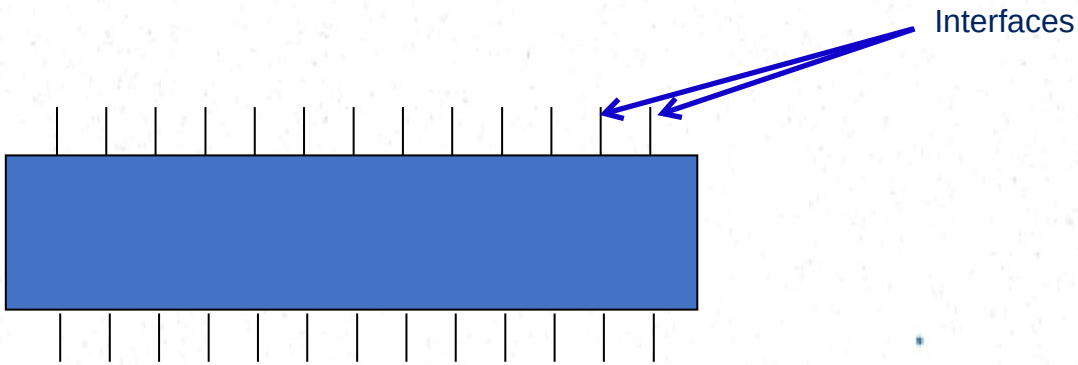
# Lecture 6 - Distributed Component frameworks

# What is a component?

A component is a reusable software module with a well-defined interface.

Similar to hardware components (e.g., semiconductor chips), software components expose interfaces that allow integration. Software development follows the same concept as hardware, where different components interact based on predefined connections.

- Very intuitive, but often vaguely defined
- A semiconductor chip is a component




The chip vendor publishes manuals that tell developers the functionality of each pin

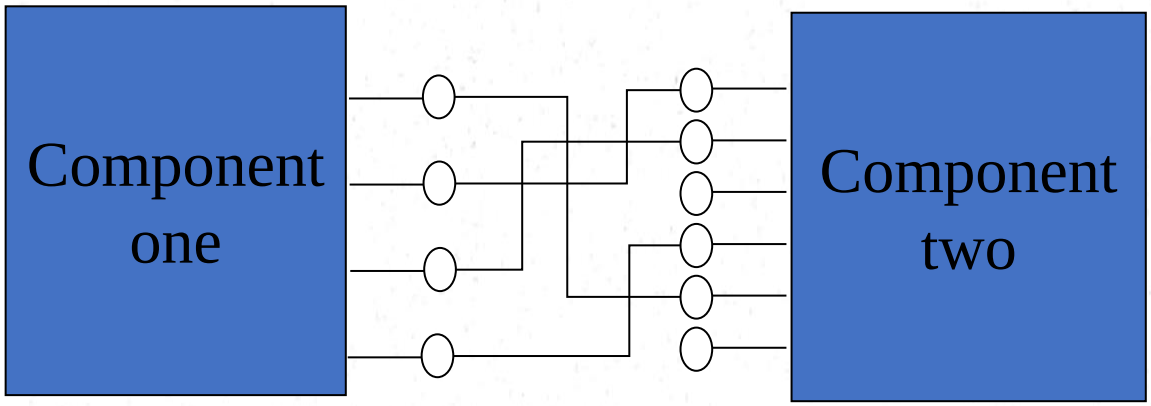
| | | |
|---|---|---|
| A0-A16 | I/O | Address bus |
| D0-D8 | I/O | Data bus |
| ACK | O | Acknowledge: Accepted when low |

# Building complex systems

In hardware, chips are connected using pins (e.g., address bus, data bus, acknowledge signals).
In software, components are connected through interfaces (a set of methods that define how they interact).

Ex: A payment processing system built by integrating a payment gateway component, notification component, database component.

- Hardware system integrators connect pins of chips together according to their functions to build complex electronic devices such as computers.



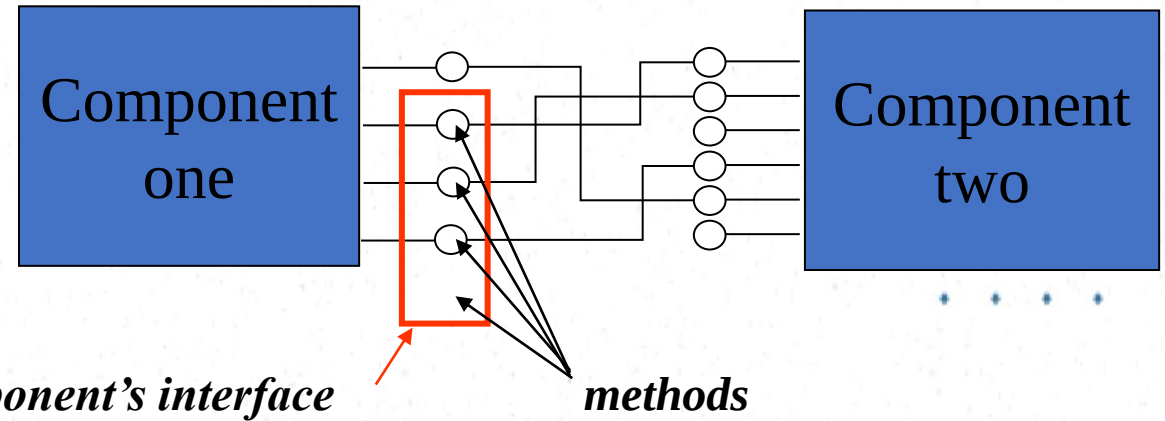

- Pins functionality defines behavior of the chip.
- Pins functionality is standardized to match functionality of the board (and take advantage of common services).

**Software Components**

- **Interfaces** in software are like the "pins" of hardware.
- Components communicate through method calls.
- **Key Features:**
  - **Encapsulation:** Hides implementation details.
  - **Reusability:** Can be reused in multiple applications.
  - **Interoperability:** Components from different vendors can work together.

# Software components

- The software component model takes a very similar approach:
  - the "pins" of software components are called interfaces
  - an interface is a set of methods that the component implements

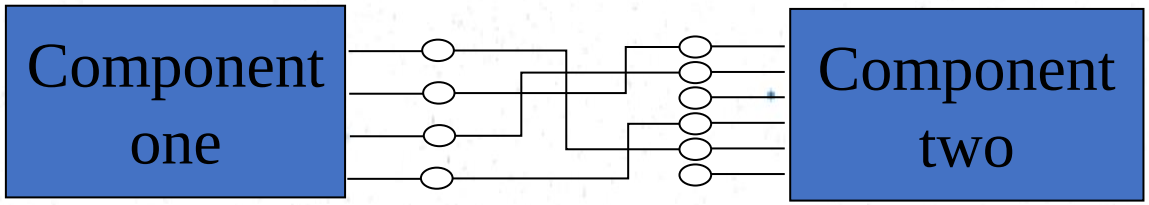



- software integrators connects components according to the descriptions of the methods within the interface.

# Software components (2)

- Component technology is still young, and there isn't even agreement on the definition of its most important element - the component.
- ***"a component is a software module that publishes or registers its interfaces"***, a definition by P. Harmon, Component Development Strategy newsletter (1998).
- Note: a component does not have to be an object!
  An object can be implemented in any language as long as all the methods of its interface are implemented.
- Objects – Design level
- Components – Architectural level

# Component wiring




Example: A user clicks "Login," and the authentication component is called.

- The traditional programming model is caller-driven (application calls methods of a component's interface, information is pulled from the callee as needed). Component never calls back.
- In the component programming model, connecting components may call each other (connection-oriented programming).
- The components can interacts with each other through event notification (a component pushes information to another component as some event arises). This enables wiring components at runtime.

Example: A stock price update system, where a price change in one component triggers updates in multiple connected components.

# Pragmatic definition

Characteristics of Software Components

- Software Components:
  - predictable behavior: implement a *common* interface
  - embedded in a framework that provides common services (events, persistence, security, transactions,…)
  - developer implements only business logic

# Distributed Component Models

- Hide implementation details — The user interacts with the interface, not the internal logic.
- Bring power of a container — container provides lifecycle management, security, and other services.
- Focus on business logic
- Enable development of third-party interoperable components — Components from different sources can work together.

# Examples of component frameworks

- CORBA (Component Object Request Broker Architecture)
- Java/Java EE - EJB (Enterprise Java Beans)
- Spring Framework
- Microsoft/.NET – DCOM, WCF Services

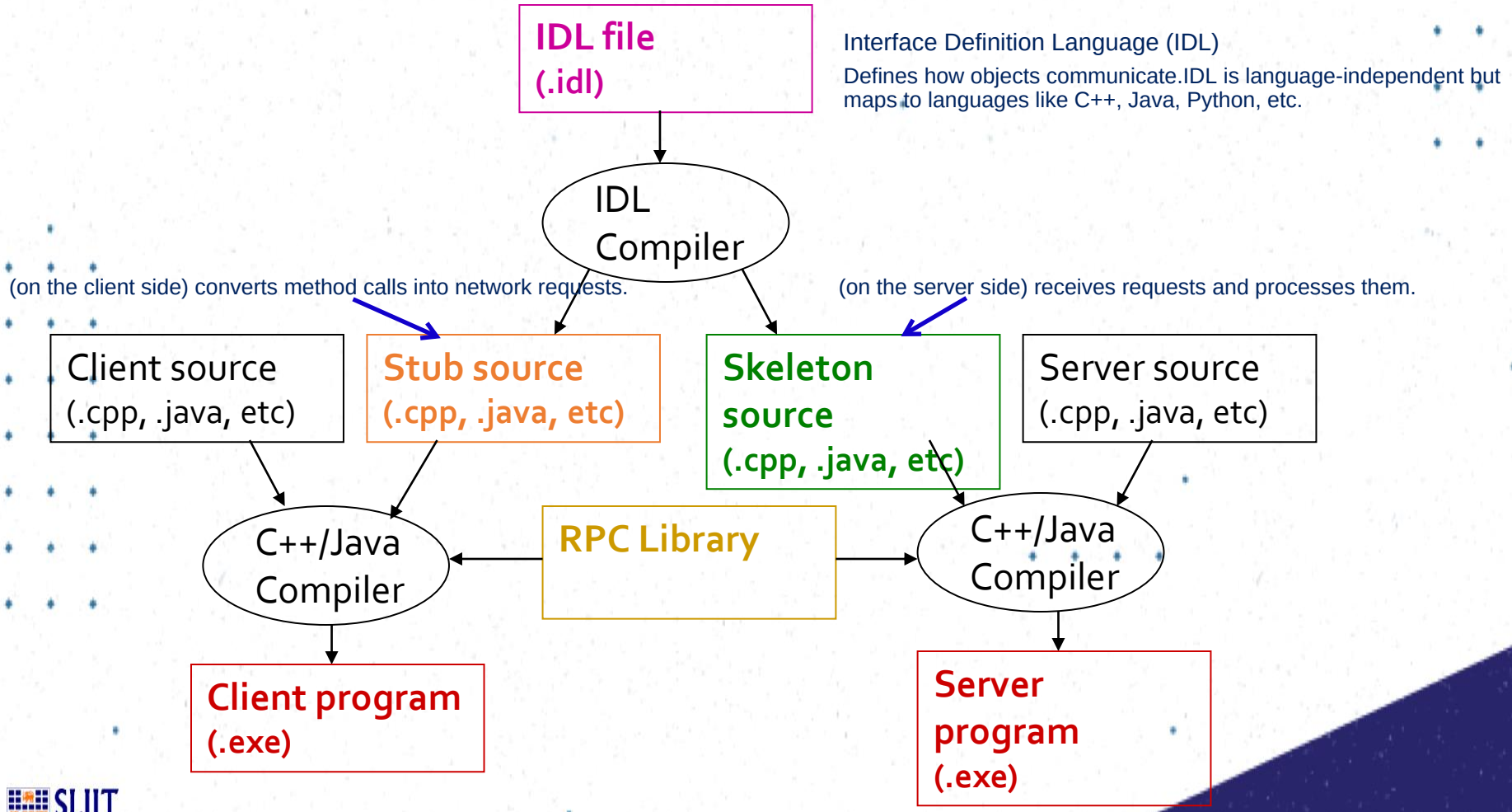
# CORBA

Allows applications written in different programming languages to communicate across different platforms.
Acts as middleware, enabling seamless interaction between distributed objects.

- **CORBA** is the acronym for **C**ommon **O**bject **R**equest **B**roker **A**rchitecture
- CORBA grew out of academic efforts to build a distributed computing framework around RPC
  - Dubbed 'middleware' since it aims to transparently connect systems running on different platforms
- CORBA specification administered by the Object Management Group (OMG)
  - Now up to CORBA 3
- Implementations of the CORBA spec are referred to as Object Request Brokers (ORBs)

  Uses Object Request Brokers (ORBs) to facilitate communication.
  - An ORB is built for a particular language (e.g. C++, Java)

# CORBA Code Generation



# CORBA and O-O

- Object-orientation gave the opportunity to hide the internal details of RPC such as marshaling
  - Interfaces are implemented as C++/Java/etc objects that inherit from an IDL-generated C++/Java/etc class
    - Uses polymorphism to direct incoming call to your object
- An object that implements an interface is roughly equivalent to the concept of a component
  - CORBA never really mentioned 'component' until the CORBA Component Model (CCM) that added things like support for authentication, persistence and transactions

# CORBA Interoperability

IIOP (Internet Inter-ORB Protocol).

- ORB's use IIOP to communicate with each other.

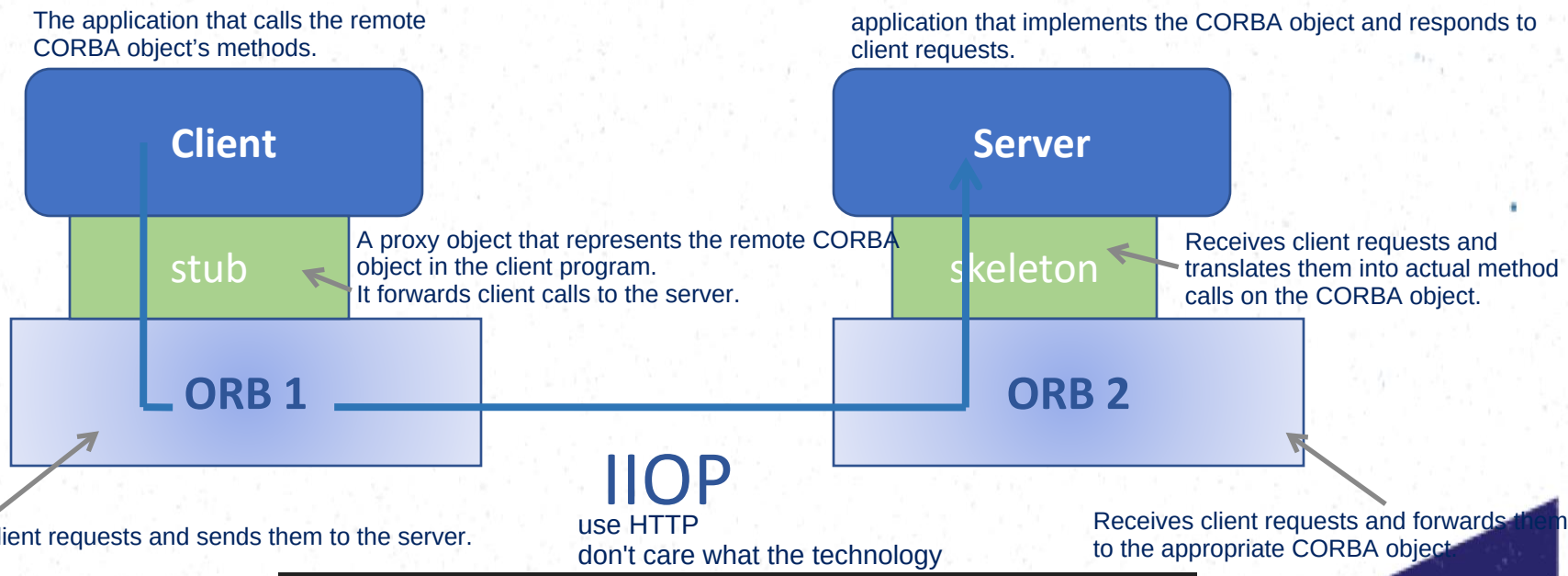
IOP ensures that CORBA applications from different vendors work together.

- Using the standard protocol IIOP, a CORBA-based program from any vendor, on almost any computer, operating system, programming language, and network, can interoperate with a CORBA-based program from the same or another vendor, on almost any other computer, operating system, programming language, and network.
  - IIOP (Internet Inter-ORB Operability Protocol); Defines:
    - Message types and binary message format
    - Data format - Binary format for data types (e.g. int, double, string)
    - Object reference format - identifies the object and its host server

# CORBA Interoperability

**. Message Flow in CORBA Interoperability**

1. The client calls a method on a remote object through a **stub**.
2. The stub forwards the request to the client-side **ORB**.
3. The **ORB** uses **IIOP** to transmit the request over the network to the server-side **ORB**.
4. The server-side **ORB** passes the request to the **skeleton**, which invokes the actual object method.
5. The CORBA object executes the method and sends the response back via the skeleton → server ORB → IIOP → client ORB → stub.
6. The client receives the response and processes it.




**IIOP Standardizes Communication** between ORBs, allowing:

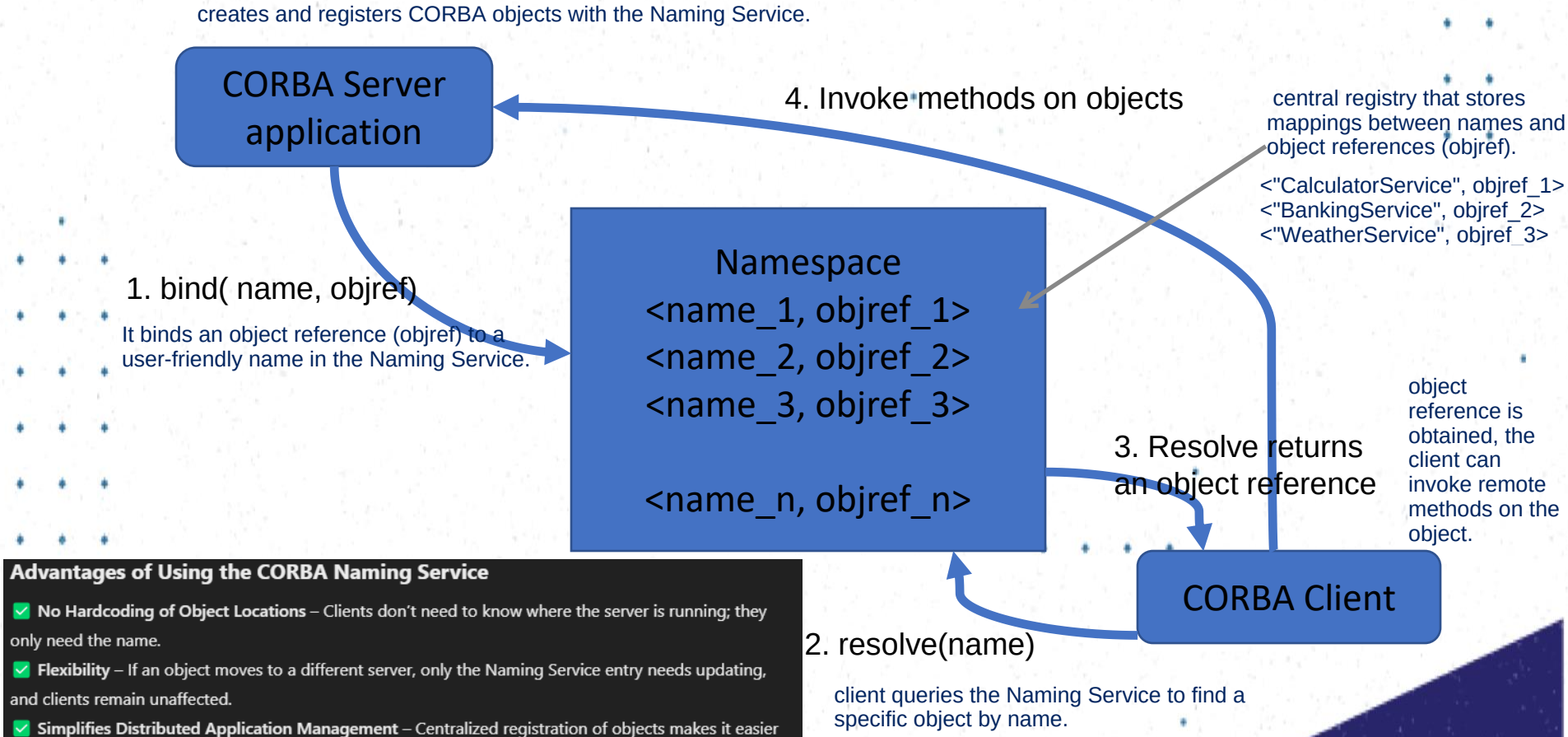
- Different programming languages to interoperate (e.g., Java and C++ applications).
- Different operating systems to communicate (e.g., Windows and Linux).
- Vendor-independent interaction (e.g., a Java-based ORB can communicate with a C++-based ORB).

# CORBA Naming Service

helps clients locate remote objects using human-readable names instead of low-level network addresses.

- CORBA Naming Service allows you to associate abstract names with CORBA objects and allows clients to find those objects by looking up the corresponding name.
- A naming service is a central server that knows where other servers can be found, like an index
  - Allows locating an object based on a user-friendly name, avoiding the need to hard-code object-server allocations
  - Means that the server machine, an object is hosted on, can be changed without breaking the distributed application

# CORBA Naming Service




**Advantages of Using the CORBA Naming Service**

- ✅ **No Hardcoding of Object Locations** – Clients don't need to know where the server is running; they only need the name.
- ✅ **Flexibility** – If an object moves to a different server, only the Naming Service entry needs updating, and clients remain unaffected.
- ✅ **Simplifies Distributed Application Management** – Centralized registration of objects makes it easier to manage remote object interactions.

# IDL Example: CORBA IDL

remote interface

it is independant language

```
module Utilities
{
   // Basic example interface
   interface Calculator
   {
      int Add(in int operand1, in int operand2);
      double Add(in double operand1, in double operand2);
   }

   // Interface inheritance
   interface ScientificCalculator : Calculator
   {
      // Returning values by the parameter list
      void SolveQuadratic(in float a, in float b, in float c, out float
  x1,
                                                          out float x2);

      // Example with a string
      double EvaluateExpression(in string expr);
   }
}
```

# CORBA Example (Java) - Server

```
public class CalculatorImpl
             extends CalculatorImplBase        ← xyzImplBase is generated by IDL compiler
{
   public CalculatorImpl() {
      super();                                 ← Let xyzImplBase do important initialisation work
   }
   public int Add(int operand1, int operand2) {      ← Actual interface function
      return operand1 + operand2;
   }
}
```

---

```
import org.omg.*;

public class CalcServer {
   public static void main(String[] args) {
      CORBA.ORB orb = CORBA.ORB.init(args, null);     ← Initialise Java's ORB
      CalculatorImpl calc;
      calc = new CalculatorImpl();
      orb.connect(calc);                              ← Tell ORB about Calculator object
      sObjRef = orb.object_to_string(calc);
      System.out.println("Object ref: " + sObjRef);   ← Print stringified obj ref for use by clients
      System.out.println("Press Enter to exit");            (using a name server is better, but more complex)
      System.in.readln();                             ← Wait for client requests
   }
}
```

# CORBA Example (Java) - Client

```
import org.omg.*;

public class CalcClient
{
   public static void Main(String[] args) {
      CORBA.ORB orb = CORBA.ORB.init(args, null);
      String sServerRef = args[0];
      CORBA.Object temp = orb.string_to_object(sServerRef);
      Calculator calc = CalculatorHelper.narrow(temp);
      System.out.println("1 + 2 = " + calc.Add(1, 2));
   }
}
```

← Initialise Java's ORB

← Assume user has reference for server (better to use name server, but more complex)

← Create Calculator on remote server, return ref

← Narrow (downcast) Object to Calculator

# CORBA Notes

- Each vendor will have slightly different ways of declaring and creating CORBA objects
  - The basic organisation of the code will be similar, just the specifics of ORB initialisation and where to get classes
- Note that a CORBA server object is *first* created by the server host process, *then* registered with the ORB
  - Means that the server object is always running awaiting new incoming client connections
  - Registration (via orb.connect) must only happen once for each object

# CORBA Today

- Although conceptually quite elegant, CORBA never really caught on, for a couple of reasons:
  - The CORBA spec is huge, making it complex to use
  - Competing frameworks were developed around the same time that had better backing and/or a broader coder base

Although CORBA was not widely adopted, some concepts that
it focused on such as interoperability and common standards were relevant for Web Services as well.

CORBA was a smart idea for letting different software systems talk to each other, but it didn't become popular. Why?

- It was really complicated because its rules (the "spec") were super long and hard to work with.
- Other easier or more popular options, like Java tools or Web Services, came out at the same time and got more support from companies and developers.

Even though CORBA didn't catch on, some of its big ideas—like making sure different systems could work together smoothly and using shared rules—helped shape Web Services, which are widely used today.

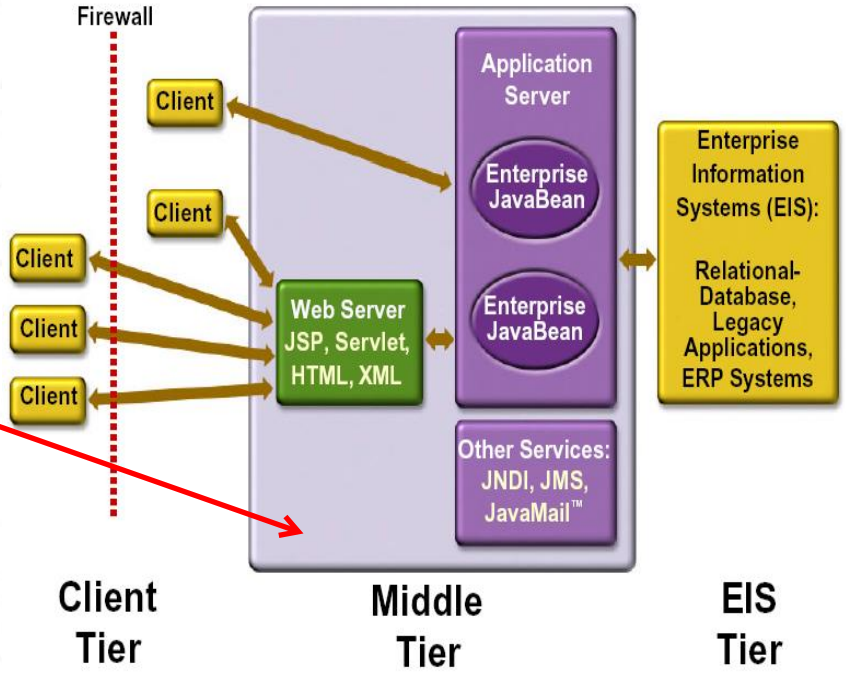
# JEE/Enterprise Java Beans

Java EE (Java Enterprise Edition) is a platform for building big, scalable apps that can handle lots of users and data. It ’s like an upgraded version of a basic app setup, making it easier to manage complex systems.

# Java EE Architecture

Normal Apps (Two-Tier): Usually, apps have just a client (like a web browser) and a server (where the data lives). Simple, but limited.

- Three/Four tiered applications that run in this way extend the standard two-tiered client and server model by placing a multithreaded application server between the client application and back-end storage




**Client**: The part users interact with (e.g., a browser or app).

**Application Server**: A middle layer that handles the app's logic and tasks (like a multitasking brain).

**Back-End Storage**: Where data (like databases) is kept.

Sometimes there's a fourth tier if the system gets even more complex (e.g., splitting tasks further).

# Java EE Containers

The application server uses containers to control and run parts of the app. Think of containers as organizers that manage specific jobs

- The application server maintains control and provides services through an interface or framework known as a *container*
- Server-side containers:
  - The server itself, which provides the Java EE runtime environment and the other two containers
  - An **EJB container** to manage EJB components
  - A **Web container** to manage servlets and JSP pages

# Java EE Components

- As said earlier, Java EE applications are made up of components
- A *Java EE component* is a self-contained functional software unit that is assembled into a Java EE application with its related classes and files and that communicates with other components
- Client components run on the client machine, which correlate to the client containers
- Web components -servlets and JSP pages
- EJB Components

Java EE apps are made of **components**—small, reusable pieces of code that do specific tasks:

- **Client Components**: Run on the user's device (e.g., a login screen).
- **Web Components**: Handle web pages (servlets and JSP).
- **EJB Components**: Do heavy lifting on the server (e.g., business logic or data management).

# Packaging Applications and Components

- Under Java EE, applications and components reside in Java Archive (JAR) files
- These JARs are named with different extensions to denote their purpose, and the terminology is important

# Various File types

- Enterprise Archive (EAR) files represent the application, and contain all other server-side component archives that comprise the application
- Web components reside in Web Archive (WAR) files
- Client interface files and EJB components reside in JAR files

**Packaging: How It's Stored**

- These components are packed into files called **JARs** (Java Archives):
  - **EAR (Enterprise Archive)**: Holds the whole app, including other files.
  - **WAR (Web Archive)**: Contains web components.
  - **JAR**: Holds EJB components or client files.

# EJB Components

EJBs (Enterprise JavaBeans) are special server-side components that handle big tasks:

- EJB components are server-side, modular, and reusable, comprising specific units of functionality
- They are similar to the Java classes we create every day, but are subject to special restrictions and must provide specific interfaces for container and client use and access
- We should consider using EJB components for applications that require scalability, transactional processing, or availability to multiple client types

# EJB Components- Major Types

- **Session beans (verbs of a system)**
  - These may be either *stateful* or *stateless* and are primarily used to encapsulate business logic, carry out tasks on behalf of a client, and act as controllers or managers for other beans
- **Entity beans (nouns of a system)**
  - Entity beans represent persistent objects or business concepts that exist beyond a specific application's lifetime; they are typically stored in a relational database
- **Message Driven Beans:**
  - Asynchronous communication with MOM
  - Conduit for non-Java EE resources to access Session and Entity Beans via JCA Resource adapters

**Session Beans** handle actions (stateful or stateless).
**Entity Beans** store lasting data (like database records).
**Message-Driven Beans** deal with messages from other systems.

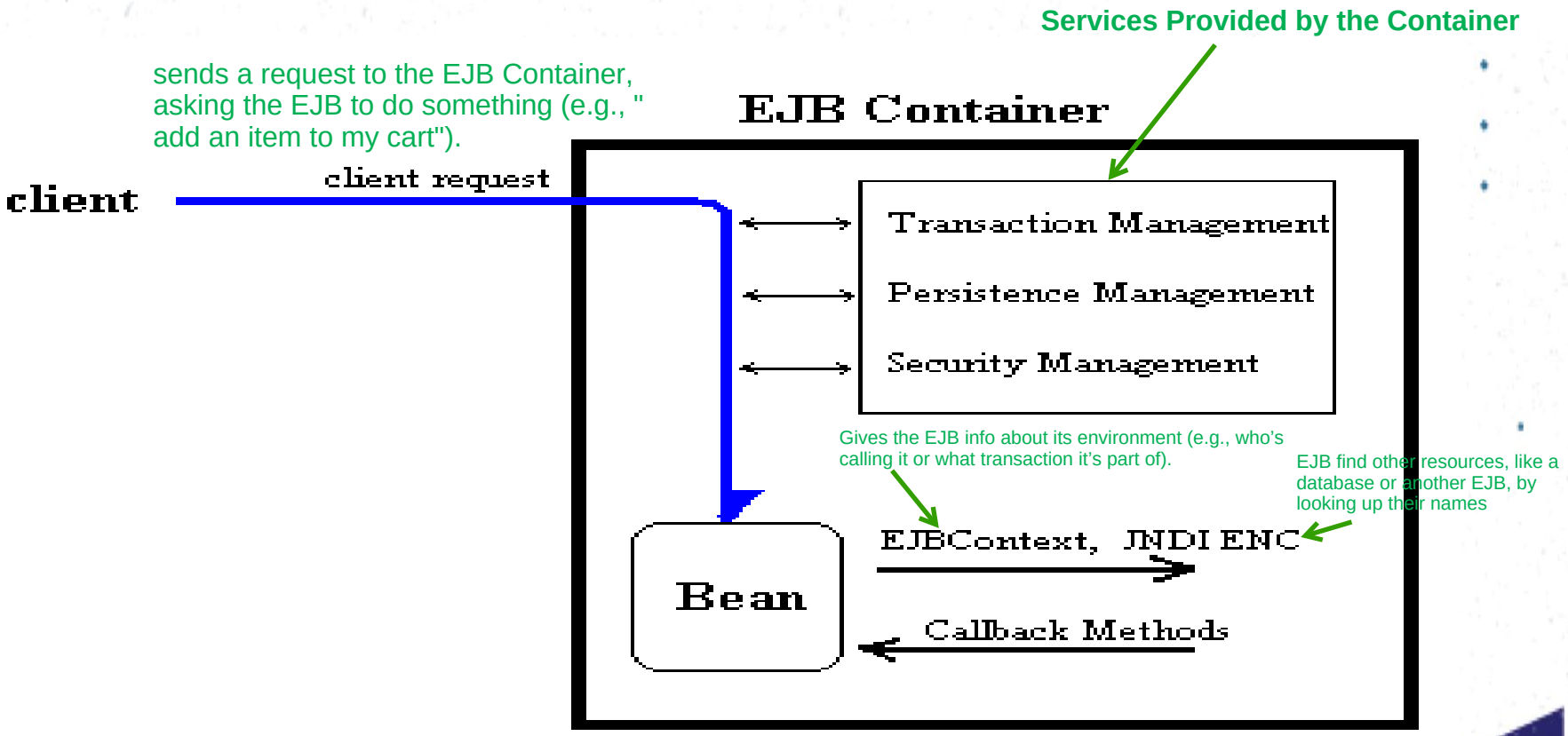
# Enterprise Java Server (EJS) / Application Server

- Part of an application server that hosts EJB containers
- EJBs do not interact directly with the EJB server
- EJB specification outlines eight services that must be provided by an EJB server:
  - Naming — Helps find EJBs by giving them names
  - Transaction — Manages "all-or-nothing" tasks
  - Security — Keeps things safe by controlling who can use the EJBs
  - Persistence — Saves data so it's not lost when the app stops
  - Concurrency — Lets multiple users or tasks use EJBs at the same time without messing things up
  - Life cycle — Controls when EJBs start, stop, or get cleaned up
  - Messaging — Handles sending and receiving messages between systems
  - Timer — Schedules tasks to happen at certain times

# EJB Container

- Functions as a runtime environment for EJB components beans
- Containers are transparent to the client in that there is no client API to manipulate the container
- Container provides EJB instance life cycle management and EJB instance identification.
- Manages the connections to the enterprise information systems (EISs)

# EJB Container(cont'd)




**EJB Containers manage enterprise beans at runtime**

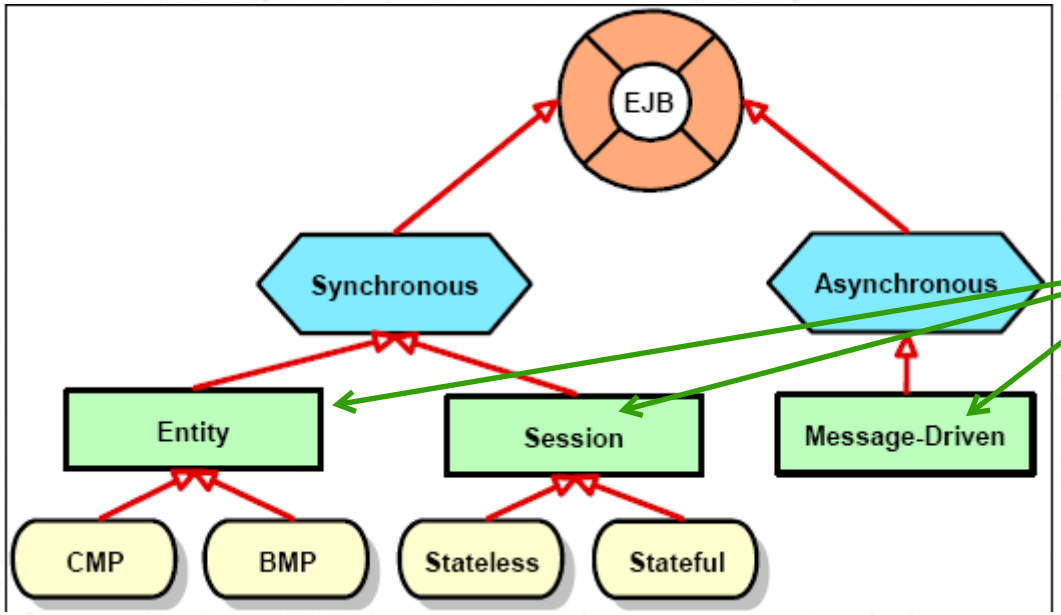
# EJB Client

- Finds EJBs via JNDI.
- Invokes methods on EJBs.

An **EJB Client** is a program or user that wants to use an EJB (Enterprise JavaBean) to do something, like adding an item to a cart or getting a total.
It has two main jobs:

1. **Finds EJBs via JNDI**: JNDI (Java Naming and Directory Interface) is like a phonebook—it helps the client look up the EJB by its name (e.g., "find me the ShoppingCart EJB").
2. **Invokes Methods on EJBs**: Once the client finds the EJB, it calls its methods to get work done (e.g., "add this product to the cart").

# EJB components



# EJB Interfaces - Local and Remote

- Local Interface
  - Used for invoking EJBs within the same JVM (process)
  - `@Local` annotation marks an interface local
  - Parameters passed by reference

Example: A web app and EJB running on the same server.

- Remote Interface

Used when the client and EJB are in different JVMs (different programs or machines).

  - Used for invoking EJBs across JVMs (processes)
  - `@Remote` annotation marks an interface remote
  - Parameters passed by value (serialization/de- serialization)

Note: An EJB can implement both interfaces if needed.

Example: A mobile app (client) calling an EJB on a remote server.

# Business Interface

- Defines business methods
- Session beans and message-driven beans require a business interface, optional for entity beans.
- Business interface do not extend local or remote component interface (unlike EJB2.x)
- Business Interfaces are POJIs (Plain Old Java Interfaces)

just a basic Java interface with no extra complexity.

# Business Interface - examples

- Shopping cart that maintains state

Stateful Shopping Cart (keeps track of the user's actions)

```
public interface ShoppingStatefulCart {
    void startShopping(String customerId);
    void addProduct(String productId);
    float getTotal();
}
```

- Shopping cart that does not maintain state

Stateless Shopping Cart (doesn't remember between actions)

```
public interface ShoppingStatelessCart {
    String startShopping(String customerId); //
  return cartId
    void addProduct(String cartId, String productId);
    float getTotal(String cartId);
}
```

This cart needs a cartId to keep track because it doesn't remember the user's actions (stateless).

# Stateless Session EJB (SLSB)

JB that does tasks for clients but doesn't remember anything between requests (no "conversational state").

- Does not maintain any conversational state with client
- Instances are pooled to service multiple clients
- `@Stateless` annotation marks a been stateless.
- Lifecycle event callbacks supported for stateless session beans (optional)
  - `@PostConstruct` occurs before the first business method invocation on the bean
  - `@PreDestroy` occurs at the time the bean instance is destroyed

# Stateless Session EJB example (1/2)

- The business interface:

```
public interface HelloSessionEJB3Interface{
    public String sayHello();
}
```

# Stateless Session EJB
## example (2/2)

- The stateless bean with local interface:

java Collapse Wrap Copy

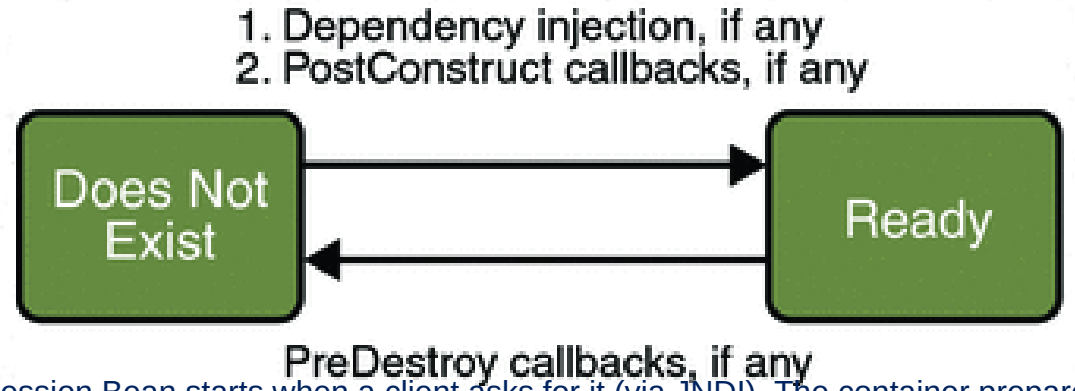
Explain

```java
import javax.ejb.*;
import javax.annotation.*;

@Local({HelloSessionEJB3Interface.class})
@Stateless
public class HelloSessionEJB3 implements HelloSessionEJB3Interface {
    public String sayHello() {
        return "Hello from Stateless bean";
    }

    @PreDestroy
    void restInPeace() {
        System.out.println("I am about to die now");
    }
}
```

- This EJB is stateless (`@Stateless`), uses a local interface (`@Local`), and implements the `sayHello()` method.
- When the EJB is about to be destroyed, it prints "I am about to die now" (`@PreDestroy`).

# Lifecycle of a Stateless Session Bean

- A client initiates the life cycle by obtaining a reference
- The container invokes the `@PostConstruct` method, if any
- The bean is now ready to have its business methods invoked by clients




The lifecycle of a Stateless Session Bean starts when a client asks for it (via JNDI). The container prepares the bean by calling @PostConstruct (if it exists), and then the bean is ready for the client to use by calling its methods. It's a simple process because the bean doesn't need to remember anything between requests!

# Stateful Session EJB (SFSB)

- Maintains conversational state with client — track of the client's actions during a session
  Each bean instance is dedicated to one client. If another client comes along, they get their own separate bean instance.
- Each instance is bound to specific client session
- Support callbacks for the lifecycle events listed on the next slide — The container can call special methods at different stages of the bean's life (e.g., when it starts, pauses, or ends).

# Stateful Session EJB – example (1/2)

meaning the client can call this EJB from a different machine.

- Define remote business interface (remote can be marked in bean class also) :

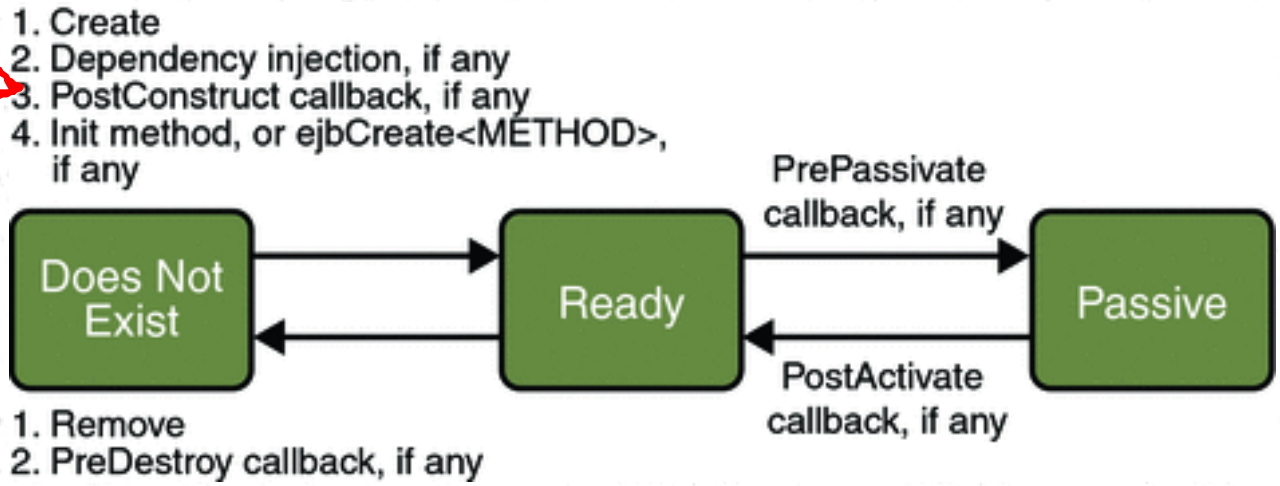
```
@Remote public interface ShoppingCart {
    public void addItem(String item);
    public void addItem(String item);
    public Collection getItems();
}
```

# Stateful Session EJB – example (2/2)

```java
@Stateful
public class CartBean implements ShoppingCart {
    private ArrayList items;

    @PostConstruct
    public void initArray() {
        items = new ArrayList();
    }

    public void addItem(String item) {
        items.add(item);
    }

    public void removeItem(String item) {
        items.remove(item);
    }

    public Collection getItems() {
        return items;
    }

    @Remove
    void logoff() {
        items = null;
    }
}
```

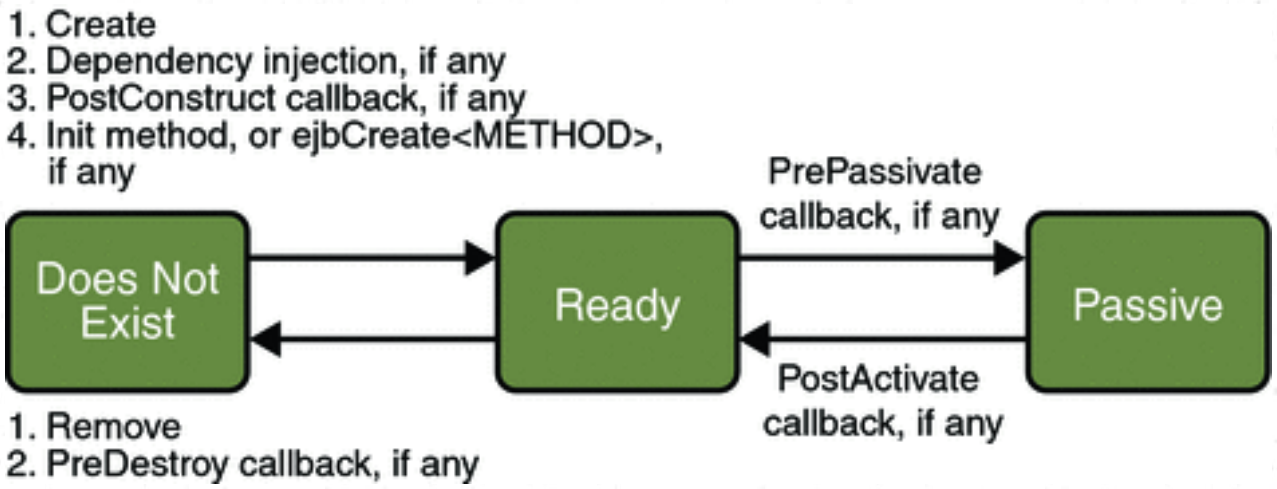
# Lifecycle of a Stateful Session Bean

- Client initiates the lifecycle by obtaining a reference
- Container invokes the `@PostConstruct` and `@Init` methods, if any
- Now bean ready for client to invoke business methods



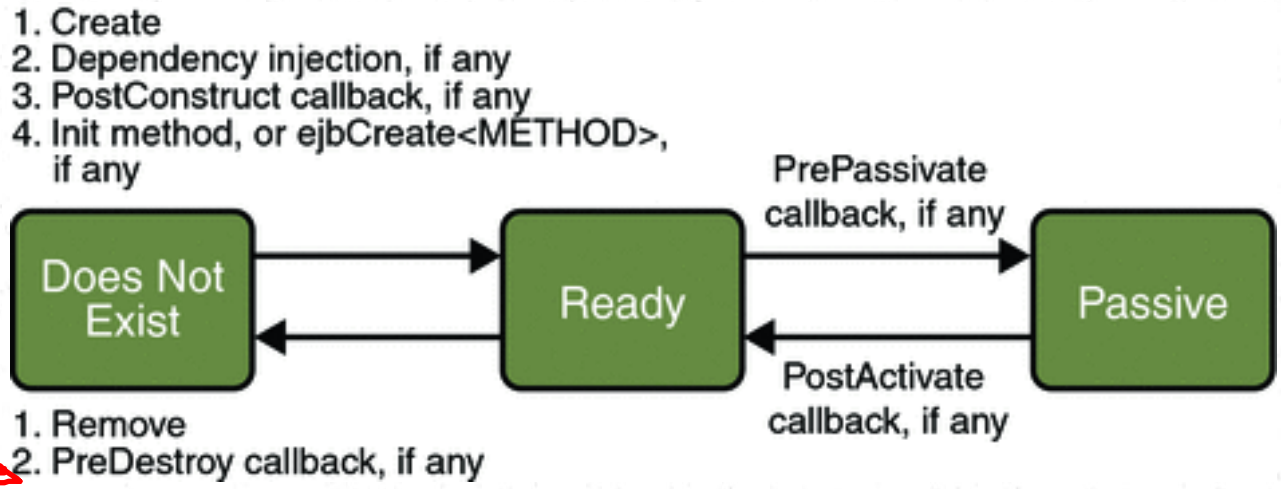
# Lifecycle of a Stateful Session Bean

- While in ready state, container may passivate and invoke the `@PrePassivate` method, if any
- If a client then invokes a business method, the container invokes the `@PostActivate` method, if any, and it returns to ready stage



# Lifecycle of a Stateful Session Bean

- At the end of the life cycle, the client invokes a method annotated `@Remove`
- The container calls the `@PreDestroy` method, if any



# Entity EJB (1)

## Entity EJB (Enterprise Java Bean)

represents persistent data stored in a database. Unlike normal Java objects, entity beans exist beyond the lifecycle of an application because they are backed by a permanent storage mechanism, usually a relational database.

- **It is permanent.** Standard Java objects come into existence when they are created in a program. When the program terminates, the object is lost. But an entity bean stays around until it is deleted. In practice, entity beans need to be backed up by some kind of permanent storage, typically a database.

Entity EJBs store data permanently in a database.Standard Java objects disappear when the program stops, but entity beans remain in the database until explicitly deleted.

- **It is identified by a primary key.** Entity Beans must have a primary key. The primary key is unique -- each entity bean is uniquely identified by its primary key. For example, an "employee" entity bean may have Social Security numbers as primary keys.

- Note: Session beans do not have a primary key.

# Entity Bean Class

```java
@Entity  // Marks this class as an Entity EJB
public class Employee implements Serializable {

    @Id  // Marks this as the Primary Key
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    private String name;
    private double salary;

    // Default constructor (Required by JPA)
    public Employee() { }

    // Getter and Setter for id
    public Long getId() { return id; }
    public void setId(Long id) { this.id = id; }
```

- `@Entity` annotation marks a class as Enity EJB
- Persistent state of an entity bean is represented by non-public instance variables
- For single-valued persistent properties, these method signatures are:

```
<Type> getProperty()
void setProperty(<Type> t)
```

- Must be a non-final concrete class
- Must have public or protected no-argument constructor
- No methods of the entity bean class may be final
- If entity bean must be passed by value (through a remote interface) it must implement `Serializable` interfa

# Entity EJB

- CMP (Container Managed Persistence)
  - Container maintains persistence transparently using JDBC calls
- BMP (Bean Managed Persistence)
  - Programmer provides persistence logic
  - Used to connect to non-JDBC data sources like LDAP, mainframe etc.
  - Useful for executing stored procedures that return result sets

# Entity EJB – example (1)

```
@Entity // mark as Entity Bean
public class Customer implements Serializable {
  private Long id;
  private String name;
  private Collection<Order> orders = new
   HashSet();
  @Id(generate=SEQUENCE) // primary key
  public Long getId() {
  return id;
  }
  public void setId(Long id) {
  this.id = id;
  }
```

# Entity EJB – example (2)

```
@OneToMany // relationship between
 Customer and Orders
public Collection<Order> getOrders() {
return orders;
}
public void setOrders(Collection<Order>
 orders) {
this.orders = orders;
}
}
```

# EntityManager

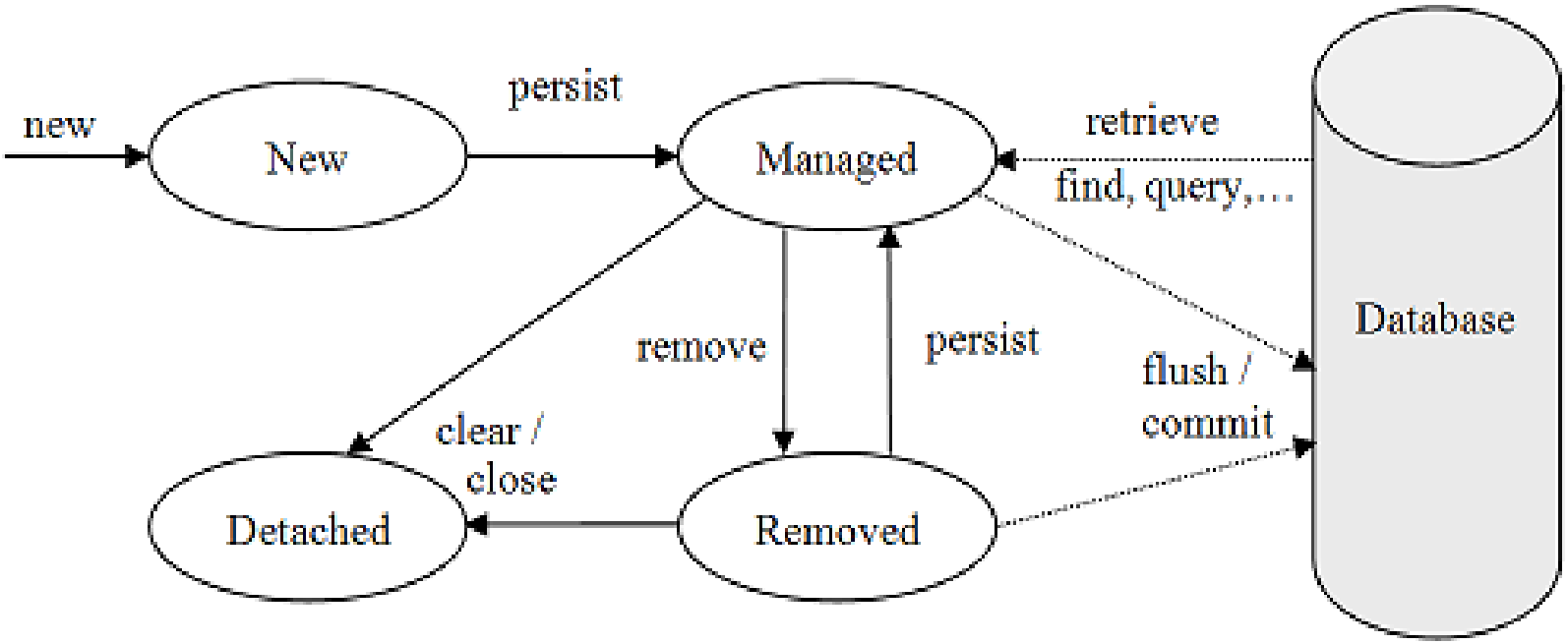
- EntityManager API is used to:
  - create and remove persistent entity instances
  - to find entities by their primary key identity, and to query over entities
- EntityManager supports EJBQL and (non-portable) native SQL

# Entity Bean Lifecycle

Entity bean instance has four possible states:

- **New** entity bean instance has no persistent identity, and is not yet associated with a persistence context.
- **Managed** entity bean instance is an instance with a persistent identity that is currently associated with a persistence context.
  Associated with a database and managed by an EntityManager.
- **Detached** entity bean instance is an instance with a persistent identity that is not (or no longer) associated with a persistence context.
  Removed from the persistence context but still exists in the database.
- **Removed** entity bean instance is an instance with a persistent identity, associated with a persistence context, scheduled for removal from the database.
  Marked for deletion and will be removed from the database.

# Entity Bean Lifecyle



# Example of Use of EntityManager API

```
@Stateless
public class OrderService {
    @Inject
    private EntityManager em;

    public void addOrder(int custID, Order newOrder) {
        Customer cust = em.find(Customer.class, custID);
        cust.getOrders().add(newOrder);
        newOrder.setCustomer(cust);
    }
}
```

# Message Driven EJB

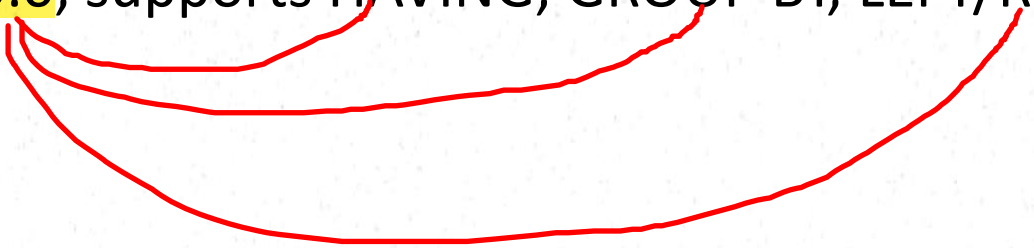
- Invoked by asynchronously by messages
- Cannot be invoked with local or remote interfaces
- `@MessageDriven` annotation with in class marks the Bean message driven
- Stateless
- Transaction aware

# Message Driven EJB example

```
import javax.jms.Message;
import javax.jms.MessageListener;
import javax.ejb.MessageDriven;
@MessageDriven
public class MessageDrivenEJBBean implements
 MessageListener {

    public void onMessage(Message message) {
     if(message instanceof MyMessageType1)
          doSomething(); // business method 1
     if(message instanceof MyMessageType2)
          doSomethingElse(); // business method 2

    }
}
```

# EJB Query Language (EJBQL)

- EJBQL : RDBMS vendor independent query syntax
- Query API supports  both static queries (i.e., named queries) and dynamic queries.
- Since EJB3.0, supports HAVING, GROUP BY, LEFT/RIGHT JOIN etc.



# EJBQL - examples

- Define named query:

**Example: Named Query**

java Copy Edit

```java
@NamedQuery(
    name="findAllCustomersWithName",
    query="SELECT c FROM Customer c WHERE c.name LIKE :custName"
)
```

Explain

**Executing Named Query**

java Copy Edit

```java
List<Customer> customers = em.createNamedQuery("findAllCustomersWithName")
                            .setParameter("custName", "John")
                            .getResultList()
```

# Deploying EJBs

- EJB 3.0 annotations replace EJB 2.0 deployment descriptors in almost all cases
- Values can be specified using annotations in the bean class itself
- Deployment descriptor *may be used* to override the values from annotations

# Some EJB Servers (Application Servers)

| Company | Product |
| --- | --- |
| • IBM | WebSphere |
| • BEA Systems | BEA WebLogic |
| • Sun Microsystems | Sun Application Server |
| • Oracle | Oracle Application Server |
| • JBoss | JBoss |

# Advantages of EJB

- Simplifies the development of middleware components that are secure, transactional, scalable & portable.
- Simplifies the process to focus mainly on business logic rather than application development.
- Overall increase in developer productivity
- Reduces the time to market for mission critical applications

# Summary

- Component based development focuses in grouping cohesive functions into reusable units called components
- Components interact with other components and clients through interfaces
- Distributed computing makes extensive use of component based development as it allows the different functionalities to be distributed over different systems.
- Different component development technologies are there by different vendors (Java EE, Spring Framework.NET WCF services, etc.)