# JAVA

# Agenda:

- **Introduction to Java**
  - About Java
  - Java vs C#
- **Java EE platform**
  - Java EE Components (web, business)
  - Java EE server
  - Enterprise JavaBeans
  - Servlet
  - JavaServer Faces
  - Session Bean

**Goal:** Overview of Java EE Technology

# Java: History

- **June 1991:** James Gosling, Mike Sheridan, and Patrick Naughton initiated the Java language project, in **Sun Microsystem**

- **1995:** Sun Microsystem released first public implementation. It promised "**W**rite **O**nce, **R**un **A**nywhere"
  - Major web browsers soon incorporated the ability to run **Java applets**

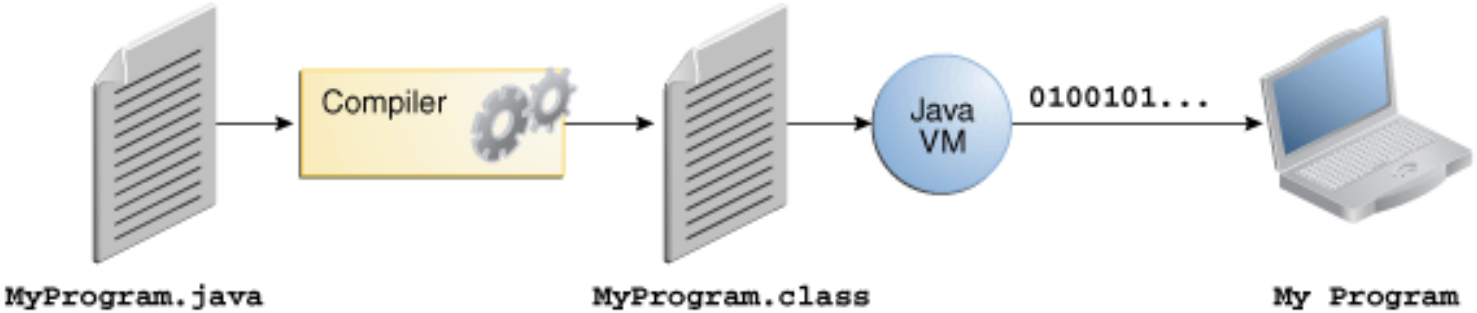
- **December 1998: Java 2** released:
  - **J2EE** includes api for enterprise applications running in server environments
  - **J2ME** includes api for mobile applications
  - **JSE** for desktop environments

- **November 13, 2006:** Sun released much of Java as *free and open-source project,* under GPL

- **January 27, 2010:** Oracle acquired Sun. The Oracle implementation is packaged into two different distributions:
  - Java Runtime Environment (**JRE**), intended for end users
  - Java Development Kit (**JDK**), intended for software developers

http://en.wikipedia.org/wiki/Java_(programming_language)

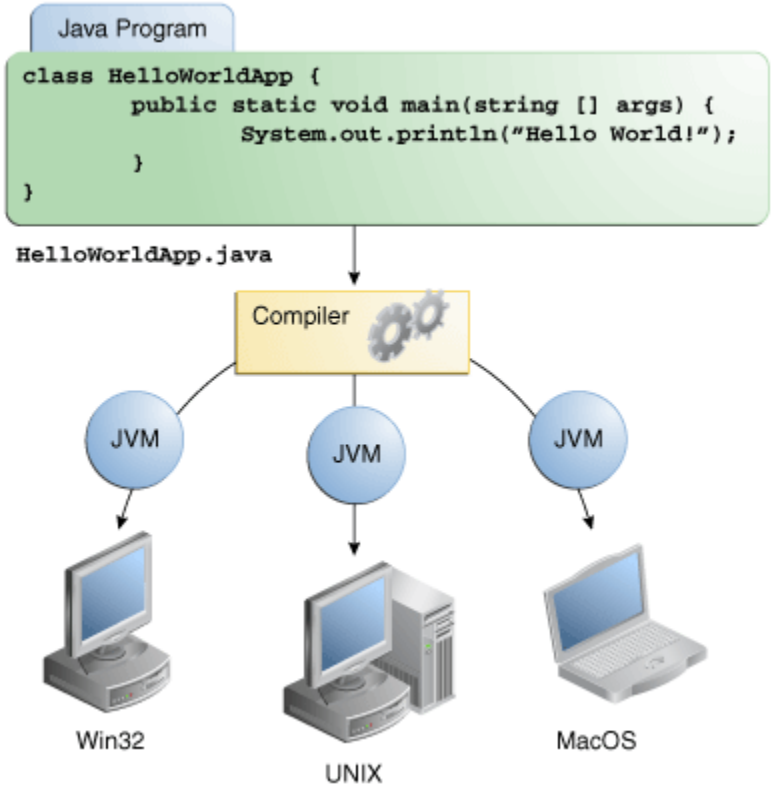
# About the Java Technology




- **javac** compiler: compile source files into .class files
- a .class file contains **bytecode**
- **bytecode** is the machine language for **Java Virtual Machine**
- **java** launcher tool runs the application

# About the Java Technology

- Java VM is available on many different operating system

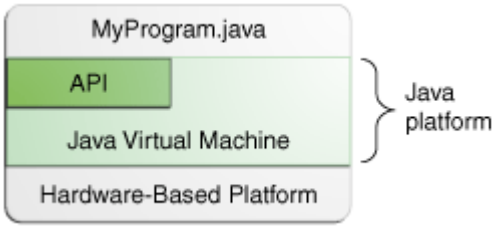


# The Java Platform

- The **Java Virtual Machine**

- The **Java Application Programming Interface (API)**

The API is a large collection of ready-made software components that provide many useful capabilities (**packages**)



# Package

- a **package** is a group of related types providing access protection and namespace management
- package name are written in all lower case to avoid conflict with names of classes or interfaces
- companies use their reverse Internet domain name

In **Java** the directory structure should match the package structure.
No such restriction in **C#**

# Java vs C#: Declaring Constants

Java:

- In Java, compile-time constant value are declared inside a class as

```
static final int K = 100;
```


C#:

- in C# the keyword **const** is used for compile time constants, while the **readonly** keyword is used for runtime constants.

```
const final int K = 100;
readonly int aField;
```

# Java vs C#: Declaring Constants

**Java**

```
class Test
{
    final int afield = 5;
    final int workday = 256;
    ...
    int getAfield() {
        return afield;
    }
}
```

```
//set method is not allowed for both finals
//cannot be written to by a constructor
```

**C#**

```
class Test
{
    readonly int afield;
    const byte workday = 256;
    Test(int n) {
        afield = n;
    }
    ...
    int getAfield() {
       return afield;
    }

}
//set method is not allowed for both.
//readonly can only be written to by a
constructor
```

# Java vs C#: Inheritance

Java:

```
class B extends A implements Comparable
{
    …
    ...
}
```

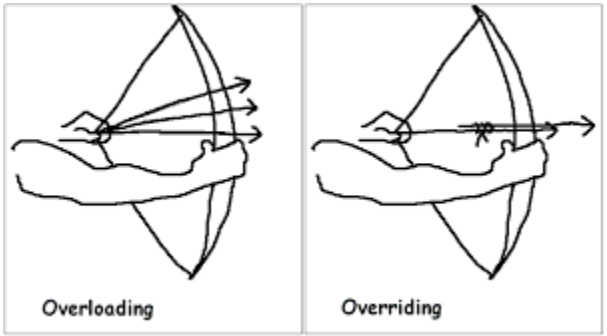
C#:

```
class B : A, IComparable
{
    …
    ...
}
```

# Polymorphism and Overriding

- In Java all methods are virtual
- In C# you must explicitily state which methods are virtual

```
using System;
public class Parent
{
   public virtual void DoStuff(string str)
         { … }
}
public class Child : Parent
{
   public override void DoStuff(string str)
         { … }
}
```




```
public new void DoStuff(string str)
```

# Multiple classes in a Single File

- In **Java**, only one class per source file with public access (it must have the same name of the source file)

- **C#** doesn't have restriction on the number of public classes that can exist in a source file

# Importing libraries

- **C#:** *using* keyword

  ```
  using System;
  using System.IO;
  using System.Reflection;
  ```

- **Java:** *import* keyword

  ```
  import java.util.*;
  import java.io.*;
  ```

# Enumerations

**Java**

```
public enum Planet {
    MERCURY (3.303e+23, 2.4397e6),
    VENUS   (4.869e+24, 6.0518e6),
    EARTH   (5.976e+24, 6.37814e6),
    MARS    (6.421e+23, 3.3972e6),
    JUPITER (1.9e+27,   7.1492e7),
    SATURN  (5.688e+26, 6.0268e7),
    URANUS  (8.686e+25, 2.5559e7),
    NEPTUNE (1.024e+26, 2.4746e7);

    private final double mass;   // in kilograms
    private final double radius; // in meters
    Planet(double mass, double radius) {
        this.mass = mass;
        this.radius = radius;
    }
    private double mass() { return mass; }
    private double radius() { return radius; }
}
```

**C#**

```
public enum Direction {
        North = 1,
        East = 2,
        West = 3,
        South = 4
};
```

Usage:

```
Direction wall = Direction.
North;
```

# Properties

**Java**

```
public int getSize()
{
        return size;
}

public void setSize(int val)
{
        size = val;
}
```

**C#**

```
public int Size
{
        get
        {
                return size;
        }
        set
        {
                size = value;
        }
}
```

# Pass by Reference

- In Java the arguments to a method are passed by value
- In C# it is possibile to specify the arguments by reference: **ref** and **out** keywords

```
ChangeMe(out param);
Swap(ref a, ref b);
```

# Delegate

- In **C#** delegate are reference types which allow to indirect calls to method

- There is no delegate concept in **Java** (it may be mimiced with reflection)

# Delegate

**C#**

```
using System;
 using System.IO;
 public class DelegateTest
 {
    public delegate void Print (String s);
    public static void Main()
    {
       Print s = new Print (toConsole);
       Print v = new Print (toFile);
       Display (s);
       Display (v);
    }
    public static void toConsole (String str)
    {
       Console.WriteLine(str);
    }
    public static void toFile (String s)
    {
       File f = new File("delegate.txt");
       StreamWriter fileOut = f.CreateText();
       fileOut.WriteLine(s);
       fileOut.Flush();
       fileOut.Close();
    }
    public static void Display(Print pMethod)
    {
       pMethod("This should be displayed in the
console");
    }
 }
```

# Event Handling: C#


```
using System;
using System.Drawing;
using System.Windows.Forms;

// custom delegate
public delegate void StartDelegate();

class EventDemo : Form
{
    // custom event
    public event StartDelegate StartEvent;

    public EventDemo()
    {
        Button clickMe = new Button();


 // an EventHandler delegate is assigned
        // to the button's Click event
        clickMe.Click += new EventHandler(OnClickMeClicked);

        // our custom "StartDelegate" delegate is assigned
        // to our custom "StartEvent" event.
        StartEvent += new StartDelegate(OnStartEvent);

        // fire our custom event
        StartEvent();
    }

    // this method is called when the "clickMe" button is pressed
    public void OnClickMeClicked(object sender, EventArgs ea)
    {
        MessageBox.Show("You Clicked My Button!");
    }

    // this method is called when the "StartEvent" Event is fired
    public void OnStartEvent()
    {
        MessageBox.Show("I Just Started!");
    }
```

# Event Handling: Java

- **Source:** is an object on which event occurs
- **Listener:** is responsible to generate response to an event


## Steps involved in event handling

- The User clicks the button and the event is generated.
- Now the object of concerned event class is created automatically and information about the source and the event get populated with in same object.
- Event object is forwarded to the method of registered listener class.
- The method is now get executed and returns.

*http://en.wikipedia.org/wiki/Observer_pattern*

# Inner and anonymous classes

Since the Java language does not permit multiple inheritance, your class cannot extend both the `Applet` and `MouseAdapter` classes. A solution is to define an *inner class* a class inside of your `Applet` subclass

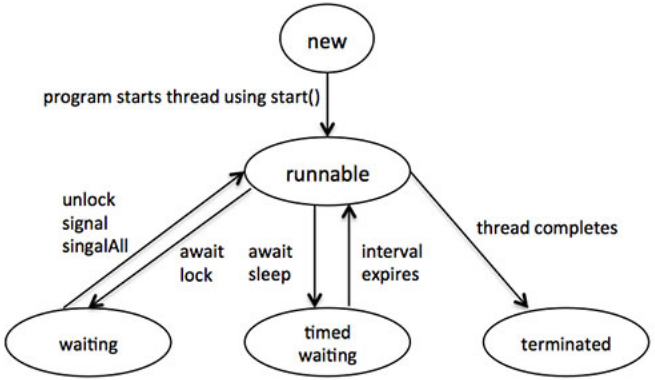
```
public class MyClass extends Applet {
   ...
      someObject.addMouseListener(new MyAdapter());
   ...
   class MyAdapter extends MouseAdapter {
      public void mouseClicked(MouseEvent e) {
         ...//Event listener implementation goes here...
      }
   }
}
```

- Inner classes can also be useful for event listeners that implement one or more interfaces directly.
- Inner classes work even if your event listener needs access to private instance variables

*When considering whether to use an inner class, keep in mind that application startup time and memory footprint are typically directly proportional to the number of classes you load. The more classes you create, the longer your program takes to start up and the more memory it will take. As an application developer you have to balance this with other design constraints you may have. We are not suggesting you turn your application into a single monolithic class in hopes of cutting down startup time and memory footprint this would lead to unnecessary headaches and maintenance burdens.*

# Java - Multithreading

- **New:** A new thread begins its life cycle in the new state. It remains in this state until the program starts the thread. It is also referred to as a born thread.
- **Runnable:** After a newly born thread is started, the thread becomes runnable. A thread in this state is considered to be executing its task.
- **Waiting:** Sometimes, a thread transitions to the waiting state while the thread waits for another thread to perform a task.A thread transitions back to the runnable state only when another thread signals the waiting thread to continue executing.




- **Timed waiting:** A runnable thread can enter the timed waiting state for a specified interval of time. A thread in this state transitions back to the runnable state when that time interval expires or when the event it is waiting for occurs.

- **Terminated:** A runnable thread enters the terminated state when it completes its task or otherwise terminates.

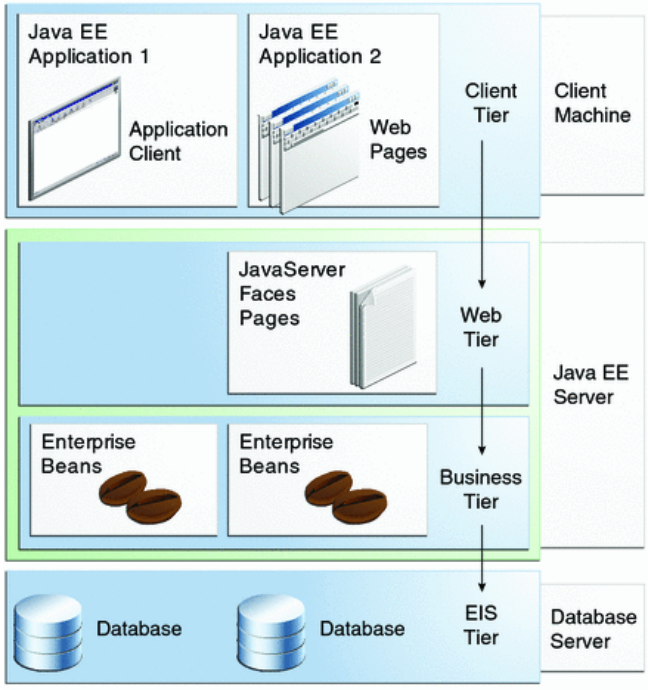
# Java Serializable

- mechanism to represent an object as a sequence of bytes (**including object's data**)
  - **serialized** object can be written in file, …
  - bytes can be used to recreate object in memory
- serialization is **JVM indipendent**

1. Class must be implement ***java.io.Serializable*** interface
2. all the fields in the class must be serializable. If a field is not serializable, it must be marked ***transient***

# Java EE platform

- large-scale, multi-tiered, scalable, reliable, secure network application
    - Client-tier components run on client (web browser)
    - Web-tier components run on the Java EE server
    - Business-tier components run on the Java EE server
    - Enterprise information system (EIS)-tier runs on the EIS server



# Java EE Components

- A **Java EE component** is a self-contained functional software unit

The Java EE specification defines the following components:

- **Application clients** and **applet** are components that run on client
- **Java Servlet, Java Server Faces, JSP** are web components that run on server
- **Enterprise Java Bean (EJB)** components are business components that run on the server

*Java EE components are assembled into a Java EE application and managed by the Java EE server*

# Java EE clients

- **Web Clients**
  - Dynamic web pages (Html, Xml, ...) which are generated by web components running in the web tier
  - Web browser, which renders the pages received from the server

- **Application Clients**
  - runs on client machine
  - graphical user interface (GUI) created from the Swing or AWT API

- **Applets**
  - small client application running in the Java virtual machine installed in the web browser

# Java EE web components

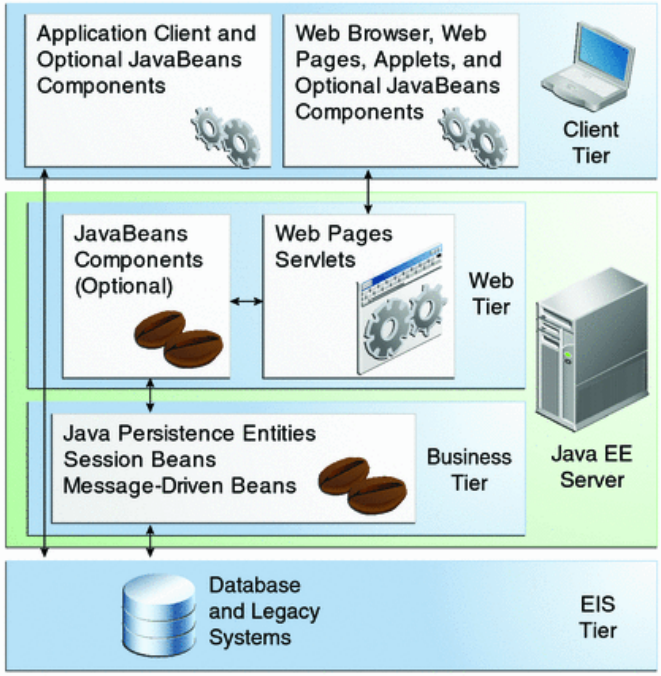

- **servlet** or **web pages** created using JSF and/or JSP.
- **Servlets** are Java classes that dynamically process requests and construct response

<gangnam-style></gangnam-style>

# Business components

- Business code is handled by enterprise bean running in either the web tier or the business tier

- The EIS tier include enterprise infrastructure system: database systems, mainframes



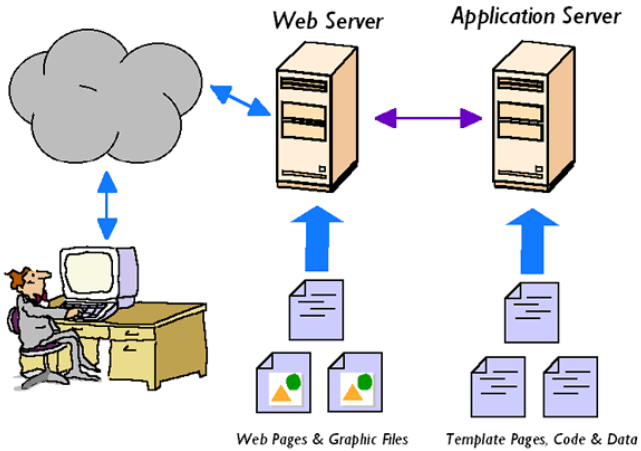
# Business tier


- **Enterprise Java Bean (EJB):** encapsulate the core functionality of application.
- **JAX-RS RESTful web services:** REST API for creating web services that respond to HTTP methods
- **JAX-WS web sercice endpoints:** api for creating and consuming SOAP web services
- **Java Persistence API entities:** api for processing data in data stores and mapping that to JAVA programming
- **Java EE managed beans:** managed components that may provide business logic, but do not require the transactional or security features of EJB

# Java EE server

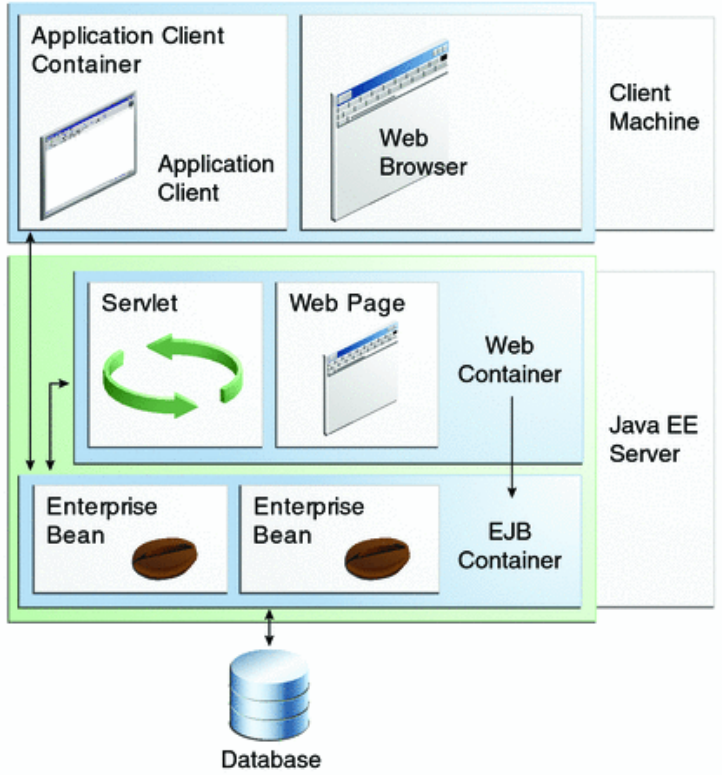
- A **Java EE server**, also called applicaton server, hosts several application component and provides the standard Java EE services in the form of **container**
    - **Java EE container:** interface between a component and the low-level platform-specific that support the component.
        - *provides services like security, transaction management, JNDI API lookups, remote connectivity…*

- A Java EE server provides EJB and web containers



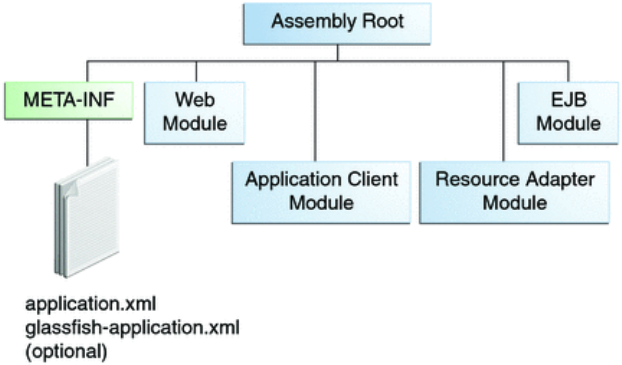
# Container

- **Enterprise JavaBeans (EJB) container:** manages the execution of enterprise beans, run on the Java EE server
- **Web container:** manages the executes of web pages, run on the Java EE server
- **Application client container:** manages the execution of application client components, run on the client
- **Applet container:** manages the execution of the applet. Consists of a web browser and Java Plug-in



# Packaging Applications


- A Java EE app is delivered in a **Java Archive (JAR)** file, a **Web Archive (WAR)** file or an **Enterprise Archive (EAR)** file.
- Using JAR, WAR and EAR files and modules makes it possibile to assemble different JEE applications.
- An EAR file contains J2EE modules (optionally) and deployment descriptors.
  - A **deployment descriptor** is an XML document that describes the deployment settings of an application, or module, or a component.

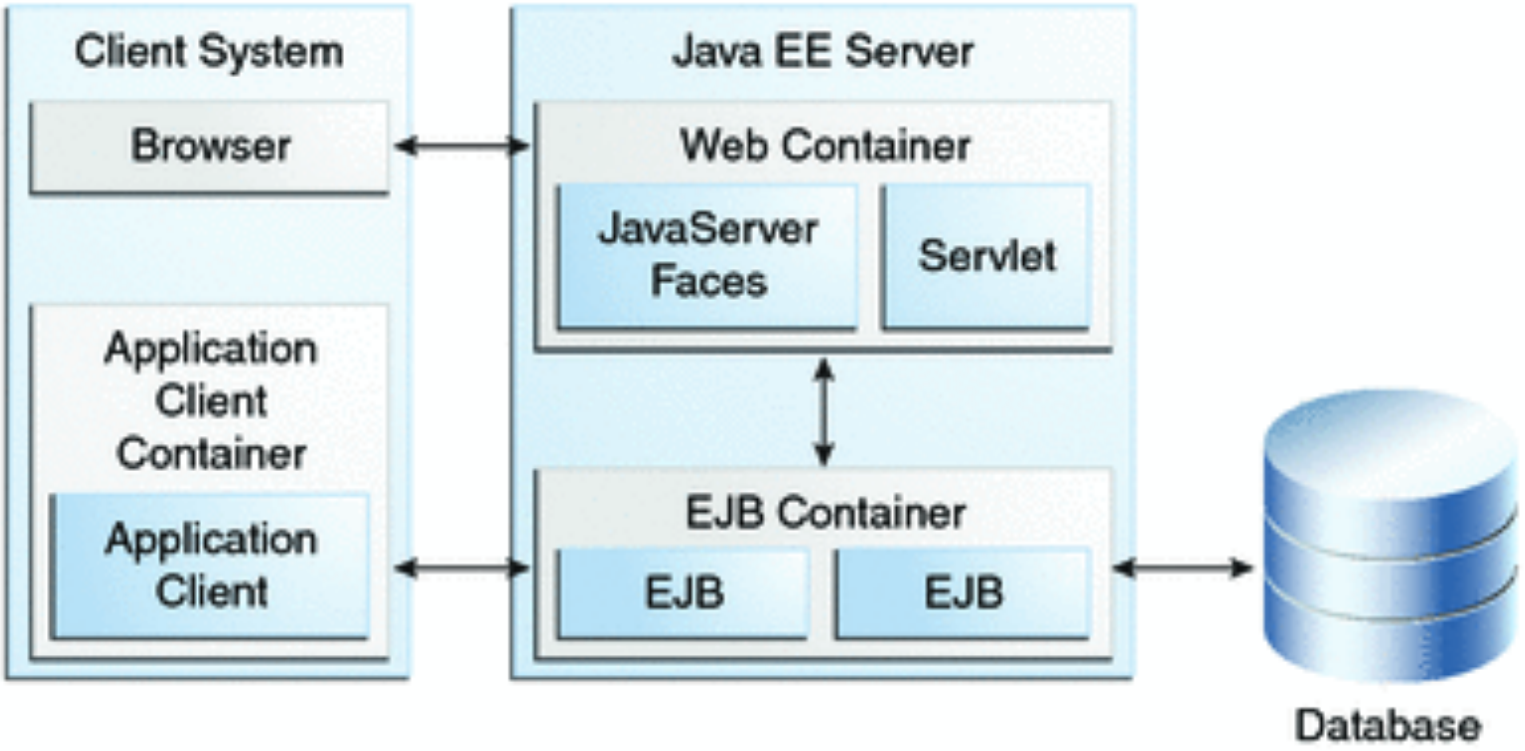


# Java EE module

- A **Java EE module** consists of one or more Java EE component for the same container type

  - EJB modules, wich contains class files for Enterprise Java Beans (.jar)
  - Web modules, which contains servlet class files, web files, HTML files, ecc. (.war)
  - Resource adapter modules, which contains classes, libraries, ecc. (.jar)

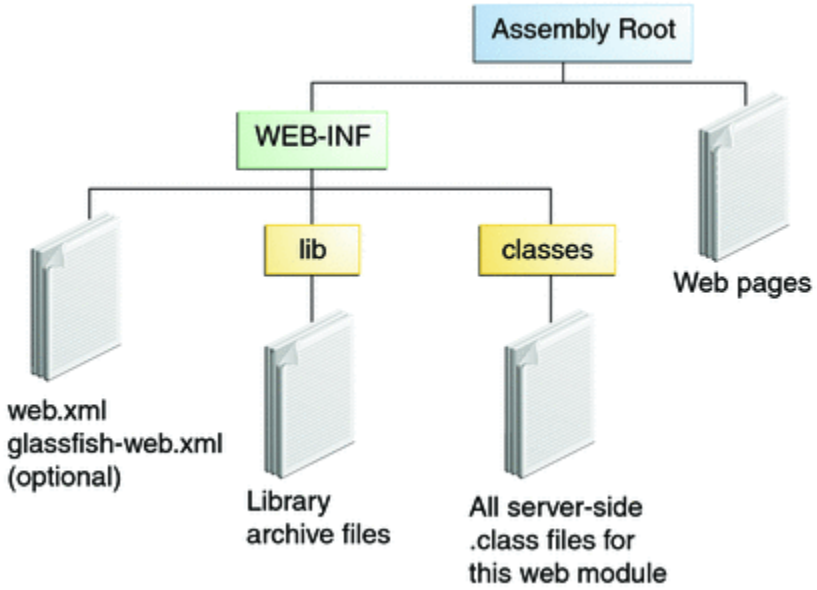
# Java EE Server



# Java EE Web Module

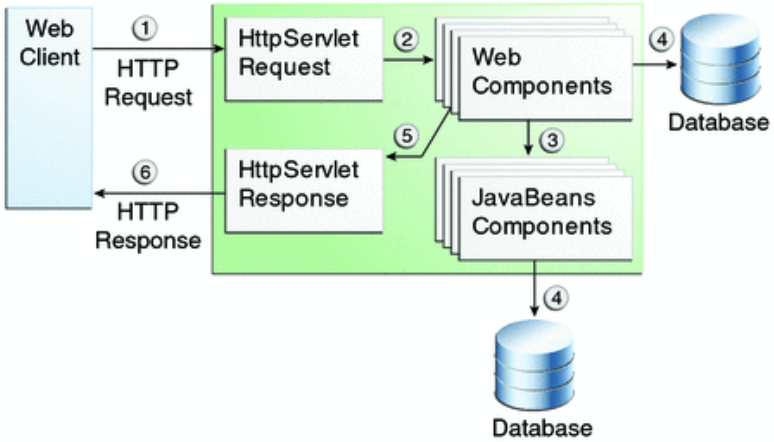
- the smallest deployable and usable unit of resources

- contains:
  - **web components**
  - **web resources (image, ...)**
  - **classes utility**



# Servlet

- client send an HTTP request to the server
- a web server, implementing a Java Servlet and JSP,* converts the request into a ***HTTPServletRequest*** object
- this object is delivered to a web component, which can interact with Java Bean component or database
- the web component generates an ***HTTPServletResponse*** object
- the web server converts it into an HTTP Response

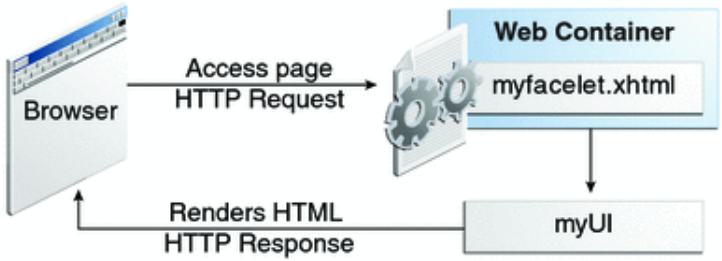



**Servlet** are Java classes that dinamically process the request and construct response.

Configuration info can be specified using JAVA EE annotation or via XML file (web application deployment descriptor)

*http://docs.oracle.com/javaee/6/tutorial/doc/geysj.html*

# JavaServer Faces




- **JSF** is a server-side component framework
  - api for representing component and managing state, handling event, validation, data-conversion, page navigation…
  - tag libraries for adding components to web pages and connecting to server-side objects

A typial JSF applications includes:

- a set of web pages
- a set of tags to add components to the web page
- a set of **managed bean**
- a web deployment descriptor (**web.xml**)
- optionally, one or more configuration files (**faces.config.xml**)**,** used to define page navigation rules and configure beans
- optionally, a set of custom objects: validators, converters, listeners, ecc…

- *JSF offers clean separation from behavior and presentation for web applications.*
- *A JSF application can map an HTTP request to a component-specific event handling and manage component as stateful objects on the server*

*http://javaserverfaces.java.net/*

# Enterprise JavaBeans

- **EJB** is a server side component that encapsulates the business logic

Types of Enterprise Beans:

- A **session bean** rapresents a transient conversation with the client.
    - **stateful bean**: business object having state. Access is limited to one client at a time.
    - **stateless bean:** business object that don't have a state. Access to  a single bean instance is limited to only one client at a time (the container routes a request to a different instance)
    - **singleton bean:** business object having a global shared state whithin a JVM

# Enterprise JavaBeans

- **Message Driven Bean:** business objects whose execution is triggered by messages instead of method calls. The message driven bean is used to provide an abstraction for the lower level JMS

EJBs are deployed in an EJB container.

- Clients not instantiate them directly, but have to obtain a reference via the EJB container.
  - through **dependency injection**
  - using **annotation**
  - using **JNDI lookup**

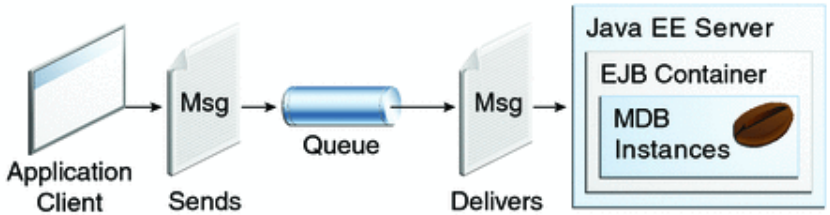
# Session Bean

- encapsulate business logic than can be invoked by a client over local, remote or web services.

- **stateful session bean:** the client interacts (talks) whith its bean (**conversional state**). It is not shared, it can have only one client
- **stateless session bean:** does not mantain a conversional state with the client. When the client invokes a method, the bean's instance variable may contain a state specific to that client but only for the duration of invocation. All instance of a stateless bean are equivalent, allowing the EJB container to assign an instance to any client. Because they support many clients, stateless bean can offer better scalability
- **singleton session bean:** is instantied once per application and exists for all the lifecycle.

# Message-driven Bean

- is an enterprise bean that allows the Java EE applications to process messages asynchronously.
- receives JMS messages (or other kind of messages)
- it resemble a stateless sesson bean:
  - no data or conversational state for a specific client
  - all instances of MdB are equivalent
  - a single MdB can process messages from multiple clients




When a message arrives, the container calls the message-driven bean's ***onMessage*** method to process the message.

# Enterprise Beans

- Enterprise Beans are Java EE components that implements the Enterprise JavaBeans (EJB) technology
- run on the EJB container
- encapsulate the business logic
- simplify the development of large and distributed applications
- EJB container is responsible for system-level services, such as transactions and security authorization

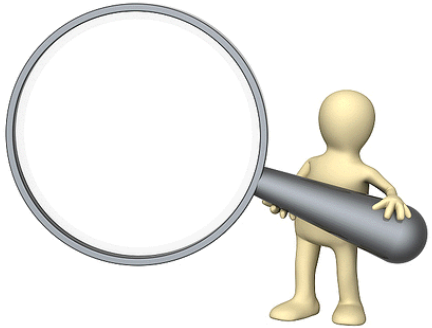
Two types of enterprise beans:

- session
- message driven

# Accessing Enterprise Beans

Clients can obtain a reference to an instance of a bean through:

- **dependency injection**, using Java annotations (`javax.ejb.EJB`)
- **JNDI lookup,** using the Java Naming and Directory Interface syntax

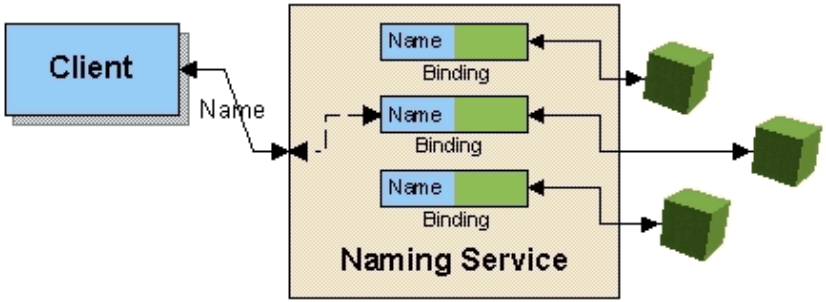


**JNDI**: is a Java api that allows clients to discover and lookup data or objects via a name. It provides:

- a mechanism to bind an object to  a name
- a directory-lookup interface that allows general queries
- an event interface that allows clients to determine which directory entries has been modified
- LDAP extensions…

JNDI is composed of five packages:

- *javax.naming*
- *javax.naming.directory*
- *javax.naming.event*
- *javax.naming.ldap*
- *javax.naming.spi*



# Lookup enterprise beans

Three namespaces are used for JNDI lookups:

- ***java:global***: to find remote enterprise bean using lookups

  `java:global[/application name]/module name/enterprise bean name[/interface name]`

- ***java:module:*** to lookup local enterprise beans whithin the same module

  `java:module/enterprise bean name/[interface name]`

- ***java:app***: to lookup local enterprise bean packaged withind the same application

  `java:app[/module name]/enterprise bean name[/interface name]`

For example, if an enterprise bean, `MyBean`, is packaged within the web application archive `myApp.war`, the module name is `myApp`. The portable JNDI name is `java:module/MyBean` An equivalent JNDI name using the `java:global` namespace is `java:global/myApp/MyBean`.

# Remote or Local Access

- **Tight or loose coupling of related beans:** when a bean depends on one another. They are good candidates for local access
- **Type of client:** if an enterprise bean is accessed by application clients, it should allow remote access. If an enterprise bean is accessed only by other beans, think on how you want to distribute your components
- **Component distribution:** Java EE applications are scalable because their server-side components can be distributed across multiple machines
- **Performance:** remote calls may be slower than local calls

# Remote or Local access


- **Define remote or local access**:

  `//@Local` or `@Remote (@Local is default)`

  ```
  @Local(InterfaceName.class)
  public class BeanName implements InterfaceName { ... }
  ```

- **Accessing local beans**
  - use the `javax.ejb.EJB` annotation

    ```
    @EJB
    ExampleBean exampleBean;
    ```

  - use the `javax.naming.InitialContext` interface's `lookup` method

    ```
    ExampleBean exampleBean = (ExampleBean)
            InitialContext.lookup("java:module/ExampleBean");
    ```

**Accessing remote beans**

```
@EJB
Example example;

      ExampleRemote example = (ExampleRemote)
        InitialContext.lookup("java:global/myApp/ExampleRemote");
```

# Contents of an Enterprise Bean

- **Enterprise bean class:** implements the business methods of the enterprise bean
- **Business interfaces:** not required if the enterprise bean expose a local, no-interface view
- **Helper classes:** other classes needed by bean (utility classes, exception, ...)

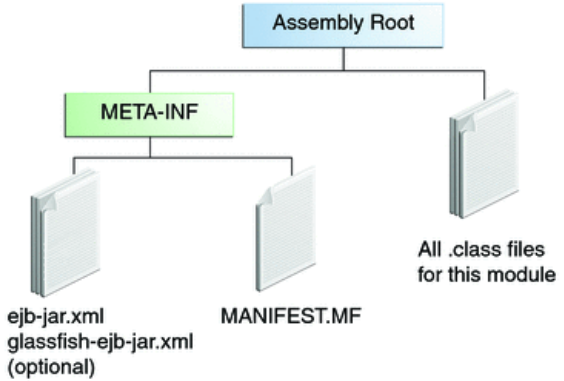
**Packaging Enteprise Beans in EJB JAR modules:**

- to assemble a Java EE application, package one or more modules, such as EJB jars, into an EAR file. You can also deploy EJB Jar not contained in an EAR file

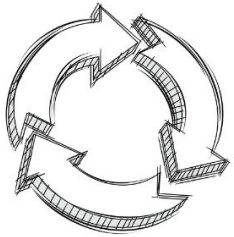
**Packaging Enterprise Beans in WAR modules:**

- to include enterprise bean class files in WAR module, the class should be in the `WEB-INF/classes` directory

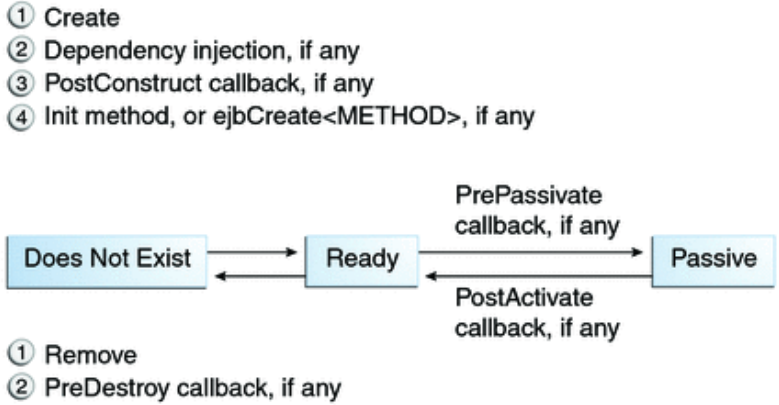
- to include a JAR containing enterprise beans in a WAR module, add the JAR to the `WEB-INF/lib` directory of the WAR module



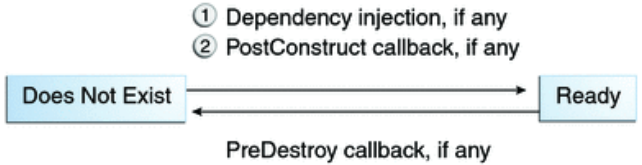
# Lifecycle of Enterprise Beans



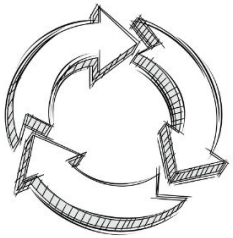
- **Stateful Session Bean's Lifecycle**
  - client obtaines a reference to a stateful session bean
  - the container perform dependency injection
  - the c. invokes the method annotated with `@PostConstruct`
  - the EJB container may decided to deactivate, or **passivate,** the bean by moving it from memory to a secondary storage
  - client invokes a method annotated `@Remove`
  - EJB container calls `@PreDestroy`
  - Bean's instance is ready for garbage collection



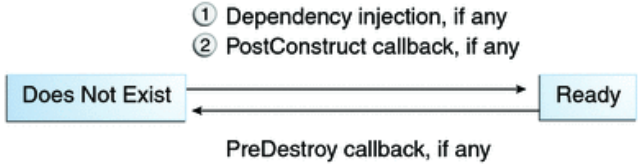

- **Stateless Session Bean's Lifecycle**
  - stateless session bean is never **passivated**
  - **EJB** container creates and mantains a pool of stateless beans



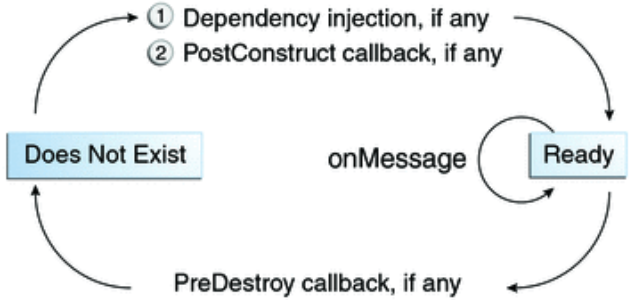
# Lifecycle of Enterprise Beans



- **Singleton Session Bean's Lifecycle**
  - EJB container creates the singleton instance
  - at the end, EJB container calls the method annotated `@PreDestroy`, if it exists




- **Message-Driven Bean's Lifecycle**
  - **EJB** container creates and mantains a pool of message-driven beans
  - like a stateless bean, it is never passivated, and only has two states: *non existent* and *ready to receive message*



# JAX-RS: Java api for RESTful WS

- **RESTful web service** are built to work best on the web
- **JAX-RS:** api that provides support in creating web services rest
- developers decorate Java class files with JAX-RS **annotations**
  - @*Path*: indicating where the Java class will be host
  - @*GET:* Java method annotated with this will process HTTP GET requests
  - @*POST*
  - @*PUT*
  - @*DELETE*
  - .
  - .

  ….

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

// The Java class will be hosted at the
URI path "/helloworld"
@Path("/helloworld")
public class HelloWorldResource {

    // The Java method will process HTTP
GET requests
    @GET
    // The Java method will produce
content identified by the MIME Media
    // type "text/plain"
    @Produces("text/plain")
    public String getClichedMessage() {
        // Return some cliched textual
content
        return "Hello World";
    }
}
```