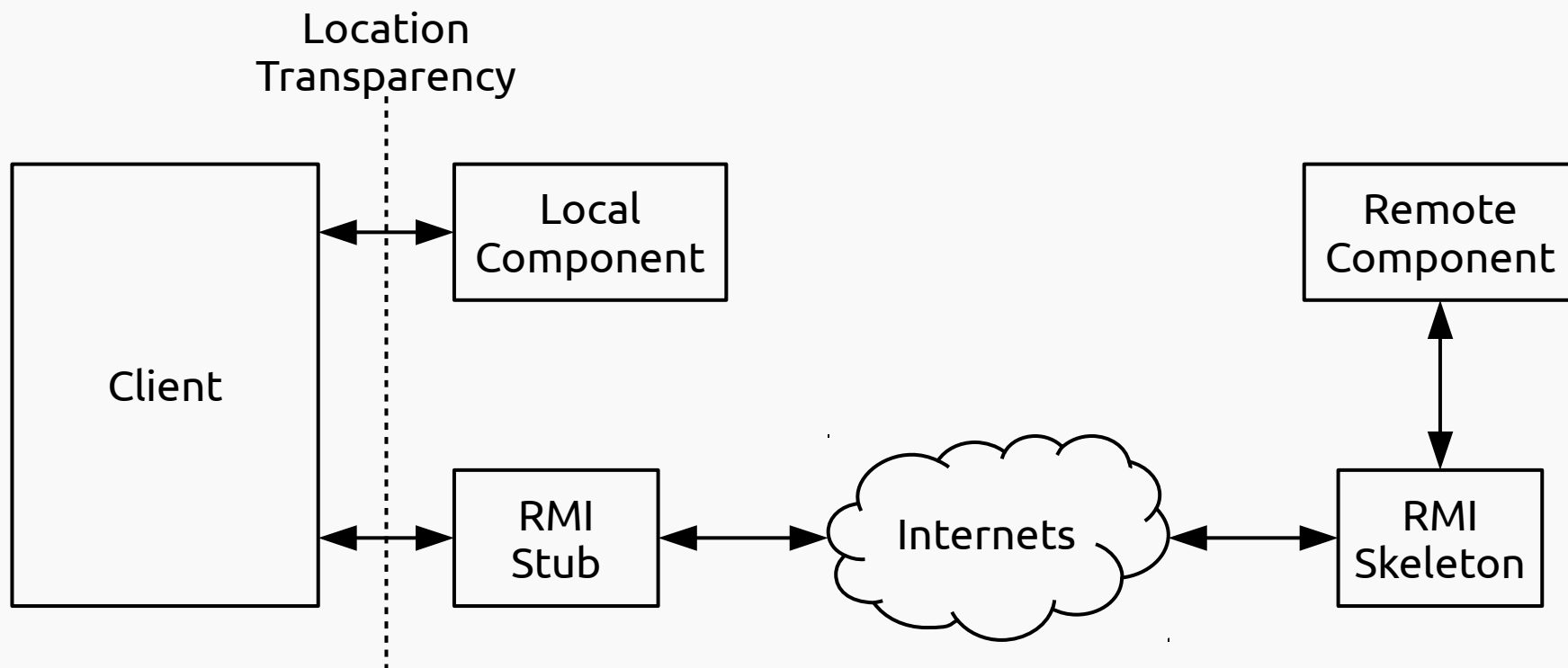


# Профили платформы Java EE

- Появились в Java EE 6.
- Позволяют сделать более «лёгкими» приложения, которым не нужен полный стек технологий Java EE.
- Существует только 2 профиля — Full и Web.
- Сервер приложений может реализовывать спецификации не всей платформы, а конкретного профиля.

# Принцип Location Transparency

Благодаря CDI не важно, где физически расположен вызываемый компонент — за его вызов отвечает контейнер.

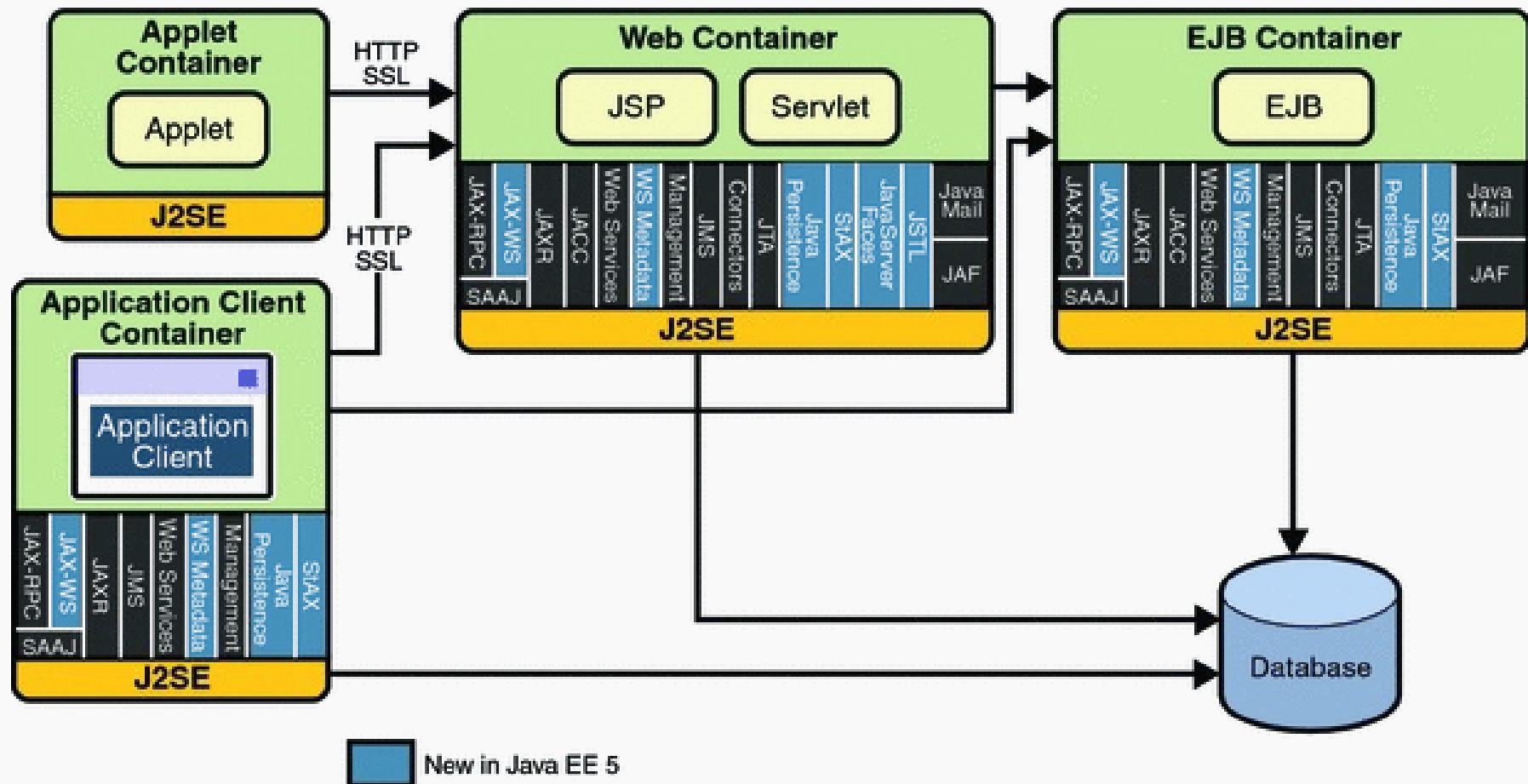




# Java EE Full & Web Profiles

API	Web Profile	Full Profile		API	Web Profile	Full Profile
Servlet	+	+		JTA	+	+
JSP	+	+		JMS		+
JSTL	+	+		JavaMail		+
EL	+	+		JAX-WS		+
JSF	+	+		JAX-RS		+
CDI	+	+		JAXB		+
EJB Lite	+	+		JACC		+
EJB Full		+		JCA		+
JPA	+	+				

# Архитектура приложения Java EE Full Profile



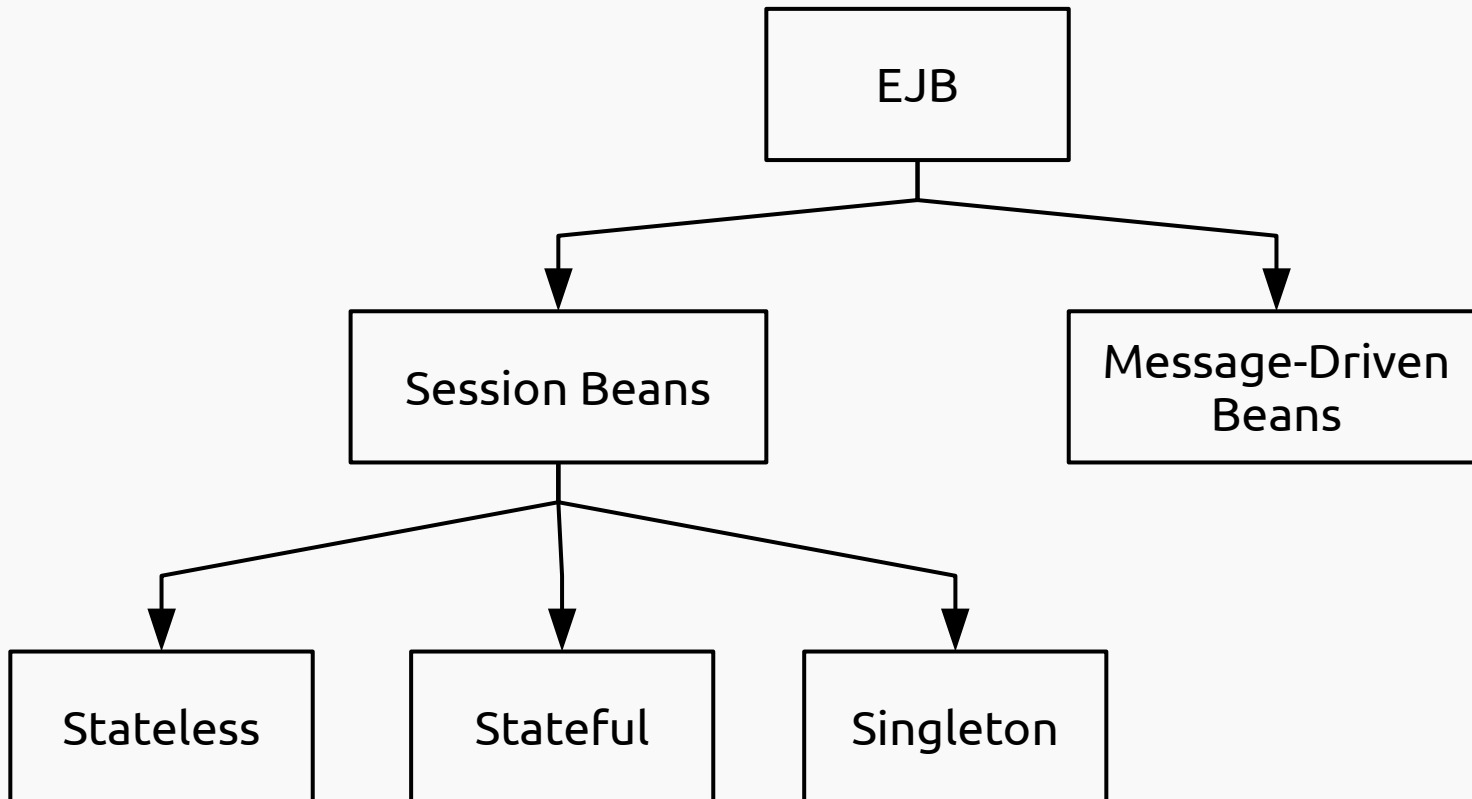
# 15. Enterprise Java Beans

EJB — технология разработки серверных компонентов, реализующих бизнес-логику.

Особенности EJB:

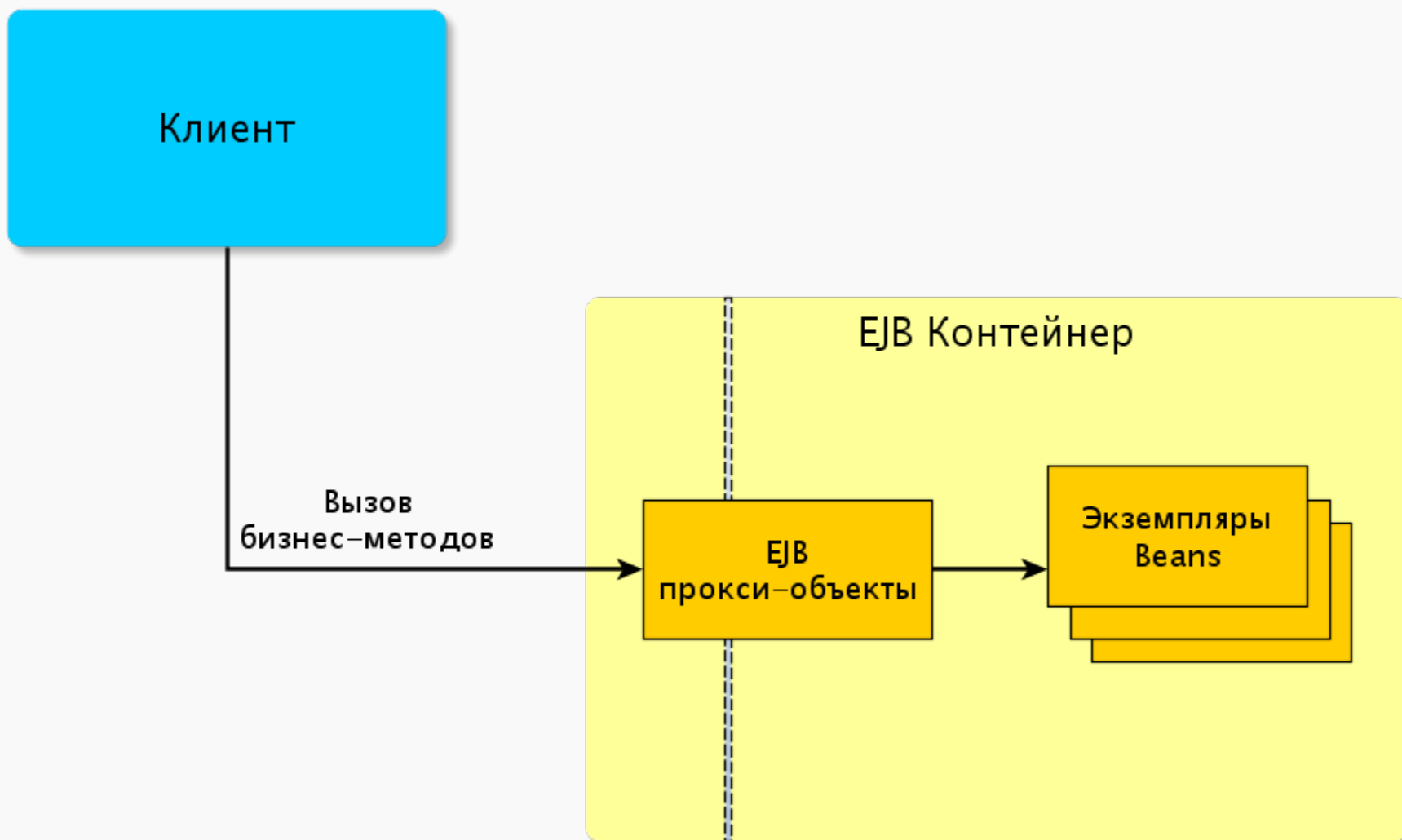
- Возможность локального и удалённого доступа.
- Возможность доступа через JNDI или Dependency Injection.
- Поддержка распределённых транзакций (с помощью JTA).
- Поддержка событий.
- Жизненным циклом управляет EJB-контейнер (в составе сервера приложений).

# Виды ЕJB



- EJB-компоненты инкапсулируются контейнером.
- Контейнер предоставляет клиентам прокси-объекты (proxy objects) для доступа к EJB-компонентам.
- Прокси-объекты реализуют бизнес-интерфейсы (Business Interfaces).
- Клиенты вызывают методы бизнес-интерфейсов.

# Компонентная модель EJB (продолжение)



# Варианты доступа к EJB

- Локальный (Local) — из той же самой JVM.
- Удалённый (Remote) — из другой JVM.
- Через веб-сервис.



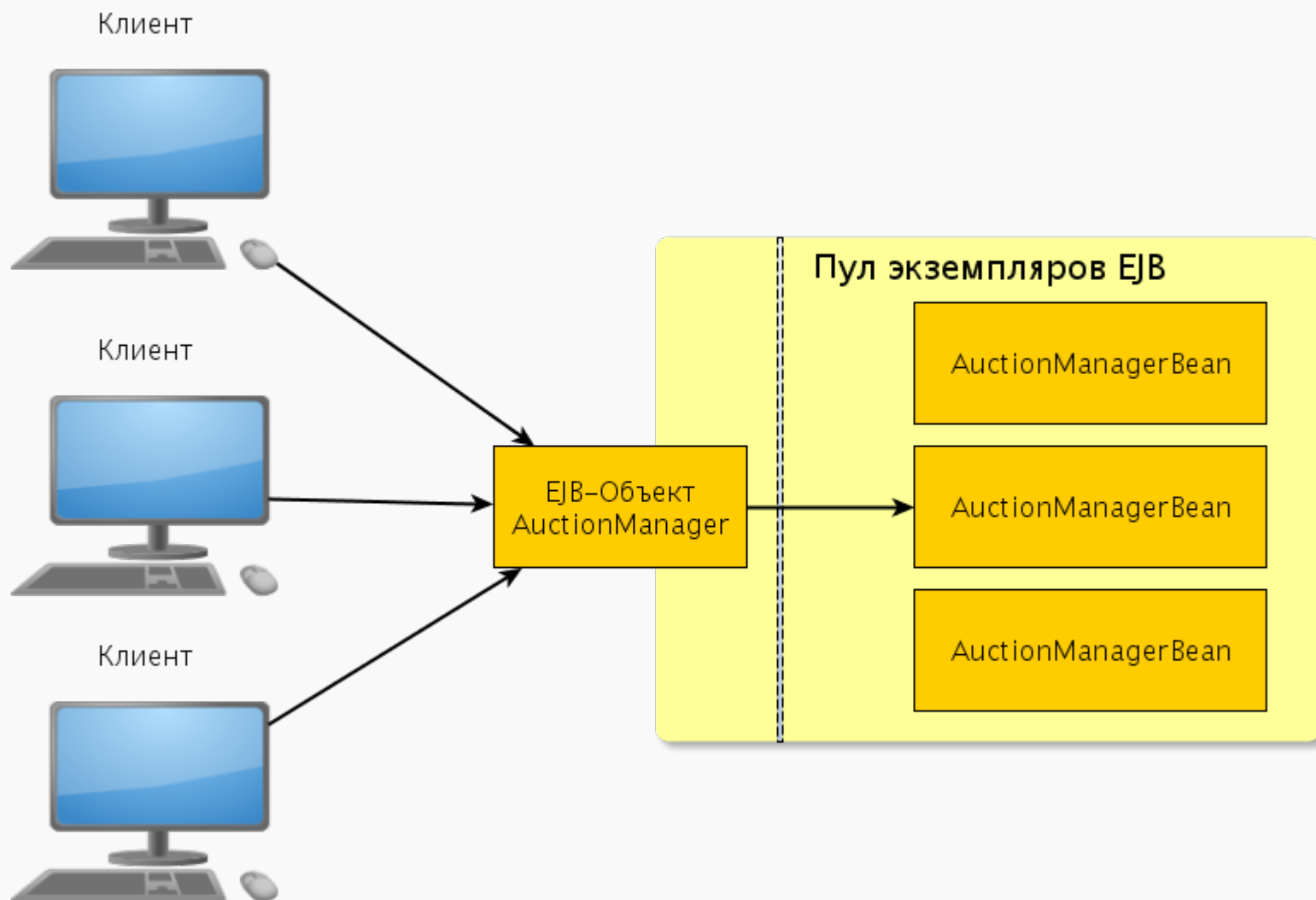
# Session Beans

- Предназначены для синхронной обработки вызовов.
- Вызываются посредством обращения через API.
- Могут вызываться локально или удалённо.
- Могут быть endpoint'ами для веб-сервисов.
- CDI — аннотация @EJB.
- Не обладают свойством персистентности.
- Можно формировать пулы бинов (за исключением @Singleton).

# Stateless Session Beans

- Не сохраняют состояние между последовательными обращениями клиента.
- Нет привязки к конкретному клиенту.
- Хорошо масштабируются.
- Объявление — аннотация `@Stateless`.

# Stateless Session Beans (продолжение)



# Stateless Session Beans (пример)

```
package converter.ejb;

import java.math.BigDecimal;
import javax.ejb.*;

@Stateless
public class ConverterBean {
    private BigDecimal yenRate = new BigDecimal("83.0602");
    private BigDecimal euroRate = new BigDecimal("0.0093016");

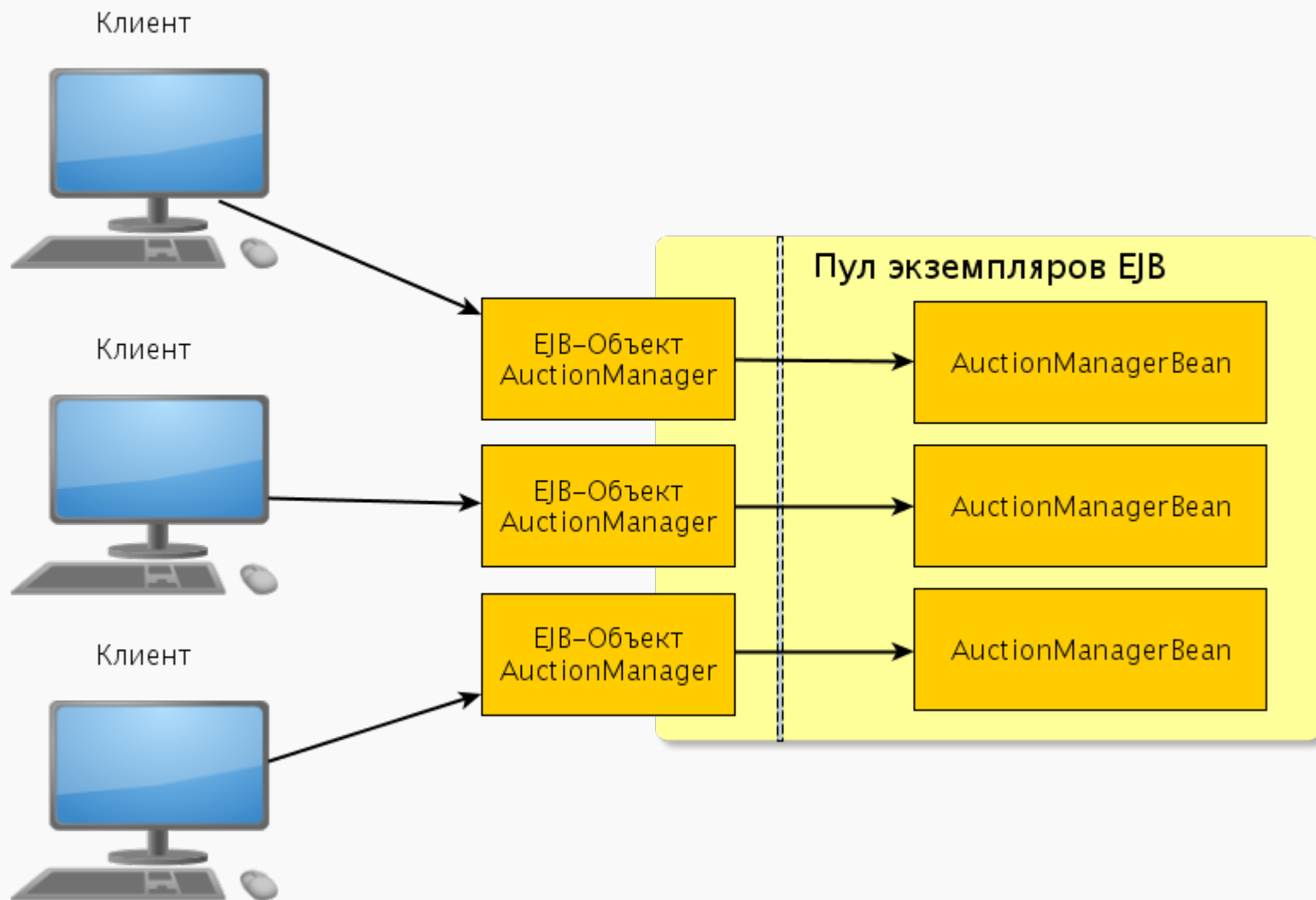
    public BigDecimal dollarToYen(BigDecimal dollars) {
        BigDecimal result = dollars.multiply(yenRate);
        return result.setScale(2, BigDecimal.ROUND_UP);
    }

    public BigDecimal yenToEuro(BigDecimal yen) {
        BigDecimal result = yen.multiply(euroRate);
        return result.setScale(2, BigDecimal.ROUND_UP);
    }
}
```

# Stateful Session Beans

- «Привязываются» к конкретному клиенту.
- Можно сохранять контекст в полях класса.
- Масштабируются хуже, чем `@Stateless`.
- Объявление — аннотация `@Stateful`.

# Stateful Session Beans (продолжение)



# Stateful Session Beans (пример)

```
@Stateful
@StatefulTimeout(unit = TimeUnit.MINUTES, value = 20)
public class CartBean implements Cart {

    @PersistenceContext(unitName = "pu", type = PersistenceContextType.EXTENDED)
    private EntityManager entityManager;

    private List products;

    @PostConstruct
    private void initializeBean(){
        products = new ArrayList<>();
    }

    @Override
    public void addProductToCart(Product product) {
        products.add(product);
    }

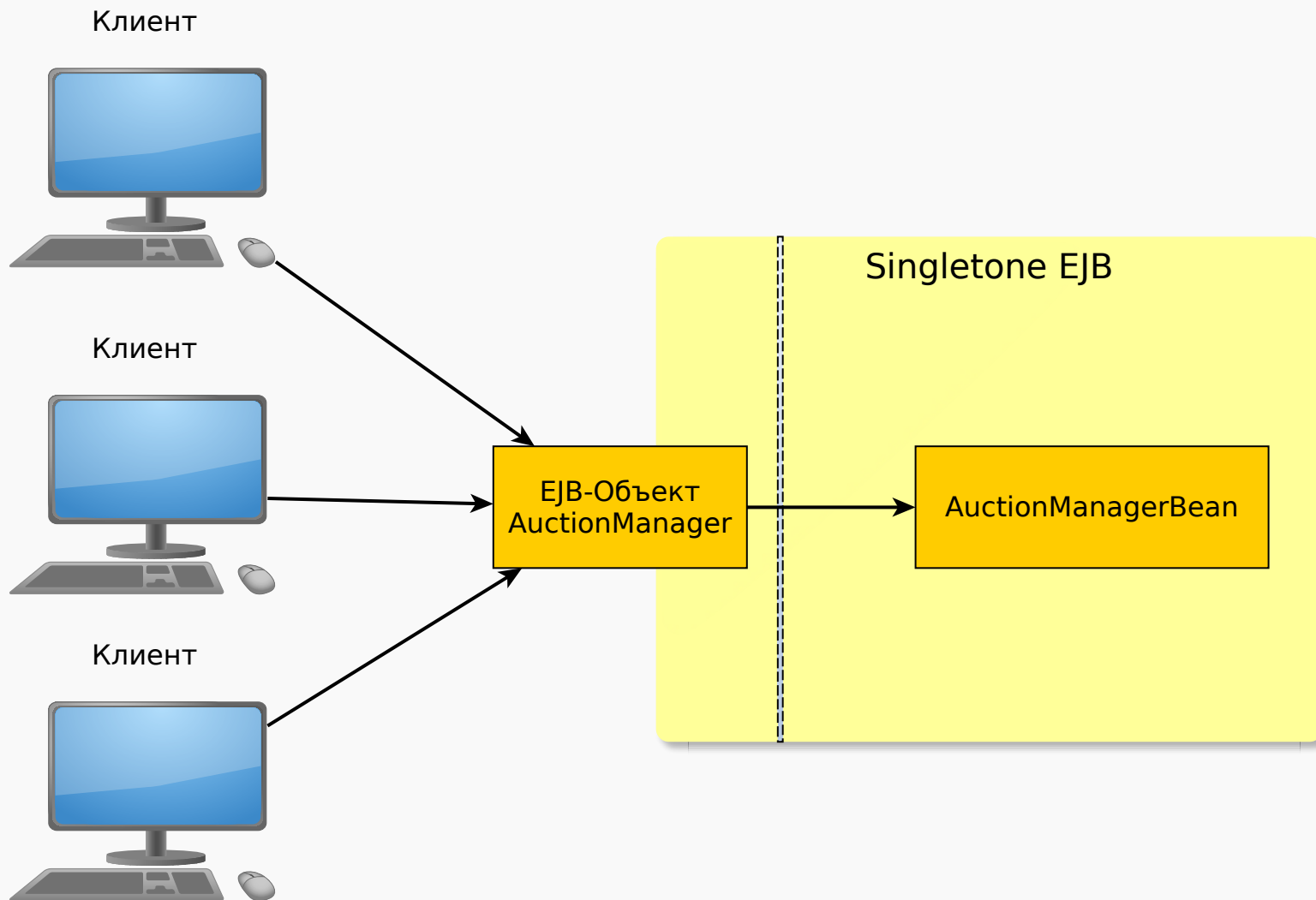
    @Override
    @TransactionAttribute(TransactionAttributeType.REQUIRED)
    public void checkOut() {
        for(Product product : products){
            entityManager.persist(product);
        }
        products.clear();
    }
}
```

# Singleton Session Beans

- Контейнер гарантирует существование строго одного экземпляра такого бина.
- Объявление — аннотация `@Singleton`.



# Singleton Session Beans (продолжение)



- 1) Создание бизнес-интерфейса компонента (Business Interface).
- 2) Создание класса компонента, реализующего бизнес-интерфейс.
- 3) Конфигурация компонента с помощью аннотаций и/или дескриптора развёртывания.

Бизнес-интерфейс содержит заголовки всех методов, реализующих бизнес-логику компонента.

Может быть локальным (`@Local`) и удалённым (`@Remote`).

Пример:

```
import javax.ejb.*;

@Remote
public interface Hello {
    public String sayHello();
}
```

# Создание класса компонента

Компонент должен реализовывать все методы бизнес-интерфейса.

Может содержать методы, реагирующие на события жизненного цикла компонента.

Пример:

```
import javax.ejb.*;

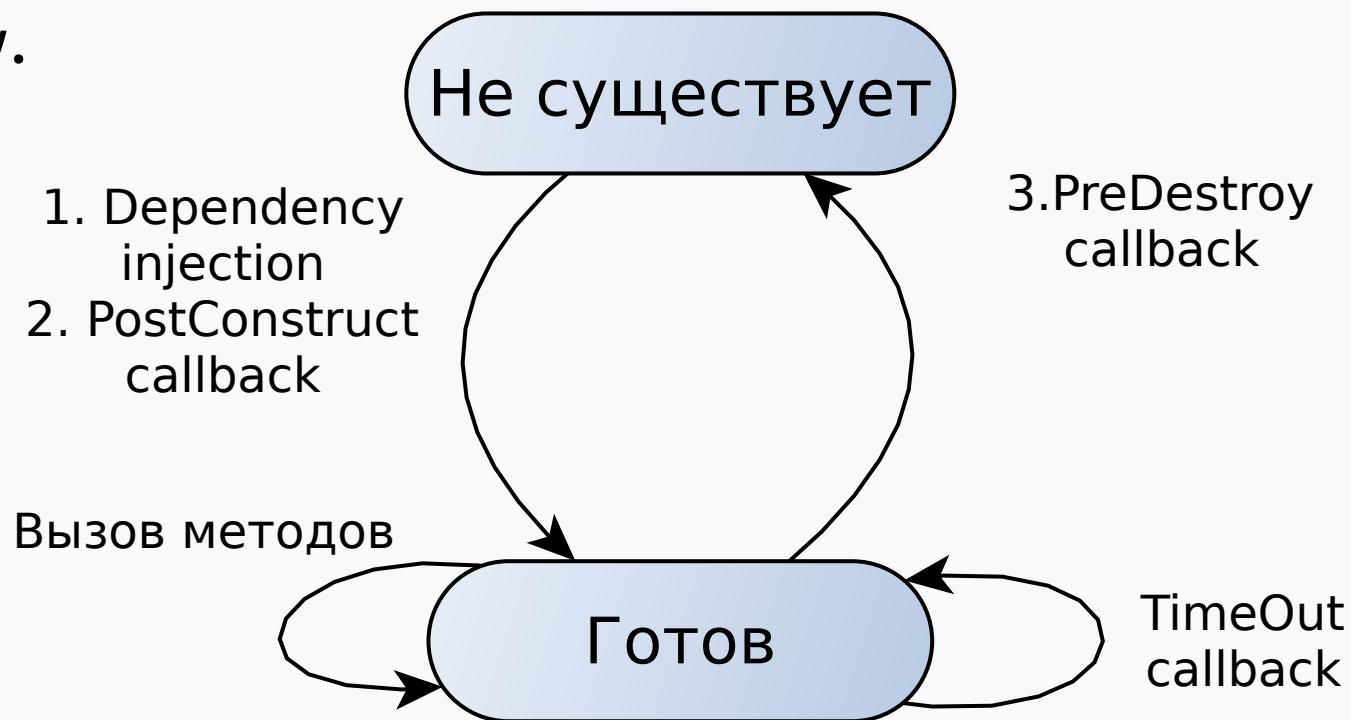
@Stateless
public class HelloBean implements Hello {
    public String sayHello() {
        return "Hello World!";
    }
}
```

# Требования к классам Session Beans

- Должен быть `public` классом верхнего уровня.
- Не должен быть `final` и/или `abstract`.
- Должен иметь `public` конструктор без параметров.
- Не должен переопределять метод `finalize()`.
- Должен реализовывать все методы бизнес-интерфейса(-ов).

Обработчики событий жизненного цикла:

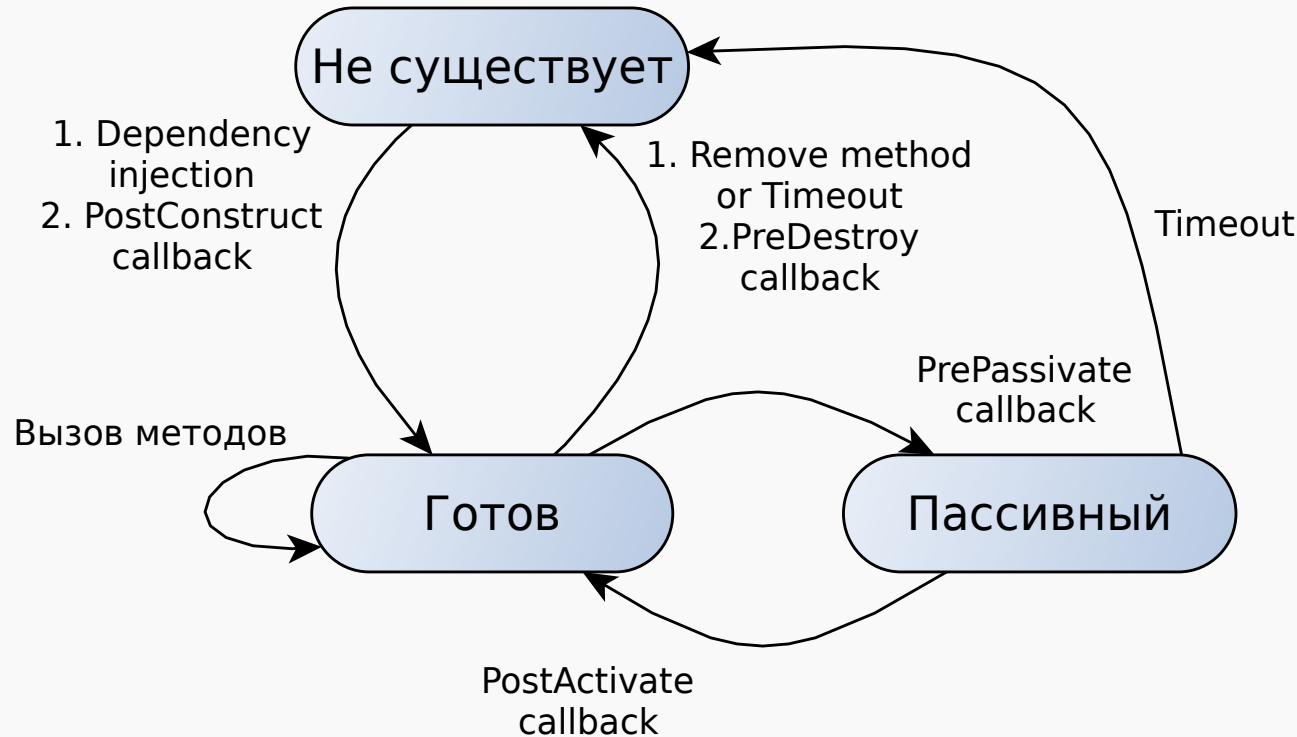
- @PostConstruct;
- @PreDestroy.



# Жизненный цикл Stateful Session Beans

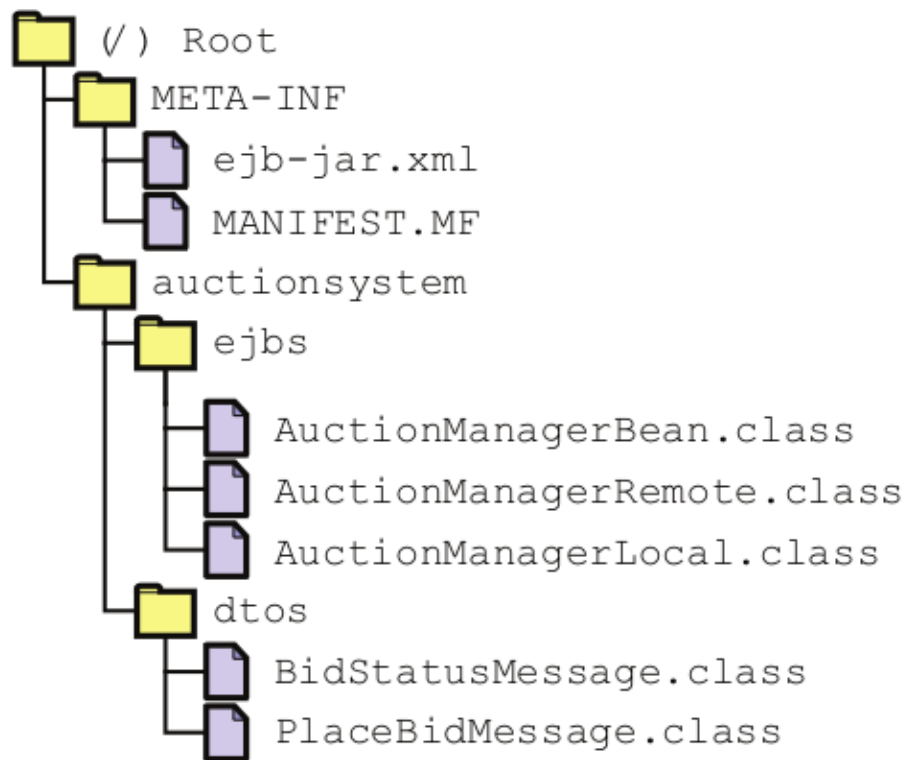
Обработчики событий жизненного цикла:

- `@PostConstruct`;
- `@PreDestroy`;
- `@PostActivate`;
- `@PrePassivate`.



# Развёртывание Session Beans на сервере приложений

- 1) Создание дескриптора развёртывания (опционально).
- 2) Создание jar-архива с компонентами.
- 3) Развёртывание архива на сервере приложений.





# Обращение к Session Beans

С помощью Dependency Injection:

```
import javax.ejb.*;

public class HelloClient {

    @EJB
    private static Hello hello;

    public static void main (String args[]) {
        System.out.println(hello.sayHello())
    }

}
```

# Java Transaction API (JTA)

Интерфейс для управления транзакциями в EJB-компонентах. Позволяет открывать, закрывать и откатывать транзакции.

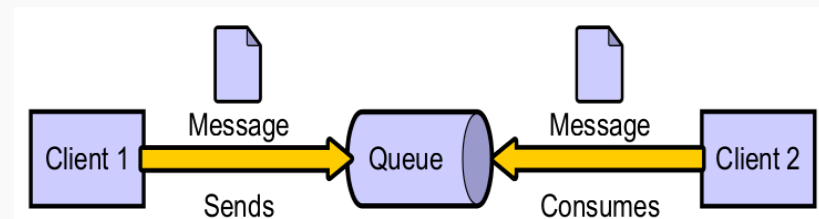
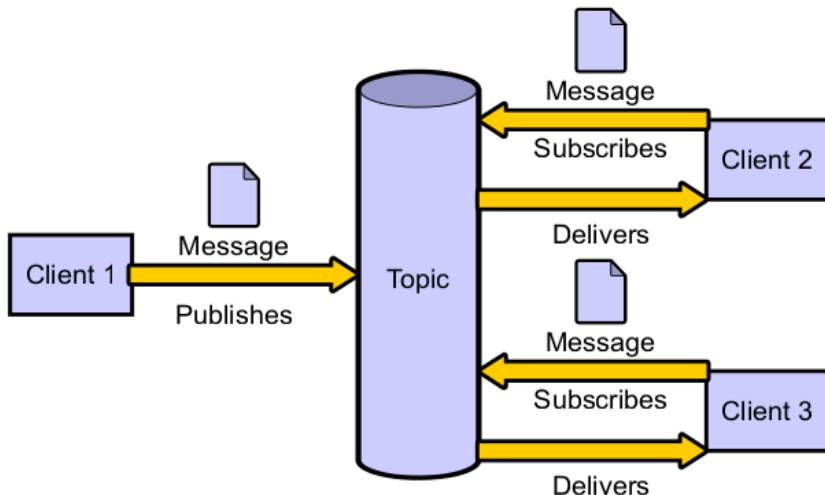
```
@Stateful
@TransactionManagement(BEAN)
public class MySessionBean implements MySession {
    @Resource UserTransaction ut;
    public void method() {
        try {
            ut.begin(); // Открываем транзакцию
            //... какие-то действия
            ut.commit(); // Закрываем транзакцию
        } catch (Exception e) {
            ut.rollback(); // Ошибка – откат транзакции
        }
    }
}
```

# EJB Lite & Full

Feature	EJB Lite	EJB		Feature	EJB Lite	EJB
Stateless beans	+	+		Asynchronous invocation		+
Stateful beans	+	+		Interceptors	+	+
Singleton beans	+	+		Declarative security	+	+
Message driven beans		+		Declarative transactions	+	+
No interfaces	+	+		Programmatic transactions	+	+
Local interfaces	+	+		Timer service		+
Remote interfaces		+		EJB 2.X support		+
Web service interfaces		+		CORBA interoperability		+

# Java Message Service (JMS)

- Позволяет организовать асинхронный обмен сообщениями между компонентами.
- Две модели доставки сообщений — «подписка» (topic) и «очередь» (queue):



# Message-Driven Beans (MDB)

MDB — это компоненты, логика которых является реакцией на события, происходящие в системе.

Особенности MDB:

- Реализуют интерфейс слушателя соответствующего типа сообщений (например, JMS).
- Нет возможности вызова методов «напрямую».
- Нет локальных и удалённых интерфейсов.
- Контейнер может создавать пулы MDB.
- При развёртывании MDB нужно регистрировать в качестве получателей сообщений.

# Пример MDB

```
import javax.ejb.*;
import javax.jms.*;

@MessageDriven(mappedName = "jms/MyQueue",
    activationConfig = {
        @ActivationConfigProperty(propertyName =
            "acknowledgeMode", propertyValue =
            "Auto-acknowledge"),
        @ActivationConfigProperty(propertyName =
            "destinationType", propertyValue =
            "javax.jms.Queue")
    })
public class SampleMDB implements MessageListener {
    public void onMessage(Message message) {
        // Do something
    }
}
```

- Позволяют организовать взаимодействие между сетевыми ресурсами по стандартизированному протоколу.
- Ресурсы могут работать на любой платформе.
- Данные «упаковываются» в XML и передаются по HTTP.
- Основные стандарты — SOAP, WSDL и WS-I.
- На платформе Java EE реализуются JAX-WS API и JAX-RS API.
- Основа SOA (Service Oriented Architecture).

```
package example;

import javax.jws.*;

@WebService
public class SayHello {

    @WebMethod
    public String getGreeting(String name){
        return "Hello " + name;
    }

}
```



# Пример реализации клиента веб-сервиса (JAX-WS)

```
import javax.xml.ws.WebServiceRef;

public class WSTest {

    public WSTest() { }

    public static void main(String[] args) {
        SayHelloService service = new SayHelloService();
        SayHello port = service.getSayHelloPort();
        System.out.println(port.sayHello("Duke"));
    }
}
```

# 16. ORM & JPA

# Объектно-реляционное отображение

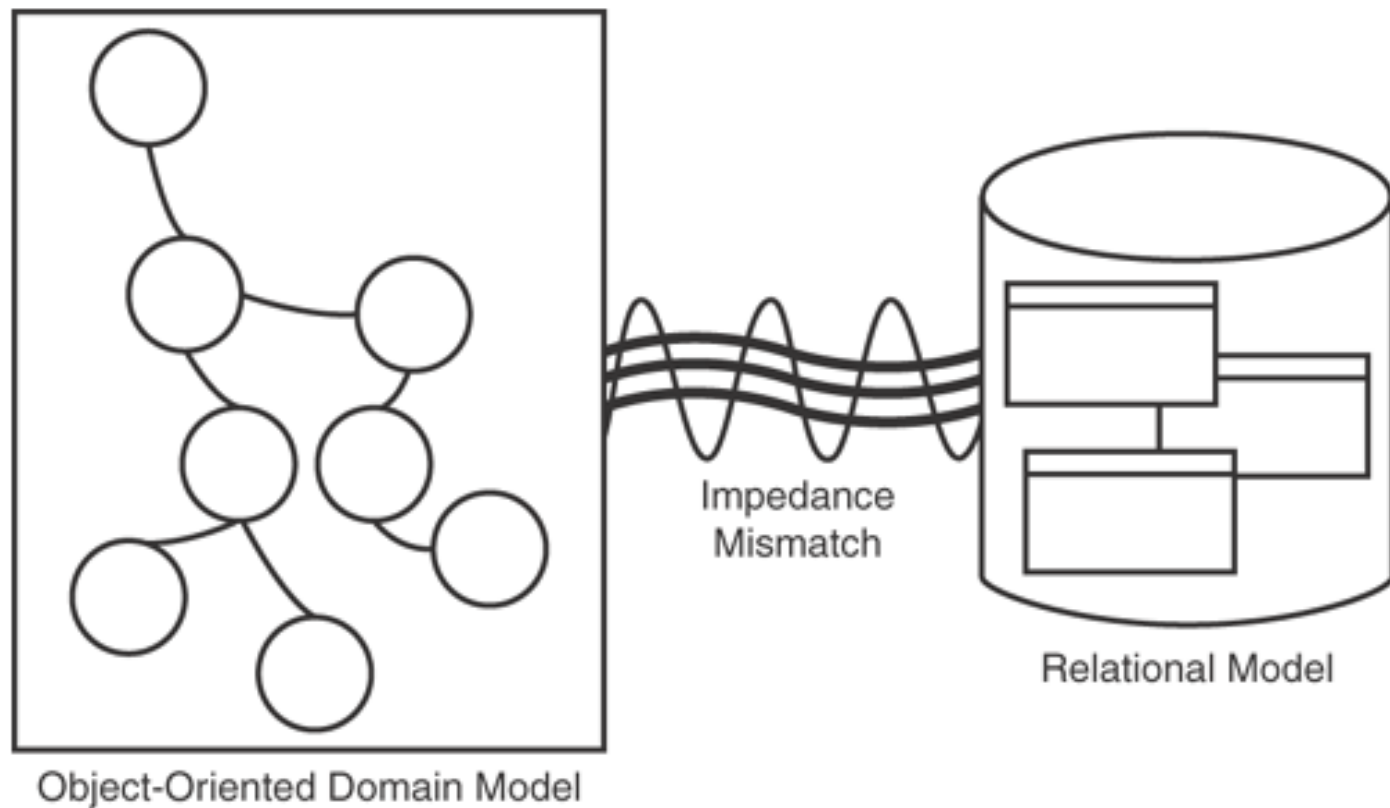
ORM — Object/Relational Mapping — преобразование данных из объектной формы в реляционную и наоборот.



Существует три подхода:

- 1) Top-Down (Сверху-Вниз) – доменная модель приложения определяет реляционную.
- 2) Bottom-up (Снизу-Вверх) – доменная модель строится на основании реляционной схемы.
- 3) Meet-in-the-Middle – параллельная разработка доменной и реляционной моделей с учетом особенностей друг друга.

# Объектно-реляционное несоответствие



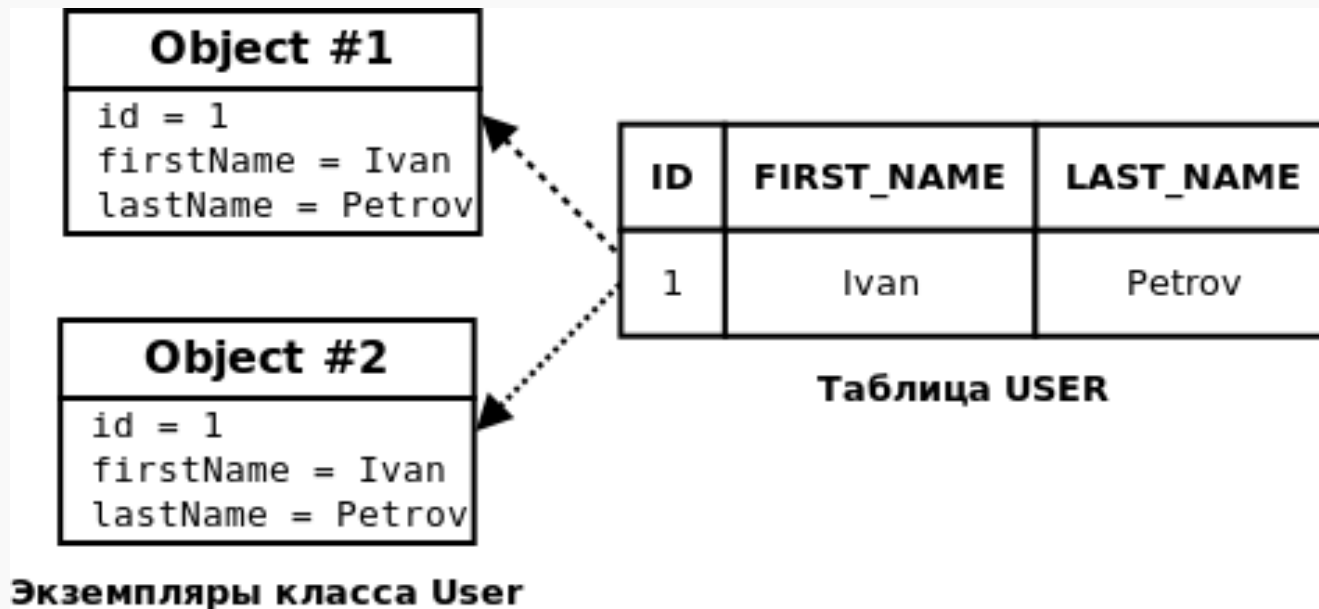
Сложности, возникающие при попытке отобразить один вид представления данных на другой, называют *объектно-реляционным несоответствием* (Object-Relational Impedance Mismatch).

Основные проявления объектно-реляционного несоответствия:

- 1) Проблема идентичности.
- 2) Представление наследования и полиморфизма.
- 3) Проблема навигации между данными.

# Проблема идентичности

Несколько неидентичных объектов представляют одну строку базы данных.



Возможны три варианта:

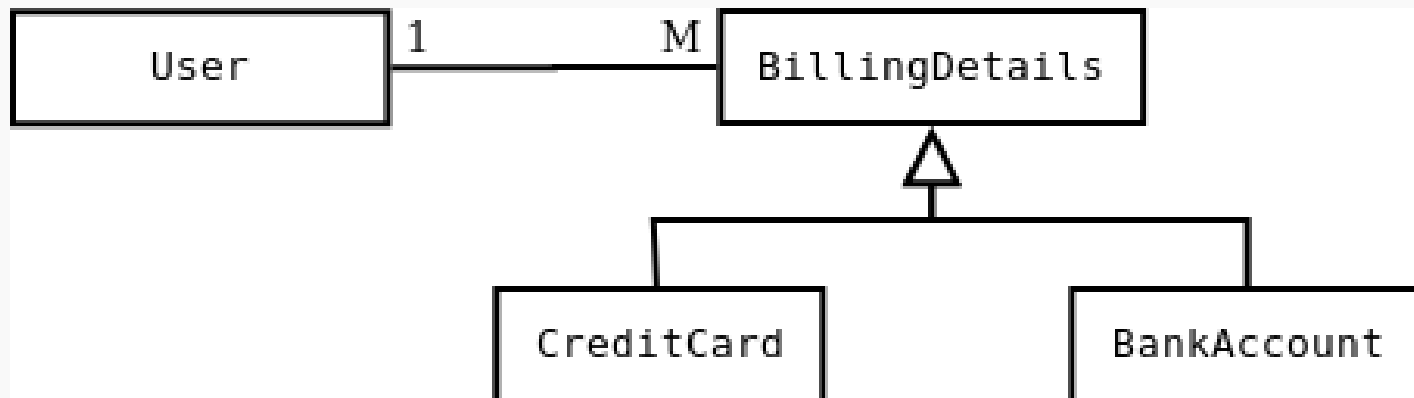
- 1) Обычный уровень хранения без управления идентичностью (no identity scope).
- 2) Уровень хранения с контекстно-управляемой (context-scoped) идентичностью .
- 3) Уровень хранения с жестким управлением идентичностью (process-scoped identity).



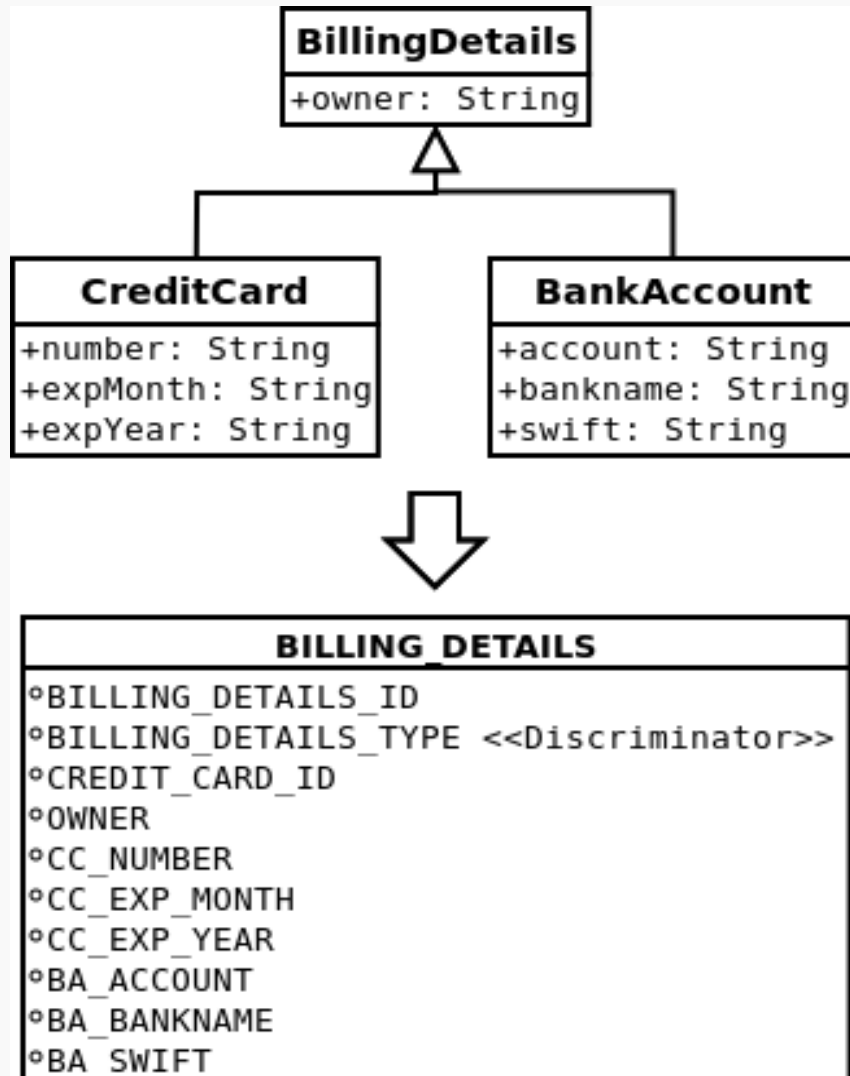
# Наследование и полиморфизм в ORM

При объектно-реляционном отображении наследование и полиморфизм тесно связаны.

Существует три способа реализации наследования в ORM.



# Одна таблица для иерархии классов (Single Table Inheritance Pattern)



Все классы иерархии отображаются на одну таблицу базы данных.

## Достоинства:

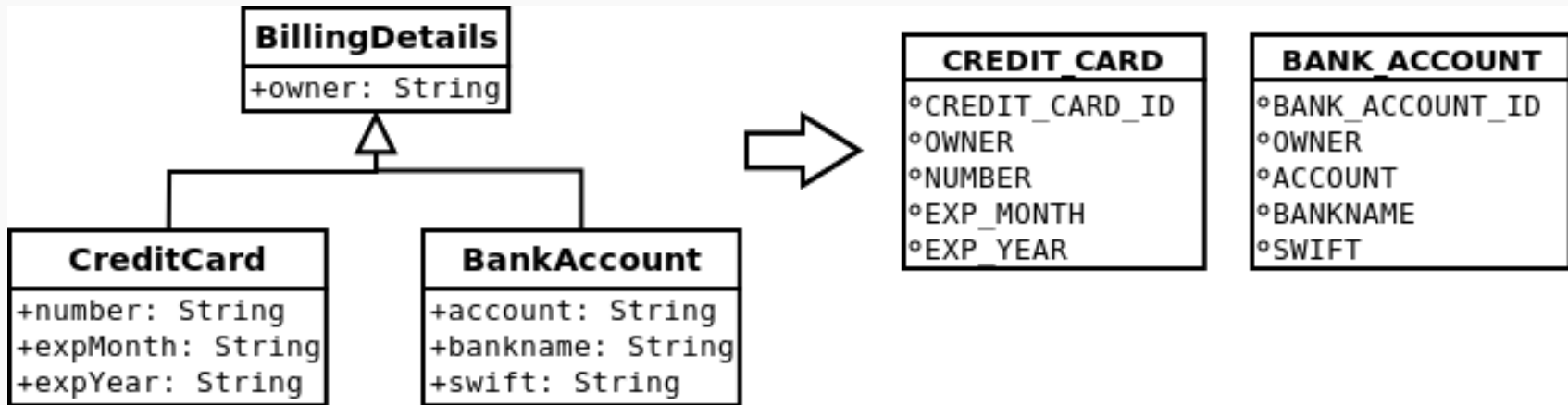
- 1) Наиболее простое решение.
- 2) Наиболее производительное решение.

## Недостатки:

- 3) На поля подклассов нельзя накладывать null-ограничения.
- 4) Полученная таблица не нормализована.

# Своя таблица для конкретного класса (Concrete Table Inheritance Pattern)

Каждый класс в иерархии отображается на отдельную таблицу БД.



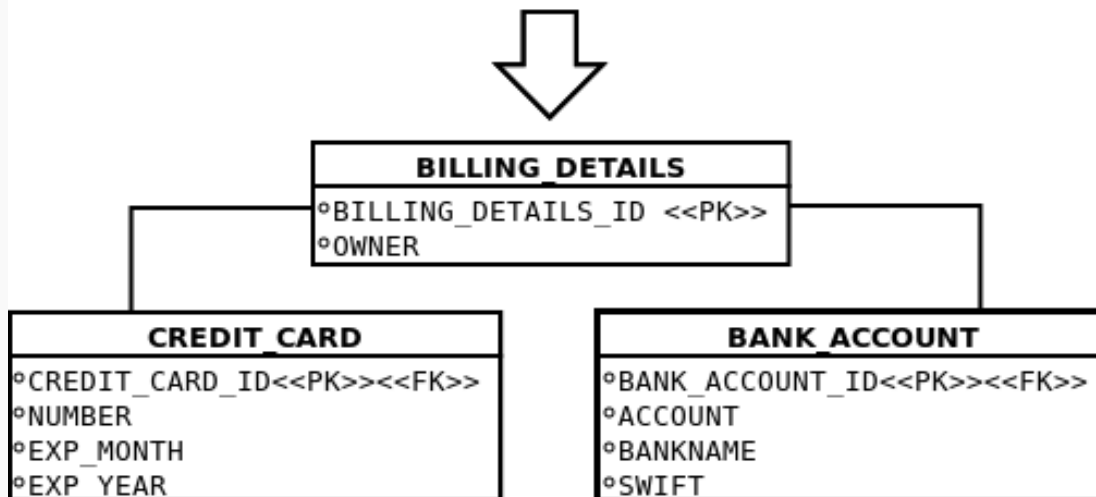
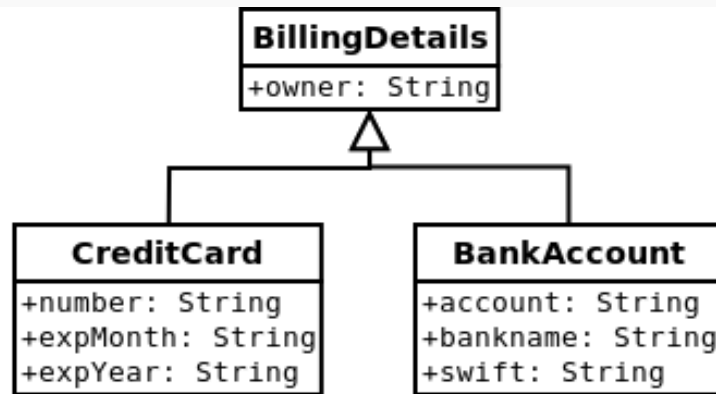
## Достоинства:

Можно накладывать ограничения (not null) на поля подклассов.

## Недостатки:

- 1) Полученные таблицы не нормализованы.
- 2) Плохая поддержка полиморфных запросов.
- 3) Низкая производительность.

# Таблица для каждого подкласса (Class Table Inheritance Pattern)



Каждый подкласс отображается на отдельную таблицу, которая содержит колонки, соответствующие только полям этого подкласса.

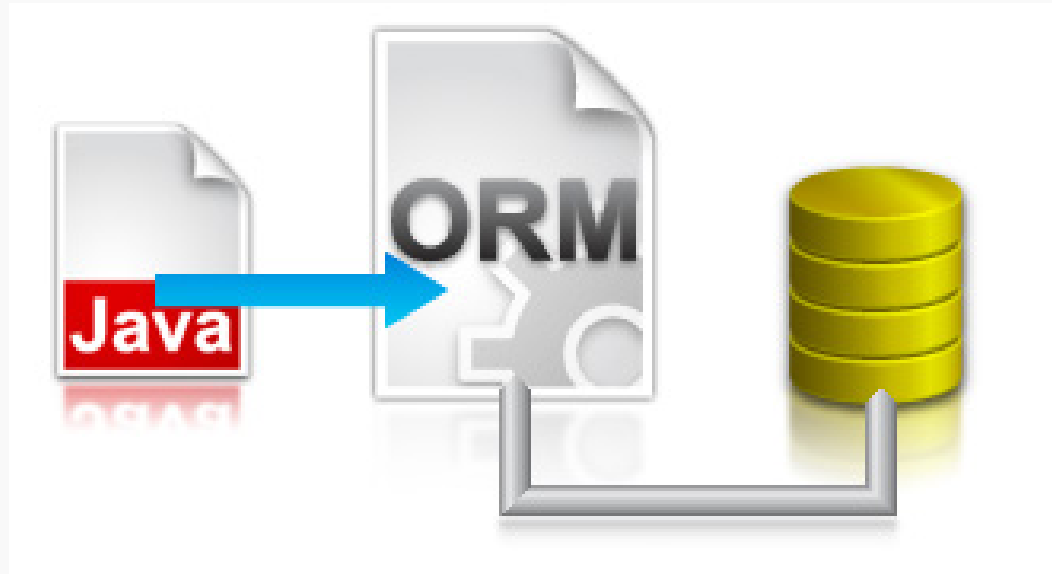
## Достоинства:

- 1) Таблицы нормализованы.
- 2) Лучший вариант для полиморфизма.
- 3) Существует возможность задавать ограничения на поля подклассов.

## Недостатки:

Запросы выполняются медленнее, чем при использовании одной таблицы.

# Как реализовать ORM на Java?



- JDBC;
- ORM-фреймворки (Hibernate, TopLink, ...);
- Java Persistence API (JPA 2.0).

ORM-фреймворк от Red Hat, разрабатывается с 2001 г.

Ключевые особенности:

- Таблицы БД описываются в XML-файле, либо с помощью аннотаций.
- 2 способа написания запросов — HQL и Criteria API.
- Есть возможность написания native SQL запросов.
- Есть возможность интеграции с Apache Lucene для полнотекстового поиска по БД (Hibernate Search).

# Hibernate ORM — пример обращения к БД

```
Session session =
HibernateSessionFactory.getSessionFactory().openSession();

Transaction tx = session.beginTransaction();

Query query = session.createQuery("update ContactEntity " +
                                "set firstName = :nameParam" +
                                ", lastName = :lastNameParam" +
                                ", birthDate = :birthDateParam"+
                                " where firstName = :nameCode");

    query.setParameter("nameCode", "Nick");
    query.setParameter("nameParam", "NickChangedName1");
    query.setParameter("lastNameParam", "LastNameChanged1" );
    query.setParameter("birthDateParam", new Date());

    int result = query.executeUpdate();

tx.commit();
session.close();
```

ORM-фреймворк от Eclipse Foundation.

Ключевые особенности:

- Основан на кодовой базе Oracle TopLink.
- Является эталонной реализацией (reference implementation) для JPA.



# Java Persistence API (JPA)

Java-стандарт (JSR 220, JSR 317), который определяет:

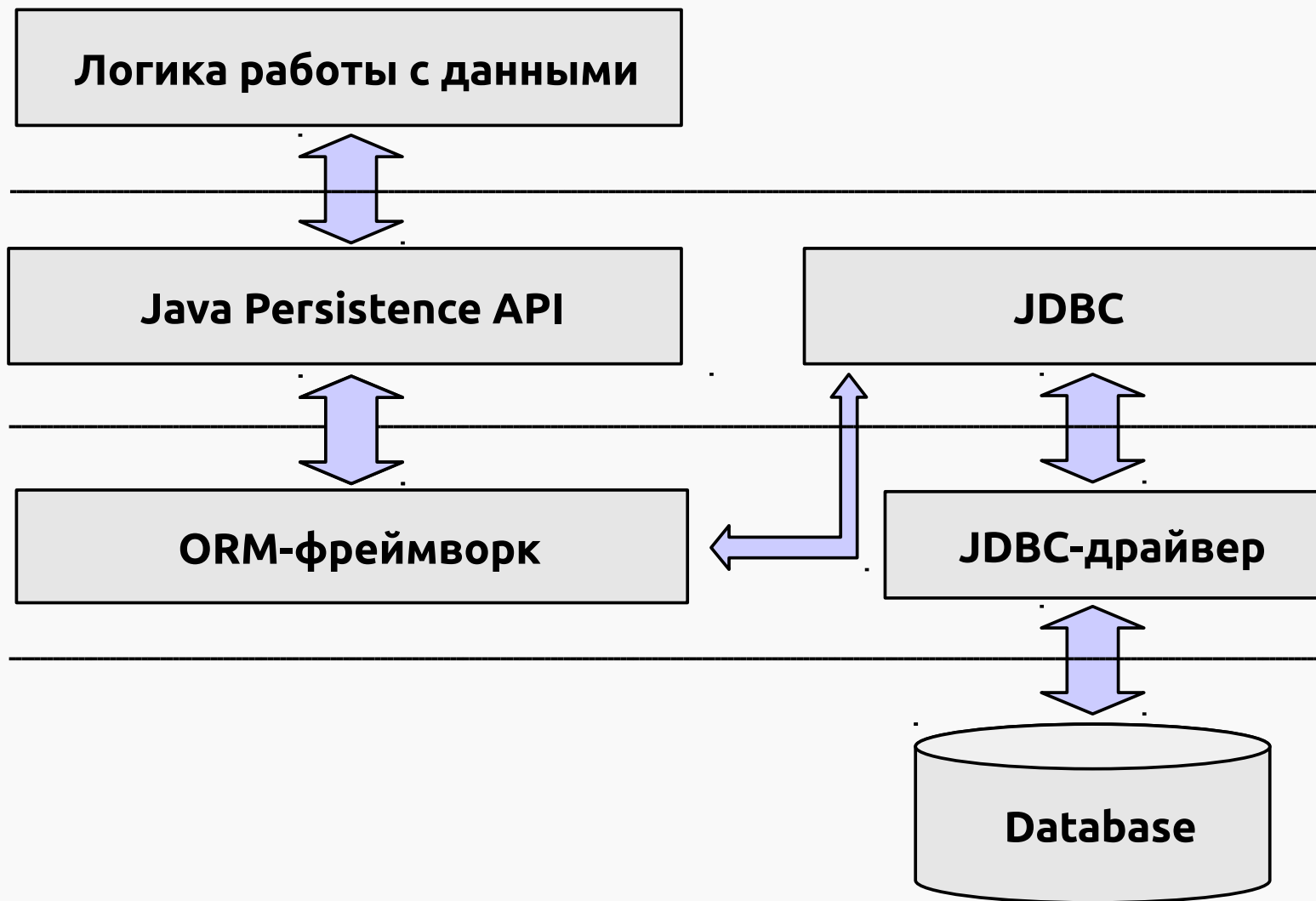
- как Java-объекты хранятся в базе;
- API для работы с хранимыми Java-объектами;
- язык запросов (JPQL);
- возможности использования в различных окружениях.

# Что даёт использование JPA?

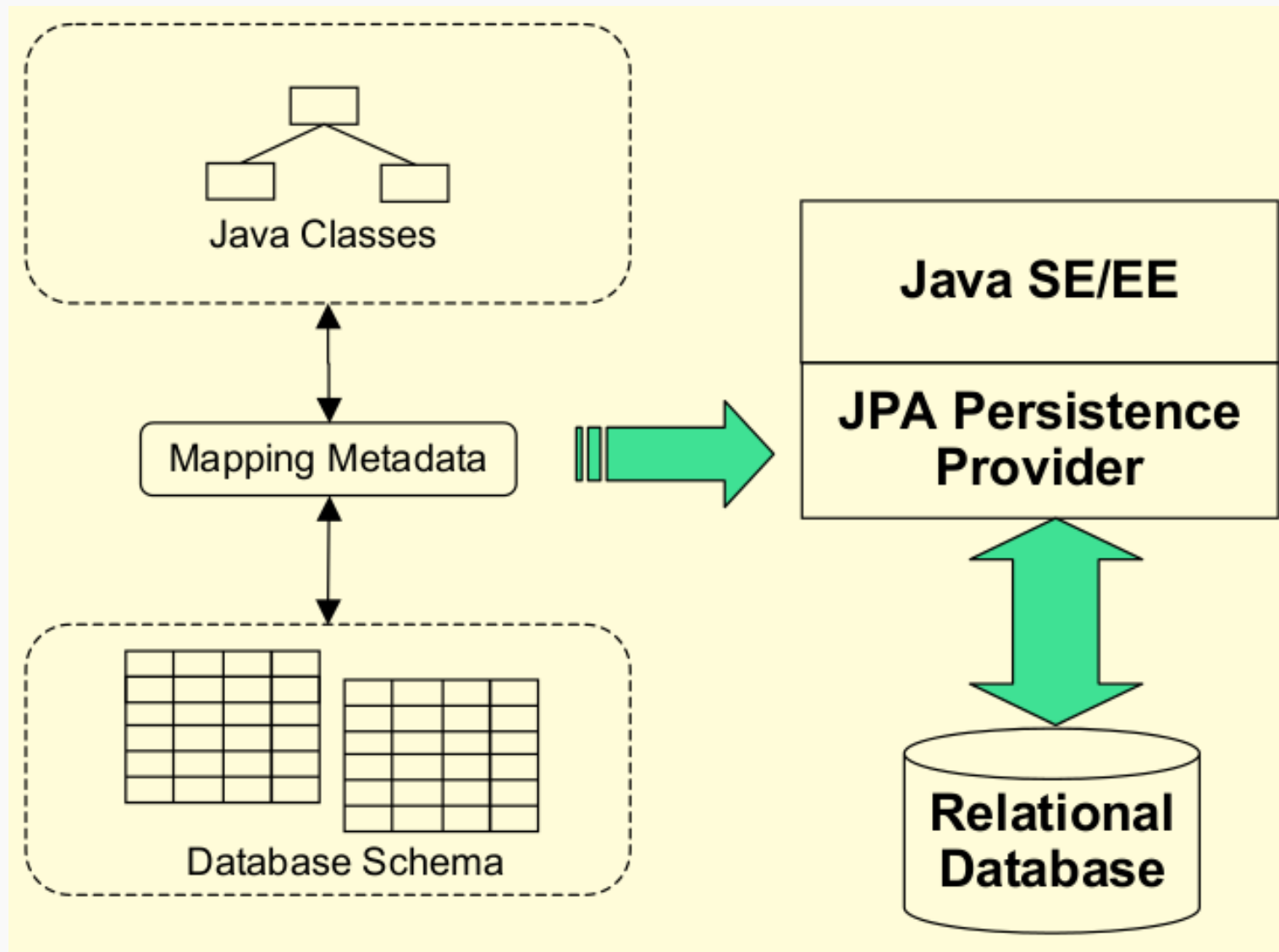


- Достижение лучшей переносимости.
- Упрощение кода.
- Сокращение времени разработки.
- Независимость от ORM-фреймворков.

# Взаимодействие приложения с БД через JPA



# Что нужно для отображения?

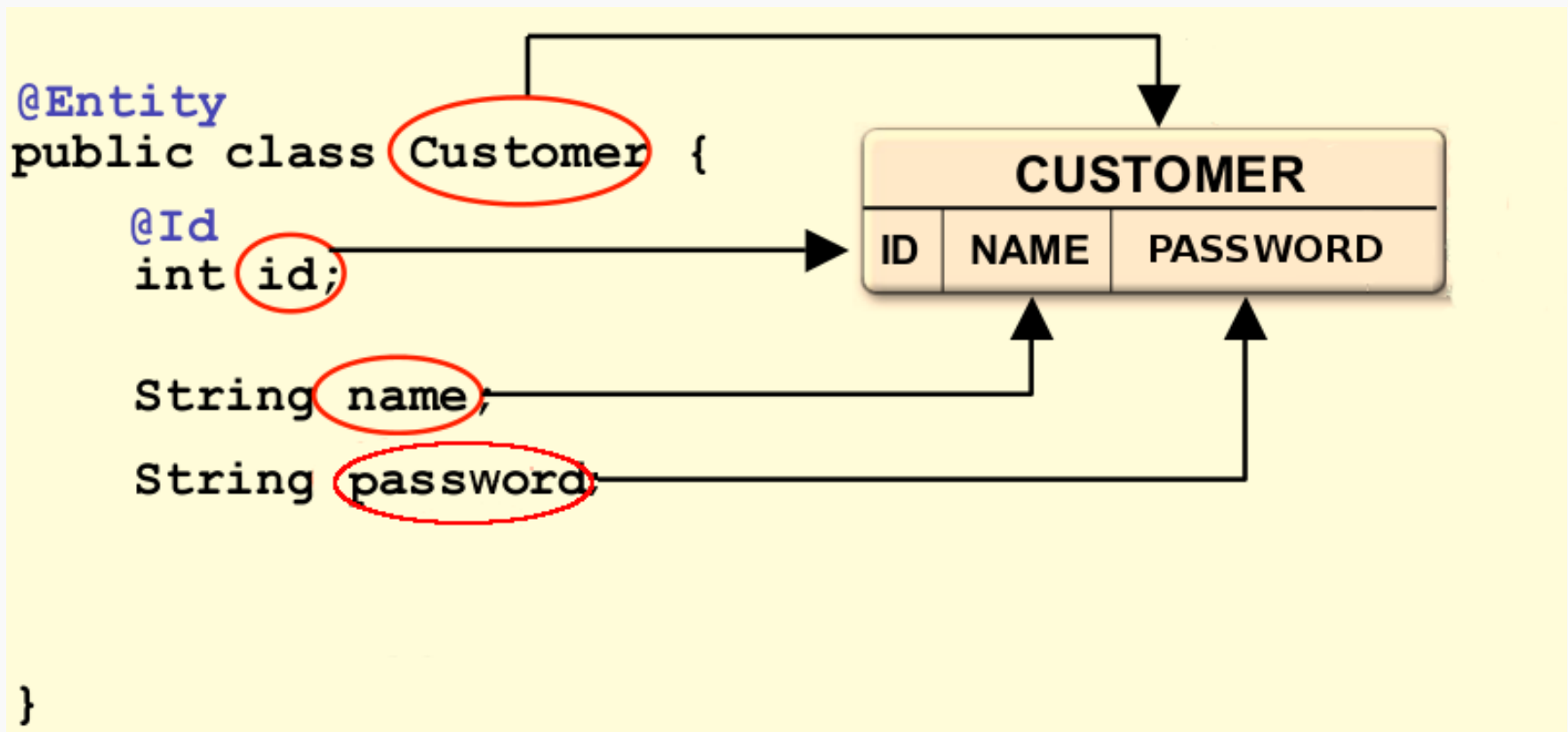


Entity — простой Java-класс (POJO), удовлетворяющий следующим требованиям:

- Не должен быть внутренним (inner).
- Не должен быть final.
- Не должен иметь final методов.
- Должен иметь public-конструктор без аргументов.
- Атрибуты класса не должны быть public.

# Entity (annotation mapping)

В классе должен быть объявлен идентификатор:



# Идентичность

- Идентификатор (id) сущности, первичный ключ в БД.
- Уникально определяет сущность в памяти и БД.

Uses  
PK  
class

1. Simple id

`@Id int id;`

2. Compound id

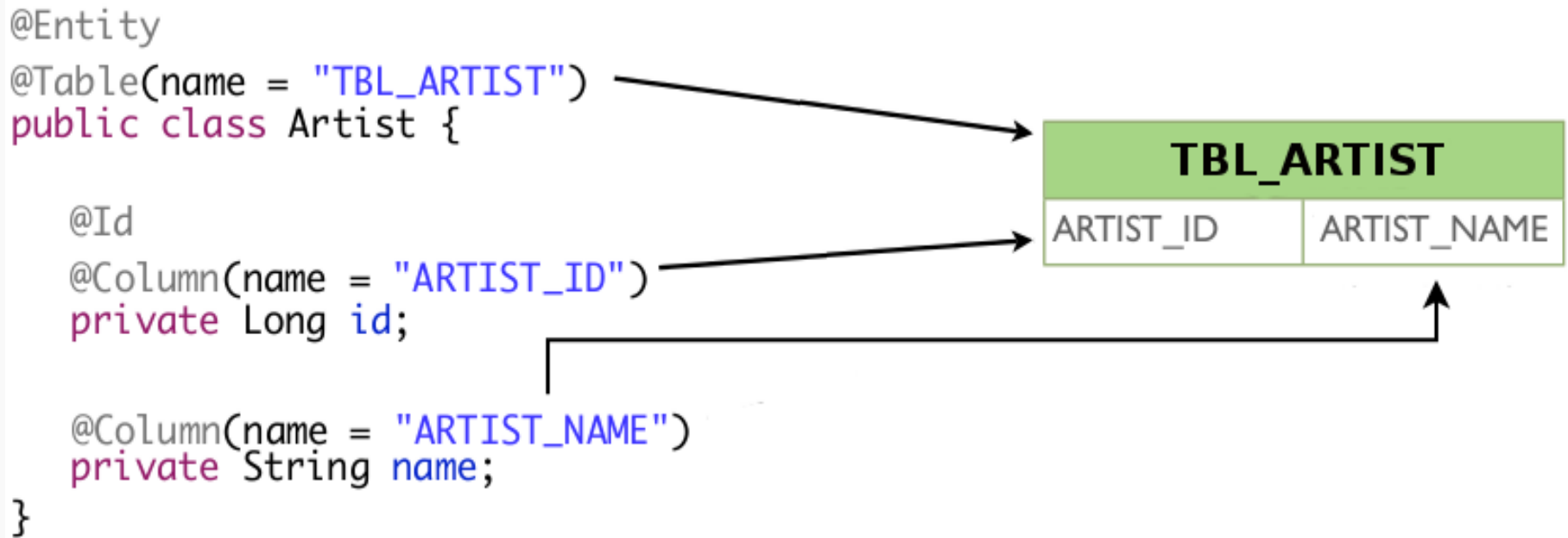
`@Id int id;`  
`@Id String name;`

3. Embedded id

`@EmbeddedId EmployeePK id;`

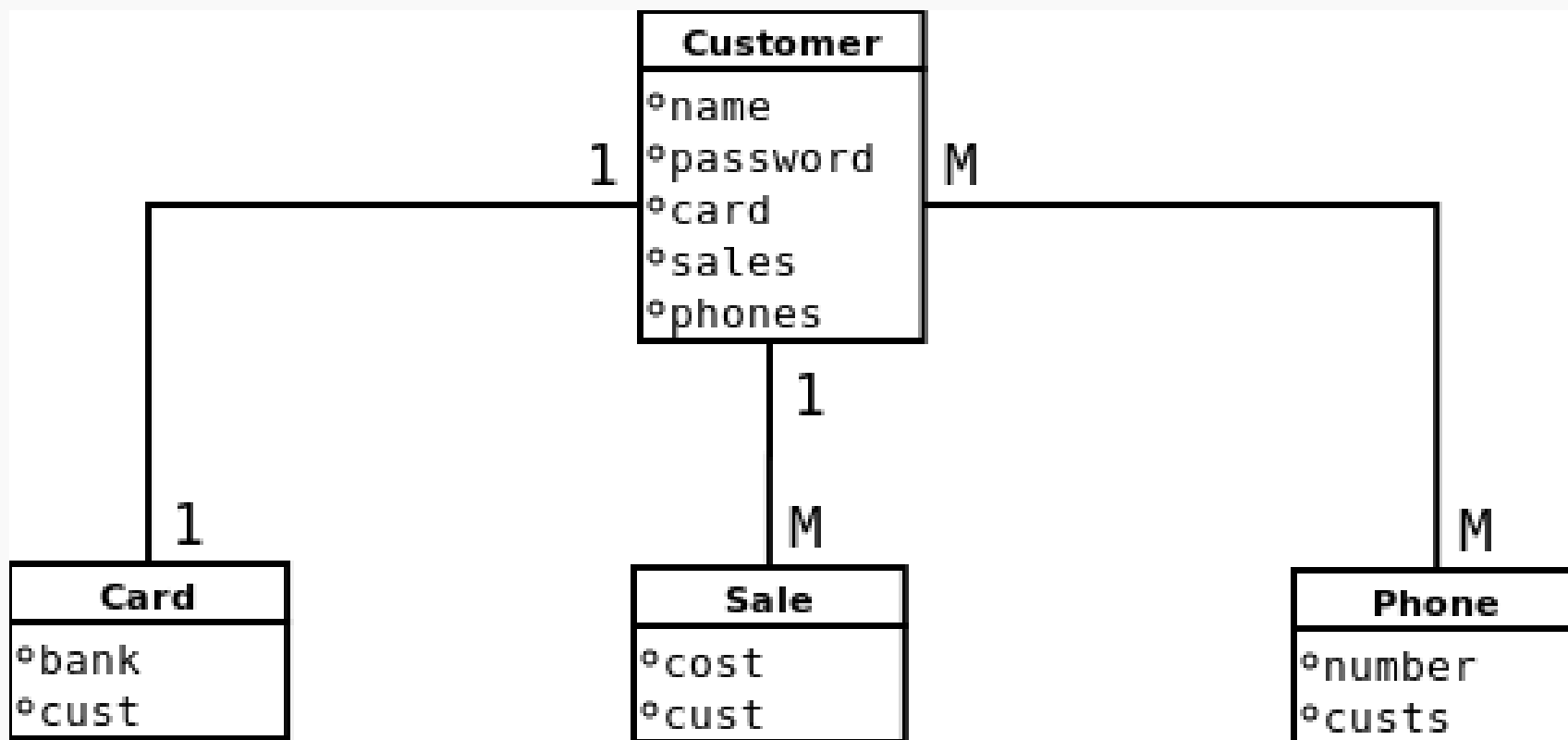
# Аннотации @Table и @Column

- Используются, чтобы определить отображаемые имена элементов.
- @Table применяется на уровне отображения класса на таблицу.
- @Column применяется на уровне отображения полей класса на столбцы таблицы.





# Доменная модель



# Связи (Relationships)

- JPA поддерживает все стандартные виды связей:

- ▶ One-To-One
- ▶ One-To-Many
- ▶ Many-To-One
- ▶ Many-To-Many

- Поддерживаются однонаправленные и двунаправленные связи.

Базовый интерфейс для работы с хранимыми данными:

- Обеспечивает взаимодействие с Persistence Context.
- Можно получить через EntityManagerFactory.
- Обеспечивает базовые операции для работы с данными (CRUD).

# Методы EntityManager

- ▶ `<T> T find(Class<T> entityClass, Object primaryKey)`
- ▶ `<T> T getReference(Class<T> entityClass, Object primaryKey)`
- ▶ `void persist(Object entity)`
- ▶ `<T> T merge(T entity)`
- ▶ `refresh(Object entity)`
- ▶ `remove(Object entity)`
- ▶ `void flush()`
- ▶ `void close();`

# persist()

- Добавляет новый экземпляр Entity в БД.
- Сохраняет состояние Entity и относящихся к ней ссылок.
- Делает экземпляр Entity управляемым РС.

```
public Customer createCustomer(int id, String name) {  
    Customer cust = new Customer(id, name);  
    entityManager.persist(cust);  
    return cust;  
}
```

# find() & remove()

- find() — получает управляемый экземпляр Entity (по идентификатору) – возвращает null, если заданный объект не найден.
- remove() — удаляет управляемую Entity. Опционально производит каскадное удаление отмеченных объектов.

```
public void removeCustomer(Long custId) {  
    Customer cust =  
        entityManager.find(Customer.class, custId);  
    entityManager.remove(cust);  
}
```

# merge()

Создается управляемая копия переданной отсоединенной (detached) Entity. Поддерживает каскадное распространение.

```
public Customer storeUpdatedCustomer(Customer cust)
    return entityManager.merge(cust);
}
```

- Поддерживаются статические и динамические запросы.
- Запросы могут быть написаны на SQL или JPQL.
- Поддерживается передача именованных и позиционных параметров.
- Поддерживается eager-доступ при использовании fetch.



Для создания запроса:

- `createQuery();`
- `createNamedQuery();`
- `createNativeQuery();`

Для получения результата:

- `getSingleResult();`
- `getResultList();`

Характерные черты:

- расширение EJB QL;
- SQL-подобный синтаксис;
- в запросах указываются объекты/свойства вместо таблиц/колонок;
- поддерживаются подзапросы.

# Criteria API (JPA 2.0)

Характерные черты:

- Объектно-ориентированный API для построения запросов.
- Есть возможность отобразить любой JPQL-запрос в Criteria.
- Поддерживает построение запросов в runtime.

# Создание Criteria Query

- Необходимо указать сущности, участвующие в запросе (query roots).
- Условие запроса задается через `where(Predicate p)`, где аргумент устанавливает необходимые ограничения.
- Метод `select()` определяет, что мы получим в результате запроса.

```
Root customer = qdef.from(Customer.class);  
qdef.select(customer).where(queryBuilder  
    .equal(customer.get("customerInfo"), ci));
```

# 17. Spring Framework

# Spring Framework

См. отдельную  
презентацию.



# 18.GWT & Vaadin

- By Google.
- На нём сделан Gmail! (нет).
- Проектируем веб как десктоп.
- Весь код пишется на Java — можно попытаться не верстать.
- Если всё-таки нужно верстать — это можно сделать, но довольно больно.



# Пример приложения на GWT

```
public class GwtApp implements EntryPoint {
    final Button confirmButton = new Button("Confirm");
    final TextBox nameField = new TextBox();
    final Label errorLabel = new Label();
    final Label helloLabel = new Label();

    VerticalPanel dialogVPanel = new VerticalPanel();
    final DialogBox dialogBox = new DialogBox();
    final HTML serverResponseHtml = new HTML();
    final Label sendToServerLabel = new Label();
    final Button closeButton = new Button("Close");

    private final GwtAppServiceIntfAsync gwtAppService = GWT.create(GwtAppServiceIntf.class);

    /** This is the entry point method.*/
    public void onModuleLoad() {

        helloLabel.setText("GwtApp Application hello world");
        final Label usernameLabel = new Label();
        usernameLabel.setText("Username: ");
        /*Связываем id=" на html странице с компонентами */
        RootPanel.get("hellold").add(helloLabel);

        RootPanel.get("usernameLabelId").add(usernameLabel);
        RootPanel.get("usernameId").add(nameField);

        RootPanel.get("confirmButtonId").add(confirmButton);
        RootPanel.get("errorLabelContainer").add(errorLabel);

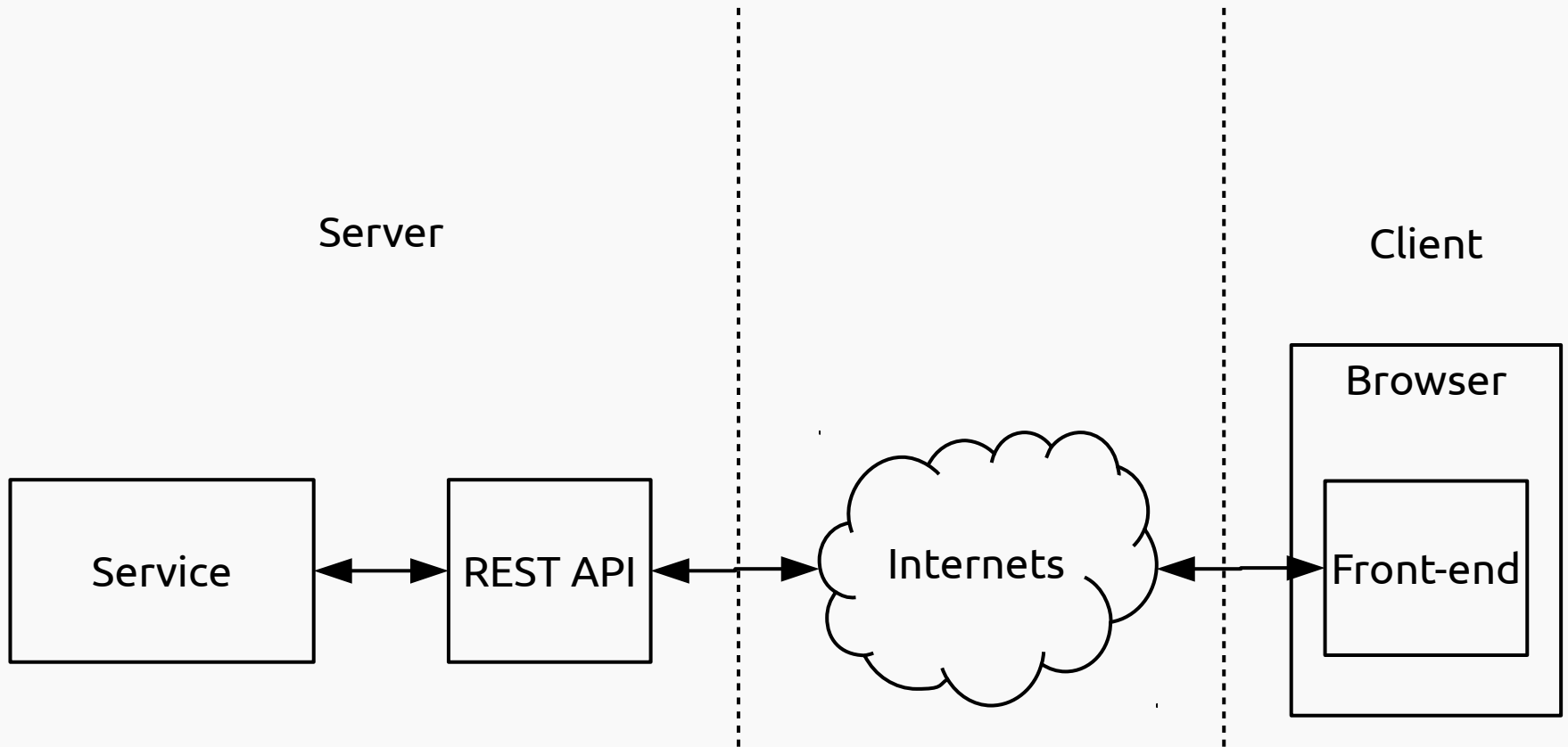
        // Create the popup dialog box
        dialogBox.setText("Remote procedure call from server");
        dialogBox.setAnimationEnabled(true);

        (...)
    }
}
```

- Финский фреймворк на базе gwt.

# 19. JS Frontend Frameworks

# Common Practice



# ReactJS + навороты

См. отдельную презентацию.

- OpenSource-фреймворк.
- Первая версия выпущена в 2009 г.
- Предназначен для разработки SPA.
- Интерфейс строится из компонентов.
- Компоненты могут рекурсивно вкладываться друг в друга.
- Компоненты могут объединяться в библиотеки.

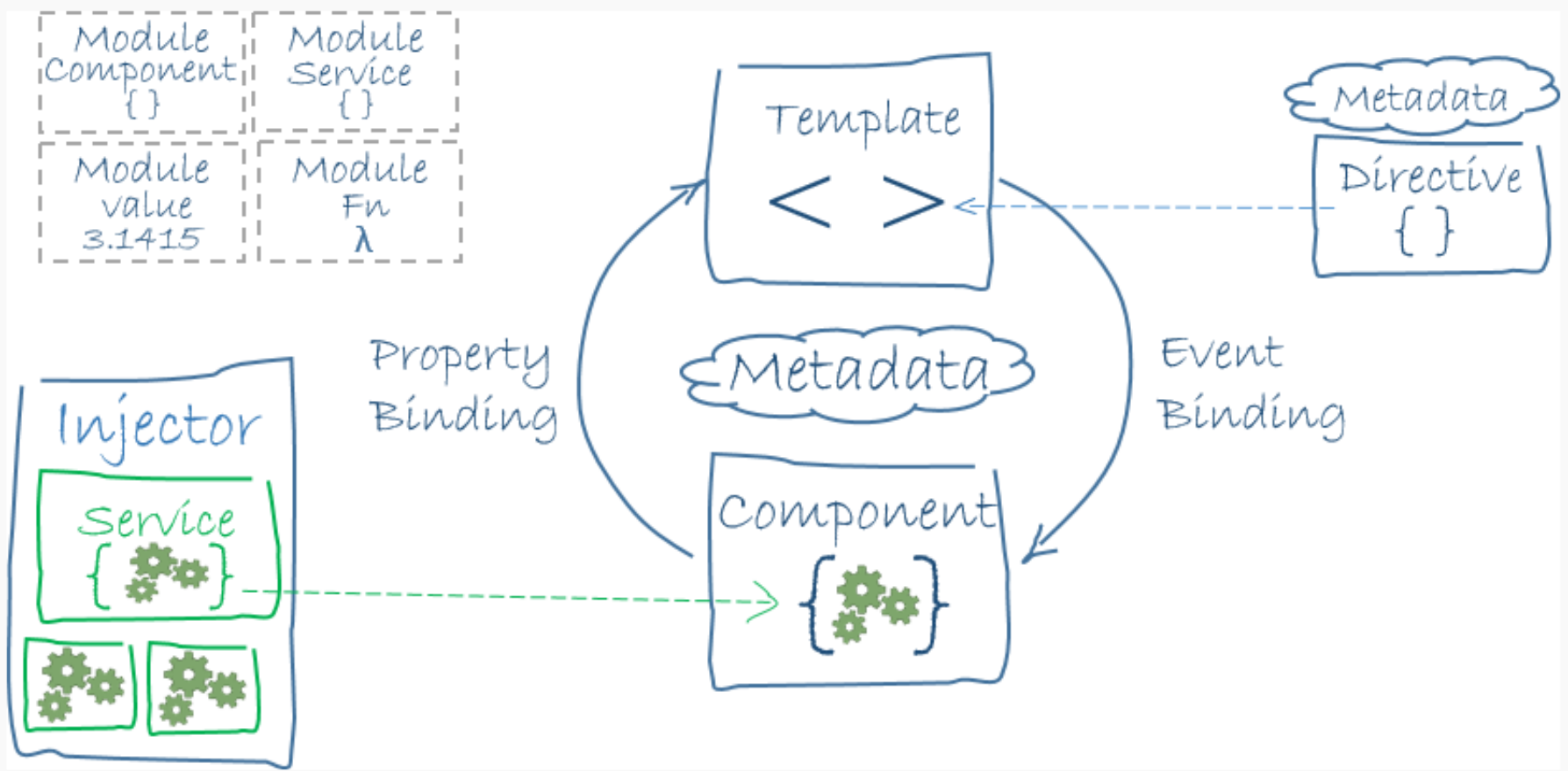
# Компоненты Angular

Состоят из шаблона HTML-разметки и класса на JS / TypeScript, управляющего поведением этого блока интерфейса.

```
-----  
import { Component } from '@angular/core';  
  
@Component({  
  selector: 'my-app',  
  template: `<h1>Hello {{name}}</h1>`  
})  
  
export class AppComponent { name = 'Angular'; }  
  
-----  
  
// index.html  
<my-app>Loading AppComponent content here...</my-app>
```



# Архитектура Angular





- Очередное «изобретение» MVC.
- Backbone.Model — данные + логика их обработки:

```
var Sidebar = Backbone.Model.extend({
  promptColor: function() {
    var cssColor = prompt("Пожалуйста, введите CSS-цвет:");
    this.set({color: cssColor});
  }
});

window.sidebar = new Sidebar;
sidebar.on('change:color', function(model, color) {
  $('#sidebar').css({background: color});
});

sidebar.set({color: 'white'});
sidebar.promptColor();
```

# Backbone.js (продолжение)

- Backbone.View — представление:

```
var DocumentRow = Backbone.View.extend({
  tagName: "li",
  className: "document-row",
  events: {
    "click .icon": "open",
    "click .button.edit": "openEditDialog",
    "click .button.delete": "destroy"
  },
  initialize: function() {
    this.listenTo(this.model, "change", this.render);
  },
  render: function() {
    ...
  }
});
```