CS209

Computer system design and application

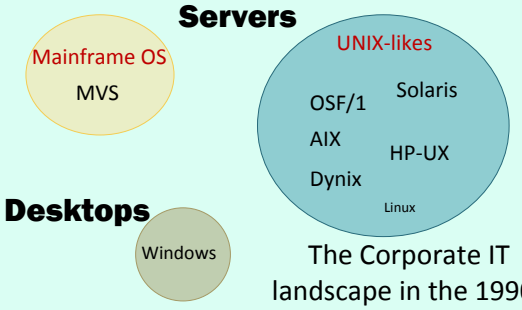
Stéphane Faroult
faroult@sustc.edu.cn

Zhao Yao zhaoy6@sustc.edu.cn
Liu Zijian liuzijian47@163.com
Li Guansong intofor@163.com

Java and the web

The last big topic of this course will be Java and the web. Complicated history, and an alphabet-soup of products. Let's start with history.

When Java appeared (and it hasn't changed that much since then, if you omit that Linux has taken a far bigger place and that many other Unixes have disappeared) this is what you found in most big companies.

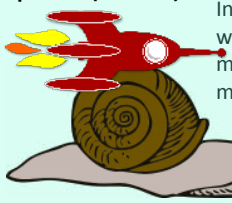


The Corporate IT landscape in the 1990s

1996 Write once, run everywhere

Gosling's promise of a language allowing to run the same .class on a lot of different computers running different systems was most attractive.

Just-in-Time Compiler (1997)
Hot-Spots (1999)

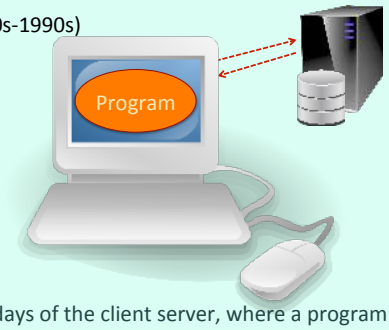


In the beginnings Java was extremely slow, but many enhancements made it a language that you could consider for business programs without looking like a clown.

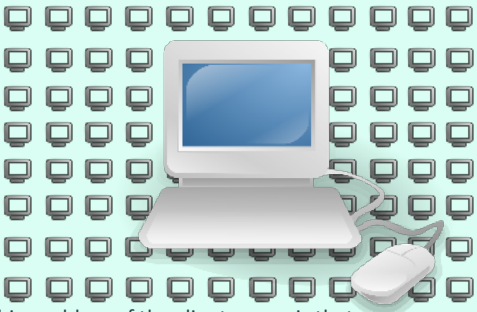
Reasonably slower than C

Client-server

(popular 1980s-1990s)



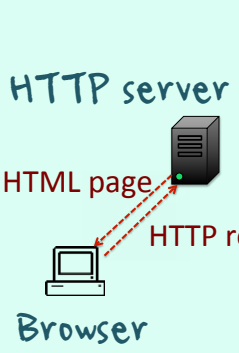
These were the days of the client server, where a program running on a desktop was talking to a (database) server.



One big problem of the client server is that every new release of a program had to be installed on perhaps thousands of computers.

Then the Web and the HTTP protocol came

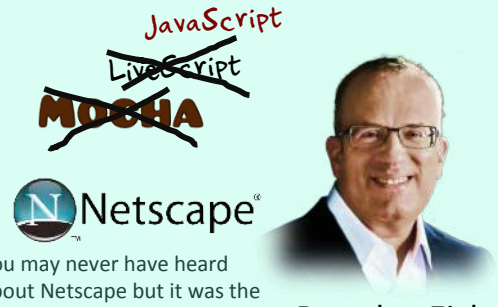
In 1990 Tim Berners-Lee invented the HTTP protocol, and the rest is history. Initially getting a static page from a web server wasn't much of an improvement (except that graphically is was nicer) over the dumb terminal that had preceded the desktop client.
But people thought that you could not only download pages, but perhaps also programs that could be run by the browser.



Put a some bits of programs between the tags, when you just need to update pages on the server and everybody will instantly get the new version.

Text between tags

Small programs between tags

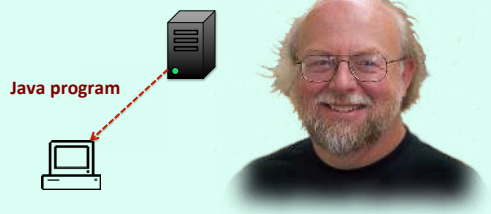


Brendan Eich

You may never have heard about Netscape but it was the first "modern" browser, and in this company Brendan Eich coded, the legend says in 10 days, a crap^H^H^H^Hlight programming language that, after some name changes, became known as Javascript.

Write once, run everywhere

Javascript was kind of inspired by Java, but that's all. Not even a cheap imitation.



Meanwhile, James Gosling was seeing his language as the perfect candidate for being downloaded. There was just one problem: Java is too powerful, and there was a security risk.

The Java security model

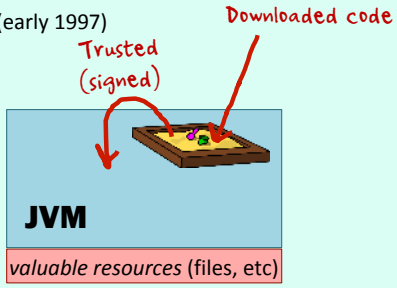
Verifier in the class loader

But a program could do nasty things on your computer!

People thought of running a Java "applet" (small application) in an isolated environment (sandbox)

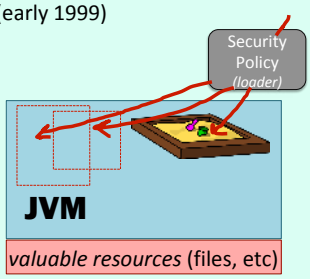
In a sandbox, you cannot, for instance, access files.

JDK 1.1 (early 1997)



But then, there was little benefit using Java compared to the easier Javascript. So people began to think about exceptions.

JDK 2.0 (early 1999)



And it became increasingly complicated. Rule of thumb: when things become too complicated, people begin to dump it for something perhaps not as good but easier to understand.

```
<script src="https://www.java.com/js/deployJava.js"></script>

    <applet code = "AppletName"
        archive = "AppletIsInside.jar"
        width = 300
        height = 300>
       <param name="permissions"
              value="sandbox" />
    </applet>
```

Additionally, permissions were set in the page. If I'm a really bad guy, I can set-up a server, and give all the permissions I want to my nasty code.

Meanwhile

hardware acceleration in browsers
(*around 2010/2011*)

Lots of excellent Javascript graphics libraries

HTML 5

Meanwhile, browser and Javascript improvements killed the one good reason (graphics) that people had of using a Java applet instead of Javascript. To make a long story short, Java applets are dead today, and no longer supported by major browsers.

What runs in the browser

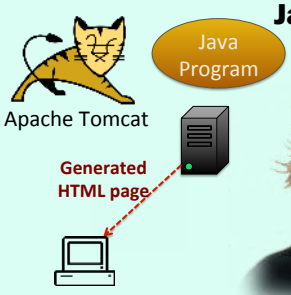
Front-end

Javascript frameworks (AngularJS, ReactJS, ...)

On the front-end, you'll mostly find JavaScript, mostly used in frameworks (Jquery used to be very popular, but there are new flavors-of-the-day).

But the front-end is only half the story ...

Java Server Pages
Servlet



Java is very much used for **generating** pages on a server (pages that may include references to Javascript). There have been several generations and technologies, and even a server (Tomcat) dedicated to Java applications.

Installation directory of Tomcat

TOMCAT_DIR
- bin
- conf
- lib
- logs
- temp
- webapps
- work

server.xml

A Tomcat installation looks like this. The port (8080 by default) is specified in server.xml.

TOMCAT_DIR
- bin
- conf
- lib
- logs
- temp
- webapps
  - ROOT
  - docs
  - examples
  - host-manager
  - manager
  - sample
  - MyServlet
    - WEB-INF
- work

Applications (.class programs generating HTML, mostly) are stored under the webapps directory.

Java Servlet API

`javax.servlet`

Web Container (also called *Servlet Container*)

URL

What Tomcat does is matching an URL that it receives with a .class program that will build the page.

Servlet

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

@WebServlet(name="MyServlet", urlPatterns={"/test"})
public class MyServlet extends HttpServlet {
```

NOT in default Java libraries!

This program must extend HttpServlet. You'll notice the heavy use of annotations, including one that says for which pattern this program should be called.

```
  public void doGet(HttpServletRequest  request,
                    HttpServletResponse response)
              throws IOException, ServletException {
    response.setContentType("text/html");
    PrintWriter pw = response.getWriter();
    try {
      pw.print("<html>");
      pw.print("<head>");
      pw.print("<title>My First Servlet</title>");
      pw.print("</head>");
      pw.print("<body>");
      pw.print("<h1>Yeepee it works!</h1>");
      pw.print("</body>");
      pw.print("</html>");
    } finally {
      pw.close();
    }
  }
}
```

Inside it, you just write HTML pages to a "PrintWriter" which is basically a text stream that will be sent back to the browser.

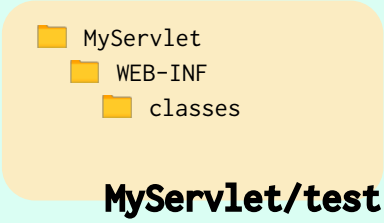
When you compile (same problem as with Jsoup) you must provide the location of the .jar that contains the HTTP stuff.

Replace with real location

```
javac -cp .:TOMCAT_DIR/lib/servlet-api.jar MyServlet.java
```

javax.servlet.* is HERE

Annotations provide everything Tomcat needs. If you are allergic to annotations, you can also write an .xml file.




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

"Application name" "Page name"

Re-doing the film database query with a servlet

Some changes

- Protocol formats results to a HTML table
- Writes directly to PrintWriter

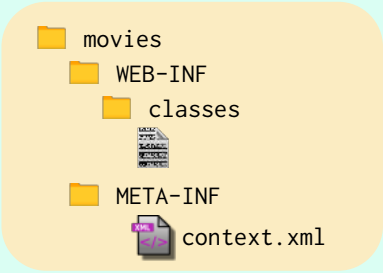
If you remember the "film server" created when talking about networking, I have redone it with a servlet.

```
...
ResultSetMetaData info = rs.getMetaData();
int               cols = info.getColumnCount();
out.print("<table class=\"films\"><tr class=\"headers\">");
for (int i = 1; i <= cols; i++) {
  out.print("<th>"+info.getColumnLabel(i)+"</th>");
}
out.print("</tr>\n");
while (rs.next()) {
  out.print("<tr>");
  for (int i = 1; i <= cols; i++) {
    if (rs.getString(i) != null) {
      out.print("<td class=\"film_col" + Integer.toString(i)
                          +"\">"+rs.getString(i)+"</td>");
    } else {
      out.print("<td></td>");
    }
  }
  out.println("</tr>");
}
...
```

I just output HTML with tags suitable for CSS formatting.

One nice feature is that Tomcat manages the database connection thanks to a component called JNDI.

Java Naming and Directory Interface



context.xml

I have briefly mentioned DataSources already, here they are again.

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

<Context>
  <Resource name="filmdb" type="javax.sql.DataSource"
            maxTotal="10" maxIdle="5"
            maxWaitMillis="3000"
            url="jdbc:sqlite:path_to_sqlite_file"
            driverClassName="org.sqlite.JDBC"
  />
</Context>
```

TOMCAT_DIR

lib

JDBC driver (.jar file)

```
import java.io.IOException;
import java.io.PrintWriter;
import java.util.ResourceBundle;

import javax.servlet.ServletException;
import javax.servlet.http.*;
import javax.servlet.annotation.*;
import javax.naming.InitialContext;
import javax.naming.NamingException;
import java.sql.Connection;
import java.sql.SQLException;
import javax.sql.DataSource;


@WebServlet(name="movies", urlPatterns={"/query"})
public class FilmServlet extends HttpServlet {
```

Let's create the Servlet. Lot of imports as usual, JDBC (of course) and also extended JDBC (javax.sql) for the DataSource.

```
  private DataSource dataSource;
  private Connection con;

  public Connection getConnection(String jndiname)
         throws SQLException {
    try {
      dataSource = (DataSource)new
                   InitialContext().lookup("java:comp/env/"
                                           + jndiname);
    } catch (NamingException e) {
      // Handle error that it's not configured in JNDI.
      throw new IllegalStateException(jndiname
                                + " is missing in JNDI!", e);
    }
    return dataSource.getConnection();
  }
```

Connection is just "looking for a service" (in context.xml) that supplies all the right parameters.

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

    response.setContentType("text/html");
    response.setCharacterEncoding("UTF-8");
    PrintWriter out = response.getWriter();

    try {
      con = getConnection("filmdb");
      con.setAutoCommit(false);
    } catch (Exception e) {
      out.println(e.getMessage());
    }
```

My service is called "filmdb" (you may want to check context.xml a few slides back)

```
FilmProtocolHTML filmP = new FilmProtocolHTML(con, out);
out.println("<!DOCTYPE html><html>");
out.println("<head>");
out.println("<meta charset=\"UTF-8\" />");
out.println("<title>Film Database Query</title>");
out.println("</head>");
out.println("<body>");

out.println("<h3>Film Database</h3>");
out.println("<p>");
out.println("<form action=\"query\" method=POST>");
```

I pass my output stream to the (new) FilmProtocolHTML because it will write the rows to it as it retrieves them (much more efficient than loading a collection and passing it back). A "POST" query can be used when you send data (you normally use it whenever you want to CHANGE a database)

```
out.println("<form action=\"query\" method=POST>");
out.println("Title");
out.println("<input type=text size=80 name=\"title\">");
out.println("<br/>");
out.println("Director");
out.println("<input type=text size=40"
            + "        name=\"director\">");
out.println("<br/>");
out.println("Actor/Actress");
out.println("<input type=text size=40 name=\"actor\">");
out.println("<br/>");
out.println("Country");
out.println("<input type=text size=15"
            + "        name=\"country\">");
out.println("<br/>");
out.println("Year");
out.println("<input type=text size=4 name=\"year\">");
out.println("<br/>");
out.println("<input type=submit>");
out.println("</form>");
```

Input form

I'm passing here a command that follows the protocol defined in the client/server example

```
// If parameters were provided, execute the query
String title = request.getParameter("title");
String director = request.getParameter("director");
String actor = request.getParameter("actor");
String country = request.getParameter("country");
String year = request.getParameter("year");
StringBuffer theCommand = new StringBuffer();
if ((title != null) && (title.trim().length() > 0)) {
   theCommand.append("TITLE " + title);
}
if ((director != null)
    && (director.trim().length() > 0)) {
  if (theCommand.length() > 0) {
    theCommand.append(',');
  }
  theCommand.append("DIRECTOR " + director);
}
```

```
if ((actor != null) && (actor.trim().length() > 0)) {
  if (theCommand.length() > 0) {
    theCommand.append(',');
  }
  theCommand.append("ACTOR " + actor);
}
if ((country != null) && (country.trim().length() > 0)) {
  if (theCommand.length() > 0) {
    theCommand.append(',');
  }
  theCommand.append("COUNTRY " + country);
}
```

The form isn't as flexible as what my "language" allows (remember I could say "or" as well as "and") but it's easier for an end-user.

```
    if ((year != null) && (year.trim().length() > 0)) {
      if (theCommand.length() > 0) {
        theCommand.append(',');
      }
      theCommand.append("YEAR " + year);
    }
    if (theCommand.length() > 0) {
      filmP.processInput(theCommand.toString());
    }
    out.println("</body>");
    out.println("</html>");
  }
```

When I'm done I call the protocol that retieves rows (if it finds something) or displays an error message or whatever, and I just terminate the page. I have reused what had previously been done with minimal transformation because I'm lazy.

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

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

}
```

I'm defining "Post" to be the same as "Get"

```
  public void doGet(HttpServletRequest  request,
                    HttpServletResponse response)
              throws IOException, ServletException {
    response.setContentType("text/html");
    PrintWriter pw = response.getWriter();
    try {
      pw.print("<html>");
      pw.print("<head>");
      pw.print("<title>My First Servlet</title>");
      pw.print("</head>");
      pw.print("<body>");
      pw.print("<h1>Yeepee it works!</h1>");
      pw.print("</body>");
      pw.print("</html>");
    } finally {
      pw.close();
    }
  }
}
```

My Servlet is mostly a Java program that writes HTML.

HTML inside Java

OK if the web site is done by a single guy

Problem of division of labor otherwise

If I have one guy working on say data retrieval and another guy working on nice webpages, it may be inconvenient.

The design guy might want to add classes to the HTML tags.

Designer

TEMPLATES

The solution for this problem (which has been adopted in several languages) is to use templates (patterns, models). A template defines the global looks of a page, and is what the designer works on. Instead of writing the page from Java, we take an opposite approach and call bits of Java from inside a HTML page. Welcome to Java Server Pages (JSP).

You have similar technologies with other languages. A Servlet is more like what you can do with CGI, which includes a lot of things, including some Python frameworks.

Servlet = HTML in Java

Similar to CGI/Fast-CGI (Common Gateway Interface)

JSP = Java embedded in HTML

Similar to PHP

JSP looks more like PHP, or a product called ColdFusion – Special tags in a HTML page are processed by a module that reads the template.

```
<html>
   <head>
      ....
   </head>
 <body>

  <% Java code %>
      <div>
         ....
      </div>
  <% Java code %>

  </body>
</html>
```

Scriptlet

You can put bits of Java between <% and %> tags.

```
<html>
   <head>
      ....
   </head>
 <body>

  <%  if (var == 0) { %>
      <div>
         ....
      </div>
  <%  } %>

  </body>
</html>
```

It can even behave a bit like the C preprocessor (for those familiar with the C preprocessor ...). Java code may decide of what will remain of the HTML in what will be sent back to users.

```
<html>
  <head>
    <title>Very, very basic JSP</title>
  </head>
  <body>
      <h1>The time is <%= new java.util.Date() %></h1>
  </body>
</html>
```

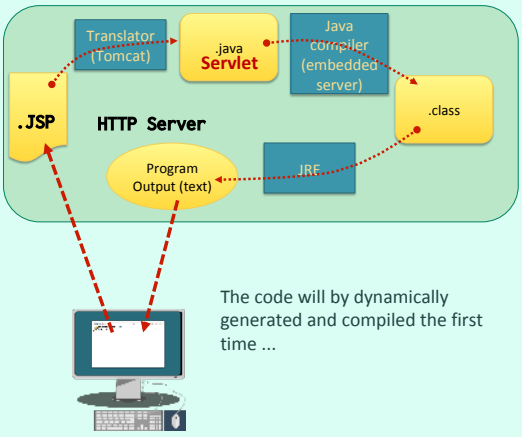
TOMCAT_DIR
- webapps
  - jspdemo
    - WEB-INF
    - date.jsp

Kind of useless demo application. Not completely useless: if you get the time in Javascript, it's the time where you are. Here it's the time on the server, which may be in a different time zone.

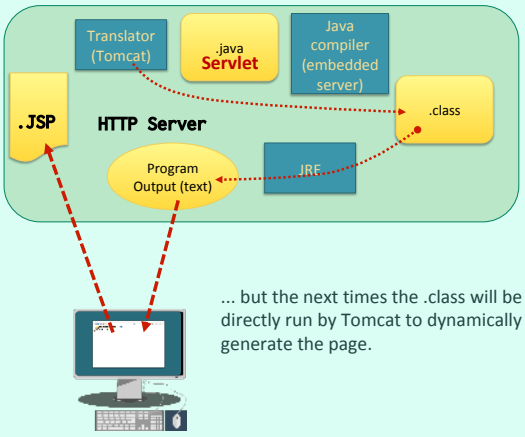
http://localhost:8888/jspdemo/date.jsp

What is really happening?

How is this done? By a number of software components that work together to build your page.



The code will by dynamically generated and compiled the first time ...



... but the next times the .class will be directly run by Tomcat to dynamically generate the page.

DEPLOYMENT

The Art of .war

(Web application Archive)

Extended .jar

For installing applications you use some .war files, which are kind of .jar files specialized for web applications.

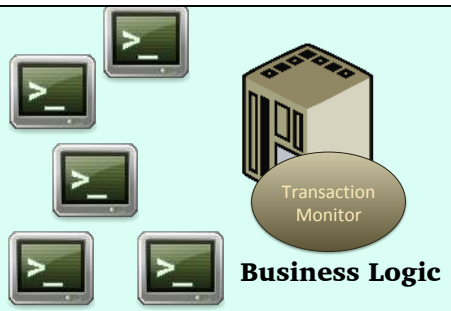
```
.class
.jsp
.xml
.html
```

and so forth

But Java-powered web applications evolved into something ...

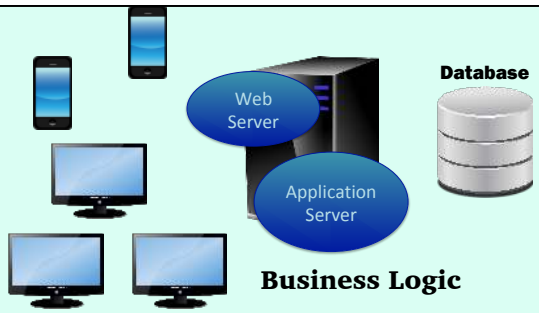
... based on an old idea.

What you have seen so far is enough for running a moderaely complex website, but for big popular websites (as well as for applications used by many employees in big companies – think about all the branches of a large retail bank) something more complex was devised.



... based on an old idea.

In the very old days of the dumb terminal, accesses to shared resources was managed by a "transaction monitor" that hosted all the "business logic" (functions performing business operations)

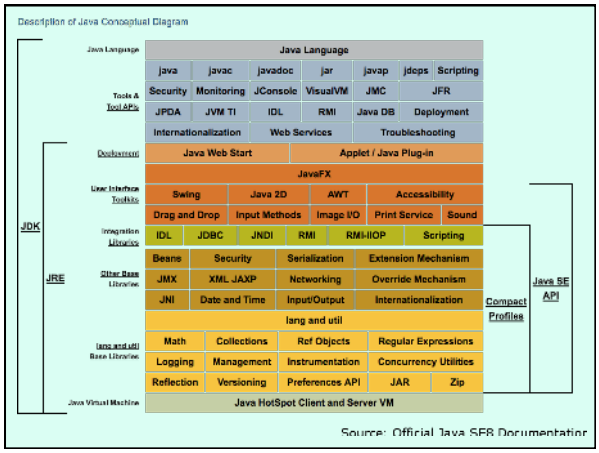


... based on an old idea.

The idea was dusted off to say "we'll have an application server distinct from the HTTP server and the database server that will contain the business logic".

Java Standard Edition

What you have been working with so far is known as "JSE", or Java Standard Edition. The following diagram comes from the Java documentation and describes JSE as a whole. I hope that you'll recognize more than a few components.



Not complicated enough ...

Far too simple for big companies. What big companies wanted (and what software vendors wanted to sell them) were ready-made components that could be reused and plugged into each other (think of Lego bricks). Application servers would mostly be the glue allowing all these components to work together. There are a few application servers that are popular on the market, Websphere (IBM), WebLogic (formerly BEA systems, bought over by Oracle that also bought Sun, owner of Java) and WildFly, formerly known as Jboss and bought by RedHat, better known for Linux distributions.

Need to inter-operate

Component-based architecture

Lego bricks are designed for interlocking. If we want software components to integrate without effort, they must be cleanly designed.

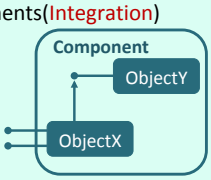
Component = Logical Processing Unit

Goal : modularity and reuse

Properties

Named, listed in a directory (Identification)
Usable alone (Independence)
Usable in different contexts (Reuse)
Can be combined with other components(Integration)

This basically lists the desirable qualities of a good software component.



Java Component : Java Bean

class

Specific Properties

Serializable

Default Constructor

Private Properties

Getters/Setters

In Java, components are called Beans. We have already encountered them in JavaFx, with TableViews and ListViews. Methods must have well-defined names so that they can be called automatically.

```
public <returntype> get<PropertyName>()
public void set<PropertyName> (parameter)
```

Java Component : Java Bean

class

Specific Properties

Methods for catching events

Use of listeners and event generation
For instance PropertyChangeListener

Beans must react to events (remember that ListViews, for instance, are backed by a collection and must refresh if the collection is modified)

```
public class InvoiceBean implements Serializable {
   private String  customer;
   private double  amount;
   private boolean paid;

   public InvoiceBean() { }

   public String getCustomer() {
     return this.customer;
   }

   public void setCustomer(String customer) {
     this.customer = customer;
   }

   public boolean isPaid() {
     return this.paid;
   }

   public void setPaid(boolean paid) {
     this.paid = paid;
   }
   ...
}
```

This is a boring example of Bean to implement accounting operations. Serializable, private properties, default constructor, getters and setters.

Java Enterprise Edition

Java EE JEE J2EE

A set of specifications

Defined by Sun microsystems

now owned by ORACLE

JEE is basically a set of standards.

Agreed to by multiple international (mostly US) companies www.jcp.org

Java Enterprise Edition

A set of specifications

Based on Java SE

+ Application Server specs

Libraries for the development of business applications (APIs)

These standards defines how components should work together.

Reference Implementation: the GlassFish application server (Open Source)

Based on Java Beans.

But you have better than the humble Java Bean: the Enterprise Java Bean!

Deal with:

Enterprise JavaBeans

Transactions

Concurrency

Messaging

Scheduling

and so forth.

EJB

It includes the support of many functions that are important in business applications.

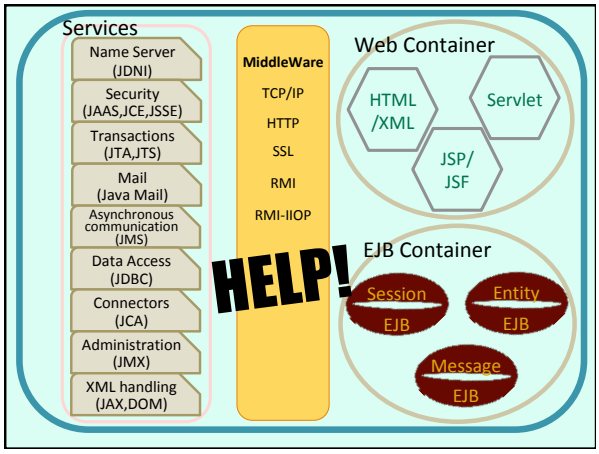
Different areas of responsibility for beans

Presentation	User interface	View
Processing	Logic	Controller
Resources	Data management	Model

Where things become ugly is that basically you don't always know where the code is running.

V V C M

But in practice almost everything can be anywhere!



What is the purpose of containers?

They simplify set-up

"managed beans"

@ManagedBean

@SessionScoped

What containers do is that they use reflection to read annotations and generate or call the necessary code. The alternative is writing configuration information into xml files – you have seen en excellent example with the Tomcat servlet example.

Java Server Faces (JSF)

Framework relying on managed beans

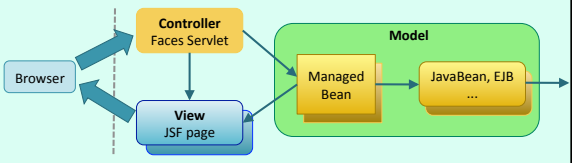
"Faces Servlet" used as controller

XHTML templates (used to be JSP)

Then the Java folks invented JSF, to try to better organize applications.

JSF separates the view from logic and data, both parts of the "model"

model = logic + data



JSF Allows to build easily data-entry forms (not the most interesting thing you can code)

Input Form ⟷ Managed Bean

properties (getters/setters)
One pair per input element

Action controller
one per form button

Called automatically

Place-holders for result data
(only getters)

NOT called automatically

As everything was getting a bit out of hand, some other folks created other development frameworks that have become quite successful.

There are other frameworks

spring by Pivotal — Integrates

Struts — Web side only
Navigates between .jsp

spring by Pivotal — Light Container

Two important ideas

Inversion of Control (IoC)

Aspect Oriented Programming (AOP)

Spring in particular stresses two things, which are yhe inversion of control (dependency injection) and "aspect oriented programming" which is about functions that aren't linked to ONE type of business operation but are often used.

Inversion Of Control
(Dependency Injection)

I have talked about dependency injection already, the idea that you get references instead of creating objects.

```
public class Car {
   private Engine engine;

   // No IoC: the constructor creates a specific engine
   public Car() {
       engine = new CombustionEngine();
   }

   // Engine passed to constructor  - IoC compatible
   public Car(Engine engine) {
       this.engine = engine;
   }
}
```

Inversion Of Control
(Dependency Injection)

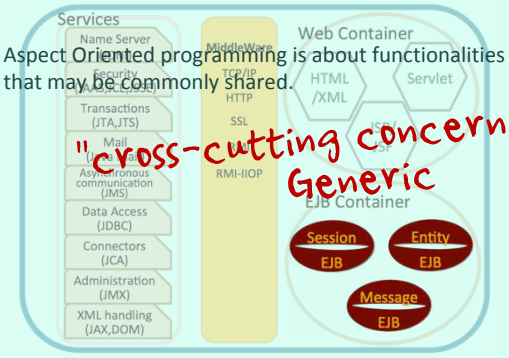
You no longer need to know what the constructor looks like and which parameters it takes – you only need to know the methods from the object that you need to call.

"Don't call us, we'll call you"

Easier to change

Easier to test

Aspect Oriented Programming



Aspect Oriented programming is about functionalities that may be commonly shared.

"cross-cutting concern"

Generic

Aspect Oriented Programming

Many business-functions are "cross-cutting"

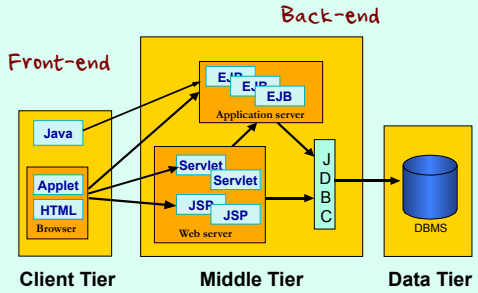
- Security
- Transaction
- Logging

A lot of business functions need this. The idea is to say "when the name of the method looks like this, then we are going to generate suitable code for this function". It's based on annotations and reflection.

As you can see, it can become VERY complicated.

Architectures are very often monsters. You have a lot of products, some of them are very trendy one year, the next year the hot product is something else ...

Fortunately Applets are now mostly gone, but JEE looks like this. And the trouble is that basically the code for your application can go anywhere, adn not only at one place.

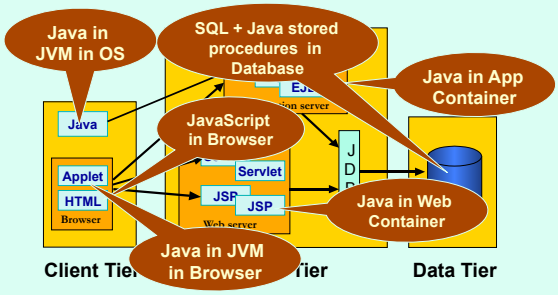


J(2)EE Architecture

Based on slides by Toon Koppelaars

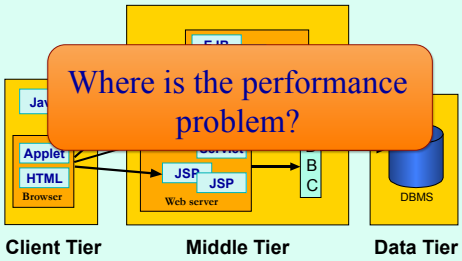
Depending on what they master best, people may write their code at many places.

You can code everywhere!



Based on slides by Toon Koppelaars

When you have performance problems or bugs, finding where to look at becomes horribly complicated, especially in a team with different people in charge of different parts, who are quick to blame the other guys.



Based on slides by Toon Koppelaars

One principle to always keep in mind is "KISS", which of course isn't what it looks.

Keep It Simple, Stupid!

When everything becomes too complicated, it's a very very bad sign. When people are only using acronyms and jargon to talk, it's also often a bad sign. It's far easier to do something complicated than something simple. Remember what St Exupéry wrote "*perfection is attained, not when there is nothing to add, but when there is nothing to remove*".

Traditional MVC

Popular *Design Pattern*

"Separate code into 3 parts"

Many Java architectures follow a pattern that was mostly popularized by a Dane called David Heinemeier Hansson, author of a (non Java) development framework called "Ruby On Rail".

Don't get me started me on Ruby on Rail, it's the nightmare of the database guy.

Based on slides by Toon Koppelaars

Traditional MVC

Popular *Design Pattern*

"Separate code into 3 parts"

1 **Model**

Handling business logic

Data retrieval & manipulation
Including all involved business rules

Based on slides by Toon Koppelaars

Traditional MVC

Popular *Design Pattern*

"Separate code into 3 parts"

2 **View**

creating user interface (UI)

Front-end generation: html/xml in browser
The LOOK of the application

Based on slides by Toon Koppelaars

Traditional MVC

Popular *Design Pattern*

"Separate code into 3 parts"

3 **Controller**

dealing with UI events

Triggered by user using the UI
The FEEL of the application

Based on slides by Toon Koppelaars

Different MVC Approaches

Remember that the code can be anywhere

Client

Application Server

Database

Thin: little code

Fat: lots of code to implement business logic

Different MVC Approaches

Your main choices:

Alternative	Client	Middle	Data
1	Thin	Fat	Thin
2	Fat	Fat	Thin
3	Fat	Thin	Thin
4	Fat	Thin	Fat
5	Thin	Fat	Fat
6	Thin	Thin	Fat
7	Fat	Fat	Fat

You can really code it in every way you want, and unfortunately people did.

Based on slides by Toon Koppelaars

Many fashions in Information Technology

Love acronyms

Love hazy concepts

HOT TECHNOLOGY OF THE DAY

... you can believe someone who has spent 30 years in the industry. One of the most useful skills is to be able to spot what is a really interesting new idea, and not a revamped old concept that was abandoned for good reasons.

Many fashions in Information Technology

Because investments are huge in software development, a lot of technologies survive many years after having dropped out of fashion.

Art by Nancy Duong