# Corso Java

per

# Credit Agricole Group Solutions

Data: 28/09/2022

Docente: Alessio Viticchié

Reti

Reti Academy
Formazione. Esperienza. Eccellenza.

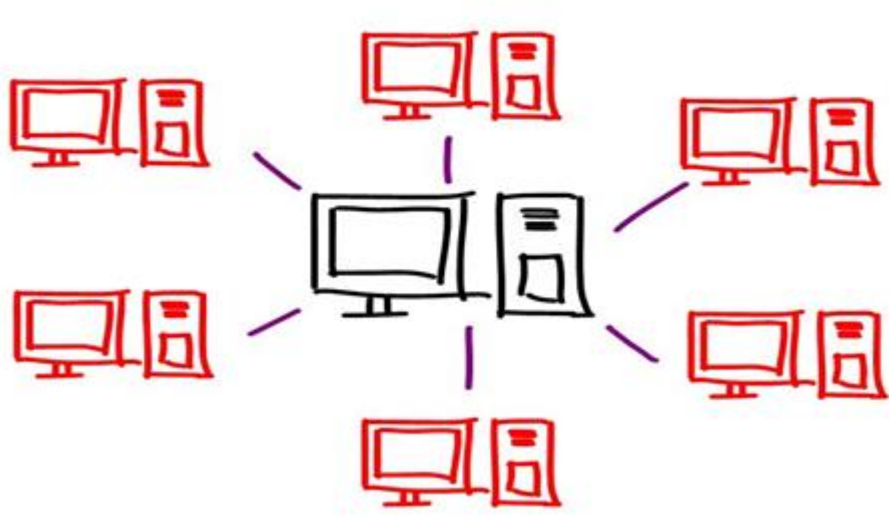
## Module 4

- Introduction to Web Applications
- JDBC: Java and relational databases
- Deploy of a Web Application on Tomcat Web Server

# Introduction to Web Applications

## Distributed Systems

- A distributed system is a system whose components are located on different networked computers, which communicate and coordinate their actions by passing messages to one another.



## Why Distributed Systems?

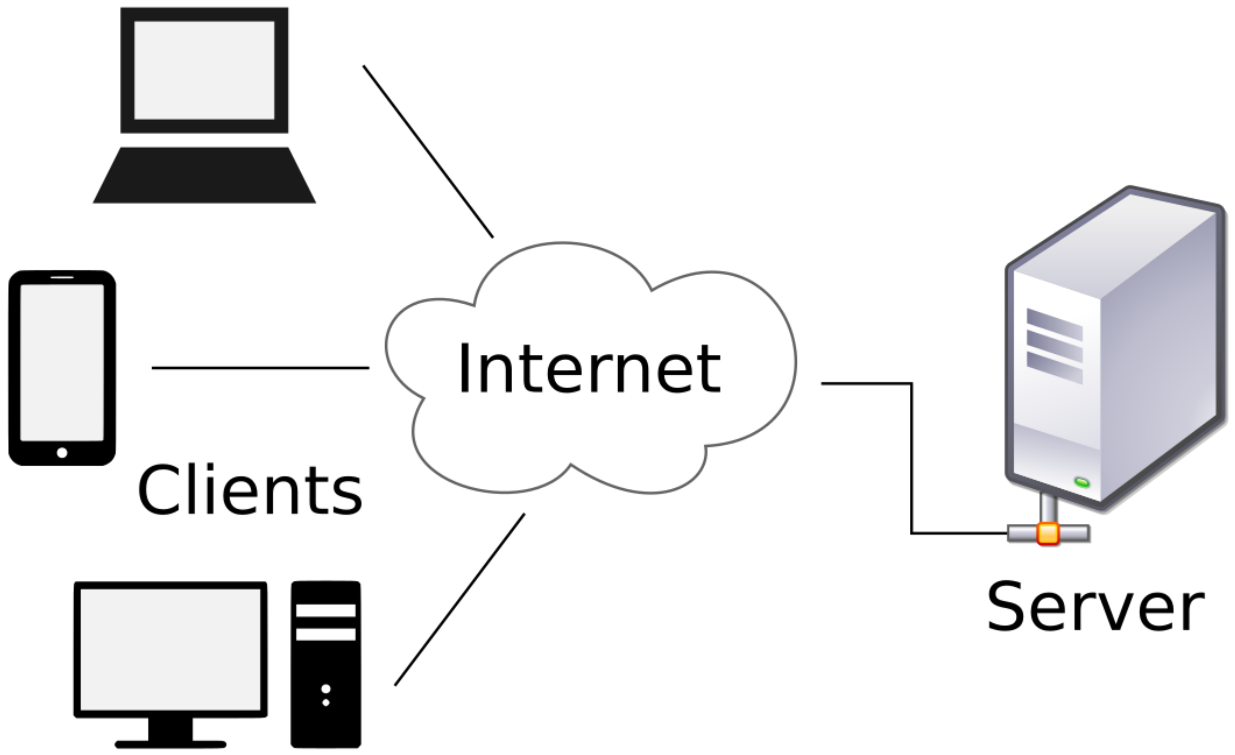
- ❑ Resource sharing
- ❑ Higher performance
- ❑ Flexible
- ❑ Reliable
- ❑ Scalable
- ❑ Cheaper
- ❑ Manageable (it depends)

## Characteristics of Distributed Systems

- ❑ Complex programming
- ❑ Heterogeneous (Hardware, OS, networking behavior, object model, programming language)
- ❑ Inherent problems
    - Network latency
    - Concurrency issues
    - Partial Failure

## Distributed Systems – Client/Server

- In Java, one of the most used paradigms is the Client/Server



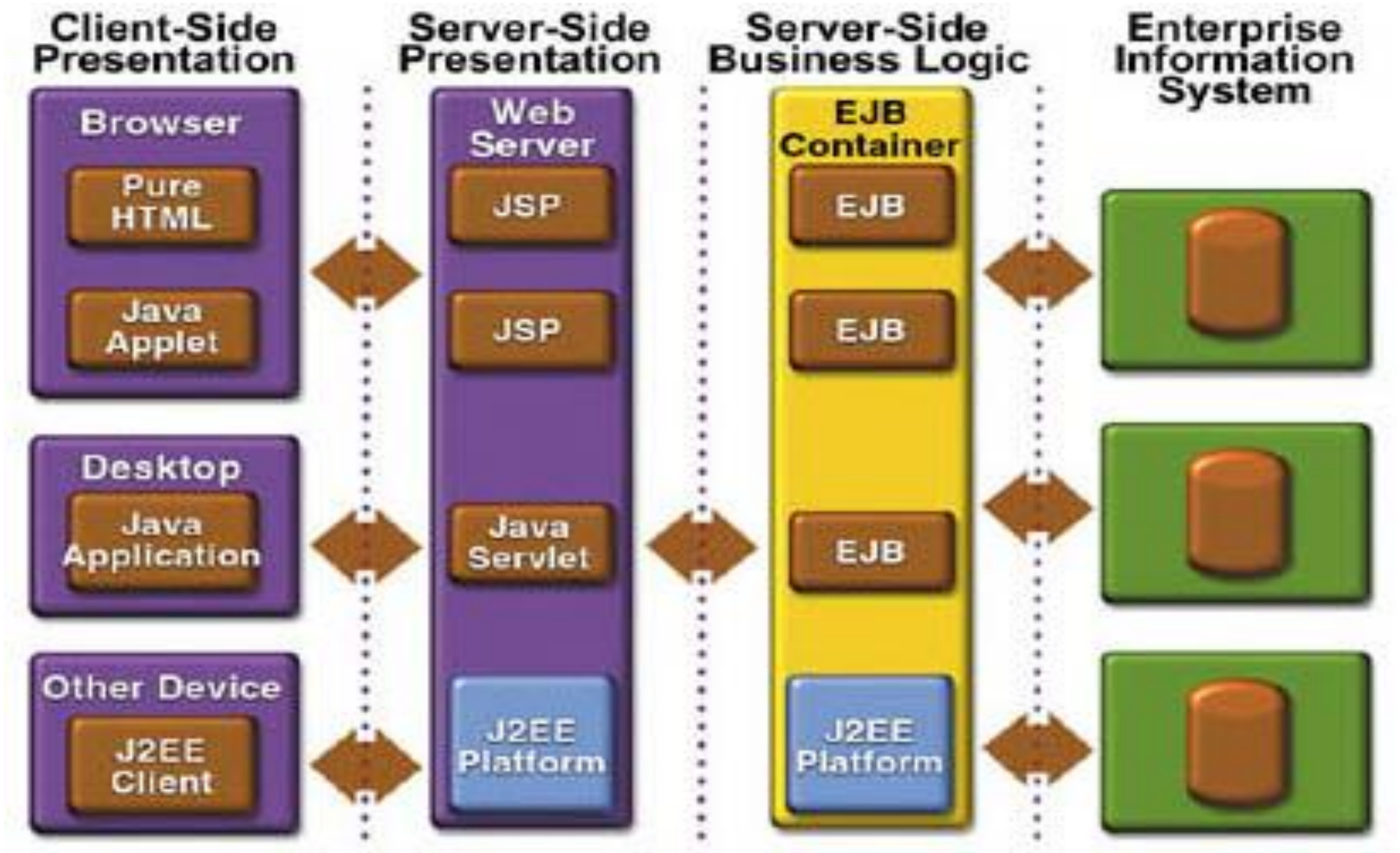
# Java Distributions used on the Web

- **Applet** : Java application downloaded from a server but executed on browser. Deprecated since Java 8

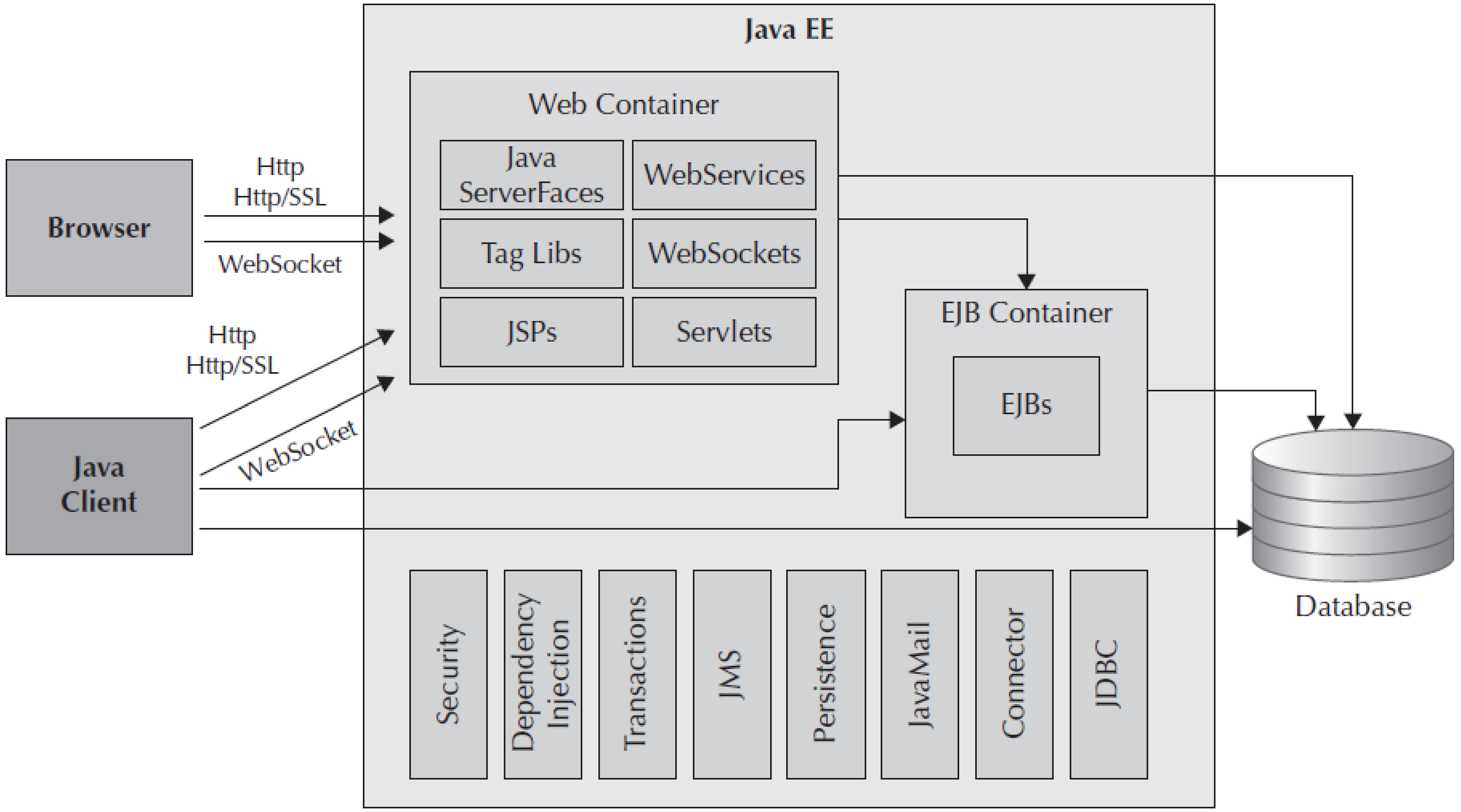
- **JNLP** : An evolution for applets. Java application downloaded from a server and installed on client. It allows to have richer apps than applets. Deprecated since Java 11 -> Use of Client apps

- **Web Application** : Java code executed on server. It allows clients to get information generally by HTTP protocol.

# Java Web Application Architecture



# Java EE Architecture



## Web Maintenance

The maintenance of a web application could be complicated:

- ❑ Multiple Technologies:
    - Client side: Angular, HTML, XML, Javascript, …
    - Server side: Web server, Application Server, Java, Database, Legacy Application
- ❑ Service has to be Just in time
- ❑ Performance
- ❑ High availability
- ❑ Fault tolerance

## HTTP Protocol

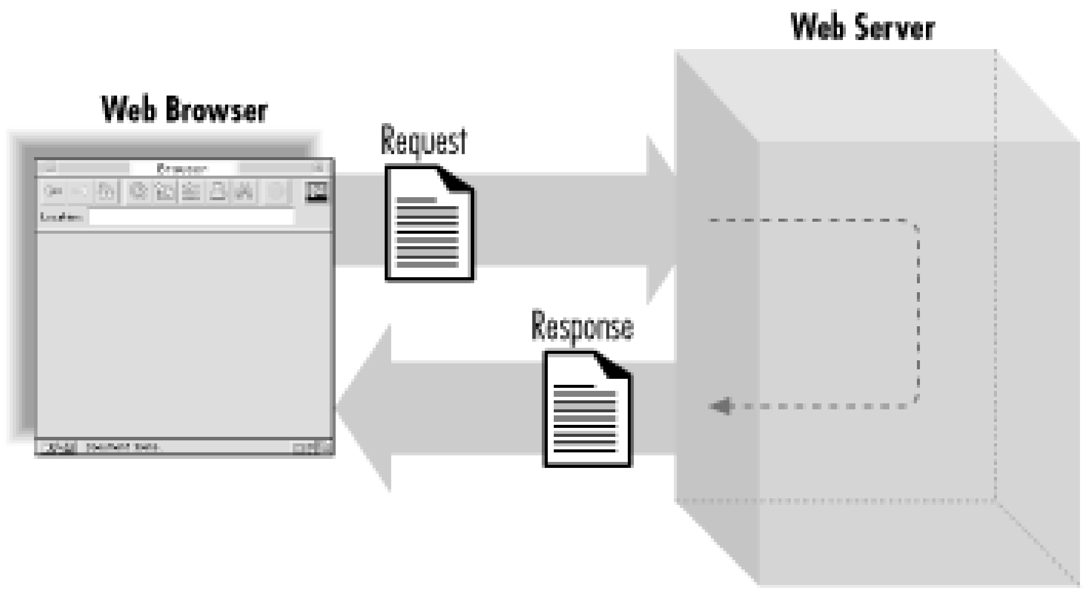
- HTTP protocol is **connectionless** and **stateless**: after the server answer to the client, the connection is closed and will not be reused. An HTTP server handle each request as a new one.

- Request method: HTTP allows different methods to make a request (GET, HEAD, POST, PUT, DELETE).

- *http://host:port/query?part=123* is an example of URL.

## HTTP Request Example

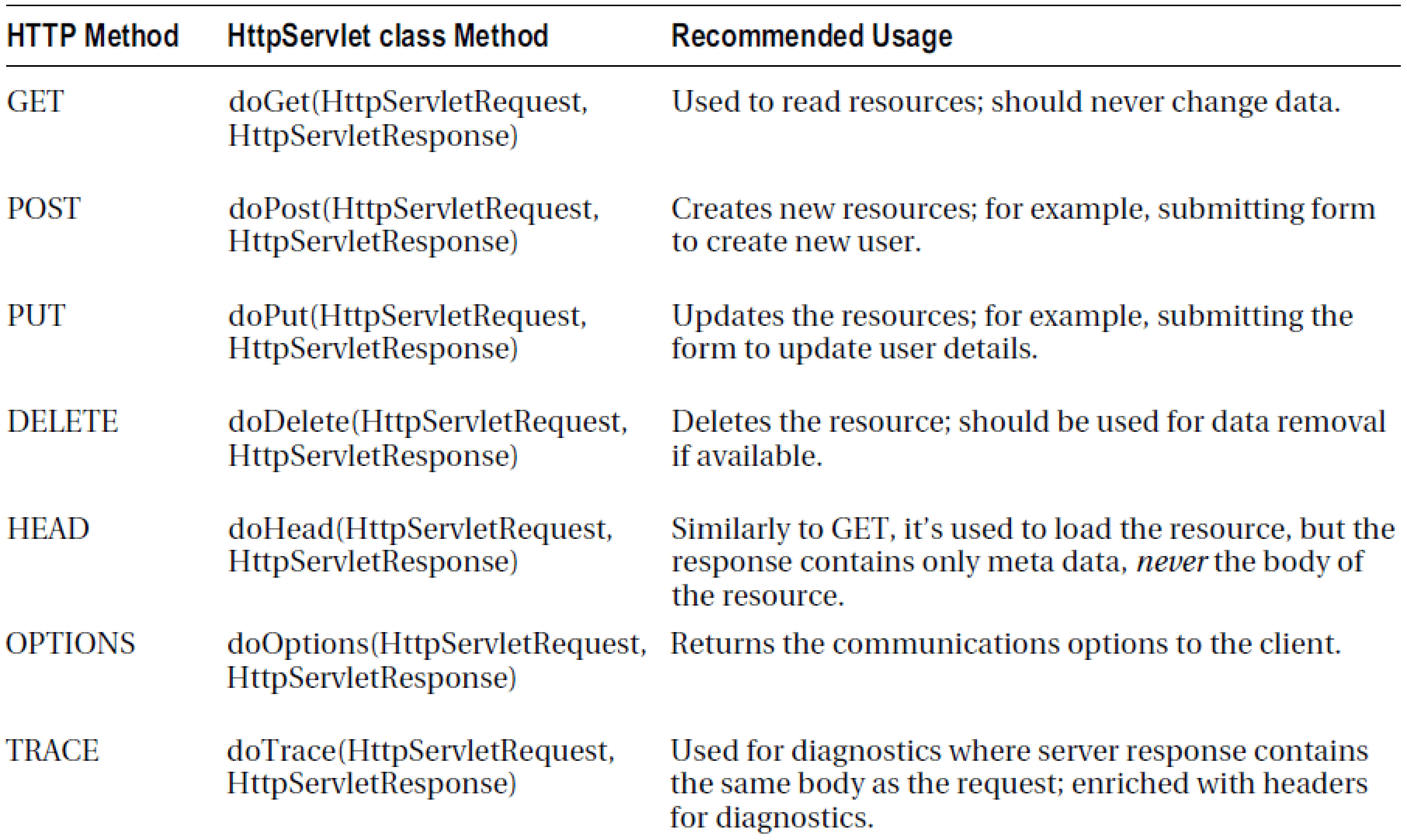
```
GET /index.html HTTP/1.1
Host: localhost
Accept: image/gif, image/x-xbitmap,
image/jpeg, image/pjpeg, image/xbm, */*
Accept-Language: en
Connection: Keep-Alive
User-Agent: Mozilla/4.0 (compatible;
MSIE 4.5; Mac_PowerPC)
Content-type:application/x….
Content-length
Org=blavla
&user=10000
```



## HTTP Response Example

```
HTTP/1.1 200 OK
Date: Sat, 18 Mar 2000 20:35:35 GMT
Server: Apache/1.3.9 (Unix)
Last-Modified: Wed, 20 May 1998 14:59:42 GMT
ETag: "74916-656-3562efde"
Content-Length: 141
Content-Type: text/html

<HTML>
<HEAD><TITLE>Sample Document</TITLE></HEAD>
<BODY>
  <H1>Sample Document</H1>
  <P>This is a sample HTML document!</P>
</BODY>
</HTML>
```

# HTTP Methods



| HTTP Method | HttpServlet class Method | Recommended Usage |
|---|---|---|
| GET | doGet(HttpServletRequest, HttpServletResponse) | Used to read resources; should never change data. |
| POST | doPost(HttpServletRequest, HttpServletResponse) | Creates new resources; for example, submitting form to create new user. |
| PUT | doPut(HttpServletRequest, HttpServletResponse) | Updates the resources; for example, submitting the form to update user details. |
| DELETE | doDelete(HttpServletRequest, HttpServletResponse) | Deletes the resource; should be used for data removal if available. |
| HEAD | doHead(HttpServletRequest, HttpServletResponse) | Similarly to GET, it's used to load the resource, but the response contains only meta data, *never* the body of the resource. |
| OPTIONS | doOptions(HttpServletRequest, HttpServletResponse) | Returns the communications options to the client. |
| TRACE | doTrace(HttpServletRequest, HttpServletResponse) | Used for diagnostics where server response contains the same body as the request; enriched with headers for diagnostics. |

## HTTP Response Codes

- ❑ **Informational (1xx)** : 100 (Continue)
- ❑ **Successful (2xx)** : 200 (OK)
- ❑ **Redirection (3xx)** : 305 (Use Proxy)
- ❑ **Client error (4xx)**: 400 (Bad Request) or 404 (Not Found)
- ❑ **Server Error (5xx)**: 500 (Internal Server Error)

## Web Container

- ❑ A web container is responsible for managing the lifecycle of servlets, mapping a URL to a particular servlet and ensuring that the URL requester has the correct access-rights.
- ❑ A web container implements the web component contract of the Jakarta EE architecture. This architecture specifies a runtime environment for additional web components, including security, concurrency, lifecycle management, transaction, deployment, and other services.
- ❑ Apache Tomcat, JBoss Server, Glassfish are some of the most famous applications that implements a web container.

# Java Web Components

- ❑ A Java web component is a server-side object used by a Web-based client (browsers) to interact with J2EE applications. Web components come in two types:
- ❑ **Java Servlet**: A server-side Web component used to process requests and construct responses.
- ❑ **JavaServer Pages (JSP)**: Used to create dynamic Web content and server/platform-independent Web-based applications.

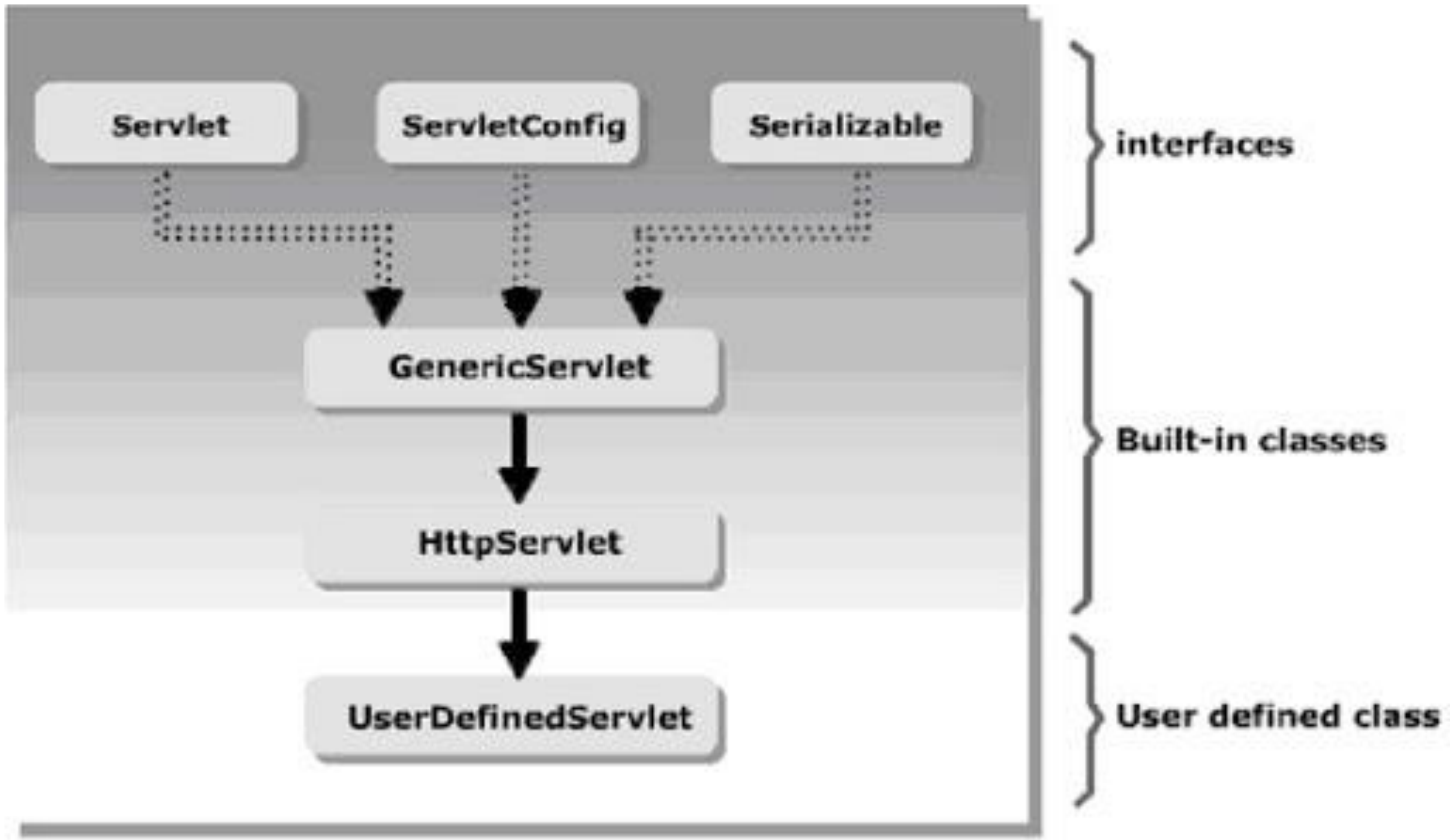
## Servlet API

- ❑ A Servlet is an object that receives a request and generates a response based on that request.
- ❑ The basic Servlet package defines Java objects to represent servlet requests and responses, as well as objects to reflect the servlet's configuration parameters and execution environment.
- ❑ The package *javax.servlet.http* defines HTTP-specific subclasses of the generic servlet elements, including session management objects that track multiple requests and responses between the web server and a client.

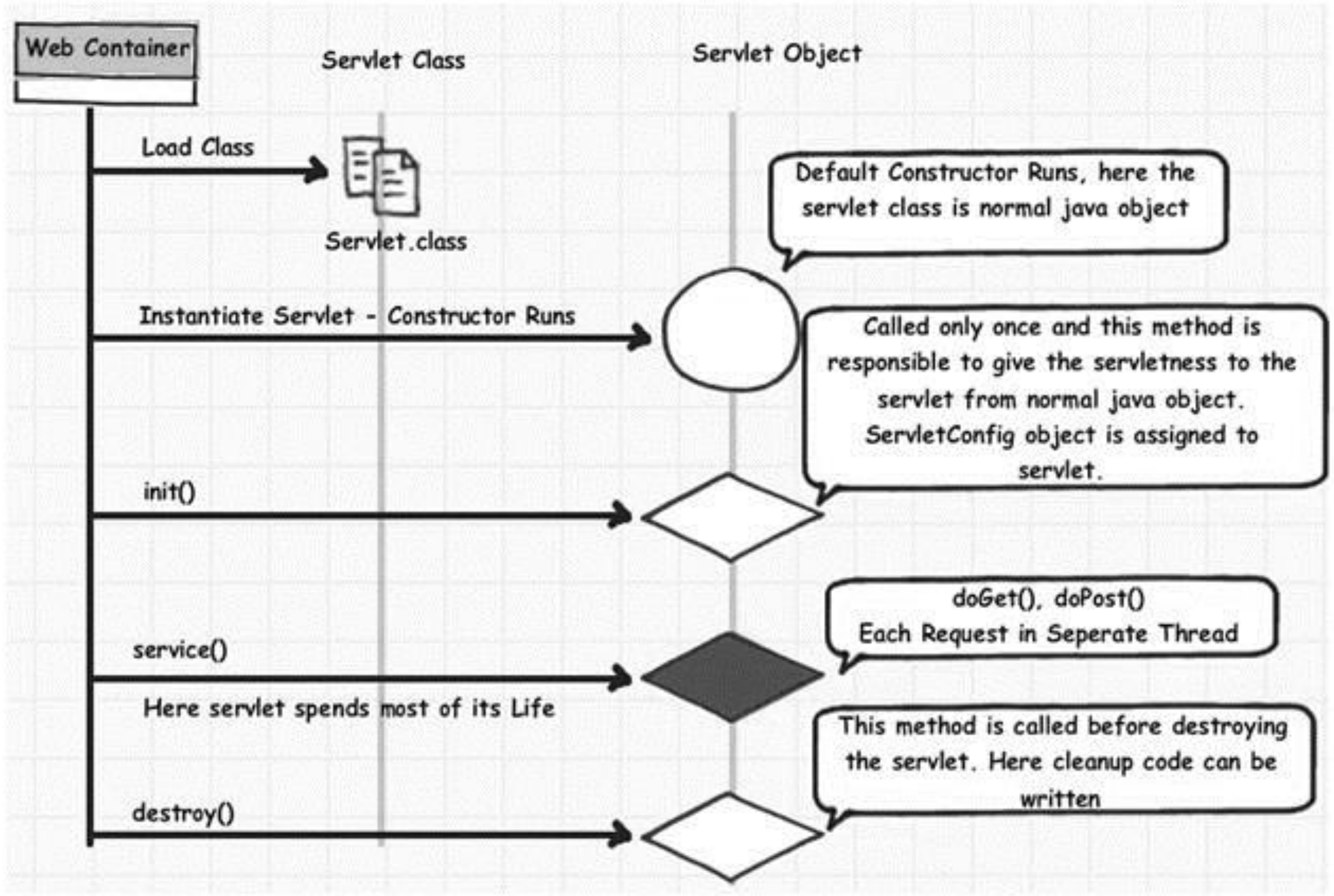
## Servlet Advantages

- ❑ Read form data from HTML
- ❑ Cookie handling
- ❑ User session tracking
- ❑ Follow Object Oriented Paradigm
- ❑ Each HTTP request is served by one Thread
- ❑ Servlet can dialog between the web server
- ❑ Exploits the HTTP protocol
- ❑ Offers security

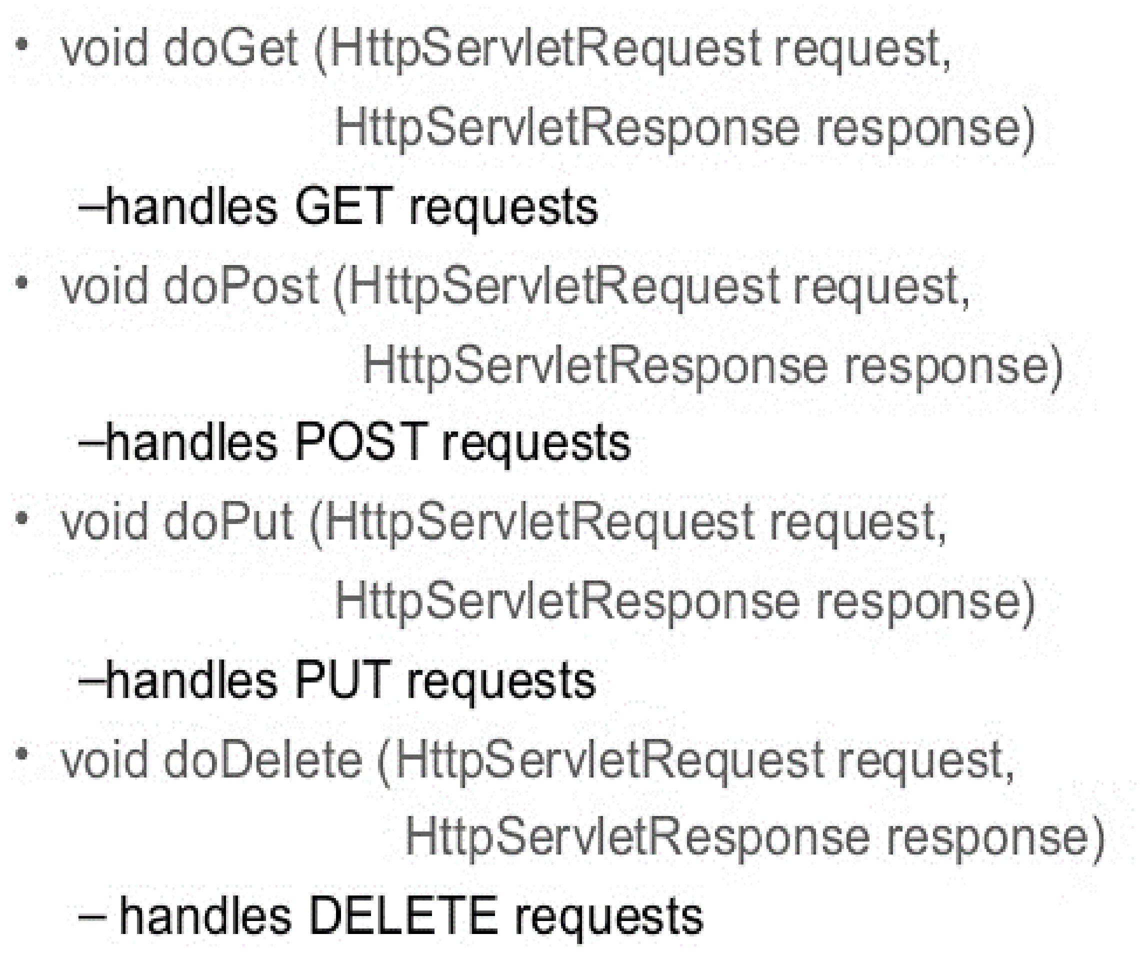
# Servlet Class diagram



# Servlet Lifecycle


Web Container
Servlet Class
Servlet Object
Load Class
Servlet.class
Default Constructor Runs, here the servlet class is normal java object
Instantiate Servlet - Constructor Runs
Called only once and this method is responsible to give the servletness to the servlet from normal java object. ServletConfig object is assigned to servlet.
init()
doGet(), doPost()
Each Request in Seperate Thread
service()
Here servlet spends most of its Life
This method is called before destroying the servlet. Here cleanup code can be written
destroy()

## HttpServlet Service methods



- void doGet (HttpServletRequest request,
  HttpServletResponse response)
  - handles GET requests
- void doPost (HttpServletRequest request,
  HttpServletResponse response)
  - handles POST requests
- void doPut (HttpServletRequest request,
  HttpServletResponse response)
  - handles PUT requests
- void doDelete (HttpServletRequest request,
  HttpServletResponse response)
  - handles DELETE requests

## Servlet Example – 1/2

```
// Import required java libraries
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

// Extend HttpServlet class
public class HelloWorld extends HttpServlet {

   private String message;

   @Override
   public void init() throws ServletException {
      // Do required initialization
      message = "Hello World";
   }
```

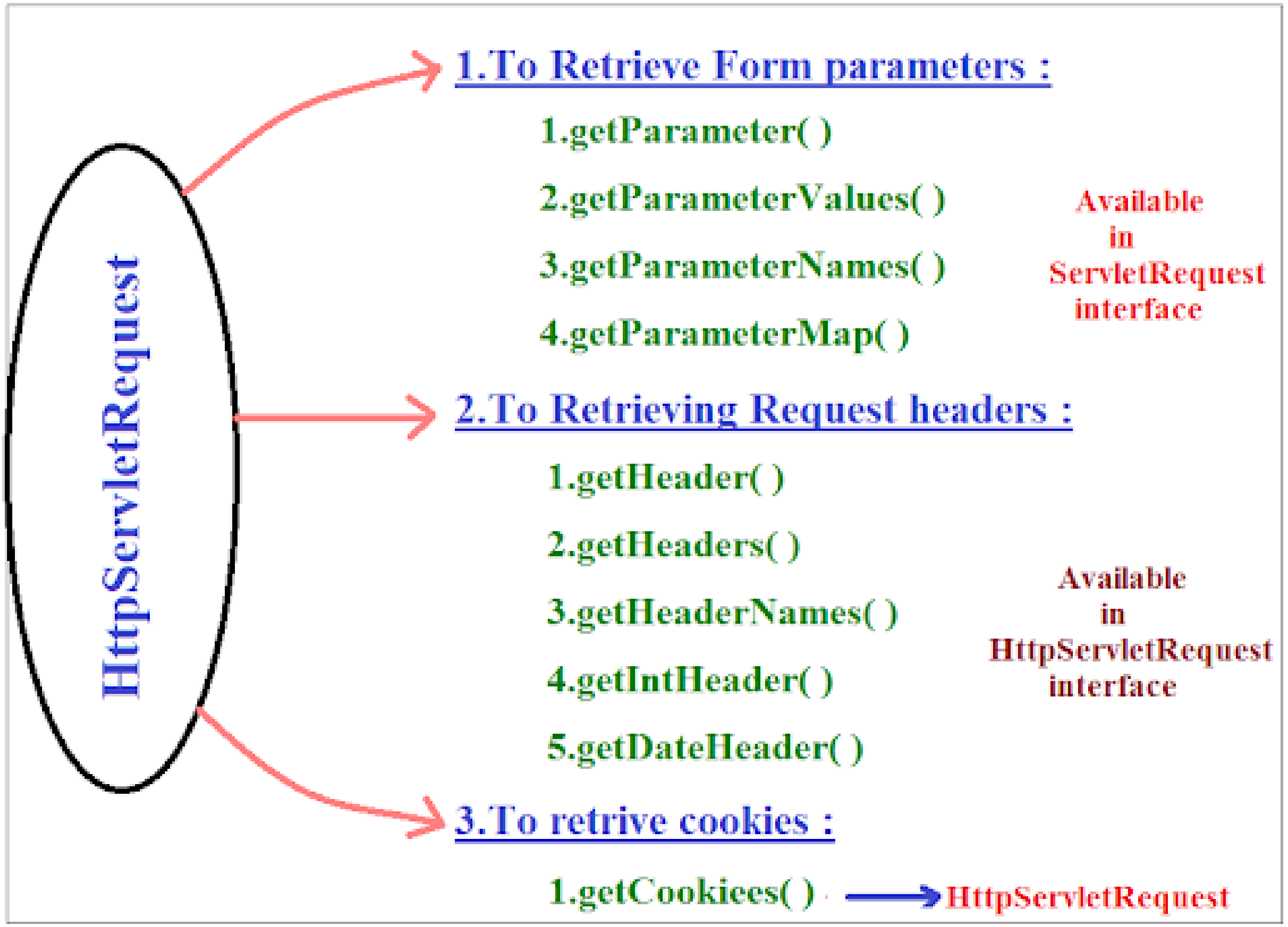
## Servlet Example – 2/2

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

      // Set response content type
      response.setContentType("text/html");

      // Actual logic goes here.
      PrintWriter out = response.getWriter();
      out.println("<h1>" + message + "</h1>");
   }

   @Override
   public void destroy() {
      // do nothing.
   }
}
```

## HttpServletRequest

- ❑ Requests contain information to send to the server.
- ❑ A http request contains the following information:
  - ▪ Parameters
  - ▪ Attributes
  - ▪ Headers
  - ▪ Protocol Information
  - ▪ Localization

# HttpServletRequest common methods

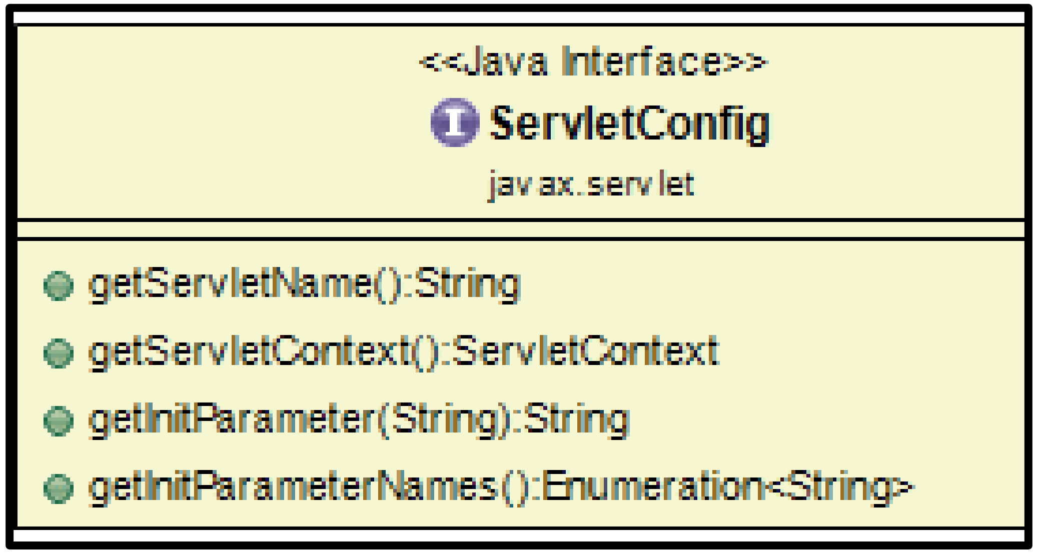


## HttpServletResponse

- ❑ Responses contain information to send back to the client.
- ❑ A http response contains the following information:
  - Output Stream
  - Content type
  - Headers

## ServletConfig

- A ServetConfig is used by the server container to pass information to a servlet during initialization.
- Its initialization parameters can ONLY be set in a deployment descriptor (web.xml).

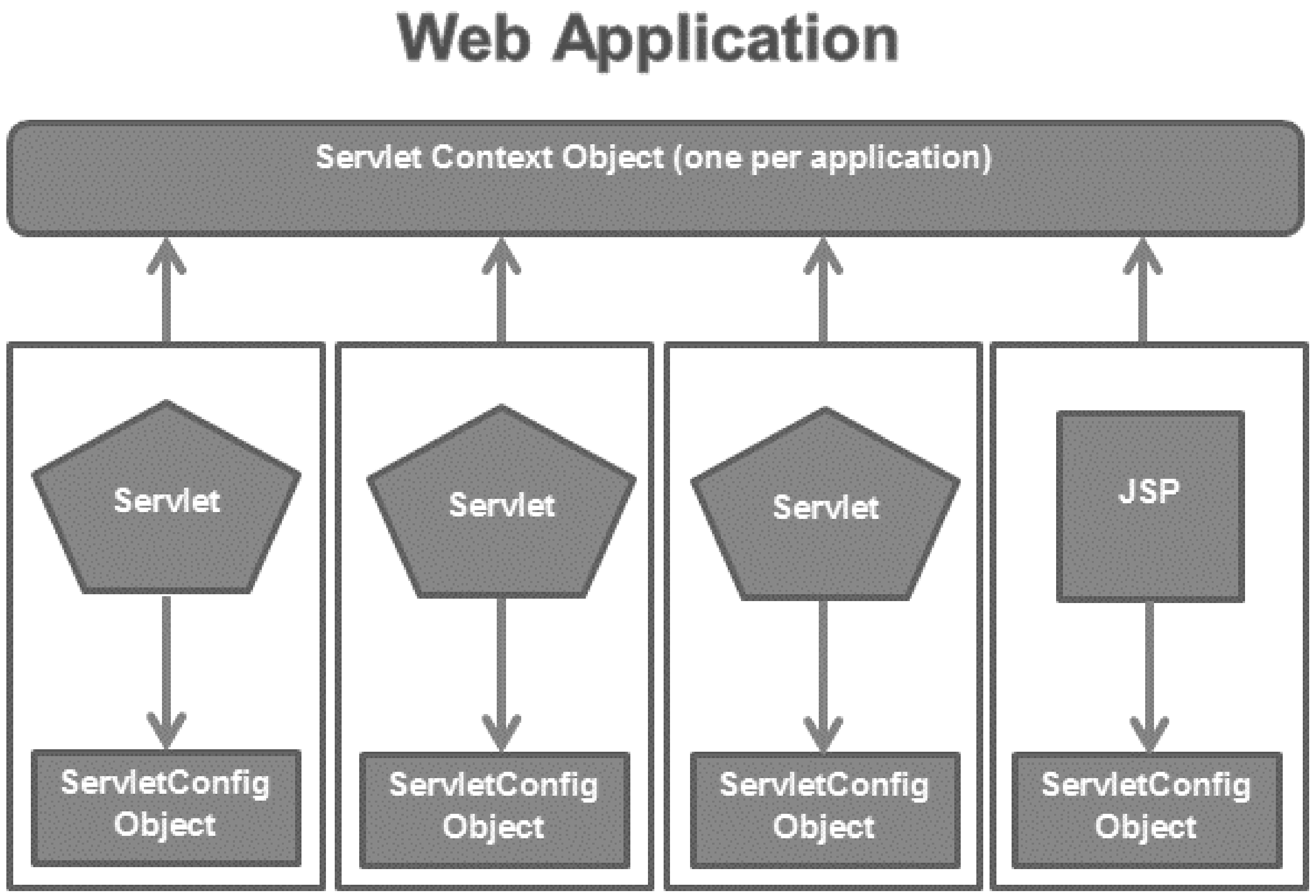


<<Java Interface>>
ServletConfig
javax.servlet
getServletName():String
getServletContext():ServletContext
getInitParameter(String):String
getInitParameterNames():Enumeration<String>

## ServletContext

- In order to access container-level services, the servlet must own a copy of the servlet context , which is a type of object used to provide the servlet with a view to its Web application.
- Each web app has only one servlet context
- ServletContext represents the application at its given context root.
- The ServletContext object can be used to get configuration information, get or remove attributes and to provide inter-application communication.

# ServletContext



# ServletContext methods – 1/2

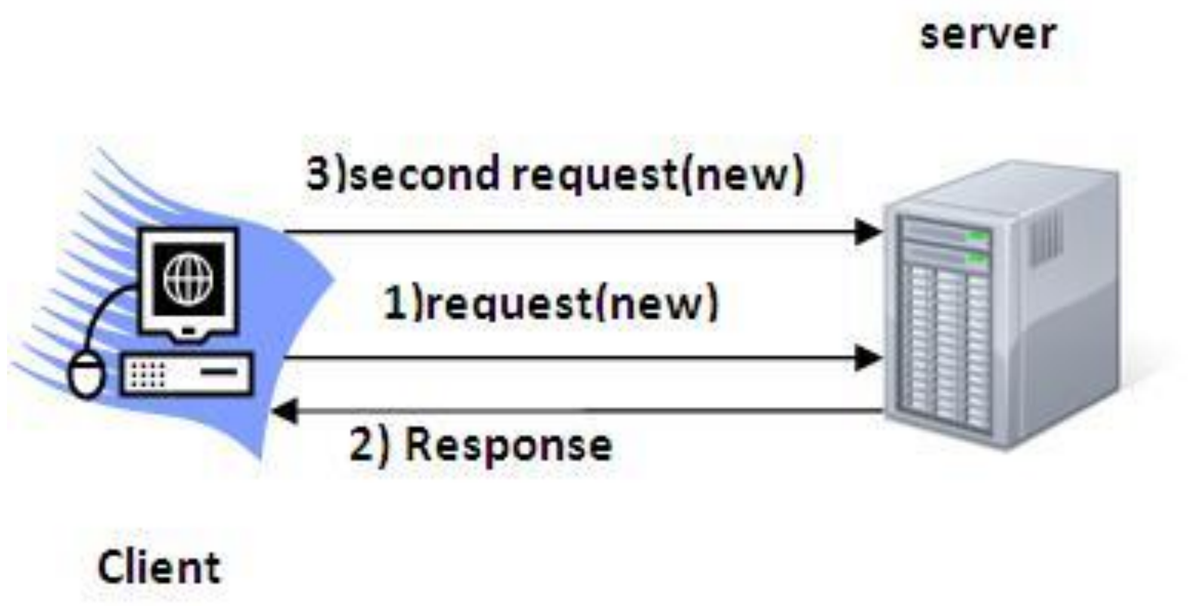
- *public String getInitParameter(String name):*Returns the parameter value for the specified parameter name.
- *public Enumeration getInitParameterNames():*Returns the names of the context's initialization parameters.
- *public void setAttribute(String name,Object object)*:sets the given object in the application scope.
- *public Object getAttribute(String name)*:Returns the attribute for the specified name.

## ServletContext methods – 2/2

- *public Enumeration getInitParameterNames():*Returns the names of the context's initialization parameters as an Enumeration of String objects.
- *public void removeAttribute(String name):*Removes the attribute with the given name from the servlet context.

## Session

- ❑ HTTP is stateless that means each request is considered as the new request.



## Session definition

- **Session** is a conversional state between client and server, and it can consist of multiple request and response between client and server.
- Since HTTP and Web Server both are stateless, the only way to maintain a session is when some unique information about the session (session id) is passed between server and client in every request and response.

## Session tracking

- **Session Tracking** is a way to maintain state (data) of a user. It is also known as session management in servlet.
- The idea is simple: on the client’s first request, the Container generates a unique session ID and gives it back to the client with the response. The client sends back the session ID with each subsequent request. The Container sees the ID, finds the matching session, and associates the session with the request.

## Session tracking techniques

- ❑ There are four techniques used in Session tracking:
  - Cookies
  - Hidden form field
  - URL rewriting
  - HttpSession

## Cookies – 1/2

- Cookies are small piece of information that is sent by web server in response header and gets stored in the browser cookies.
- When client make further request, it adds the cookie to the request header and we can utilize it to keep track of the session.
- We can maintain a session with cookies but if the client disables the cookies, then it won’t work.

## Cookies – 2/2

- ❑ Cookies are used a lot in web applications to personalize response based on your choice or to keep track of session.
- ❑ Notice the cookie that we are setting to the response..

```
Cookie loginCookie = new Cookie("user",user);
//setting cookie to expiry in 30 mins
loginCookie.setMaxAge(30*60);
response.addCookie(loginCookie);
```

## URL Rewriting

- ❑ If your browser does not support cookies, URL rewriting provides you with another session tracking alternative.
- ❑ We can append a session identifier parameter with every request and response to keep track of the session. This is very tedious because we need to keep track of this parameter in every response and make sure it’s not clashing with other parameters.
- ❑ Disadvantages: It will work only with links ad can send only text!

## Hidden Form field

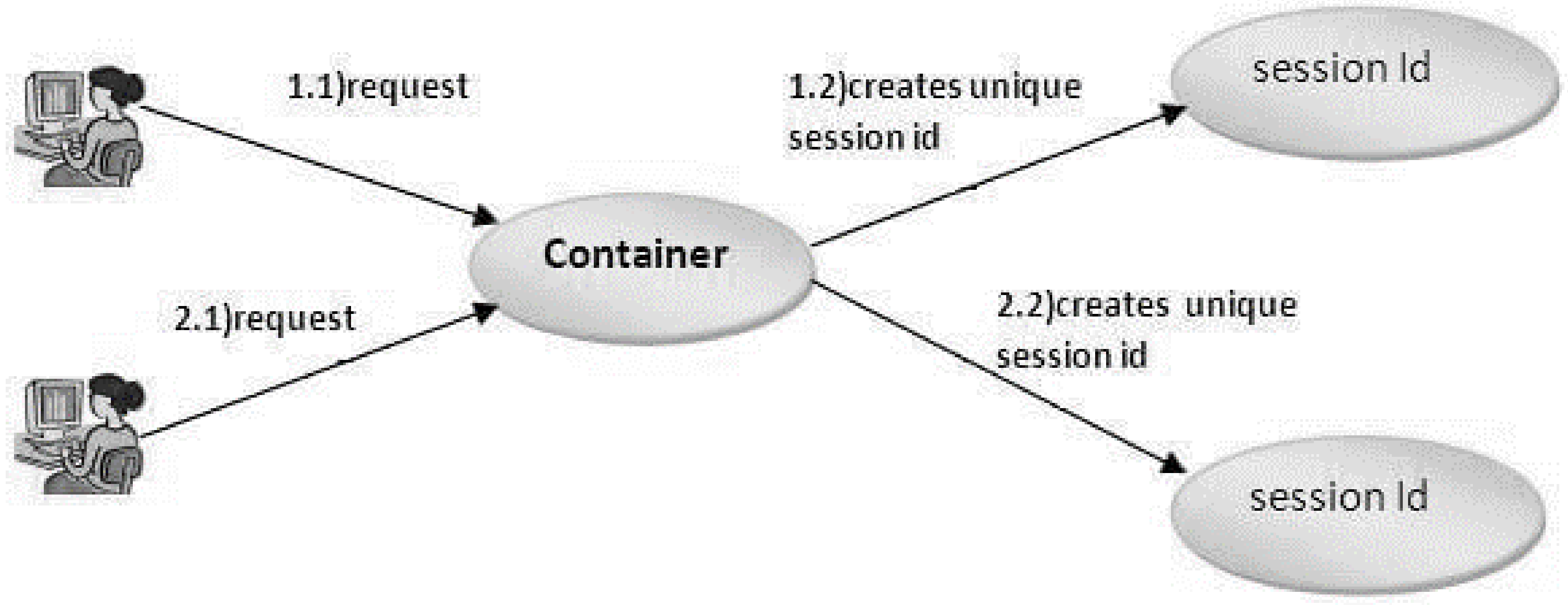
- ❑ A hidden form field is a way of session tracking so that developers can save the information in the client browser itself.
- ❑ For that, developers can use a hidden field for securing the fields.
- ❑ At the time of page display, no one can see these fields. When a client submits the form, the browser also transfers these hidden value to the server with the other fields.

```
<input type="hidden" name="testHidden" value="testHiddenValue" />
```

## HttpSession

- ❑ Container creates a session id for each user.
- ❑ The container uses this id to identify the particular user.
- ❑ An object of HttpSession can be used to perform two tasks:
    - ▪ bind objects
    - ▪ view and manipulate information about a session, such as the session identifier, creation time, and last accessed time.

# HttpSession Example



## How to use HttpSession?

- The HttpServletRequest interface provides two methods to get the object of HttpSession.
- *public HttpSession getSession()*:Returns the current session associated with this request, or if the request does not have a session, creates one.
- *public HttpSession getSession(boolean create):*Returns the current HttpSession associated with this request or, if there is no current session and create is true, returns a new session.

## HttpSession methods

- public String *getId*():Returns a string containing the unique identifier value.
- public void *setAttribute*(String name, Object obj): save an attribute on session.
- public void *getAttribute*(String name): get an attribute from session.
- public long *getCreationTime*():Returns the time when this session was created, measured in milliseconds since midnight January 1, 1970 GMT.
- public long *getLastAccessedTime*():Returns the last time the client sent a request associated with this session, as the number of milliseconds since midnight January 1, 1970 GMT.
- public void *invalidate*():Invalidates this session then unbinds any objects bound to it.

## Exercise

- develop a JavaEE application that exposes one endpoint service
- the endpoint should be able to accept any http method
- the endpoint acts as an echo request
- the servlet, at each request, add a new element to the session
  - key = random integer string
  - value = string containing the current date-time
- it returns a simple html document containing
  - all the received http headers and relative values
  - the session id and any data key from the session
  - the url parameters
  - the body parameters

## Java Server Pages (JSP) – 1/2

- Architecturally, JSP may be viewed as a high-level abstraction of Java servlets.
- JSPs are translated into servlets at runtime, therefore JSP is a Servlet.
- Each JSP servlet is cached and re-used until the original JSP is modified (hot-deploy).
- Java Server Pages can be used independently or as the view component of a server-side model–view–controller design, normally with JavaBeans as the model and Java servlets as the controller.
- JSP allows Java code and certain predefined actions to be interleaved with static web markup content, such as HTML.

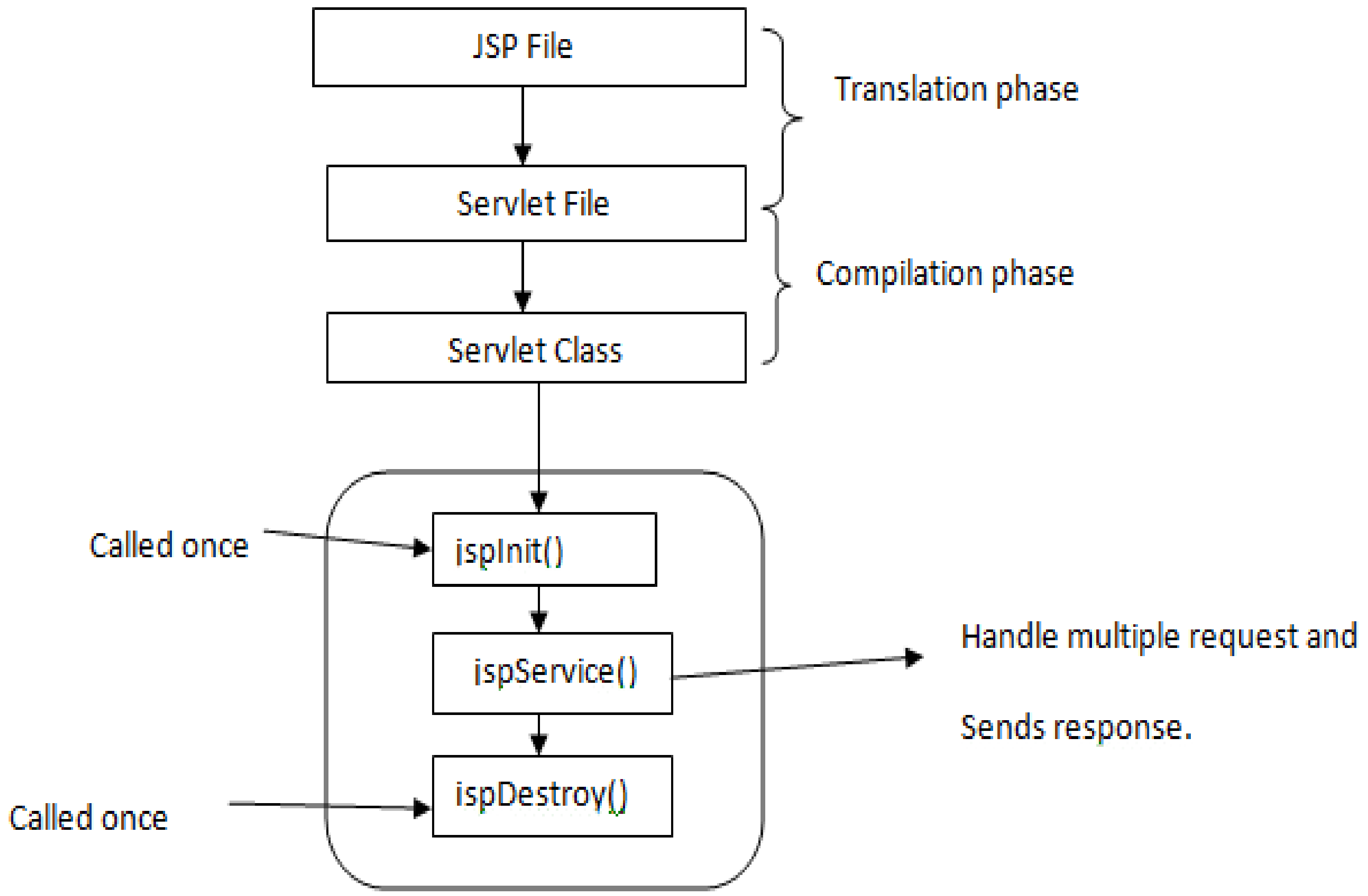
## Java Server Pages (JSP) – 2/2

- The resulting page is compiled and executed on the server to deliver a document.
- The compiled pages, as well as any dependent Java libraries, contain Java bytecode rather than machine code. Like any other .jar or Java program, code must be executed within a Java virtual machine (JVM) that interacts with the server's host operating system to provide an abstract, platform-neutral environment.
- JSPs are usually used to deliver HTML and XML documents, but using OutputStream, they can deliver other types of data as well.
- The Web container creates JSP implicit objects like request, response, session, application, config, page, pageContext, out and exception. JSP Engine creates these objects during translation phase.

## JSP Lifecycle

1. Jsp creation
2. Deployment: the author deploys to the container
3. Validation (tag libraries, EL, actions)
4. Translation: Jsp translated into servlet source code
5. Compilation: servlet compiled into Java bytecode
6. Load class: servlet is loaded into memory using application class loader

# JSP Lifecycle methods



## JSP Example

```
<html>
<head><title>First JSP</title></head>
<body>
  <%
    double num = Math.random();
    if (num > 0.95) {
  %>
      <h2>You'll have a luck day!</h2><p>(<%= num %>)</p>
  <%
    } else {
  %>
      <h2>Well, life goes on ... </h2><p>(<%= num %>)</p>
  <%
    }
  %>
  <a href="<%= request.getRequestURI() %>"><h3>Try Again</h3></a>
</body>
</html>
```

# JSP Elements

1. Directives
2. Declarations
3. Scriptlets
4. Expressions

## Directives

- A JSP directive affects the overall structure of the servlet class. It usually has the following form

```
<%@ directive attribute = "value" %>
```

- Directives can have several attributes which you can list down as key-value pairs and separated by commas.
- The blanks between the @ symbol and the directive name, and between the last attribute and the closing %>, are optional.

# Directive types

| Type | Directive & Description |
|---|---|
| page | **<%@ page ... %>**<br>Defines page-dependent attributes, such as scripting language, error page, and buffering requirements. |
| include | **<%@ include ... %>**<br>Includes a file during the translation phase. |
| taglib | **<%@ taglib ... %>**<br>Declares a tag library, containing custom actions, used in the page |

## Main Page Directive attributes

| Attribute | Purpose |
|---|---|
| buffer | Specifies a buffering model for the output stream. |
| autoFlush | Controls the behavior of the servlet output buffer. |
| contentType | Defines the character encoding scheme. |
| errorPage | Defines the URL of another JSP that reports on Java unchecked runtime exceptions. |
| isErrorPage | Indicates if this JSP page is a URL specified by another JSP page's errorPage attribute. |
| extends | Specifies a superclass that the generated servlet must extend |
| import | Specifies a list of packages or classes for use in the JSP as the Java import statement does for Java classes. |
| info | Defines a string that can be accessed with the servlet's getServletInfo() method. |
| isThreadSafe | Defines the threading model for the generated servlet. |
| language | Defines the programming language used in the JSP page. |
| session | Specifies whether or not the JSP page participates in HTTP sessions |
| isELIgnored | Specifies whether or not EL expression within the JSP page will be ignored. |
| isScriptingEnabled | Determines if scripting elements are allowed for use. |

## Include directive

- The include directive is used to include a file during the translation phase.
- This directive tells the container to merge the content of other external files with the current JSP during the translation phase.
- You may code the include directives anywhere in your JSP page.
- The general usage form of this directive is as follows:

```
<%@ include file = "relative url" >
```

## Declarations

- JSP declaration is used to declare methods and variables global for a JSP page.
- Declarations start with a <%! Delimiter and ens with %>
- You can declare your own method:

```
<%! Private int generateNextNumber(){
         return randomObj.nextInt();
}
%>
```

## Scriptlet

- They contain any Java processing code for the body of a JSP. They provide the logic functions for the processing in the _jspService method.
- A scriptlet has <% %> delimiter.
- Variables declared in a scriptlet are method-local to the jsp’s service method.
- Used to cycle on collections.

## Scriplet Example

```
<%
    String userID = request.getParameter("userid");
    String userStatus = getUserStatus(userID);
    String audioFile = "";
    if(userStatus.equals(UserStatus.NEW)) {
        audioFile = "new.wav";
    } else {
        audioFile = "existing.wav";
    }
%>
```

## JSP Implicit Objects – 1/2

- **request** : the type is HttpServletRequest under the http protocol.
- **response** : the type is HttpServletResponse
- **pageContext** : class javax.servlet.jsp.PageContext
- **session** : interface javax.servlet.http.HttpSession
- **application** : interface javax.servlet.ServletContext

## JSP Implicit Objects – 2/2

- **out** : an outputStream, your JSP scripting elements should always write to the out implicit object and not to the response stream.(JspWriter)
- **config:** Javax.servlet.ServletConfig
- **page:** the instance of this page’s implementation class (type Object)
- **exception:** instance of Exception object

## JavaServerFaces (JSF)

- Java Server Faces (JSF) is a Java-based web application framework intended to simplify development integration of web-based user interfaces.
- JSF is a standardized display technology, which was formalized in a specification through the Java Community Process.
- A JSF is a better version of a JSP.
- It takes many of the common tasks that developers build into web applications and provides specific mechanisms, tags, and APIs to help those tasks be built into a web application more easily.

## JSF Example – 1/2

```
import javax.faces.bean.ManagedBean;

@ManagedBean(name = "helloWorld", eager = true)
public class HelloWorld {

   public HelloWorld() {
      System.out.println("HelloWorld started!");
   }

   public String getMessage() {
      return "Hello World!";
   }
}
```

## JSF Example – 2/2

```
<!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>JSF Tutorial!</title>
   </head>

   <body>
      #{helloWorld.getMessage()}
   </body>
</html>
```

## EJB (Enterprise Java Bean)

- EJB is an essential part of a J2EE platform. J2EE platform has component-based architecture to provide multi-tiered, distributed and highly transactional features to enterprise level applications.
- EJB provides an architecture to develop and deploy component-based enterprise applications considering robustness, high scalability, and high performance.

## EJB Types

- **Session Bean**: Session bean stores data of a particular user for a single session. It can be stateful or stateless. It is less resource intensive as compared to entity bean. Session bean gets destroyed as soon as user session terminates.
- **Entity Bean:** Entity beans represent persistent data storage. User data can be saved to database via entity beans and later can be retrieved from the database in the entity bean.
- **Message Driven Bean:** Message driven beans are used in context of JMS (Java Messaging Service). Message Driven Beans can consume JMS messages from external entities and act accordingly.

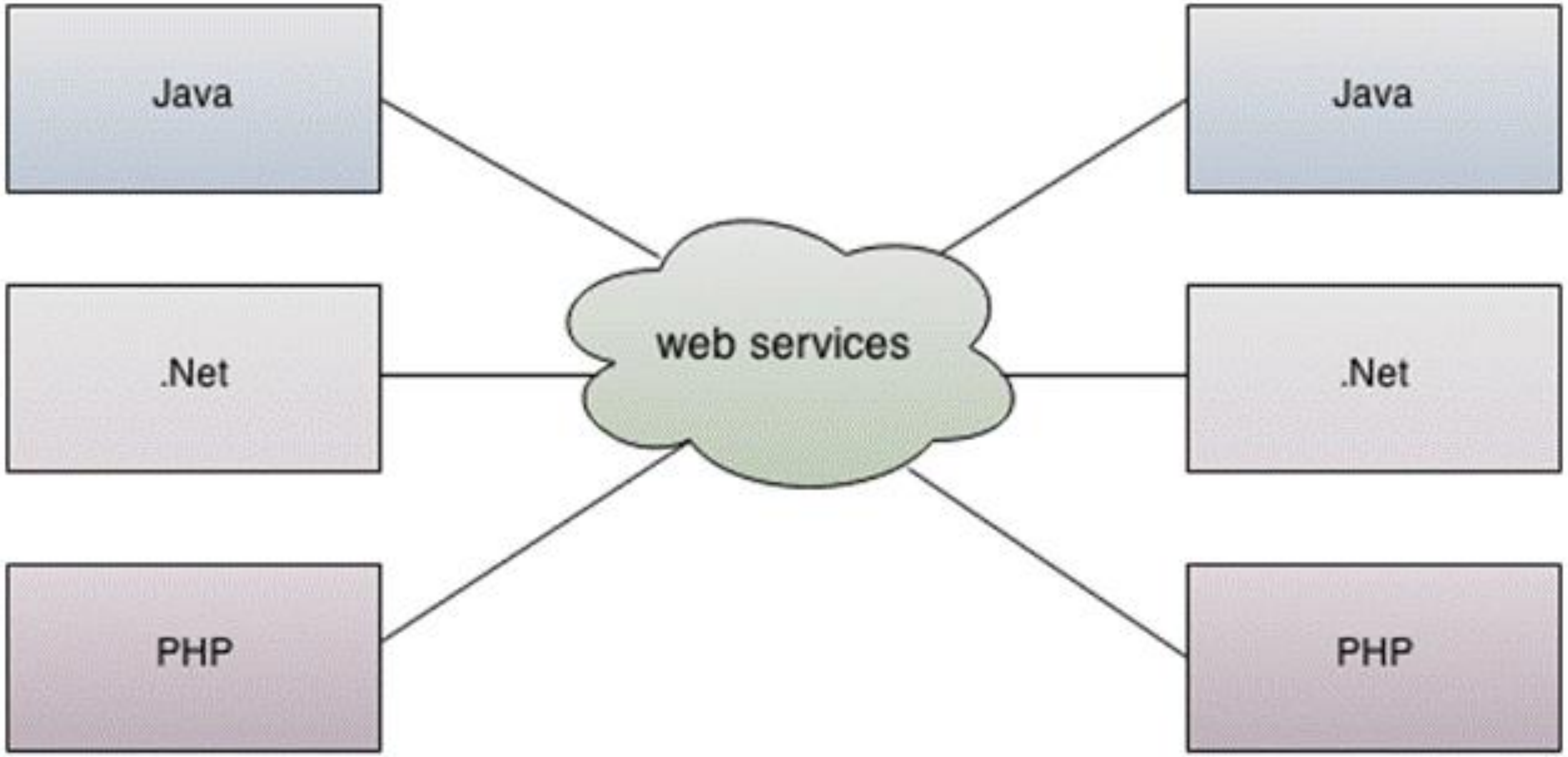
## EJB Benefits

- ❑ Simplified development of large-scale enterprise level application.
- ❑ Application Server/EJB container provides most of the system level services like transaction handling, logging, load balancing, persistence mechanism, exception handling, and so on. Developer must focus only on business logic of the application.
- ❑ EJB container manages life cycle of EJB instances, thus developer needs not to worry about when to create/delete EJB objects

## Web Service

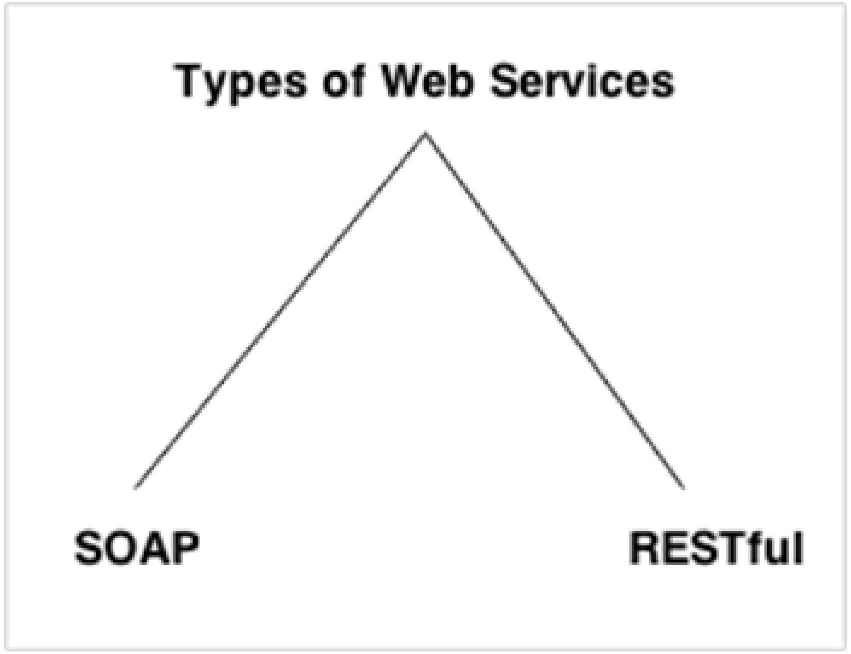
- A Web Service is can be defined by following ways:
  - It is a client-server application or application component for communication.
  - The method of communication between two devices over the network.
  - It is a software system for the interoperable machine to machine communication.
  - It is a collection of standards or protocols for exchanging information between two devices or application.

# Web Service Example



## Web Service Types

- There are mainly two types of web services.
  - SOAP web services.
  - RESTful web services.



## Web Service Features – 1/2

- ❑ **XML-Based/Json-Based** : Web services use XML/JSON at data description and data transportation layers. Using these exclude any networking, operating system, or platform binding. Web services-based operation is extremely interoperable at their core level.
- ❑ **Loosely Coupled** : A client of a web service is not fixed to the web service directly. The web service interface can support innovation over time without negotiating the client's ability to communicate with the service. Accepting a loosely coupled architecture tends to make software systems more manageable and allows more straightforward integration between various systems.

# Web Service Features – 2/2

- **Coarse-Grained** : Object-oriented technologies such as Java expose their functions through individual methods. Web services technology implement a natural method of defining coarse-grained services that approach the right amount of business logic.
- **Ability to be Synchronous or Asynchronous** : In synchronous invocations, the client blocks and delays in completing its service before continuing. Asynchronous operations grant a client to invoke a task and then execute other functions.

## SOAP

- ❑ SOAP stands for Simple Object Access Protocol. It is an XML-based protocol for accessing web services.
- ❑ SOAP is a W3C recommendation for communication between two applications.
- ❑ SOAP web services can be written in any programming language and executed in any platform.
- ❑ SOAP uses XML format that must be parsed to be read. It defines many standards that must be followed while developing the SOAP applications. So, it is slow and consumes more bandwidth and resource.

# RESTful

- ❑ REST stands for REpresentational State Transfer.
- ❑ REST is an architectural style not a protocol.
- ❑ RESTful Web Services are fast because there is no strict specification like SOAP. It consumes less bandwidth and resource.
- ❑ RESTful web services can be written in any programming language and executed in any platform.
- ❑ RESTful web service permits different data format such as Plain Text, HTML, XML and JSON.

# RESTful vs SOAP

| SOAP | REST |
|---|---|
| SOAP is a **protocol**. | REST is an **architectural style**. |
| SOAP stands for **Simple Object Access Protocol**. | REST stands for **REpresentational State Transfer**. |
| SOAP **can't use REST** because it is a protocol. | REST **can use SOAP** web services because it is a concept and can use any protocol like HTTP, SOAP. |
| SOAP **uses services interfaces to expose the business logic**. | REST **uses URI to expose business logic**. |
| **JAX-WS** is the java API for SOAP web services. | **JAX-RS** is the java API for RESTful web services. |
| SOAP **defines standards** to be strictly followed. | REST does not define too much standards like SOAP. |
| SOAP **requires more bandwidth** and resource than REST. | REST **requires less bandwidth** and resource than SOAP. |
| SOAP **defines its own security**. | RESTful web services **inherits security measures** from the underlying transport. |
| SOAP **permits XML** data format only. | REST **permits different** data format such as Plain text, HTML, XML, JSON etc. |
| SOAP is **less preferred** than REST. | REST **more preferred** than SOAP. |

## Java Web Services

- ❑ Java web service application perform communication through WSDL (Web Services Description Language).

- ❑ There are two main API's defined by Java for developing web service applications since JavaEE 6.

  1. **JAX-WS**: for SOAP web services. The are two ways to write JAX-WS application code: by RPC style and Document style.

  2. **JAX-RS**: for RESTful web services. There are mainly 2 implementation currently in use for creating JAX-RS application: Jersey and RESTeasy.

## JAX-WS – RPC Style

- RPC style web services use method name and parameters to generate XML structure.
- The generated WSDL is difficult to be validated against schema.
- In RPC style, SOAP message is sent as many elements as possible.
- RPC style message is tightly coupled.
- In RPC style, SOAP message keeps the operation name.
- In RPC style, parameters are sent as discrete values.

# JAX-WS – RPC Style Example – 1/3

*File: HelloWorld.java*

```
package com.reti;
import javax.jws.WebMethod;
import javax.jws.WebService;
import javax.jws.soap.SOAPBinding;
import javax.jws.soap.SOAPBinding.Style;
//Service Endpoint Interface
@WebService
@SOAPBinding(style = Style.RPC)
public interface HelloWorld{
    @WebMethod String getHelloWorldAsString(String name);
}
```

*File: HelloWorldImpl.java*

```
package com.reti;
import javax.jws.WebService;
//Service Implementation
@WebService(endpointInterface = "com.reti.HelloWorld")
public class HelloWorldImpl implements HelloWorld{
    @Override
    public String getHelloWorldAsString(String name) {
        return "Hello World JAX-WS " + name;
    }
}
```

## JAX-WS – RPC Style Example – 2/3

*File: Publisher.java*

```
package com.reti;
import javax.xml.ws.Endpoint;
//Endpoint publisher
public class HelloWorldPublisher{
    public static void main(String[] args) {
       Endpoint.publish("http://localhost:7779/ws/hello", new HelloWorldImpl());
        }
}
```

After running the publisher code, you can see the generated WSDL file by visiting the URL:

http://localhost:7779/ws/hello?wsdl

# JAX-WS – RPC Style Example – 3/3

*File: HelloWorldClient.java*

```
package com.reti;
import java.net.URL;
import javax.xml.namespace.QName;
import javax.xml.ws.Service;
public class HelloWorldClient{
    public static void main(String[] args) throws Exception {
        URL url = new URL("http://localhost:7779/ws/hello?wsdl");
        //1st argument service URI, refer to wsdl document above
        //2nd argument is service name, refer to wsdl document above
        QName qname = new QName("http://reti.com/", "HelloWorldImplService");
        Service service = Service.create(url, qname);
        HelloWorld hello = service.getPort(HelloWorld.class);
        System.out.println(hello.getHelloWorldAsString("test rpc"));
     }
 }
```

```
Output:

Hello World JAX-WS test rpc
```

## JAX-WS – Document Style

- ❑ Document style web services can be validated against predefined schema.
- ❑ In document style, SOAP message is sent as a single document.
- ❑ Document style message is loosely coupled.
- ❑ In Document style, SOAP message loses the operation name.
- ❑ In Document style, parameters are sent in XML format.

# JAX-WS – Document Style Example – 1/3

*File: HelloWorld.java*

```
package com.reti;
import javax.jws.WebMethod;
import javax.jws.WebService;
import javax.jws.soap.SOAPBinding;
import javax.jws.soap.SOAPBinding.Style;
//Service Endpoint Interface
@WebService
@SOAPBinding(style = Style.DOCUMENT)
public interface HelloWorld{
    @WebMethod String getHelloWorldAsString(String name);
}
```

*File: HelloWorldImpl.java*

```
package com.reti;
import javax.jws.WebService;
//Service Implementation
@WebService(endpointInterface = "com.reti.HelloWorld")
public class HelloWorldImpl implements HelloWorld{
    @Override
    public String getHelloWorldAsString(String name) {
        return "Hello World JAX-WS " + name;
    }
}
```

## JAX-WS – Document Style Example – 2/3

*File: Publisher.java*

```
package com.reti;
import javax.xml.ws.Endpoint;
//Endpoint publisher
public class HelloWorldPublisher{
    public static void main(String[] args) {
       Endpoint.publish("http://localhost:7779/ws/hello", new HelloWorldImpl());
        }
}
```

After running the publisher code, you can see the generated WSDL file by visiting the URL:
http://localhost:7779/ws/hello?wsdl

If you run the publisher class, it may generate following error:
 *Wrapper class com.javatpoint.GetHelloWorldAsString is not found.*
 *Have you run APT to generate them?*
To solve the problem, go to bin directory of your current project in command prompt and run the following command:
 *wsgen -keep -cp . com.reti.HelloWorldImpl*
Now, it will generator 2 files: SayHello and SayHelloResponse
Paste these files in com.reti directory and then run the publisher class.

## JAX-WS – Document Style Example – 3/3

*File: HelloWorldClient.java*

```
package com.reti;
import java.net.URL;
import javax.xml.namespace.QName;
import javax.xml.ws.Service;
public class HelloWorldClient{
    public static void main(String[] args) throws Exception {
        URL url = new URL("http://localhost:7779/ws/hello?wsdl");
        //1st argument service URI, refer to wsdl document above
        //2nd argument is service name, refer to wsdl document above
        QName qname = new QName("http://reti.com/", "HelloWorldImplService");
        Service service = Service.create(url, qname);
        HelloWorld hello = service.getPort(HelloWorld.class);
        System.out.println(hello.getHelloWorldAsString(“test rpc"));
     }
 }
```

```
Output:

Hello World JAX-WS test rpc
```

## JAX-RS Operations

- ❑ RESTful web services use HTTP methods to perform operations on resources.
- ❑ **HTTP GET :** for read operations on a resource to obtain all the information it can give about its identity.
- ❑ **HTTP PUT :** for update operations to allow the client to pass information to the resource so that the resource can be modified as a result of the operation.
- ❑ **HTTP POST :** to create new web services resources to allow the client to pass information to a web service resource that it uses to create a new web resource.
- ❑ **HTTP DELETE :** for delete operations to allow the client to send an HTTP DELETE request to a web service resource, with the result that the resource is removed from the web service.

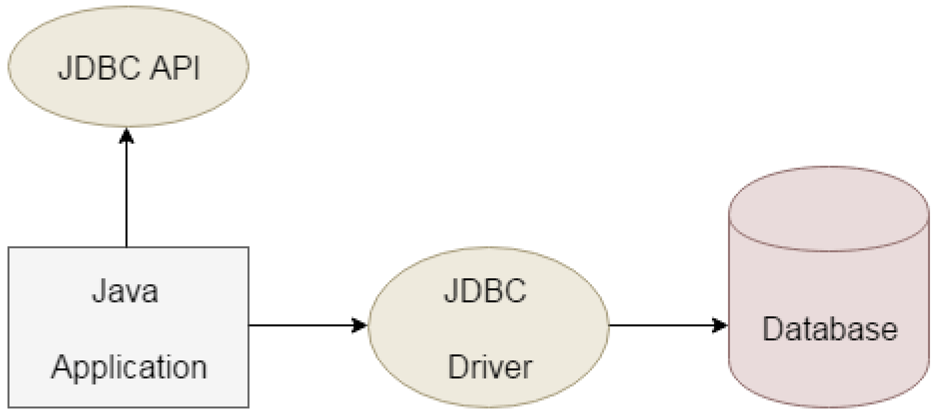
## JAX-RS – Example

```
import javax.ws.rs.GET;
import javax.ws.rs.Path;

@Path("hello")
public class HelloResource {

  @GET
  public String sayHello() {
    return "Hello you !";
  }

}
```

# JDBC: Java and relational databases

# JDBC

- JDBC stands for Java Database Connectivity.
- JDBC is a Java API to connect and execute the query with the database.
- We can use JDBC API to access tabular data stored in any relational database. By the help of JDBC API, we can save, update, delete and fetch data from the database.
- It is like Open Database Connectivity (ODBC) provided by Microsoft.

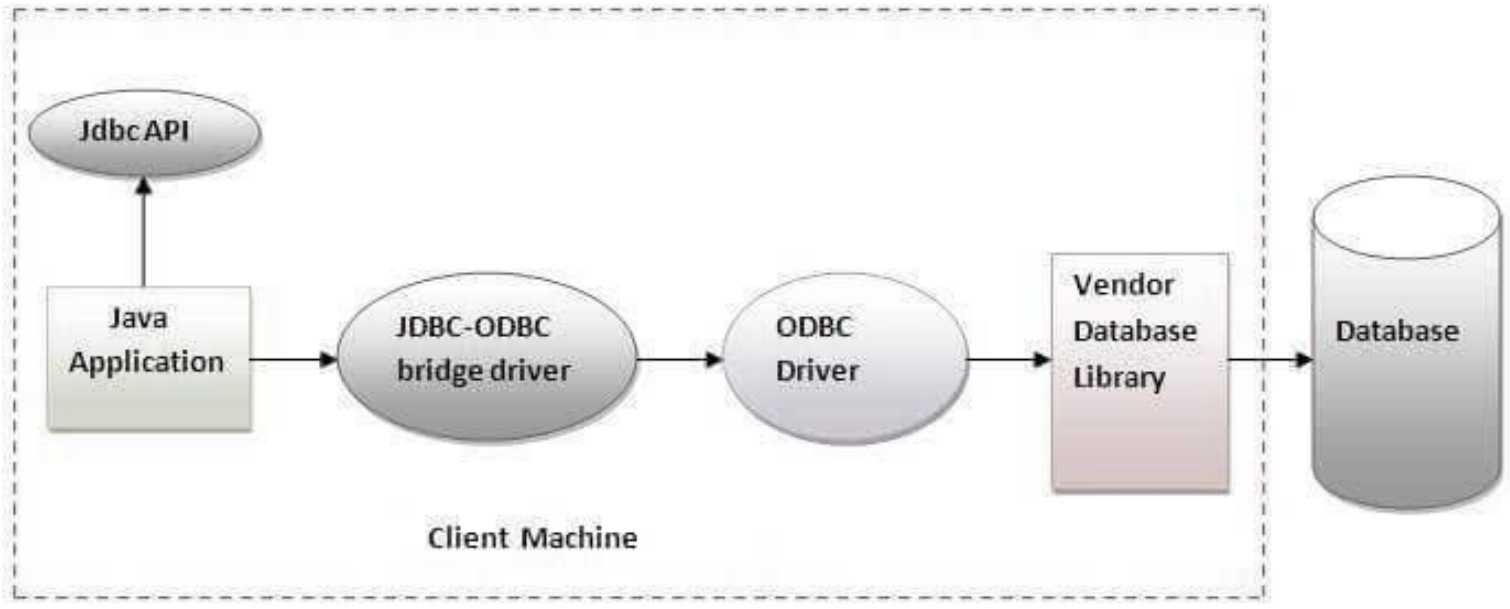


## JDBC Driver

- JDBC Driver is a software component that enables java application to interact with the database. There are 4 types of JDBC drivers:
  - JDBC-ODBC bridge driver
  - Native-API driver (partially java driver)
  - Network Protocol driver (fully java driver)
  - Thin driver (fully java driver)

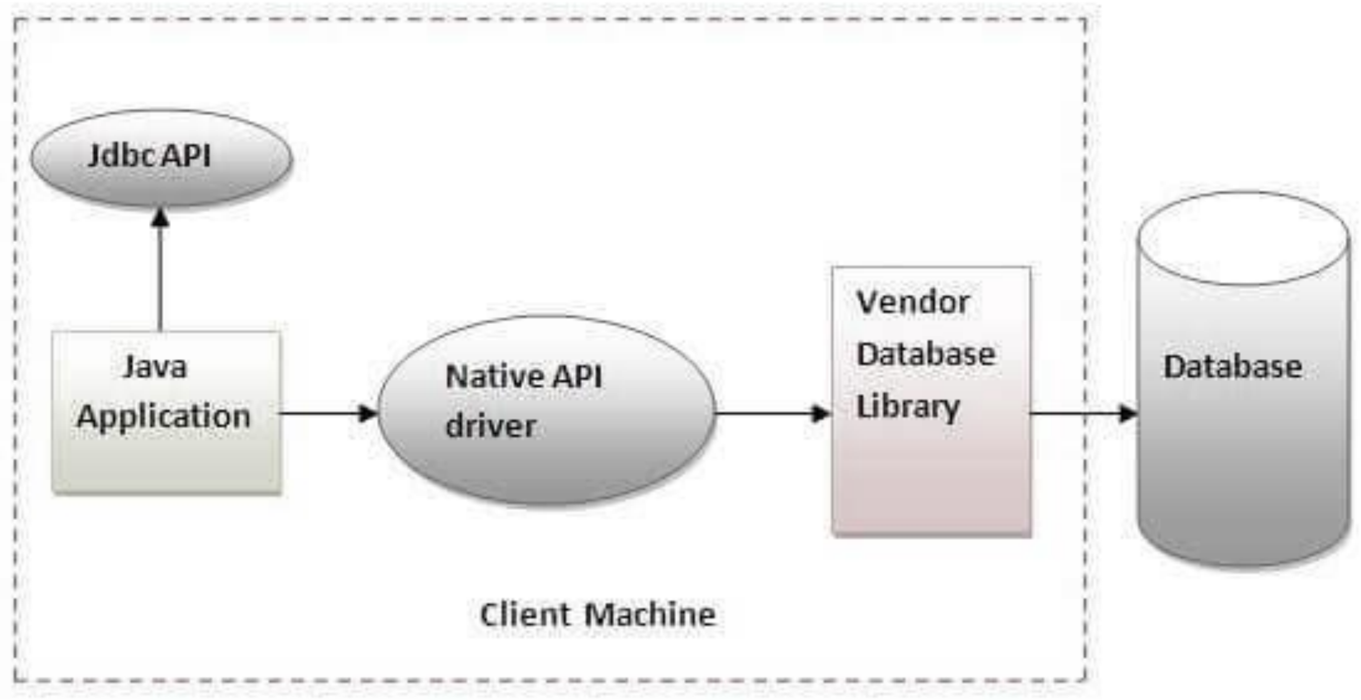
# JDBC-ODBC bridge driver

- The JDBC-ODBC bridge driver uses ODBC driver to connect to the database. The JDBC-ODBC bridge driver converts JDBC method calls into the ODBC function calls. This is now discouraged because of thin driver.



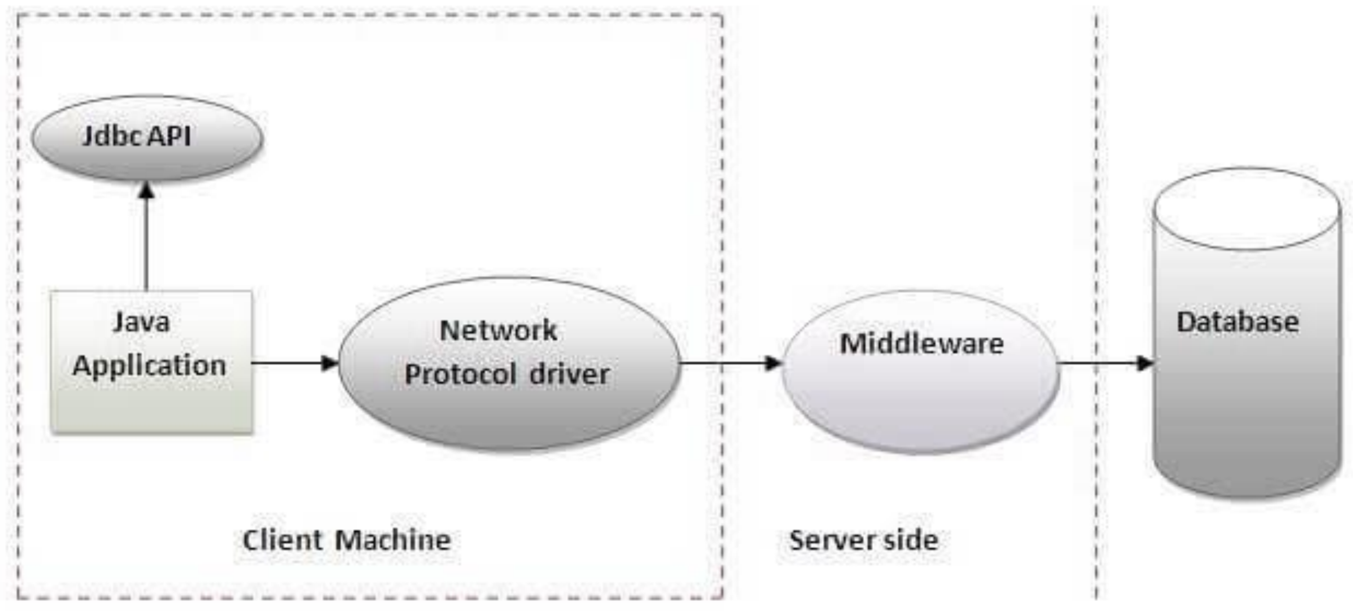
## Native-API driver

- The Native API driver uses the client-side libraries of the database. The driver converts JDBC method calls into native calls of the database API. It is not written entirely in java.



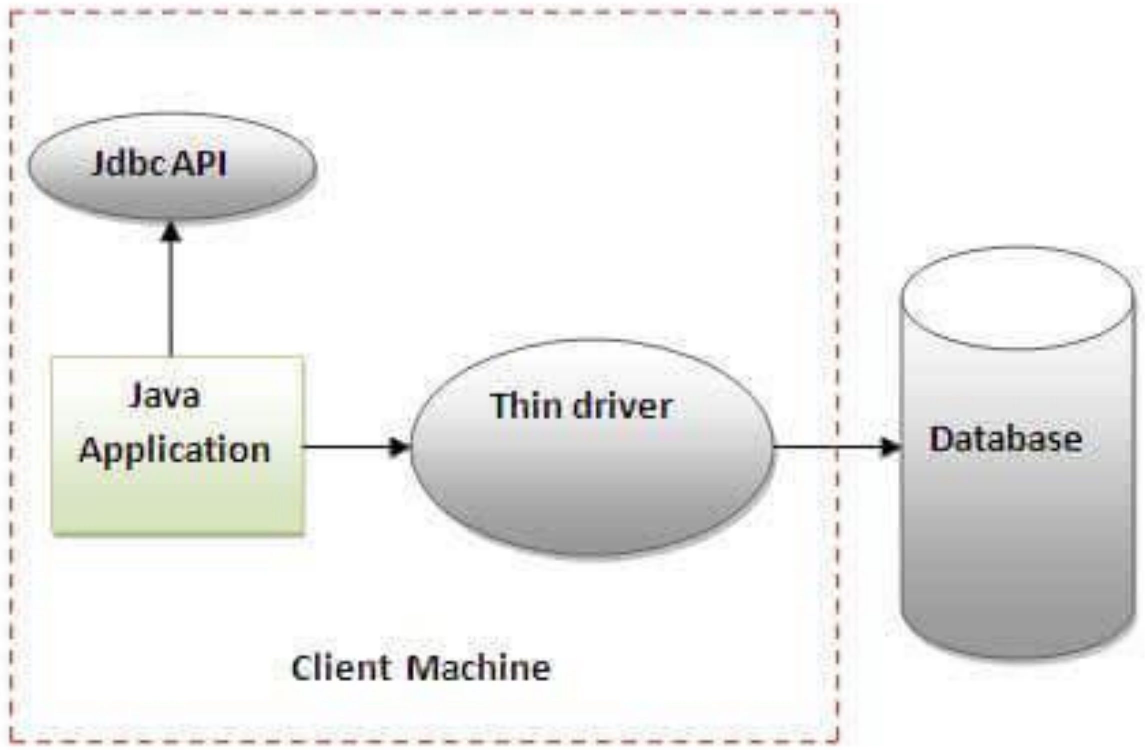
# Network Protocol driver

- The Network Protocol driver uses middleware (application server) that converts JDBC calls directly or indirectly into the vendor-specific database protocol. It is fully written in java.



# Thin driver

- The thin driver converts JDBC calls directly into the vendor-specific database protocol. That is why it is known as thin driver. It is fully written in Java language.



## Database Connection

- There are 5 steps to connect any java application with the database using JDBC. These steps are as follows:

    1. Register the Driver class
    2. Create connection
    3. Create statement
    4. Execute queries
    5. Close connection

## Connection to Oracle DB Example

```
import java.sql.*;
class OracleCon{

  public static void main(String args[]){
    try{
      //step1 load the driver class
      Class.forName("oracle.jdbc.driver.OracleDriver");
      //step2 create  the connection object
      Connection con = DriverManager.getConnection("jdbc:oracle:thin:@localhost:1521:xe","system","oracle");
      //step3 create the statement object
      Statement stmt = con.createStatement();
      //step4 execute query
      ResultSet rs = stmt.executeQuery("select * from emp");
      while(rs.next())
        System.out.println(rs.getInt(1)+"  "+rs.getString(2)+"  "+rs.getString(3));
      //step5 close the connection object
      con.close();
    }catch(Exception e){ System.out.println(e);}
  }
}
```

# Connection to SQL Server Example

```
import java.sql.*;
class OracleCon{

  public static void main(String args[]){
    try{
      //step1 load the driver class
      Class.forName(“com.microsoft.sqlserver.jdbc.SQLServerDriver");
      //step2 create  the connection object
      Connection con = DriverManager.getConnection("jdbc:sqlserver://localhost:1521","system",“sqlserver");
      //step3 create the statement object
      Statement stmt = con.createStatement();
      //step4 execute query
      ResultSet rs = stmt.executeQuery("select * from emp");
      while(rs.next())
        System.out.println(rs.getInt(1)+"  "+rs.getString(2)+"  "+rs.getString(3));
      //step5 close the connection object
      con.close();
    }catch(Exception e){ System.out.println(e);}
  }
}
```

# Insert Data Example

```
import java.sql.*;
class OracleCon{

  public static void main(String args[]){
    try{
      Class.forName("oracle.jdbc.driver.OracleDriver");
      Connection con = DriverManager.getConnection("jdbc:oracle:thin:@localhost:1521:xe","system","oracle");
      PreparedStatement stmt=con.prepareStatement("insert into Emp values(?,?)");
      stmt.setInt(1,101);//1 specifies the first parameter in the query
      stmt.setString(2,"Ratan");
      int i=stmt.executeUpdate();
      System.out.println(i+" records inserted");
      con.close();
    }catch(Exception e){ System.out.println(e);}
  }
}
```

## Delete Data Example

```
import java.sql.*;
class OracleCon{

  public static void main(String args[]){
    try{
      Class.forName("oracle.jdbc.driver.OracleDriver");
      Connection con = DriverManager.getConnection("jdbc:oracle:thin:@localhost:1521:xe","system","oracle");
      Statement stmt = con.createStatement();
      int result = stmt.executeUpdate("delete from emp765 where id=33");
      System.out.println(result+" records affected");
      con.close();
    }catch(Exception e){ System.out.println(e);}
  }
}
```

## Transaction Management in JDBC

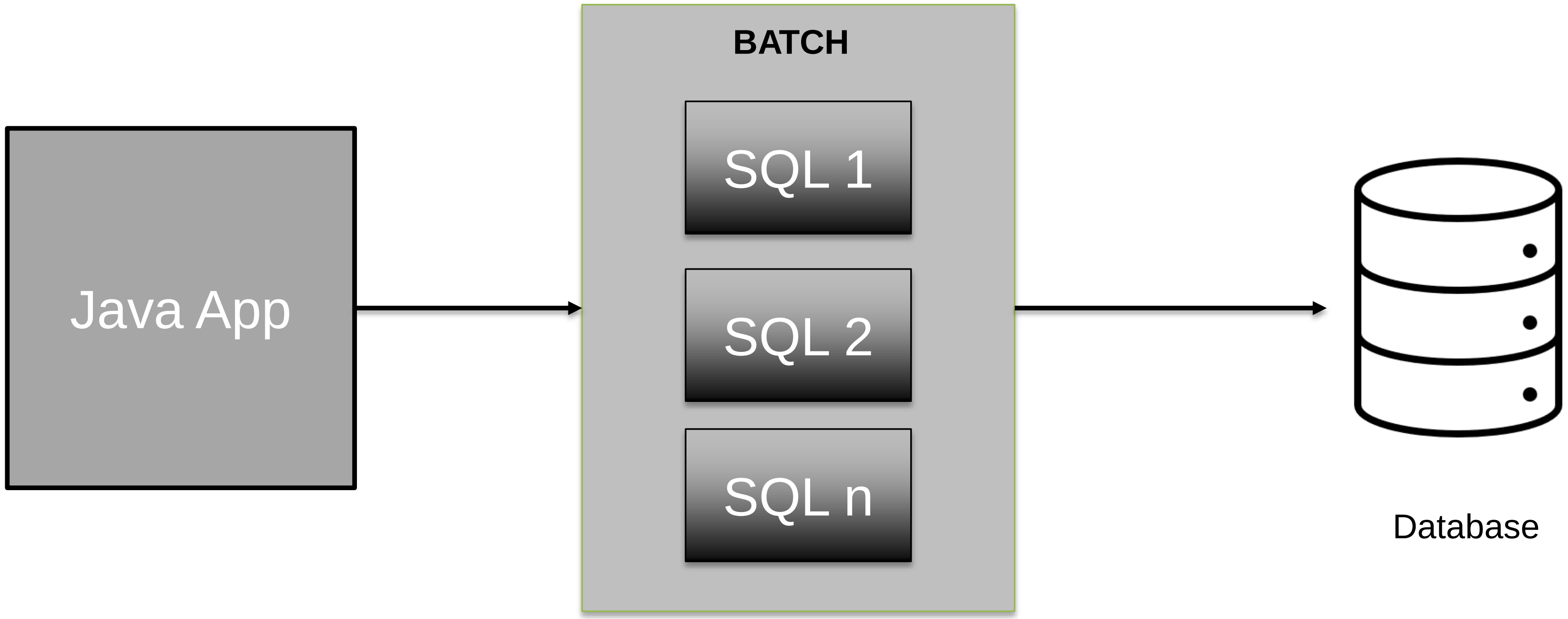
- Transaction represents a single unit of work.
- The ACID properties describes the transaction management well. ACID stands for Atomicity, Consistency, isolation and durability.
  - **Atomicity** means either all successful or none.
  - **Consistency** ensures bringing the database from one consistent state to another consistent state.
  - **Isolation** ensures that transaction is isolated from other transaction.
  - **Durability** means once a transaction has been committed, it will remain so, even in the event of errors, power loss etc.

## Transaction Example

```
import java.sql.*;

class FetchRecords{
  public static void main(String args[])throws Exception{
    Class.forName("oracle.jdbc.driver.OracleDriver");
    Connection con=DriverManager.getConnection("jdbc:oracle:thin:@localhost:1521:xe","system","oracle");
    con.setAutoCommit(false);

    Statement stmt=con.createStatement();
    stmt.executeUpdate("insert into user420 values(190,'abhi',40000)");
    stmt.executeUpdate("insert into user420 values(191,'umesh',50000)");

    con.commit();
    con.close();
}}
```

## Batch Processing in JDBC

- Sometimes data has be to be proceeded in batch modality, so transactions can be faster.



## Transaction Example

```
import java.sql.*;
class FetchRecords{
  public static void main(String args[])throws Exception{
    Class.forName("oracle.jdbc.driver.OracleDriver");
    Connection con=DriverManager.getConnection("jdbc:oracle:thin:@localhost:1521:xe","system","oracle");
    con.setAutoCommit(false);

    Statement stmt=con.createStatement();
    stmt.addBatch("insert into user420 values(190,'abhi',40000)");
    stmt.addBatch("insert into user420 values(191,'umesh',50000)");

    stmt.executeBatch();//executing the batch

    con.commit();
    con.close();
}}
```

## Configure DB Connection – 1/5

- Database connections can be stored directly on the web application, leaving Tomcat to handle only the driver storage.

- Follow the following steps for the configuration:

    - **Install the JDBC Driver** : Install the .jar file(s) containing the JDBC driver in Tomcat's $CATALINA_BASE/lib folder. You do not need to put them in your application's WEB-INF/lib folder.

## Configure DB Connection – 2/5

- **Create context.xml** : the file can be created on
  - META-INF/context.xml of your web application if you choose to save database configuration inside your application;
  - TOMCAT_HOME/conf/context.xml of tomcat folder if you choose to save database configuration inside your tomcat
- The preferred way is inside the Tomcat application because it separate concepts between system administrator and developers.
- In that way developers can connect to multiple datasources during developing phase

## Configure DB Connection – 3/5

```
<?xml version="1.0" encoding="UTF-8"?>
<Context>

  <Resource name="jdbc/WallyDB" auth="Container"
            type="javax.sql.DataSource" username="wally" password="wally"
            driverClassName="com.microsoft.sqlserver.jdbc.SQLServerDriver"
            url="jdbc:sqlserver://localhost;DatabaseName=mytest;SelectMethod=cursor;"
            maxActive="8"
            />

</Context>
```

- This example shows how to configure a DataSource for a SQL Server database named mytest located on the development machine. Simply edit the Resource name, driverClassName, username, password, and url to provide values appropriate for your JDBC driver.

## Configure DB Connection – 4/5

❑ **Add web.xml info:**

```
<web-app xmlns="http://java.sun.com/xml/ns/j2ee"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee
http://java.sun.com/xml/ns/j2ee/web-app_2_4.xsd"
    version="2.4">
  <description>MySQL Test App</description>
  <resource-ref>
      <description>DB Connection</description>
      <res-ref-name>jdbc/WallyDB</res-ref-name>
      <res-type>javax.sql.DataSource</res-type>
      <res-auth>Container</res-auth>
  </resource-ref>
</web-app>
```

## Configure DB Connection – 5/5

- ❑ **Access the DataSource in Your Application**

```
InitialContext ic = new InitialContext();
DataSource ds = (DataSource) ic.lookup("java:comp/env/jdbc/WallyDB");
Connection c = ds.getConnection();
...
c.close();
```

**Notice that, when doing the DataSource lookup, you must prefix the JNDI name of the resource with java:comp/env/**

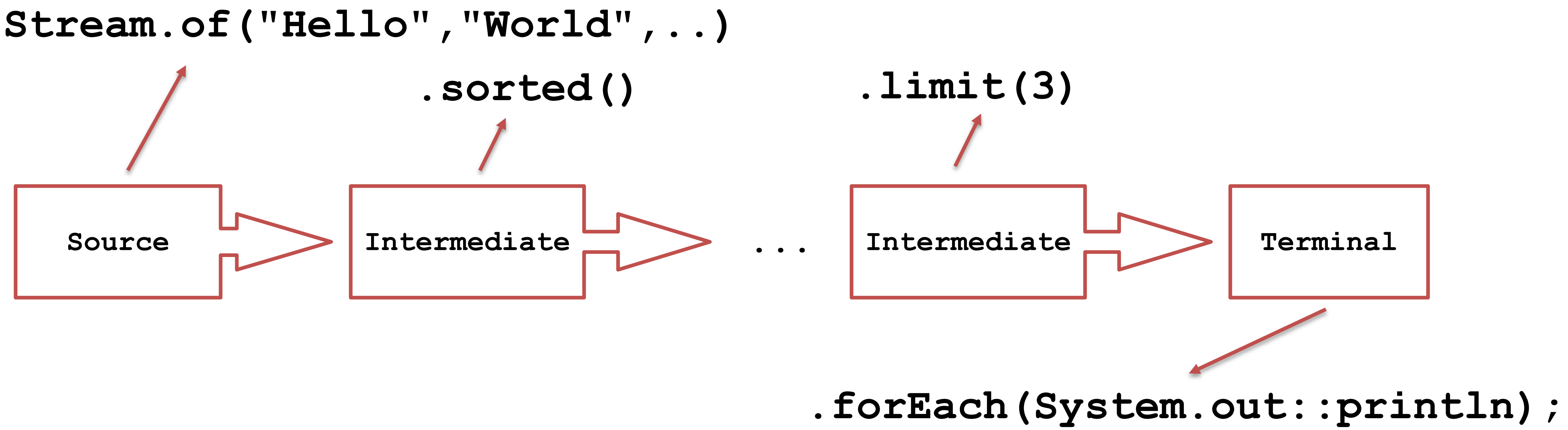
## Java Stream API

## Streams

- A sequence of elements from a source that supports data processing operations.
    - Operations are defined by means of behavioral parameterization

- Basic features
    - Pipelining
    - Internal iteration
        - no need to write explicit loops statements
    - Lazy evaluation (pull)
        - no work until a terminal operation is invoked

## Pipelining


Stream.of("Hello","World",..)
.sorted()
.limit(3)
Source
Intermediate
...
Intermediate
Terminal
.forEach(System.out::println);

## Streams != Collections

- No storage
  - not a data structure
  - conveys elements from a source, through a pipeline of computational operations, to a terminal operation
- Functional in nature
  - operations produce results, but does not modify its source
- Laziness-seeking
  - operation evaluated only when actually needed
  - many optimization opportunities
  - intermediate operations are always lazy

## Streams != Collections

- Possibly unbounded
    - streams do not ensure a finite size
    - infinite streams can be made finite, e.g. limit(n) or findFirst()
- Consumable
    - elements are only visited once (like an Iterator)
    - new stream needed to revisit the same elements

## Source operations

| Operation | Args | Purpose |
|---|---|---|
| `static Arrays.stream` | `T[]` | Returns a stream from an existing array |
| `default Collection.stream` | - | Returns a stream from a collection |
| `static Stream.of` | `T…` | Creates stream from the variable list of arguments |

# Stream source

- Arrays → **`Stream<T> stream()`**

```
String[] s={"Red", "Green", "Blue”}.
Arrays.stream(s).forEach(System.out::println)
```

- Stream of → **`static Stream<T> of(T... values)`**

```
Stream.of("Red", "Green", Blue”).forEach(System.out::println);
```

- Collection → **`Stream<T> stream()`**

```
Collection<Student> oopClass = new LinkedList<>();
oopClass.add(newStudent(100,"John","Smith"));
...
oopClass.stream().forEach(System.out::println);
```

## More…

- Parallelism

```
int sumOfWeights = widgets.parallelStream()
                          .filter(b -> b.getColor() == RED)
                          .mapToInt(b -> b.getWeight())
                          .sum();
```

  - pipeline is executed in parallel
  - check stream nature **`isParallel()`**
  - the orientation can be modified with
    - **`BaseStream.sequential()`**
    - **`BaseStream.parallel()`**

## Intermediate operations

- Mapping → transformation
    - `Map`
    - `FlatMap`

- Filtering
    - `Filter`
    - `Unique Elements`
    - `Limiting`
    - `Skipping`
    - …

- Sorting
- aggregate operations

# Terminal operations – predicate matching

- **`boolean anyMatch`**
    - Checks if any element in the stream matches the predicate
- **`boolean allMatch`**
    - Checks if all the element in thestream match the predicate
- **`boolean noneMatch`**
    - Checks if none element in thestream match the predicate
- **`Optional<T> findFirst`**
    - seturns the first element
- **`Optional<T> findAny`**
    - Returns any element
- **`Optional<T> min`**
    - Finds the min element base on the comparator argument
- **`Optional<T> max`**
    - Finds the max element base on the comparator argument

## Optional

- Optional represents a potential value

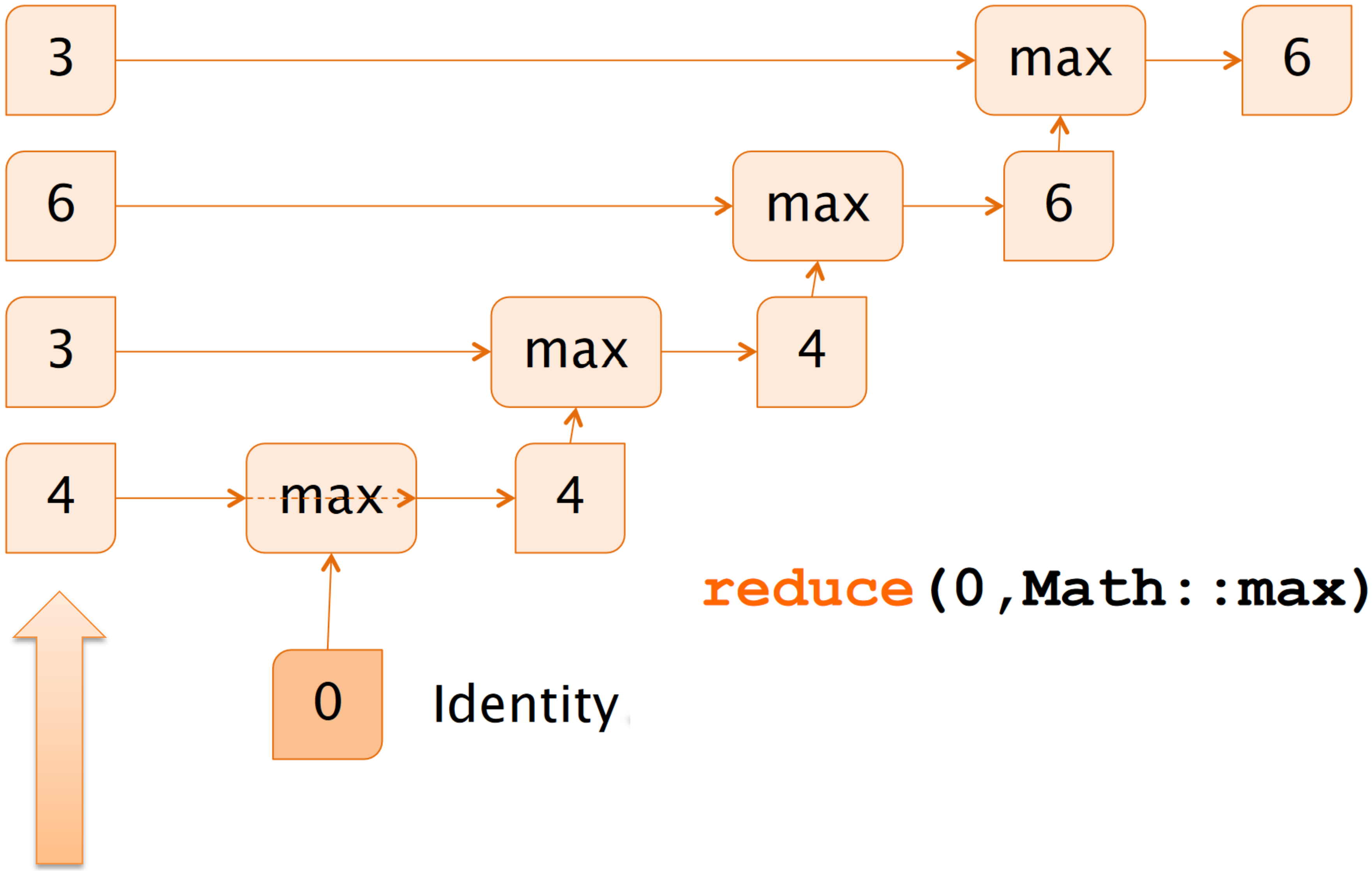
- Methods returning Optional<T> make explicit that return value may be missing
    - For methods returning a reference we cannot know whether a null could be returned
    - Force the client to deal with potentially empty optional

- access to embedded value through methods
    - isPresent(), ifPresent(), get(), orElse(supplier), orElseThrow(Exception)

# Terminal operations

- `forEach (Consumer<T>)`
    - Consumes each element from a stream and applies a operator to each of them
- `count`
    - Returns the number of elements in a stream as a long
- `reduce(T, BinaryOperator<T>)`
    - Reduces the elements using an identity value and an associative merge operator
- `collect(Collector<T,A,R>)`
    - Reduces the stream to create a collection such as a List, a Map, or even an Integer

# Reduce



`reduce(0,Math::max)`

## Collect

- **`Stream.collect()`**
    - applies a recipe for accumulating stream’s elements into a summary result
    - stateful operation

- Typical recipes available to
    - Summarize (reduce)
    - Accumulate
    - Group or partition

## Collector

```
interface Collector<T,A,R>{

   // Creates the accumulator container
   Supplier<A> supplier()

   //Adds a new element into the container
   BiConsumer<A,T> accumulator();

   // Combines two containers (used for parallelizing)
   BinaryOperator<A> combiner();

   // Performs a final transformation step
   Function<A,R> finisher();

}
```

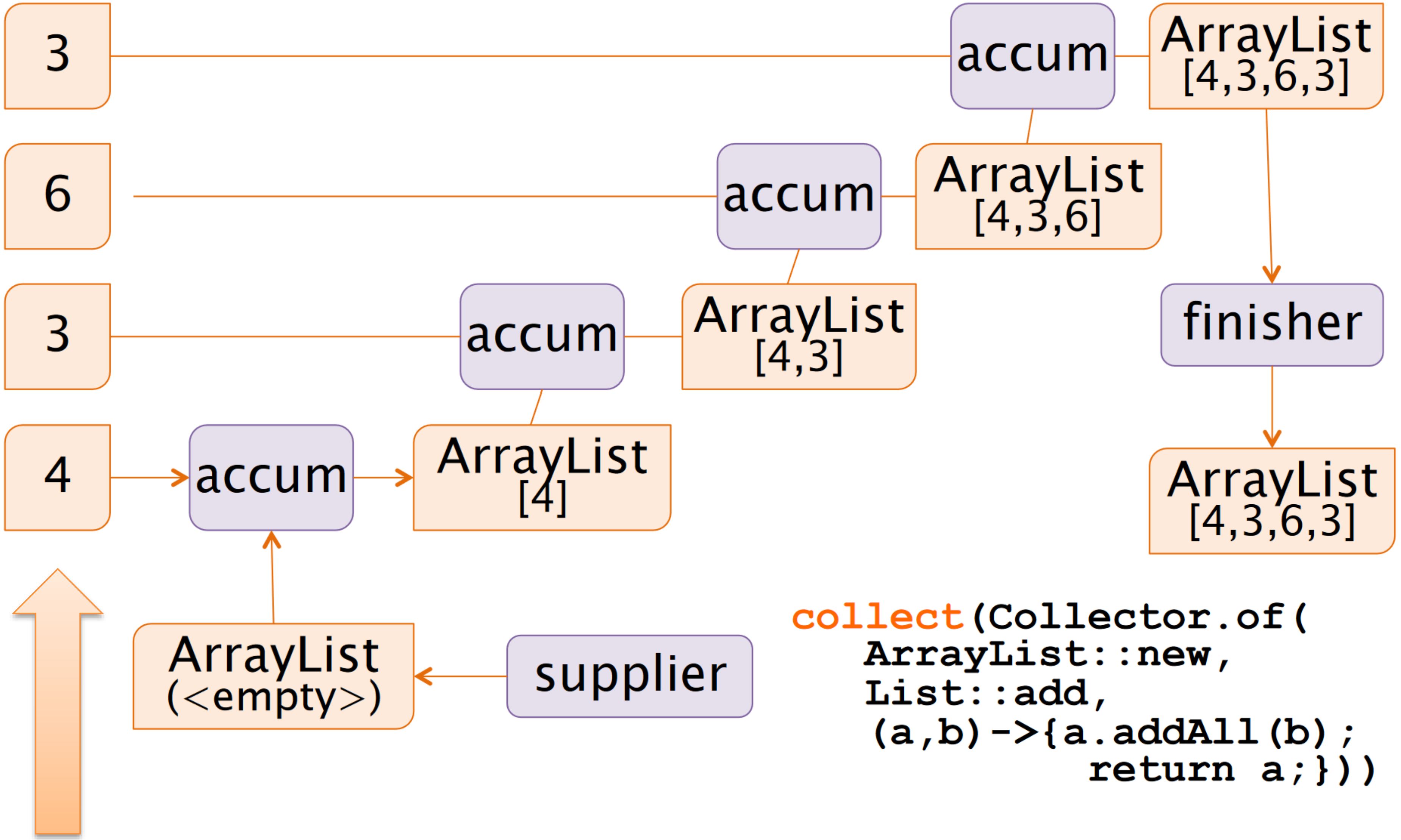
## Collector

```
class addToList<T> implements Collector<T,List<T>,List<T>>{

      public Supplier<List<T>> supplier(){
         return ArrayList<T>::new;}

      public BiConsumer<List<T>,T> accumulator(){
         return ArrayList<T>::add;}

      public BinaryOperator<List<T>> combiner() {
         return(a,b)->{a.addAll(b); return a;}; }

      public Function<List<T>,List<T>> finisher (){
         return Function.identity(); }

}
```

## Collector

```
Collector<Student, List<Student>,List<Student>> ctl =

    Collector.of(
            ArrayList::new,                      ← supplier (kinda initializer)

            List::add,                           ← accumulator (collect next element)

            (a,b)->{a.addAll(b);return a;}       ← combiner (combine partial/parallel results)

            () -> Function.identity()            ← implicit finisher
            );
```

## Collect



```
collect(Collector.of(
    ArrayList::new,
    List::add,
    (a,b)->{a.addAll(b);
            return a;}))
```

Grazie

http://www.reti.it/
https://academy.reti.it/

Via  Dante, 6, 21052 Busto Arsizio VA
Tel.: +39.0331.357.400

Reti