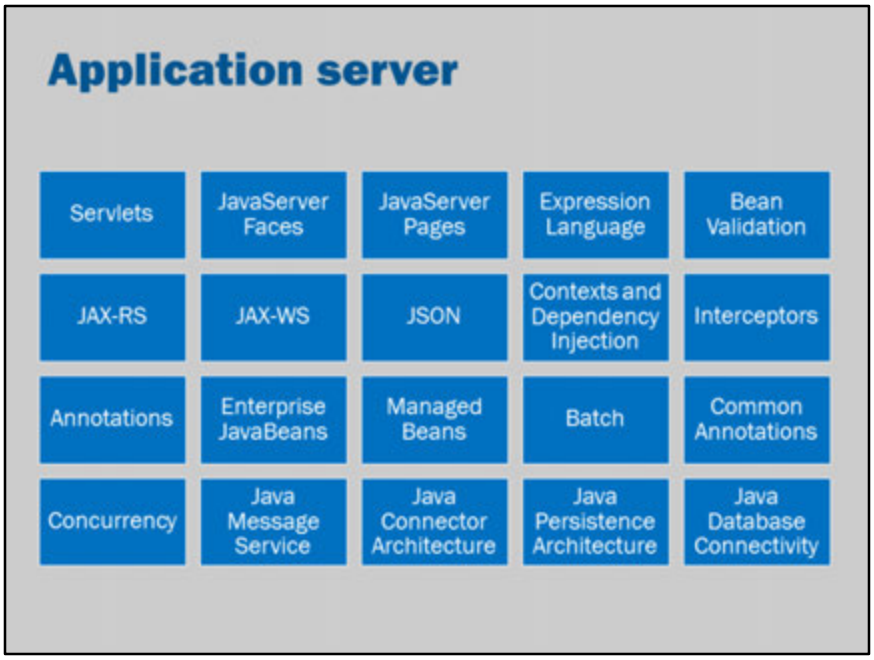
For this week, I recommend studying Chapter 2 of "Beginning Java EE 7".
http://find.lib.uts.edu.au/?R=OPAC_b2874770

We have been using a few container services and annotations but they have not been properly explained.

We used InitialContext.doLookup to access a JDBC connection that has been configured by the container.

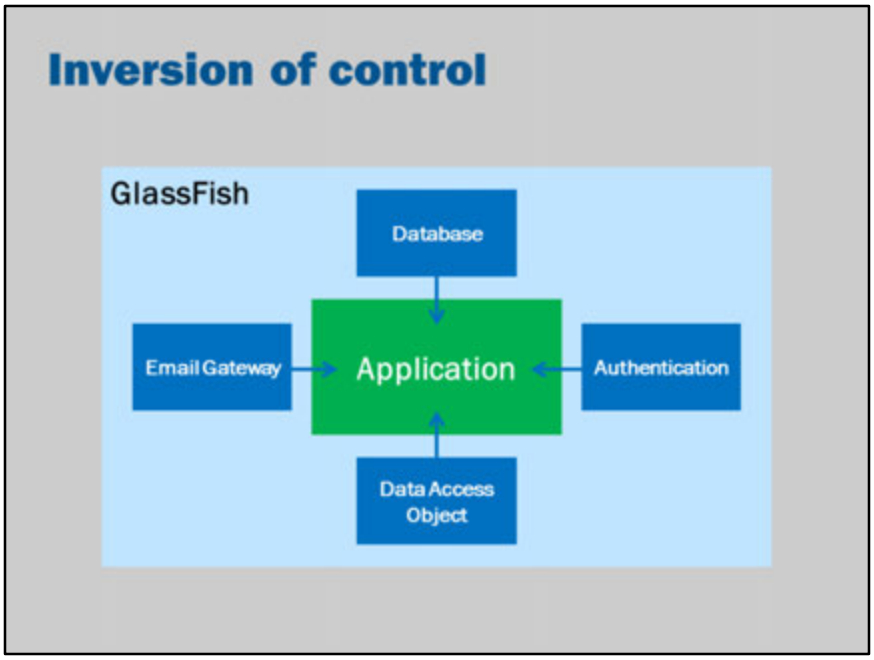
We declared our JSF backing beans @Named and @RequestScoped.

In this lecture, I hope to give you some more detail about what these mean.



Recall that an application server provides many features and services.
These services need to be configured for the deployment environment.
The services may be implemented differently for different Application servers.
The same service may need to be configured in two different ways (e.g., two separate email gateways, for sending internal and external email).

How can we access these services from within Java?
How can we access the services in a convenient and uniform way?

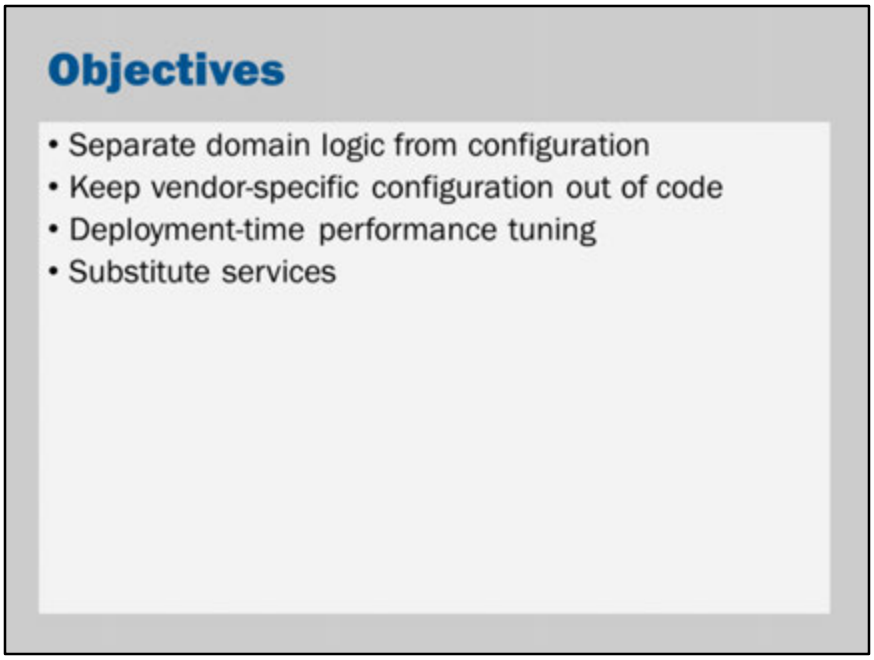


This is our situation.
We have an application running in GlassFish.
The application makes use of many different services.
We need a way to make these services available to the application.

In traditional applications, the application is in "control".
It would create all the services it needs, and use those services directly.

When the application is running in an application server, we can use "Inversion of Control".
The application server is responsible for creating all the services.
The application simply makes use of those services created by the application server.

Note: an application server is sometimes known as a "container".

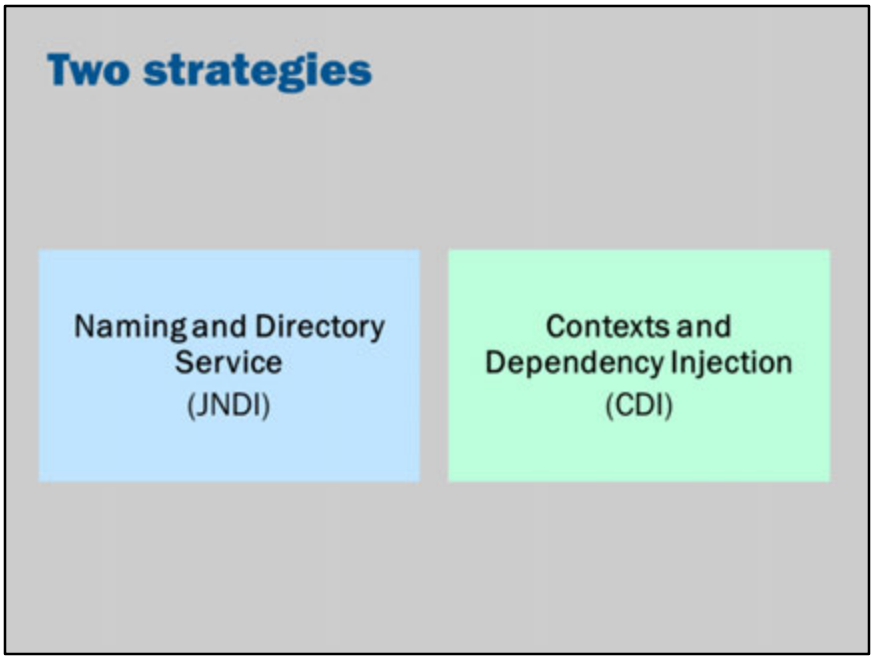


Why do we want to do this?

We shouldn't need to change our code to make deployment-specific settings:
- Database connection string changes
- Email server changes
- Web server changes
- Vendor changes (e.g., switch database technology)
- Services change (e.g., switch from JDBC authentication to LDAP authentication
- and so on…

We should also be able to tune our application at runtime:
- Change connection pooling strategies
- Change number of threads
- Change the way we distribute our application

Ideally, we should be able to make these deployment changes, without needing to modify our code.



In Java EE, there are two main strategies for getting this application server configuration "into" an application.

JNDI is where the application "asks" for configuration by name.
The container responds when it is asked.

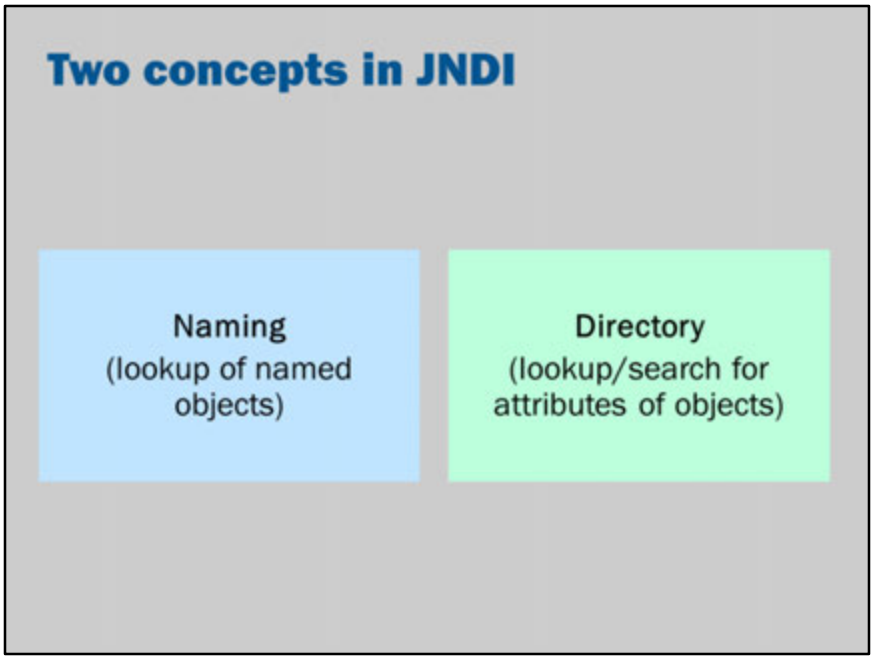
CDI is where the application server just "tells" the application what configuration to use.
The container detects places in the code that use various services.
It automatically sets the variables accordingly.

We look at each in this lecture.

JNDI

JNDI

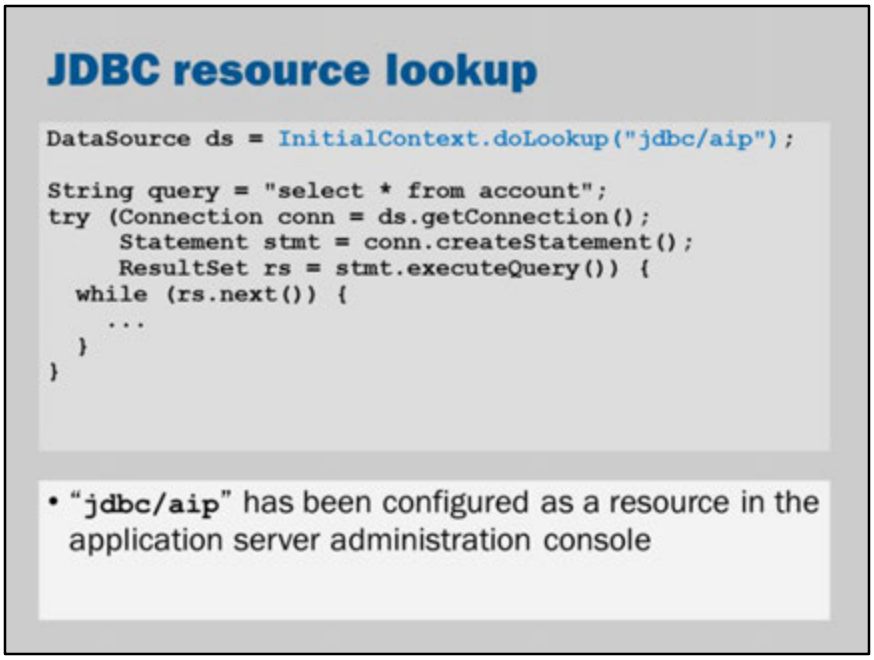
- Java Naming and Directory Interface
- A standard way for Java applications to interface with a variety of naming and directory services
- A standard way to access resources provided by the application container (i.e., GlassFish)



JNDI can be used for looking up "names".
e.g., "jdbc/aip" maps to a particular JDBC data source

JNDI can also be used for looking up a directory.
A directory is like a name lookup, except it also has the ability to query attributes.
e.g., "uts.edu.au" maps to 54.251.117.70 but it also has other attributes such as the mailserver (MX), nameservers (NS) and so on.

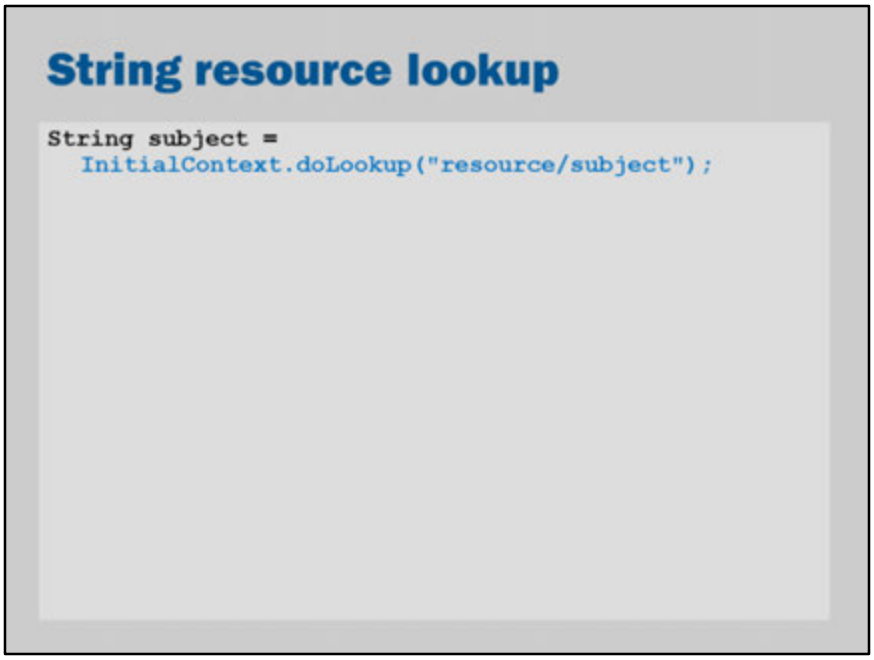
Naming



There is an example of naming in the Week 5 tutorials.

Here we are looking up a DataSource by the name "jdbc/aip".
The container does all the configuration and management of the DataSource.
In our application, we simply obtain the DataSource and can run queries against the database.

For this to work, I added a Resources\JDBC\JDBC Resources (and a JDBC Connection Pool) configuration in the GlassFish administration console.



We can look up other resources. In this case, a simple string.

For this to work, I added a Resources\JNDI\Custom Resources configuration in the GlassFish administration console:

JDNI Name: resource/subject
Resource Type: java.lang.String
Factory Class: org.glassfish.resources.custom.factory.PrimitivesAndStringFactory

Additional Property:
value = Advanced Internet Programming

JavaMail session resource lookup

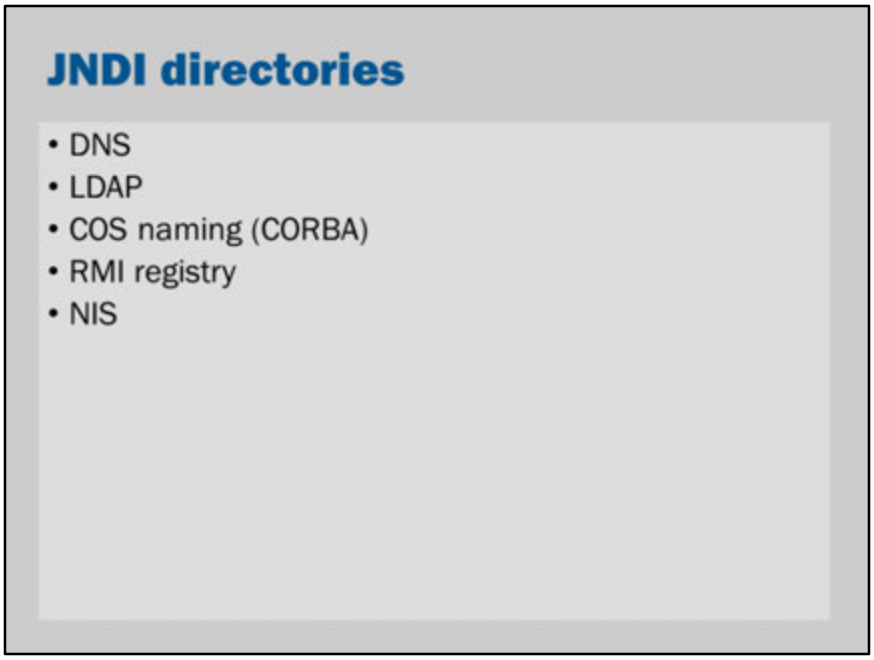
```
Session session =
  InitialContext.doLookup("mail/aip");

Message message = new MimeMessage(s);
Address[] to = InternetAddress.parse("you@...");
message.addFrom(InternetAddress.parse("me@..."));
message.addRecipients(Message.RecipientType.TO, to);
message.setSubject("Greeting");
message.setText("Hello, World!");

Transport.send(message);
```

… and JavaMail for email send/receive

For this to work, I added a Resources\JavaMail Sessions configuration in the GlassFish administration console.

You need to enter server names as appropriate for your environment.


Directory



- **DNS**: The Domain Name System is used to map names on the internet (e.g., www.uts.edu.au) to internet addresses for routing.
- **LDAP**: Lightweight Directory Access Protocol is typically used for storing user, account and system configuration information inside corporate networks (i.e., you can use your UTS username/password on the library, UTS Online and in the faculty thanks to LDAP integration).
- **COS naming**: Used by the CORBA distributed object protocol to store object references.
- **RMI registry**: Used by the Java Remote Method Invocation protocols to register the services that it provides to clients.
- **NIS**: Network Information Services, originally created by Sun microsystems, is used to store user and host configuration information in a network. This is largely replaced by LDAP nowadays.

All of these directories are available within JNDI, in addition to the naming services provided by the Java EE container / application server.

DNS directory lookup

```
Properties env = new Properties();
env.put(
    Context.INITIAL_CONTEXT_FACTORY,
    "com.sun.jndi.dns.DnsContextFactory"
);

InitialDirContext idc = new InitialDirContext(env);

Attributes attributes =
  idc.getAttributes("uts.edu.au");

System.out.println(attributes.get("MX"));
```

JNDI provides a number of built-in directory services.
DNS is one.
Here we are using JNDI directly to configure the directory service and query it.

This query gets the "MX" attribute of "uts.edu.au".
In other words, it finds the IP address (or name) of the server to which any email addressed @uts.edu.au should be delivered to.

DNS directory lookup

```
DirContext ctx =
    InitialContext.doLookup("resource/dns");

Attributes attributes =
  ctx.getAttributes("uts.edu.au");

System.out.println(attributes.get("MX"));
```

Because we are running on a Java EE application server, we can move the configuration into the application server.

This slide does exactly the same as the previous slide. Except, now, it is simpler and focuses on the domain logic rather than the implementation-specific configuration.

For this to work, I added a Resources\JNDI\External Resources configuration in the GlassFish administration console:

JDNI Name: resource/dns
Resource Type: com.sun.jndi.dns.DnsContext
Factory Class: com.sun.jndi.dns.DnsContextFactory
JNDI Lookup: . (i.e., just a dot)

Additional Property:
java.naming.factory.initial = com.sun.jndi.dns.DnsContextFactory

LDAP directory lookup

```
Properties env = new Properties();
env.put(
    Context.INITIAL_CONTEXT_FACTORY,
    "com.sun.jndi.ldap.LdapCtxFactory"
);
env.put(
    Context.PROVIDER_URL,
    "ldap://ldap.uts.edu.au"
);

InitialDirContext ctx = new InitialDirContext(env);

Attributes attributes = ctx.getAttributes(
    "uid=107624,ou=Staff,o=uts.edu.au,o=UTS"
);

System.out.println(attributes.get("mail"));
```

JNDI provides a number of built-in directory services.
LDAP is another.
Here we are using JNDI directly to configure the directory service and query it.

This particular LDAP server is the UTS staff directory.
This query is looking up the email address of the Vice-Chancellor (staff id 107624).

We can also use LDAP to perform searches, such as "all staff with a surname of Johnston".
This is beyond the scope of this course.

LDAP can also be used for authenticating passwords.
This is handy when developing internal systems for large organizations.
You can connect your application's login username/password to use the same password as the organization.

In fact, where we used JDBCRealm for user authentication in the tutorials, you could also have configured an LDAPRealm to use an LDAP server for authentication.

LDAP directory lookup

```
DirContext ctx =
    InitialContext.doLookup("resource/ldap");

Attributes attributes = ctx.getAttributes(
    "uid=107624"
);

System.out.println(attributes.get("mail"));
```

Because we are running on a Java EE application server, we can move the configuration into the application server.

This slide does exactly the same as the previous slide. Except, now, it is simpler and focuses on the domain logic rather than the implementation-specific configuration.

For this to work, I added a Resources\JNDI\External Resources configuration in the GlassFish administration console:

JDNI Name: resource/ldap
Resource Type: com.sun.jndi.ldap.LdapCtx
Factory Class: com.sun.jndi.ldap.LdapCtxFactory
JNDI Lookup: ou=Staff,o=uts.edu.au,o=UTS

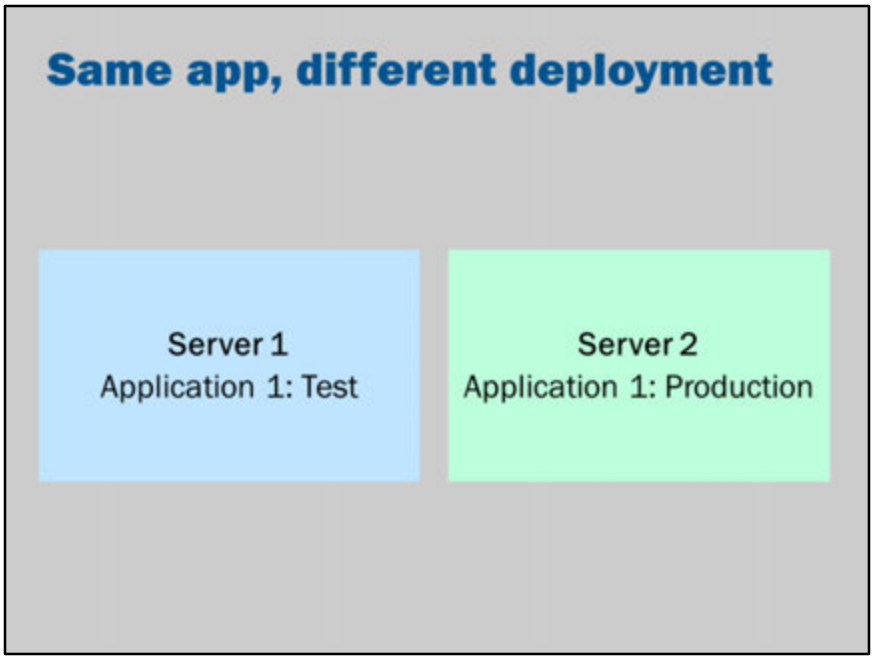
Additional Property:
java.naming.factory.initial = com.sun.jndi.ldap.LdapCtxFactory
java.naming.provider.url = ldap://ldap.uts.edu.au

Global names

There is an example of naming in the Week 5 tutorials.

Here we are looking up a DataSource by the name "jdbc/aip".
The container does all the configuration and management of the DataSource.
In our application, we simply obtain the DataSource and can run queries against the database.

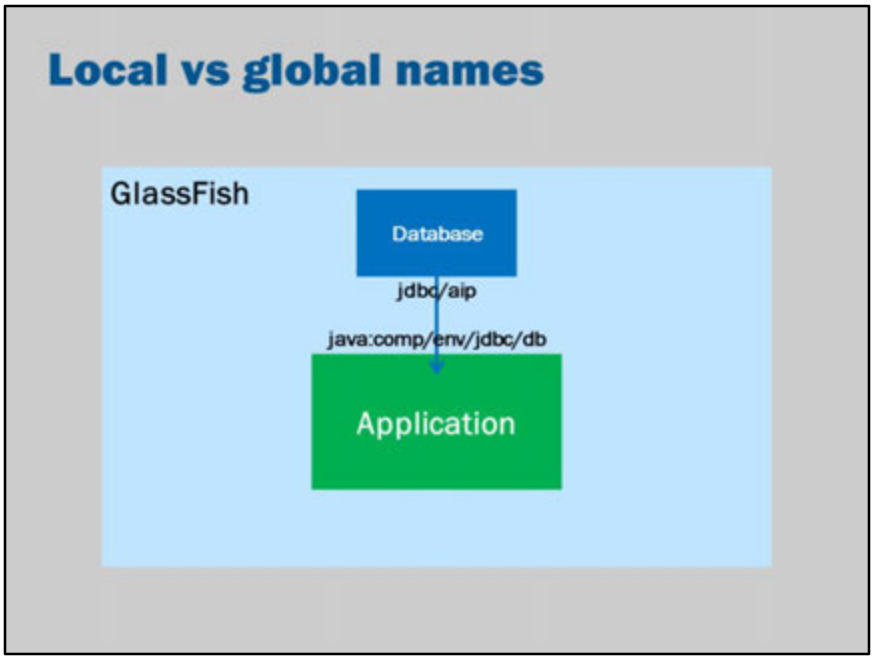
For this to work, I added a Resources\JDBC\JDBC Resources (and a JDBC Connection Pool) configuration in the GlassFish administration console.



The problem with using a global name such as jdbc/aip, is that we can't connect the same app to two separate databases.
You might want the same application to run on the same server, but in two separate modes.
For example, you might have www.example.com and test.example.com serving the same application running on two separate databases.
You can experiment with the test system and corrupt the data, but the production system will be safe.

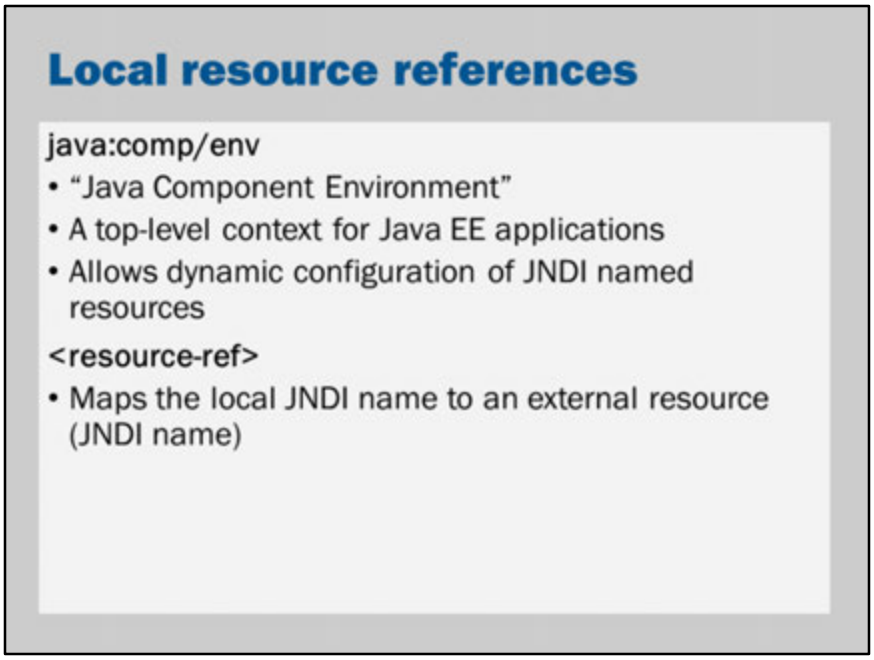
Another possibility is that two separate application developers might use the same name to refer to different databases.

As a simple example in this subject, two separate students might use the same database configuration (jdbc/aip).
If I wanted to run both applications on the same server, there would be problems if they used the same table names.



If we run multiple applications on one server, we can have a problem.
Two applications might use the same name: jdbc/db to refer to their database.
If we want separate databases, there needs to be a way to handle these requests.

Rather than needing to change the code, the solution is to use local names.
The application refers to its resources by a local name.
The local names are mapped to global names by the deployment descriptor.

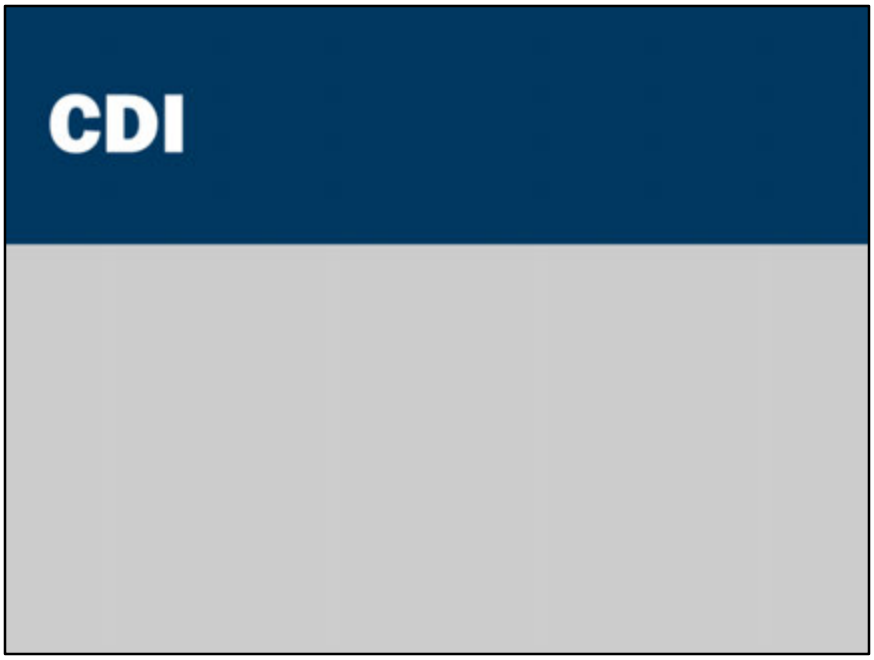


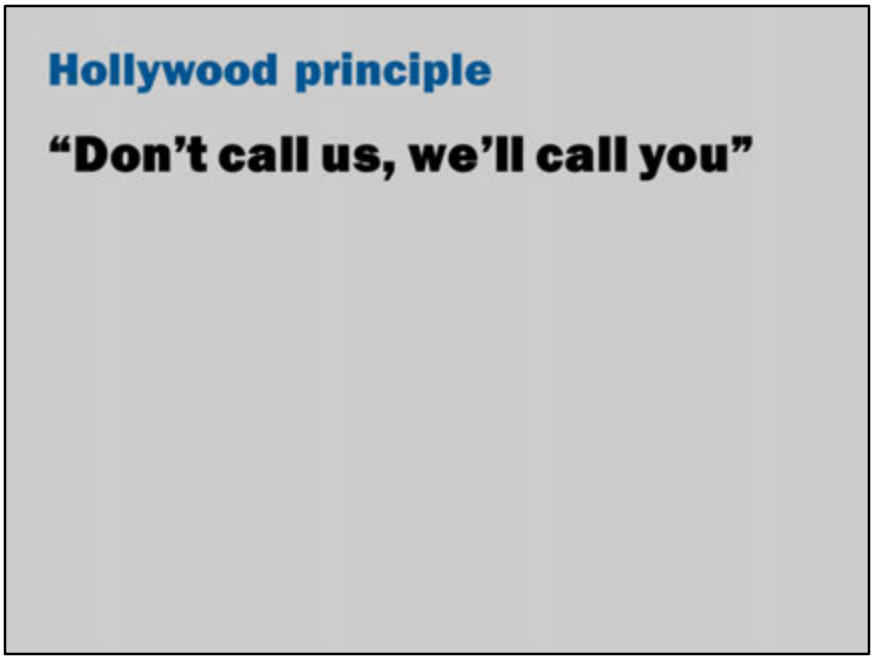
This is done by a special JNDI namespace: "java:comp/env".
Use the namespace by prepending it to JDNI names.
For example, instead of "jdbc/db", use "java:comp/env/jdbc/db".

Then, in the deployment descriptors, you can map the local names to global names.

You can create local resources by adding a sun-web.xml or glassfish-web.xml configuration to your application.

Then, from the application, you lookup that database using the java:comp/env context.

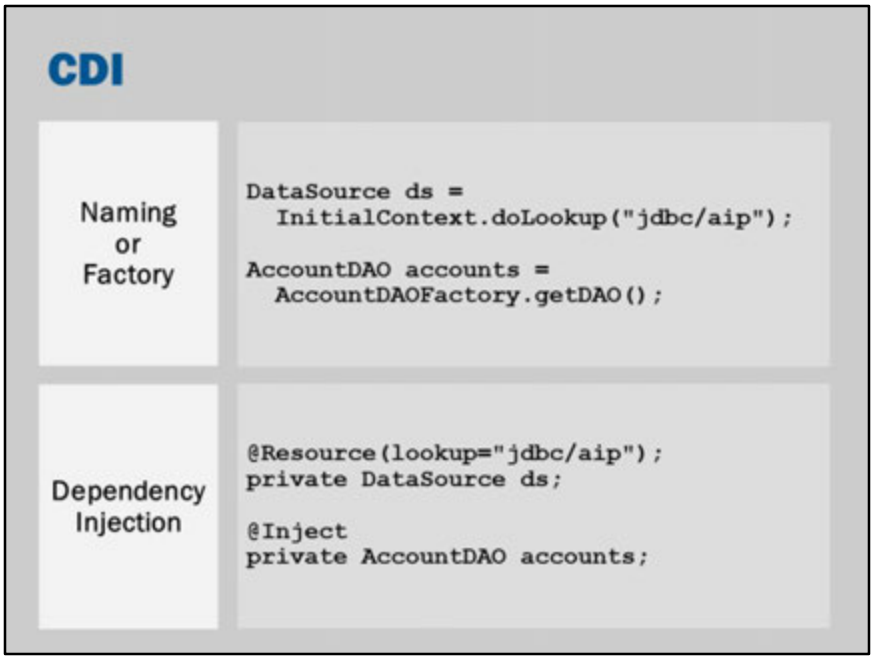

CDI



The "Hollywood principle" refers to the cliché of directors telling amateur actors, "don't call us, we'll call you".
This was, perhaps, a more polite way to tell overly-enthusiastic actors they weren't wanted.

In computer science, it is taken to refer to the use of call-backs or dependency injection.

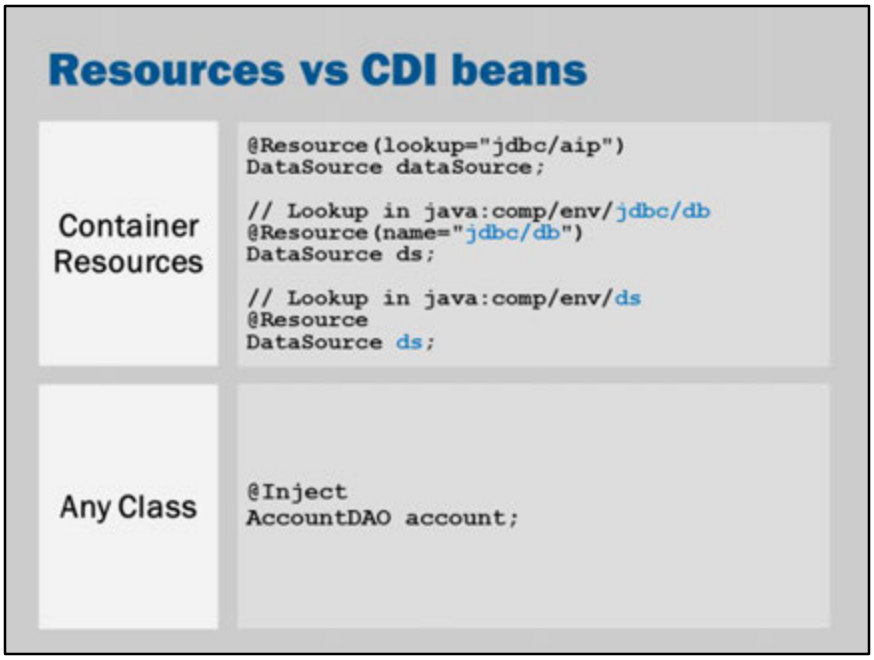
In traditional applications, you would calling a function to get some data or configuration.
The "Hollywood principle" refers to when you have set up a function or variable to hold the data, and you wait for the container to provide you with the information.



In dependency injection, we add special annotations to fields of our class. Dependency injection then figures out which class to create, instantiates it, and automatically sets those fields.

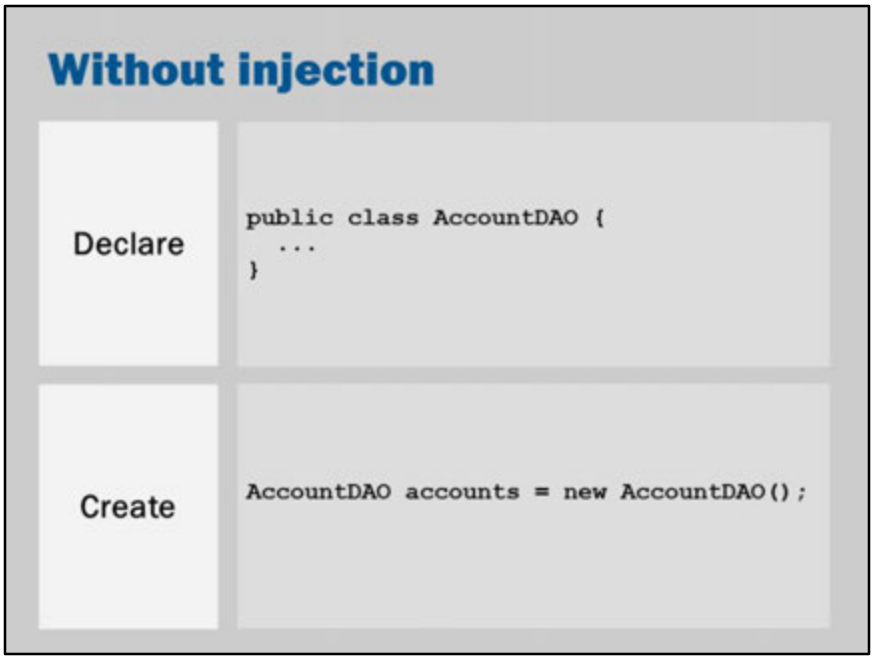
In other words, you just declare the fields and CDI will make sure that the appropriate objects are set into those fields.

It works, even if the fields are private.

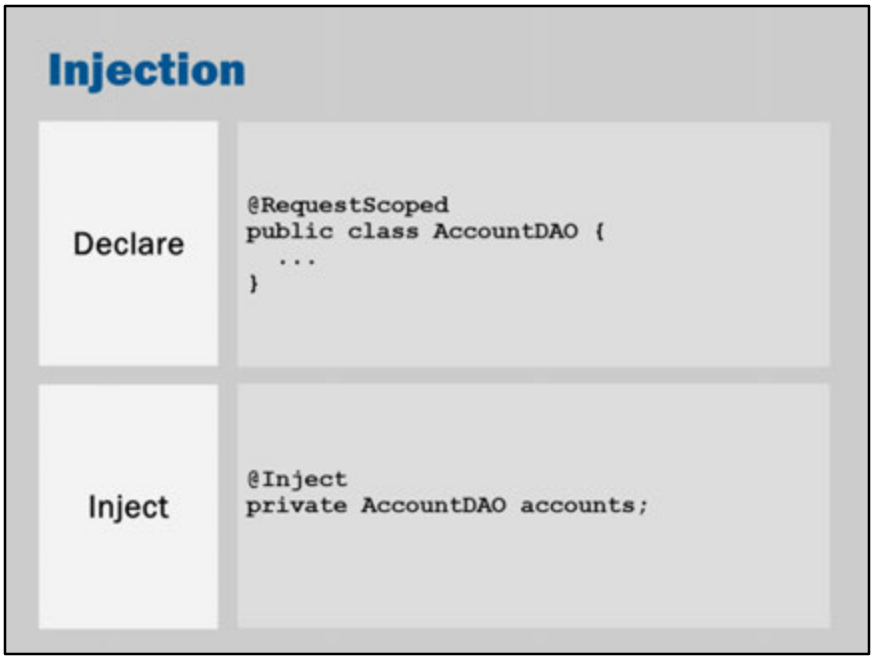


You can use @Resource to do JNDI lookups.

CDI has a more general @Inject process that allows you to inject instances of any class.

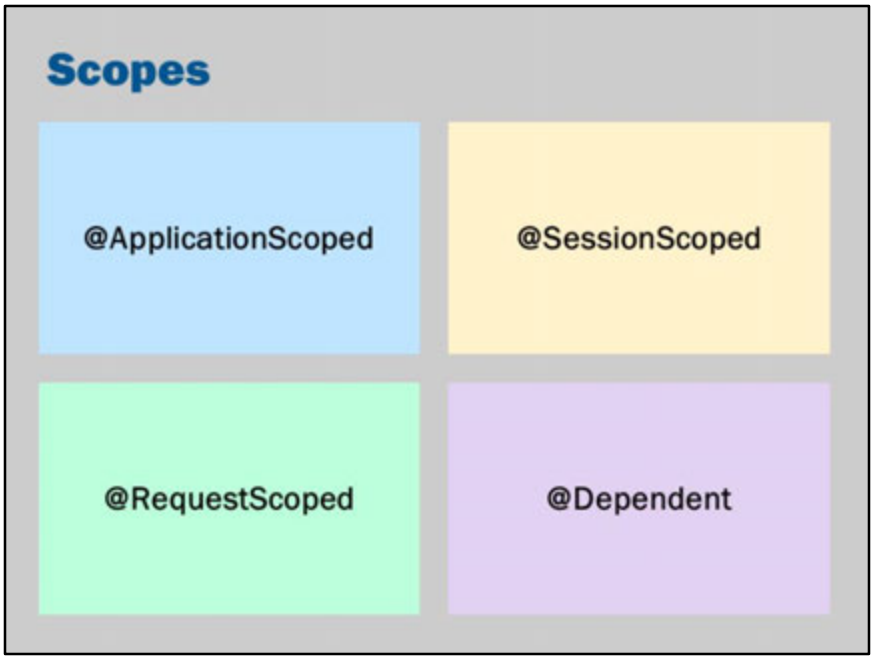


This is an example of how you would create objects without using dependency injection.



This is the same example as the previous slide, but using dependency injection.

To use CDI on any class, simply declare the class with a scope annotation.
Then, when you need the class, you can use it by "@Inject"ing it.



We have seen @RequestScoped many times in JavaServer Faces.
This creates a new instance every time a new request comes in.

@ApplicationScoped creates only one instance across the entire application.

@SessionScoped creates one instance per user.

@Dependent creates a new instance every time it is used. The scope is "dependent" on the scope of the class that it was injected into.

Proxies

Behind the scenes, CDI uses 'proxies':

- 'Fake' instances that like the real target object
- Method calls to the proxy are forwarded to another object
- The same proxy instance can represent multiple target objects: e.g., by looking up the correct object according to the user session

Session counter

```
public class MyServlet extends HttpServlet {

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

    int visit_no;
    synchronized (this) {

      HttpSession session = request.getSession();
      Integer count = (Integer)session.getAttribute("count");

      if (count == null)
        visit_no = 1;
      else
        visit_no = (int)count + 1;


      session.setAttribute("count", visit_no);
    }

    PrintWriter out = response.getWriter();
    out.println("<html><body><p>");
    out.println("Your visits: " + visit_no);
    out.println("</p></body></html>");
  }
}
```

So, here's an example of a Servlet that doesn't use CDI.
How could we improve it?

1. The domain logic code that does the counting can be moved into a separate bean
2. The association between that counting logic and the current session could be managed by CDI @SessionScoped
3. Then the servlet can just use the @SessionScoped bean

SessionScoped bean

```
@SessionScoped
public class Counter implements Serializable {

  private int count;

  public synchronized int getCount() {
    return count;
  }

  public synchronized int increment() {
    count++;
    return count;
  }
}
```

Here's a simple counter bean.

Session counter

```
public class MyServlet extends HttpServlet {

  @Inject
  Counter counter;

  @Override
  protected void doGet(HttpServletRequest request,
                       HttpServletResponse response)
                            throws ServletException, IOException {
    counter.increment();

    PrintWriter out = response.getWriter();
    out.println("<html><body><p>");
    out.println("Your visits: " + counter.getCount());
    out.println("</p></body></html>");

  }
}
```

Using CDI we can avoid all the hassle of dealing with the session object.
We just @Inject a @SessionScoped bean into our Servlet and CDI does all the hard work of figuring out the lifecycle.

Advanced comment:
Recalling back to Week 2 lectures, is there anything surprising about this?

Remember that there is normally only one instance of a Servlet.
The one instance is used to serve all requests simultaneously.

However, the counter field refers to a session scoped bean.
This means that the counter field should be different for each user.

This "trick" is achieved in CDI using proxies.
CDI creates a special object called a proxy.
The counter field contains the proxy.
When you call a method on the proxy, CDI will redirect the method call to the appropriate session-scoped object.

Other CDI features

Alternatives

Interface

```
public interface AccountDAO {
  public void create(AccountDTO a);
  public AccountDTO find(int id);
  public void update(AccountDTO a);
  public void delete(AccountDTO a);
}
```

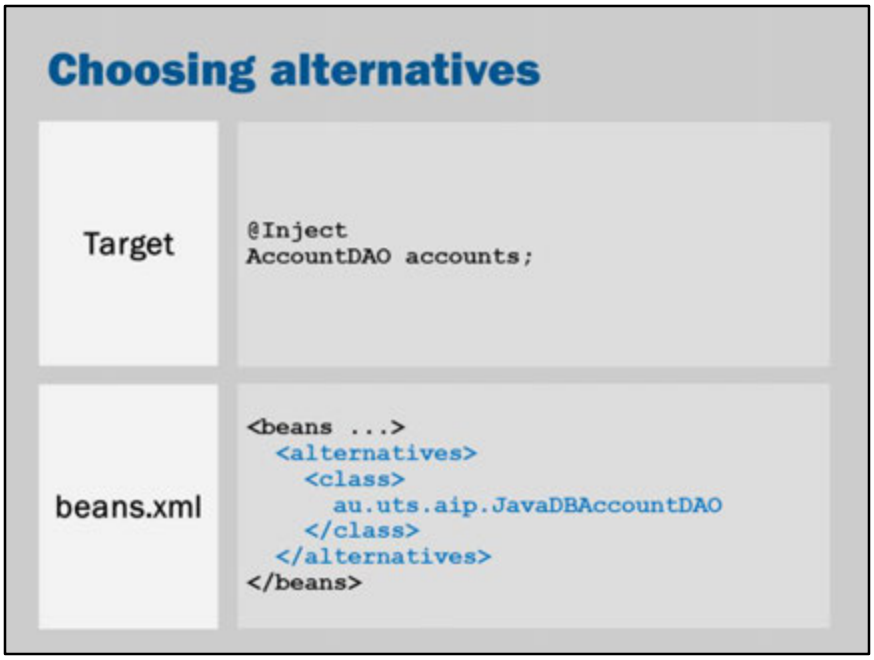
Java DB

```
@Dependent
@Alternative
public JavaDBAccountDAO
    implements AccountDAO {
 ...
}
```

Oracle

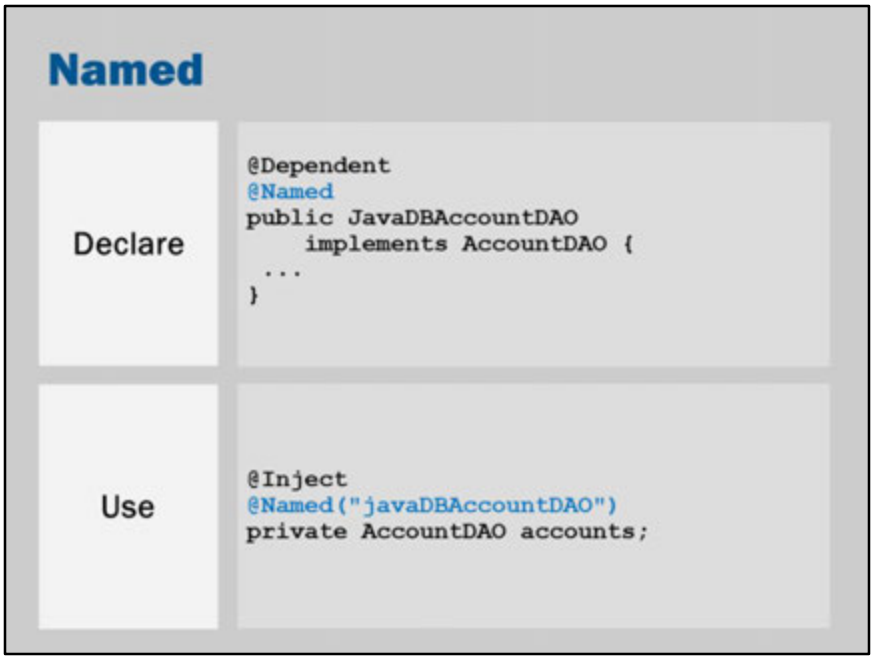
```
@Dependent
@Alternative
public OracleAccountDAO
    implements AccountDAO {
 ...
}
```

CDI allows you to choose one of multiple instances of an interface.
For example, you may implement multiple DAOs.
The correct DAO depends on the underlying database.



The choice of alternative can be chosen at deployment using the deployment descriptors, rather than changing/recompiling the code.

The accounts field will be injected with the alternative chosen in beans.xml. In this example, it will use the JavaDBAccountDAO.

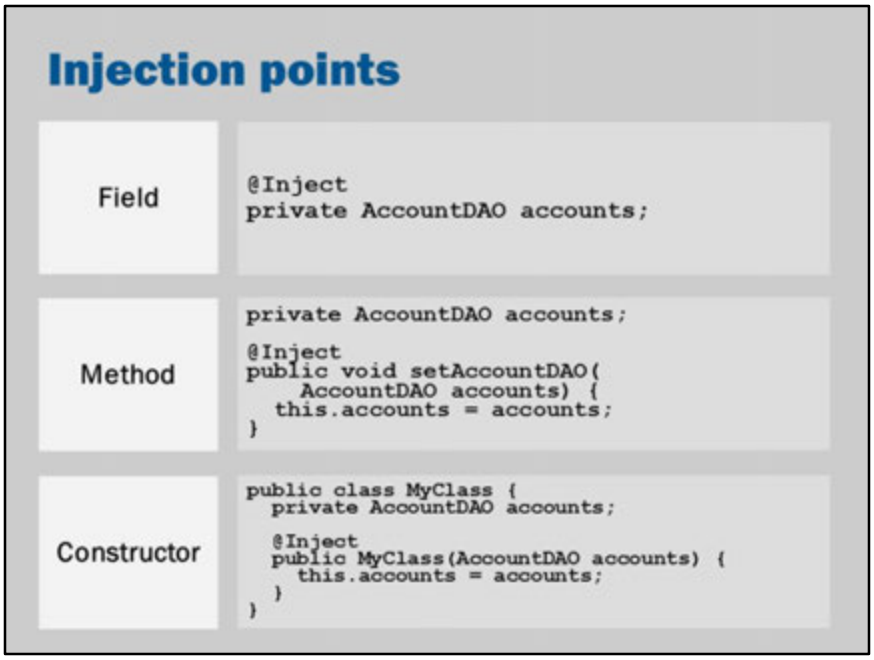


@Named allows you to qualify beans with a string.
This should be familiar to you in that it used by JavaServer Faces to access a backing bean.

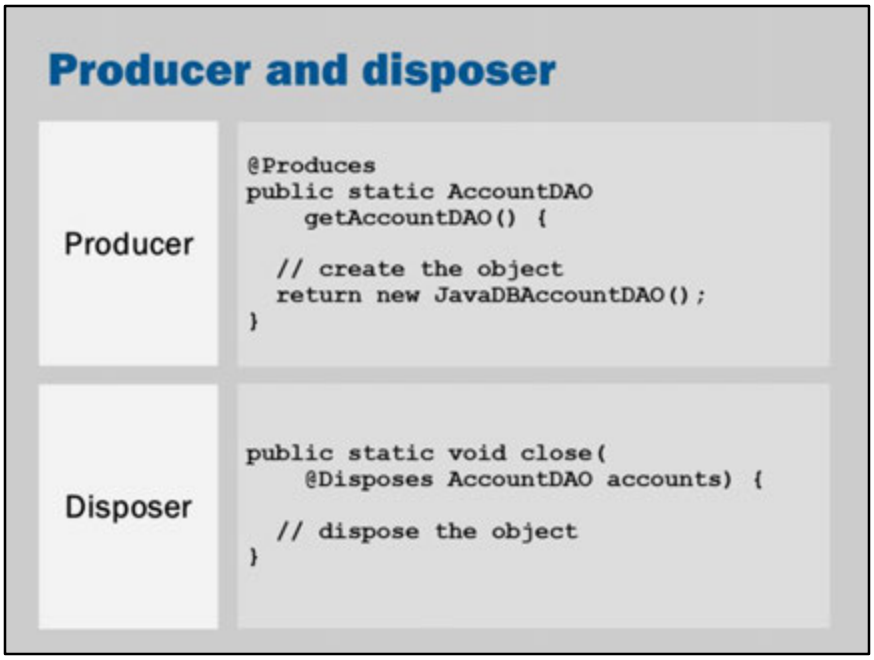
In practice, you should avoid the use of @Named.
This is because the compiler cannot check that the name in the string is correct.

The exception to this rule is when using JavaServer Faces.
It makes sense for JavaServer Faces to used @Named.
This is because XHTML code needs to refer to your classes by name (there is no type-checking).
That is, we already have the problem of the compiler not being able to check your XHTML.
Thus, nothing is lost by using @Named but by using that annotation, we can tell JavaServer Faces that the bean is accessible by its name.

Note: You can also declare a specific name, rather than using the default naming rule.



While we have injected into fields, CDI allows injection in different places.
All of these examples are rougly equivalent.
CDI looks for @Inject annotations, and supplies the appropriate object during creation.



By default, CDI creates beans using the "default constructor" of your bean.

The default constructor is a constructor that takes no arguments.

```
public MyClass {
  public MyClass() {  //   <- this is the default constructor
    // init
  }
}
```

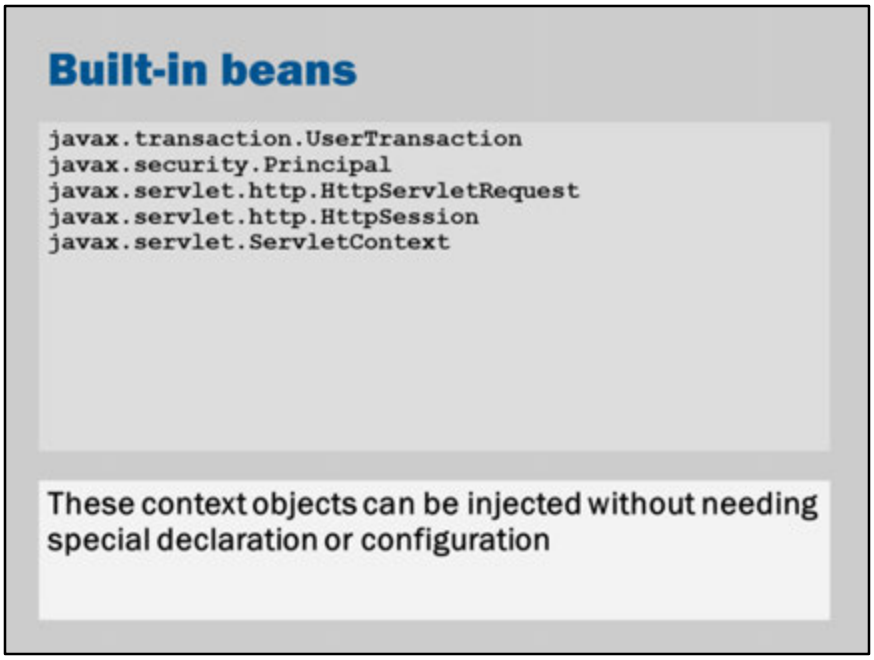
In Java, if you have not declared any constructor, Java automatically creates a default constructor for you.

```
public MyClass {
  //   <- there are no constructors, but that is ok because a default constructor is created for us automatically
}
```

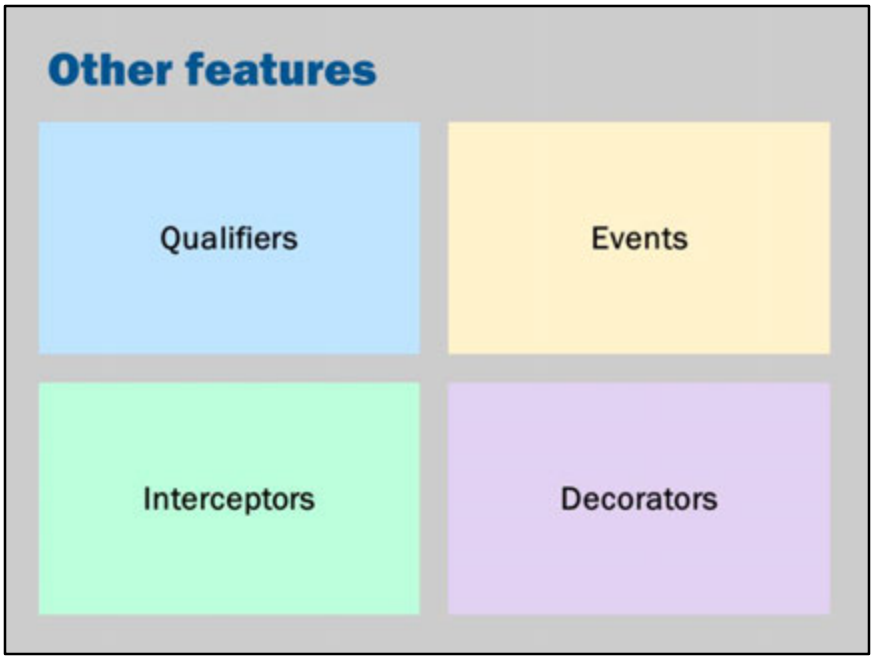
CDI normally creates beans using the default constructor.
However, if you annotate a method with @Produces, CDI will use it as a producer.
A produce is called to create a new instance of your bean.

CDI calls the method instead of using the object's constructor.

This allows you to use special constructors.
This allows you to use special logic before construction.
This allows you to have dynamic type selection based on, for example, configuration information.



These are beans that should be available for injection in a Java EE application.
You do not need to declare them.
They are "just there", and available for use with @Inject.



CDI has a variety of other advanced features:

- **Qualifiers**: Allow you to specify different types of objects to inject. E.g., you can create @Oracle and @JavaDB qualifiers so that can request the injection of a particular type of class: @Inject @Oracle AccountDAO accounts;
- **Events**: Allow you to declare events in CDI and have other classes subscribe to those events. Events are a way to communicate information between modules, without needing to explicitly declare the function call.
- **Interceptors**: Allow you to have an 'interceptor' function called whenever a different method is called (i.e., it "intercepts" every call to the method).
- **Decorators**: Allow to you intercept every method of a class. It allows you to create wrapper classes that act just like an existing class, but provide extra facilities, processing or error checking.

For more information on the extra features of CDI, I recommend reading Chapter 2 of "Beginning Java EE".
The book is available for free download from the UTS library catalog:
http://find.lib.uts.edu.au/?R=OPAC_b2874770

A tip

Do not use "new" and CDI to construct the same beans:

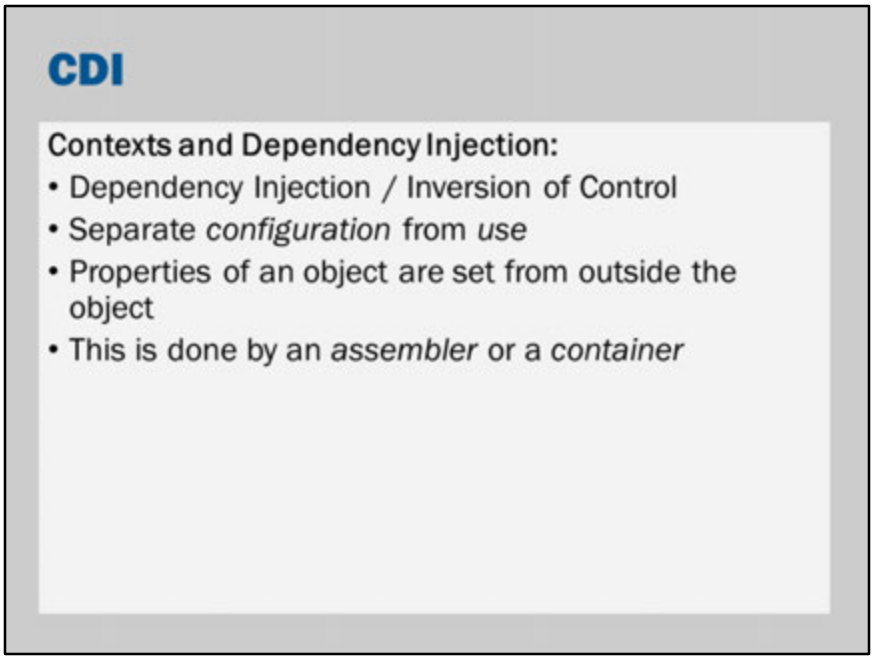
- "new" bypasses CDI – it does not give CDI a chance to do injection on the instance

Key points

- Remember the scopes:
  - *Application Scoped*
  - *Session Scoped*
  - *Request Scoped*
  - *Dependent*
- Inject beans with CDI scopes using @Inject
- Inject JNDI resources using @Resource
- Don't use `new` with CDI

Bonus slides

Now that you've watched/read these slides, you should be able to explain what each of these are doing.



An excellent summary of Dependency Injection:
http://www.martinfowler.com/articles/injection.html

The key idea is that the container does the work of setting properties of your beans/objects.