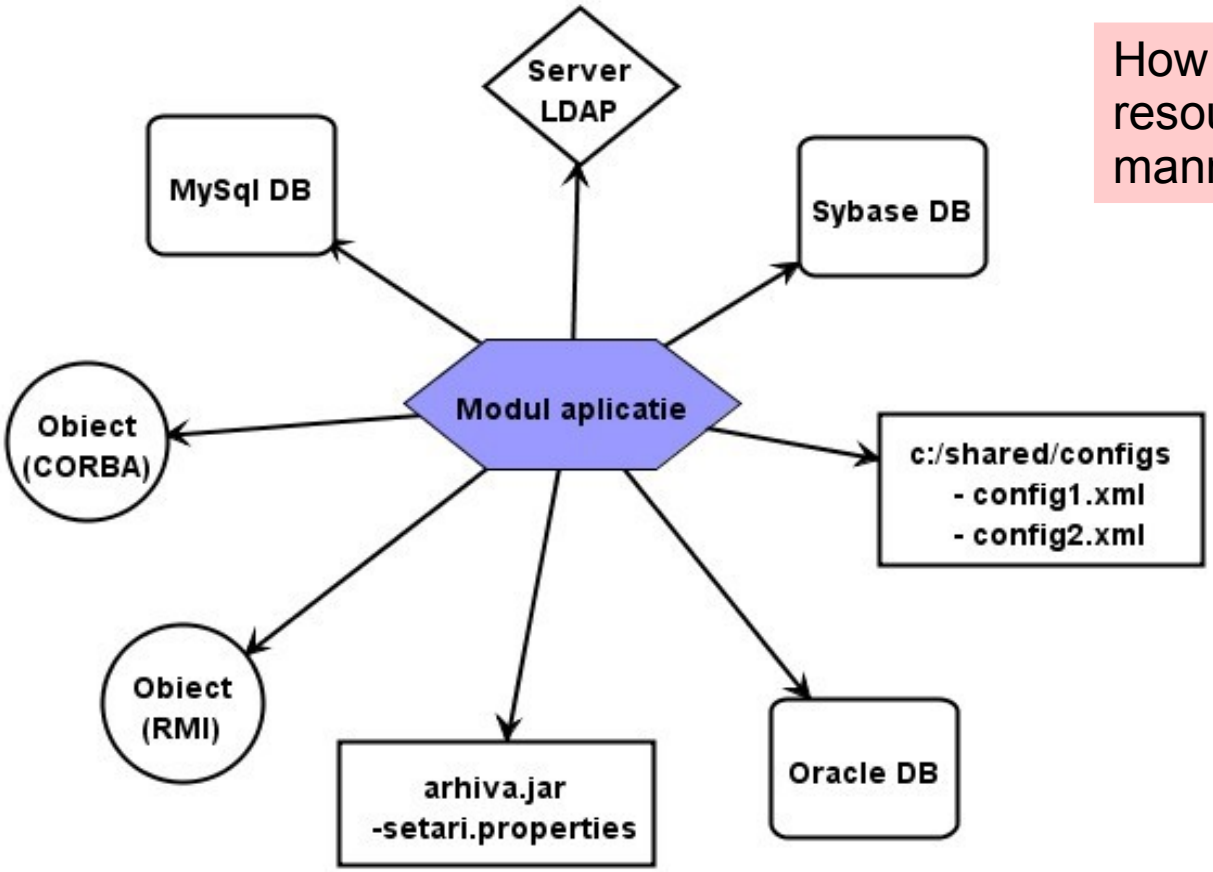
Java Technologies

Resources and JNDI

The Context

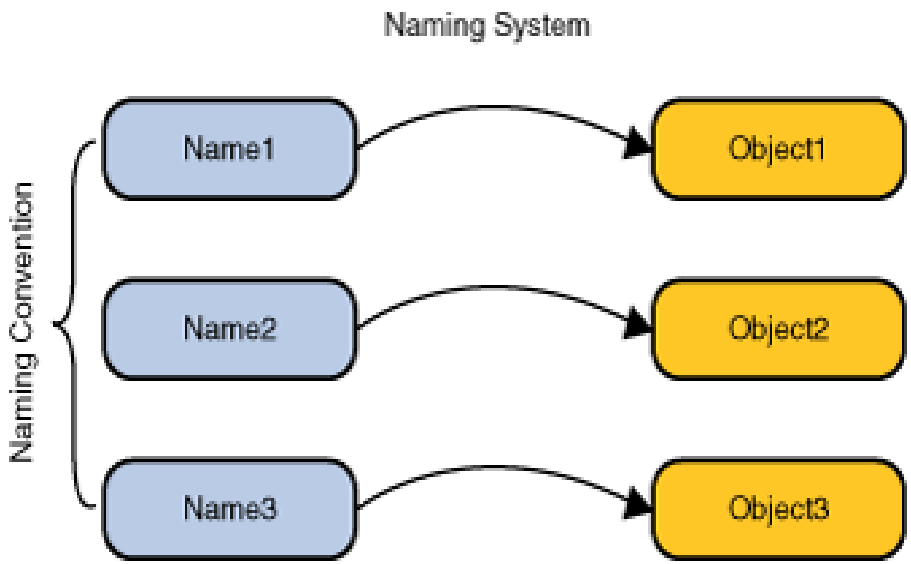


How to access all these resources in a similar manner?

A **resource** is a program object that provides connections to other systems such as: database servers, messaging systems, etc.

Naming Services

A **naming service** represents a mechanism by which *names* are associated with *objects* and objects are found based on their names.



file name	→ file content
domain name	→ IP address
RMI name	→ Java object

Examples: DNS, RMI Registry, COS Naming

Naming Concepts

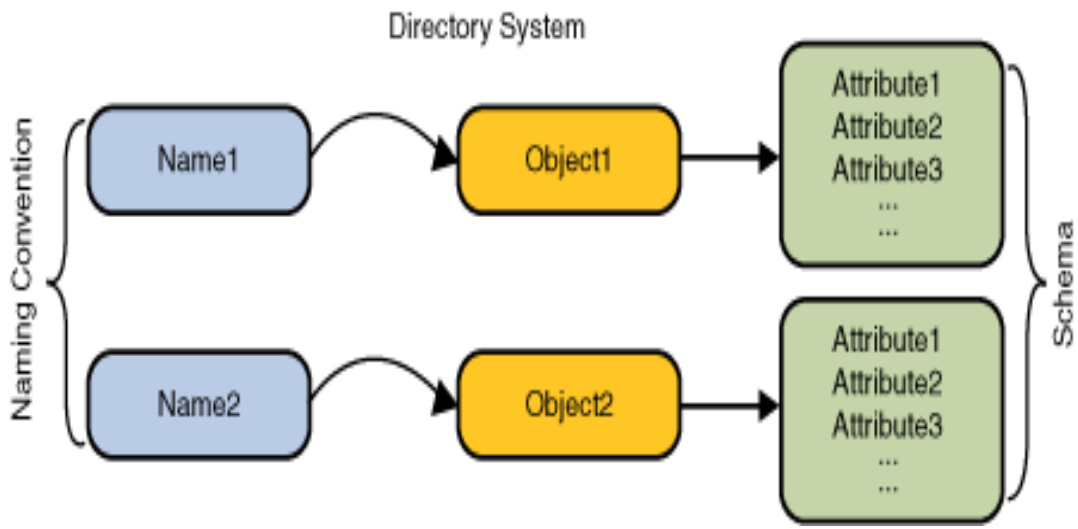
- **Binding**: The association of a name with an object
  - fenrir.info.uaic.ro → 85.122.23.145
- **Context:** A set of name-to-object bindings using an associated *naming convention*. A context provides a *lookup (resolution)* operation
  - File System: **c:** (*c:\bin*), Domain: **.ro** (*uaic.ro*)
- **Naming System**: a connected set of contexts of the same type. A naming system provides a *naming service* for performing naming-related operations
- **Namespace**: the set of all possible names in a naming system.

Directory Services

A directory service associates *names with objects* and *also* associates such *objects with attributes.*

Directory service =

naming service + objects containing attributes



name → person
+ address
+ phone
+ mail

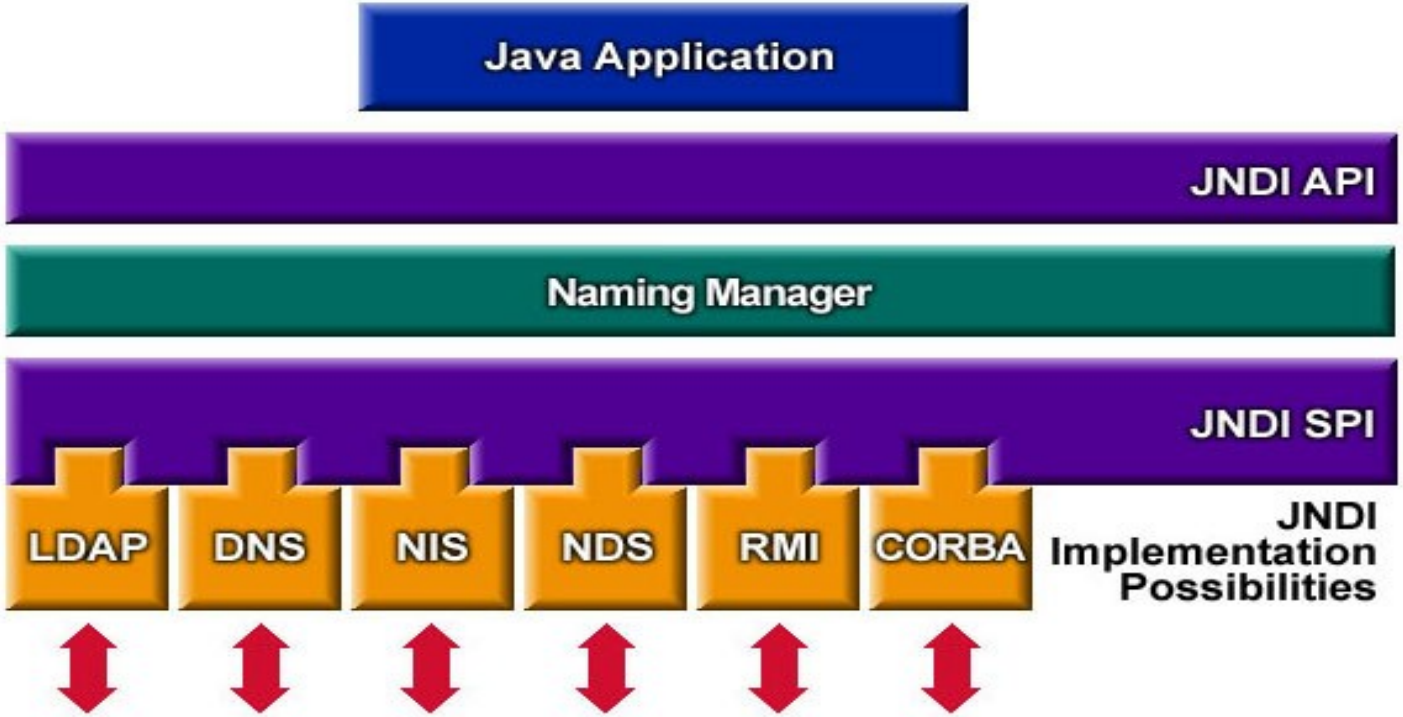
Examples: LDAP, NIS, Oracle Directory Server, ADS

Directory Concepts

- **Attributes**: A directory object can have attributes. An attribute has an *attribute identifier* and a *set of attribute values*.
  - John Doe + (**mail**: *john.doe@yahoo.com, john.doe@gmail.com*)
- **Directory:** a connected set of directory objects
- **Directory Service**: a service that provides operations for creating, adding, removing, and modifying the attributes associated with objects in a directory.
- **Search service**: *reverse lookup* or *content-based searching*, using *queries (search filters)* that specify logical expressions based on the attributes:
  - (&(cn=John Doe)(l=Dallas))

What is JNDI?

- Java Naming and Directory Interface
- JNDI service enables components to locate other components and resources.



Common JNDI Operations

- Setting the *initial context*

```
Context ctx = new InitialContext(properties);
```

- *Looking* for an object

```
Printer printer = (Printer)ctx.lookup("treekiller");

printer.print(report);
```

- Managing the *bindings*

```
context.bind(name, object);

        .rebind

        .unbind

        .listBindings
```

Remote Method Invocation

- The administrator configures the initial context, at the application server level:
  - `java.naming.factory.initial = com.sun.jndi.rmi.registry.RegistryContextFactory`
  - `java.naming.provider.url = rmi://someAddress:port`
- In an application component:

```
RemoteObject ref = (RemoteObject) registry.lookup("objectName");
```

- How will we actually use this:

```
@EJB(lookup="objectName")
RemoteObject ref;
```

Accessing Resources using JNDI

- A **resource** is a program object that provides connections to other systems, such as database servers and messaging systems.
- Each resource object is identified by a *unique*, *people-friendly* name → **the JNDI name**.
- Resources are **created by an administrator**, in a JNDI namespace, using server management tools, such as GlassFish Admin Console.
- Applications access resources:
  - either using annotations to inject them,
  - or by making direct calls to the JNDI API.

@Resource

- The **Resource annotation** marks a resource that is needed by the application.
- When the annotation is applied <u>to a field or method</u>, the container will **inject** an instance of the requested resource into the application component when the component is initialized.
- Example:

```
@Resource(name="sampleDB")
private javax.sql.DataSource myDB;
```

javax.sql.DataSource

- **javax.sql**: provides the API for server side data source access and processing.
- **DataSource**: represents a factory for connections to the physical database.
  - The preferred means of getting a connection (alternative to the DriverManager)
  - Typically registered with a JNDI naming service
- The DataSource is implemented by a vendor:
  - Basic implementation → standard Connection
  - **Connection pooling** implementation
  - Distributed transaction implementation

“Classical“ JDBC Connection

```
public class TheWellKnownDatabaseConnectionSingleton {

  private static Connection connection = null;

  public static Connection getConnection() {
    if (connection != null) {
      return connection;
    }
    try {
      Class.forName("org.postgresql.Driver").newInstance();
      connection = DriverManager.
          getConnection("jdbc:postgresql://localhost:5432/sample");
    } catch(Exception e) {
      return null;
    }
    return connection;
  }
}
```

- When to create the connection?
- When to close the connection?

DataSource Connection

```
// Instantiate a DataSource object
org.postgresql.ds.PGSimpleDataSource ds;
ds = new org.postgresql.ds.PGSimpleDataSource();

// Set up connection properties
ds.setUser("user");
ds.setPassword("passwd");
ds.setDatabaseName("sample");
ds.setServerName("localhost");
ds.setPortNumber(5432);

// Open a connection
Connection conn = ds.getConnection();
System.out.println("Connection successful!");
```

Still ugly...

A DataSource object has properties that can be modified when necessary. This is good...

Using a DataSource with JNDI

1. Registering the DataSource object in a naming service

Administrator

```
org.postgresql.ds.PGSimpleDataSource ds;
...
Context ctx = new InitialContext(env);
ds = new org.postgresql.ds.PGSimpleDataSource();
ds.setDatabase("jdbc:postgresql://localhost:5432/sample");
ctx.bind("jdbc/sample", ds);
ctx.close();
```

2. Creating a connection using JNDI

Programmer

```
...
Context ctx = new InitialContext(env);
DataSource ds = (DataSource)ctx.lookup("jdbc/sample");
connection = ds.getConnection();
```

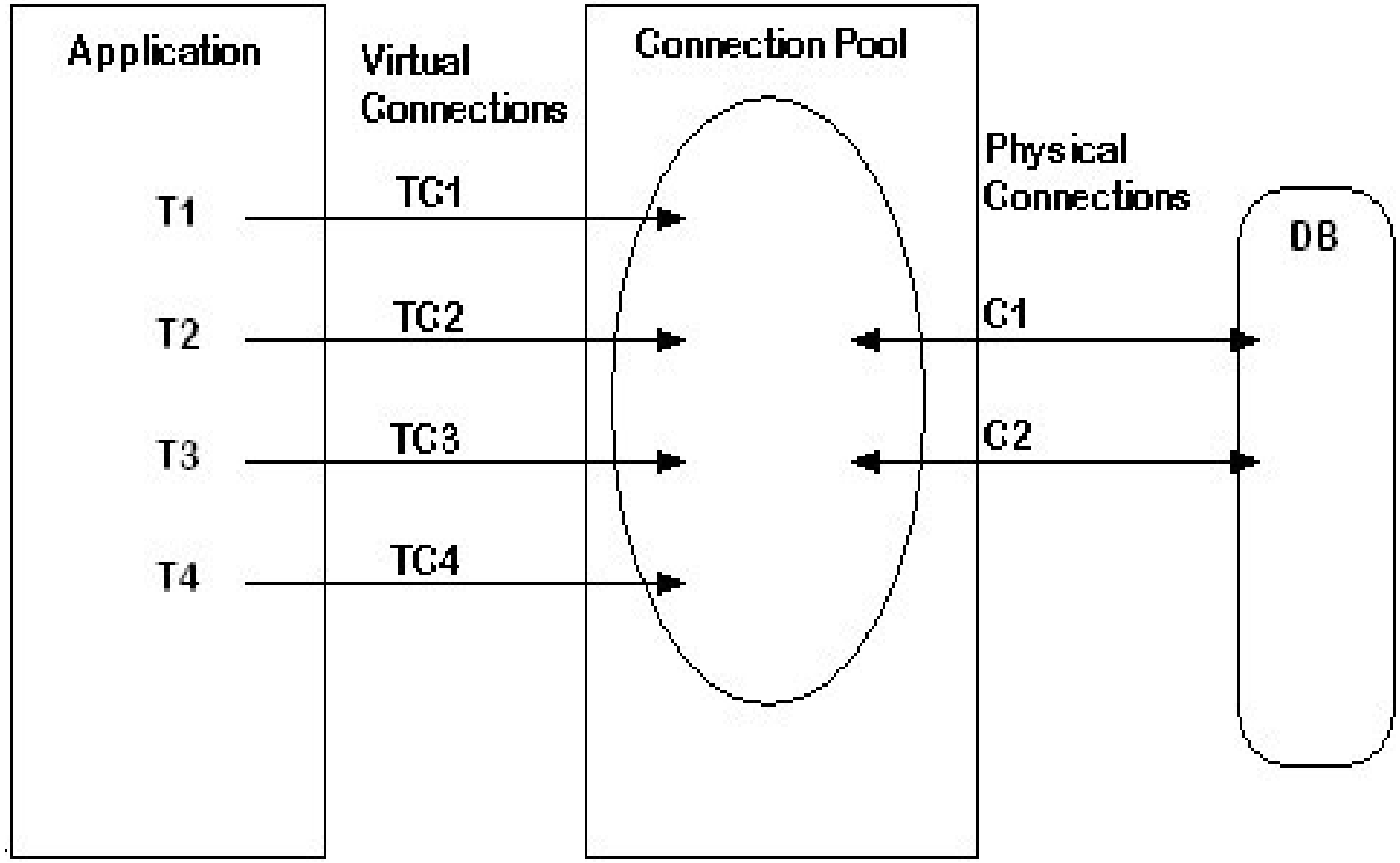
```
@Resource(name="jdbc/sample")
private DataSource ds;
```

Connection Pool

- **Reusable set (cache) of database connections**
- Helps to alleviate connection management overhead and decrease development tasks for data acces
- Improves the response time of any application that requires connections, especially Web-based applications
- **How it works**:
  - The Application Server enables administrators to establish a pool of backend connections that applications can share.
  - An application **obtains** a connection from the pool, **uses** it for a specified period and then **returns** it to the pool.

Connection Pool

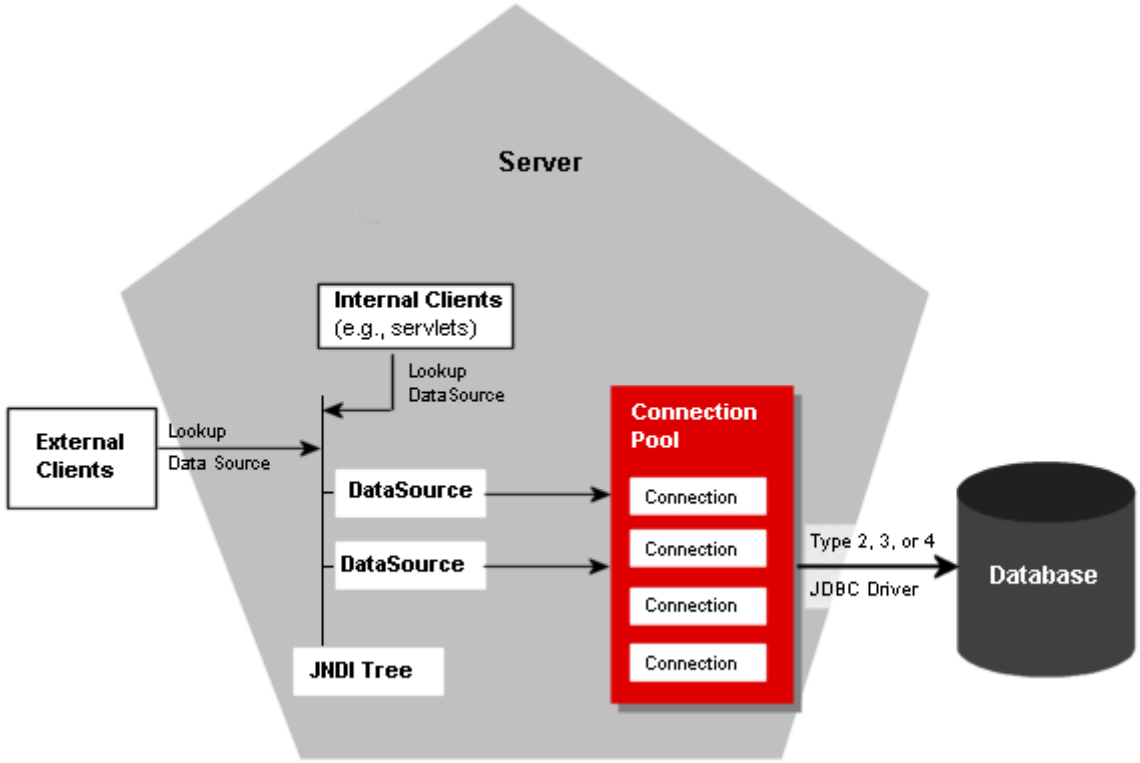
Reusable set (cache) of database connections



Object Pool - Creational Design Pattern

DataSource

- Applications use a **data source** to obtain connections to a relational database: associated with a *connection pool*, using a *JDBC resource*.



JDBC Connection Pool

- localhost:4848 → Admin console
- Resources → JDBC Connection Pools
- Resource Type: *javax.sql.DataSource*
  - *ConnectionPoolDataSource*
  - *XADataSource* (two-phase commit)
- Datasource Classname (vendor specific)
- Properties: configure database access params
  - serverName, portNumber, databaseName, user, password, *url*

Configuring a CP

- **Pool Settings**
  - Initial and Minimum Pool Size (8 connections)
  - Maximum Pool Size (32)
  - Pool Resize Quantity (2)

    Number of connections to be removed when pool idle timeout expires
  - Idle Timeout (300 seconds)

    Maximum time that connection can remain idle in the pool
  - Max Wait Time (60000 ms)

    Amount of time caller waits before connection timeout is sent
- **Transactions**
  - Non Transactional Connections

    Avoiding the overhead incurred in enlisting and delisting connections in transaction contexts (accessing a read-only database, for example)
  - Transaction Isolation

Advanced Configuration

- Statement Timeout

  Enables termination of abnormally long running queries

- Statement Cache Size

  Allows statement caching

- Init SQL

  An SQL string to be executed whenever a connection is created from the pool

- Slow Query Log Threshold

  SQL queries that exceed this time in seconds will be logged

- Log JDBC Calls

  Tracing of all JDBC interactions including SQL

- SQL Trace Listeners

Connection Leaking and Validation

- A **connection leak** means some of the database request/transaction are not getting closed properly or are not getting committed and finally those connections are getting abondoned and closed permanently
  - Connection Leak Timeout, Reclaim, etc.
- **Connection validation** ensures that connections aren't assigned to your application after the connection has already gone **stale** (no longer connected to the DB actively).
  - Validation method: table, auto-commit, meta-data, custom
  - Close all connections and reconnect on failure, otherwise reconnect only when used

JDBC Resource

- A data source is called a *JDBC resource*.

```
Admin Console
    -Resources
        -Connection Pools
            *postgres/sample_pool
        -JDBC Resources
            *postgres/sample
```

- Accessing the resource
  - using JNDI lookup

```
InitialContext ic = new InitialContext();
DataSource ds = (DataSource) ic.lookup("jdbc/sample");
```

  - using annotations

```
@Resource(mappedName = "jdbc/sample")
private DataSource ds;
```

Dependency Injection

- Using annotation and the JNDI name

```
public class MyServlet extends HttpServlet {
  @Resource(mappedName = "jdbc/sample")
  private DataSource sample;
  ...
}
```

- Using an abstract resource name mapped to the JNDI name in the server descriptor (glassfish-web.xml)

```
<glassfish-web-app>
  <resource-ref>
    <res-ref-name>myData</res-ref-name>
    <jndi-name>jdbc/sample</jndi-name>
  </resource-ref>
  ...
</glassfish-web-app>
```

```
...
jdbc/sample1
jdbc/sample2
jdbc/test
...
```

```
@Resource(name = "myData")
private DataSource ds;
```

Configuring the DataSource

- glassfish-resources.xml → /WEB-INF

```
<resources>
  <jdbc-connection-pool name="sample_pool">
    <property name="serverName" value="localhost"/>
    <property name="portNumber" value="5432"/>
    <property name="databaseName" value="sample"/>
    <property name="user" value="dba"/>
    <property name="password" value="sql"/>
    <property name="driverClass" value="org.postgresql.Driver"/>
  </jdbc-connection-pool>

  <jdbc-resource jndi-name="java:app/jdbc/sample"
     pool-name="sample_pool"/>

</resources>
```

- When the application is deployed, the server reads in the resource declarations, and creates the necessary resources.

GlassFish *asadmin*

- The **asadmin** utility performs administrative tasks for Oracle GlassFish Server from the command line or from a script. You can use this utility instead of the Administration Console interface.
- $GLASSFISH-PATH$/bin
- Commands:
  - create-jdbc-connection-pool
  - create-jdbc-resource
  - ...

Example

- *sample-create-resources.bat*

```
c:\glassfish4\glassfish\bin\asadmin.bat

  multimode --file sample-create-resources.txt
```

- *sample-create-resources.txt*

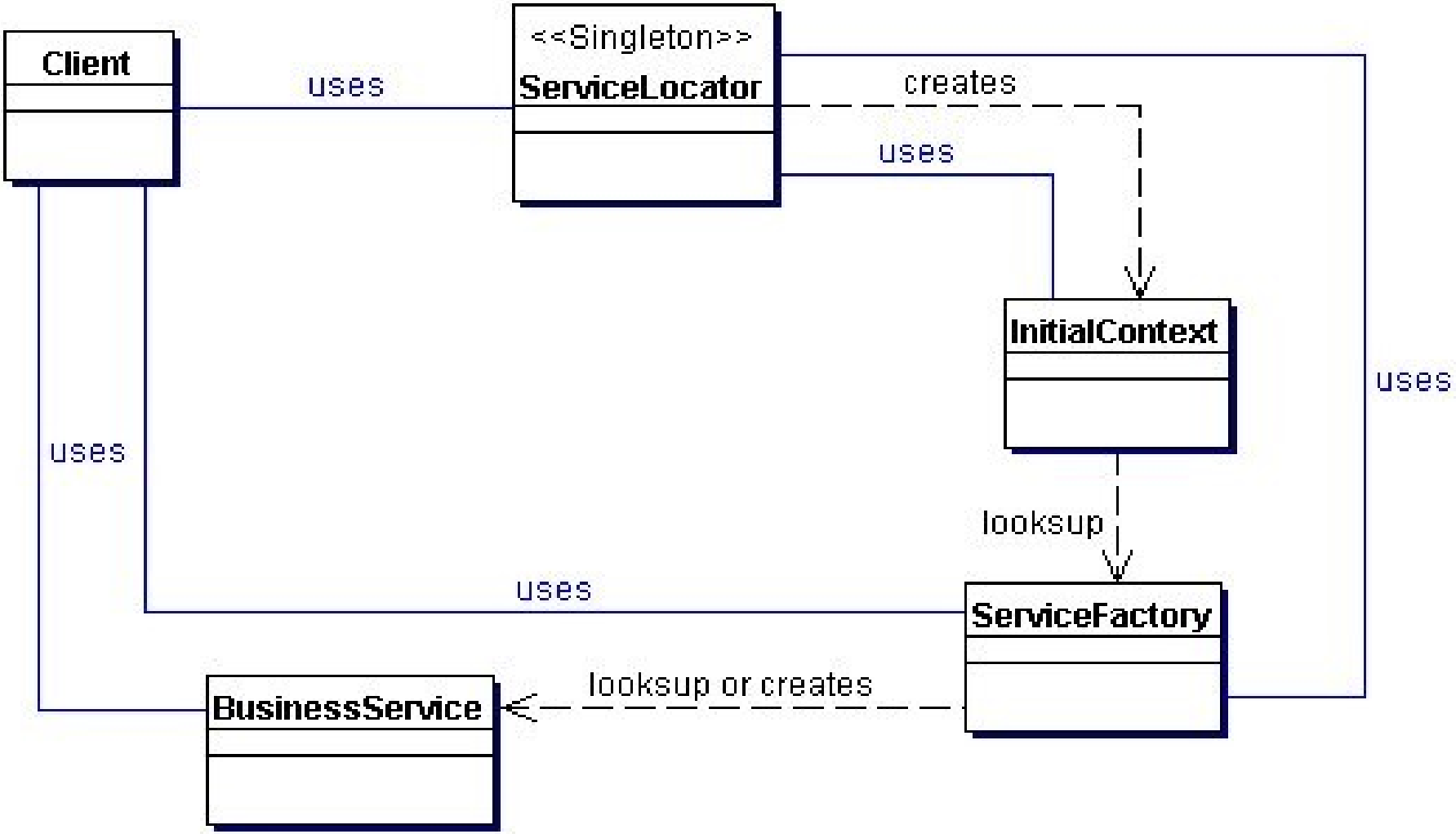
```
create-jdbc-connection-pool
 --datasourceclassname  org.postgresql.ds.PGSimpleDataSource
 --restype  javax.sql.DataSource
 --driverclassname=org.postgresql.Driver
 --property
serverName=localhost:portNumber=5432:databaseName=sample:user=dba:pass
word=sql sample_pool

create-jdbc-resource --connectionpoolid sample_pool jdbc/sample
```

Service Locator Pattern

- **The Context**: Service lookup and creation involves complex interfaces and network operations.
- **The Solution:**
  - abstract the JNDI usage
  - hide the complexities of initial context creation and object lookup.
  - multiple clients can reuse the Service Locator object to reduce code complexity, provide a single point of control, and improve performance by providing a caching facility.

Service Locator Class Diagram



Service Locator Example

```
public class MyServiceLocator {
  private InitialContext ic;

  public MyServiceLocator() {
    try {
      ic = new InitialContext();
    } catch (NamingException ne) {
      throw new RuntimeException(ne);
  }
}

  private Object lookup(String jndiName) throws NamingException {
    return ic.lookup(jndiName);
  }

  public DataSource getDataSource(String dataSourceName)
      throws NamingException {
    return (DataSource) lookup(dataSourceName);
  }
}
```

Caching Service Locator Example

```
public class MyCachingServiceLocator {
  private InitialContext ic;
  private Map<String,Object> cache;

  private MyCachingServiceLocator() throws NamingException {
    ic = new InitialContext();
    cache = Collections.synchronizedMap(new HashMap<>());
  }

  private Object lookup(String jndiName) throws NamingException {
    Object cachedObj = cache.get(jndiName);
    if (cachedObj == null) {
      cachedObj = ic.lookup(jndiName);
      cache.put(jndiName, cachedObj);
    }
    return cachedObj;
  }
}
```