



CHAPTER 2

CONCEPTS



OBJECTIVES

After completing “Concepts,” you will be able to:

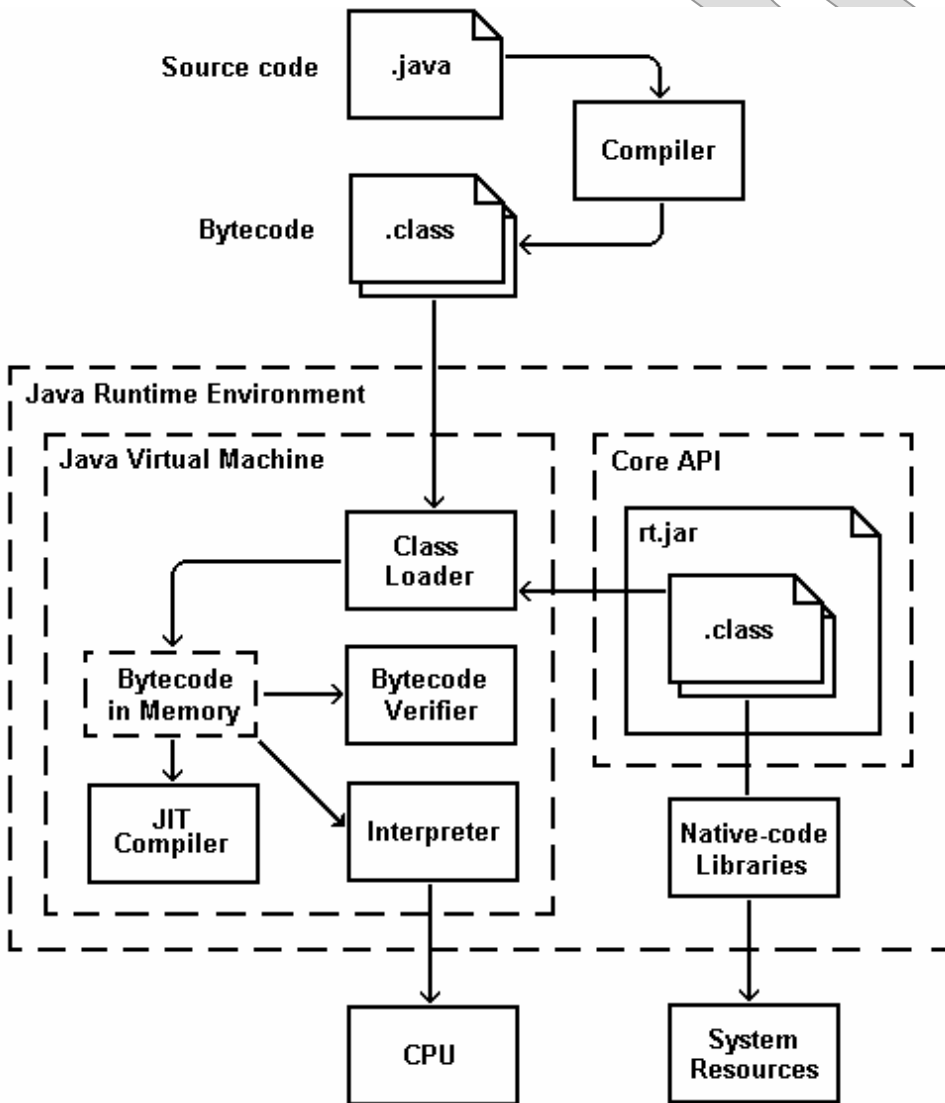
- Describe the importance of intermediate bytecode to the portability of Java software.
- Describe the Java EE architecture of containers and components, and identify the three containers that must be supported by a Java EE application server.
- Explain the lifecycle of a Java EE application and the roles that various people play in development, deployment, and administration.
- Understand the power of declarative development, using XML to declare facts and parameters to an application server so that it can implement many enterprise features “for free.”

Bytecode

- Java software, deployed in a single form or version, runs on a wide range of platforms – this is **portability**.
- It is a hybrid of two traditional approaches to high-level languages:
 - **Compiled** languages enjoyed build-time type checking and other error checking, but had to be **cross-compiled** for different target platforms.
 - **Interpreted** languages could be portable, but relied on run-time error checking, a frustrating and inefficient development process.
- Java is **compiled and interpreted**.
 - Source files are compiled to intermediate **bytecode**, which is very compact like the machine languages that traditional compilers write.
 - But then the bytecode is interpreted by the **virtual machine** – which is just what it sounds like!

From Source Code to CPU

- Here is a summary of development, deployment and execution of a Java program:



The Virtual Machine and Runtime

- The virtual machine, or **JVM**, loads compiled Java classes and interprets them in a machine-specific way.
 - Resulting machine code is fed to the actual processor in use at runtime.
- The Java runtime environment, or **JRE**, includes the JVM and the **Core API**.
 - This is the large library of classes that Java applications and components can use to do their work.
 - The Core API is loosely analogous to the standard C and C++ libraries, or the Visual Basic runtime.
 - A key difference is that C programs may use the standard libraries to, say, write a file; Java programs must use the Core API, as the virtual machine prevents direct use of the CPU or operating-system API.
 - This is both a way of fostering good practice and a pillar of Java's **code-security architecture**.

- Let's look at a dead-simple Java application that simply prints a greeting to the console.
- The source code is as follows, though we'll not worry about any details of Java coding in this course.

```
public class Hello
{
    public static void main (String[] args)
    {
        System.out.println ("Hello, Java!");
    }
}
```

- What we will consider for the moment:
 - How to build and test the application
 - How the application works for a given platform

- There are two simple scripts for this demo:
 - A build script, **build.bat**, to compile the source code in .java files to bytecode in .class files
 - A run script, **run.bat**, to launch the application class in a new JVM
- View a listing of the files in the project directory:

```
dir
build.bat
doc
run.bat
src
```

- Under **src** is the Java source file:

```
dir src
Hello.java
```

- Build, and view the directory listing again: you'll see the new file **build/Hello.class**.

```
build
dir build
Hello.class
```

- Run the run script and see that the application class runs and prints the greeting back to you:

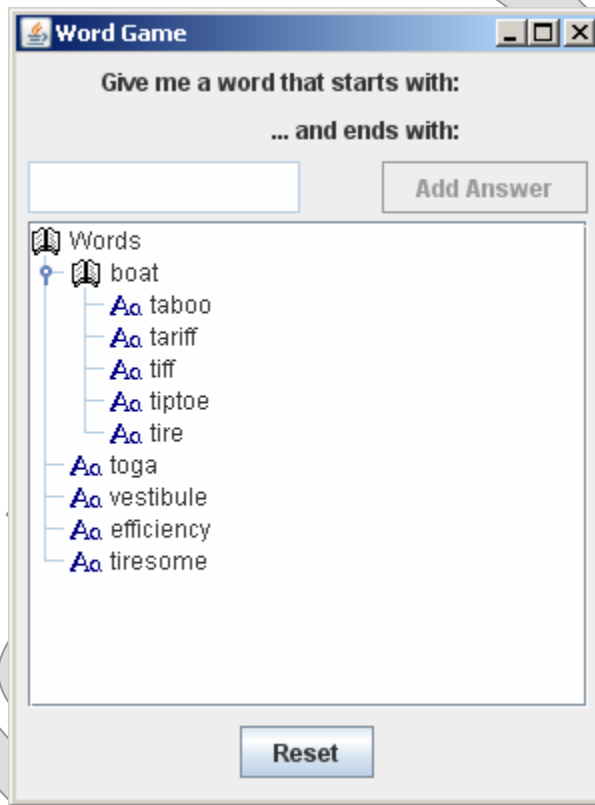
```
run
Hello, Java!
```

How Does It Work?

- How does this application work on your platform?
- Java software achieves portability on two levels.
- Rather than compiling to machine code for a specific CPU, Java is compiled to portable bytecode.
 - Here's a listing of the bytecode in **Hello.class**:

```
0:  getstatic      #2; // gets the printing stream
3:  ldc           #3; // loads the greeting string
5:  invokevirtual #4; // calls the print method
8:  return
```
 - This is interpreted to machine instructions, say for a Pentium chip, at runtime, by the JVM.
 - It would of course be interpreted differently for a RISC processor or a Sun Alpha - etc.
- Where the application needs to interact with the physical system, it makes Core API calls such as **System.out.println** to write to the console output.
 - The Core API implementation for your platform holds the logic for connecting to the console's standard output stream and producing characters to that stream.
 - The operating system is ultimately responsible for implementing the rendering of characters on the screen.

- Now we'll see an example of a graphical user interface (GUI) application: a word game!



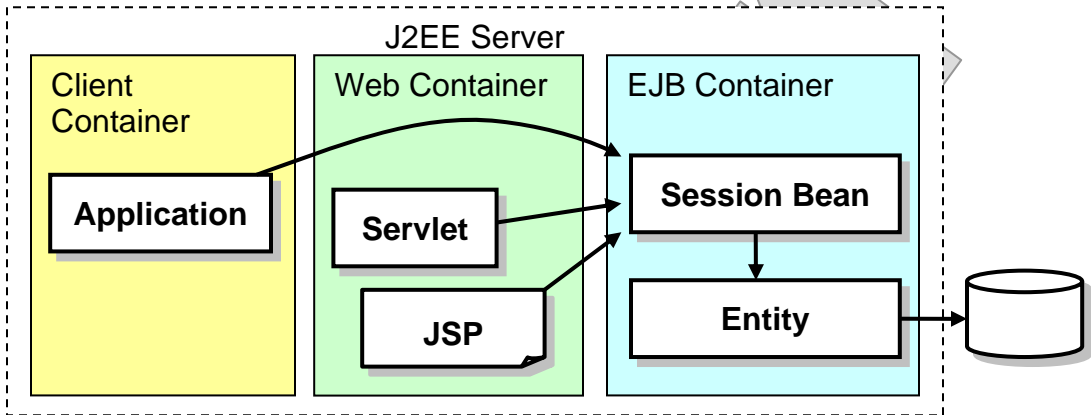
- Here, the Java runtime loads a main class that uses the Core API to interact with the OS' windowing system.
- Specifically, this application uses the **Java Foundation Classes**, or **JFC** – and a subset of JFC that is sometimes known as **Swing**.
- User gestures – mouse clicks, keystrokes, etc. – trigger **events** in the JFC system, in the form of method calls on event **handlers**.
- Handlers then respond by updating the GUI in some way.

Containers and Components

- From basic code portability and the capacity to create standalone applications, we now move to key concepts in the Java EE architecture.
- The organizing principle of Java EE is the **container**.
 - A container holds objects or **components**.
 - The container is responsible for the **lifecycle** of the component: it creates and destroys the component (and others like it) by the container's own policies.
 - The container also **mediates**: it “sees” every interaction between the component and the outside world.
- By assuming this control, the container is able to provide many useful features:
 - **Remote connectivity**
 - **Scalability**
 - **Availability**
 - **Security**
 - **Transaction support**
- All these features come “for free” to the component developer, as they can be implemented once in the container.
 - We'll see how in a moment.

Three Containers

- In its current version, Java EE defines three types of container:



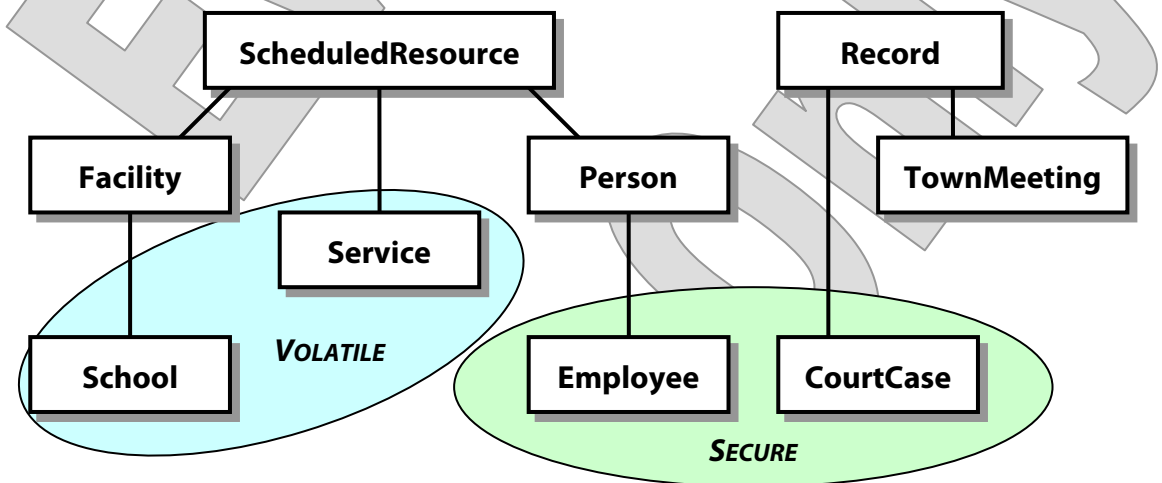
- The **web container** supports Servlets and JSPs.
 - The **EJB container** is exclusively for EJBs.
 - The **application client container** (sometimes ACC) is the least used; it supports otherwise ordinary standalone applications, but gives them visibility to Java EE APIs so that they can, for instance, invoke an EJB.
- A Java EE **application server** is required to support all three containers.

Roles

- Java EE also promotes a really inspired idea: recognizing that software development is carried out by people playing various **roles**.
 - On one hand, we knew this!
 - But Java EE formalizes these roles, and gives different responsibilities and privileges to each.
 - This facilitates good workflow and development practices.
 - It also organizes certain logic or key values in the application structure so that they can be managed most appropriately.
- **The four most common roles are:**
 - The **component provider** builds a given component.
 - The **application assembler** creates a comprehensive application from a set of components and possibly "glue" code.
 - The **deployer** installs the application on a given physical system.
 - The **administrator** manages the application in his or her domain.
- **Two more roles are played only by vendors of Java EE application servers:**
 - **Container provider**
 - **Server provider**

Aspect-Oriented Programming

- In the previous chapter we put the Java programming language in the context of major conceptual developments:
 - **Structured programming**, which promises reuse through callable **functions** and instantiable **data types**
 - **Object-oriented development**, which promises reuse through **encapsulation** and **inheritance** of state and behavior
- The newest wave is **aspect-oriented programming**, or **AOP**.
 - AOP recognizes **cross-cutting concerns** – features that are required of many otherwise disparate types, also known as **aspects**.
 - Where OO achieves reuse by classification, AOP recognizes that a taxonomy – which is a single-parent, many-child hierarchy – is insufficient to address a multitude of independent aspects.
 - For example, how would an OO inheritance tree encapsulate the attributes **volatile** and **secure** in this domain model?



Managing Metadata

- Java is basically an OO language and, from an OO perspective, aspects are seen as **metadata** – meaning that they are not fundamental to the OO type model, but nevertheless compose a body of formal and useful information about a given type.
- Any AOP system must define means to two activities:
 - **Expressing** metadata – this is how we define the aspects
 - **Processing** metadata – some body of generic code must be able to implement the features we’re declaring
- How can we do this?
 - We might **describe** the aspects of a class in an external file.

```
class X  
{  
    public void foo ();  
}
```

```
<component>  
<class>X</class>  
<poolOf>50</poolOf>  
<method>  
    <name>foo</name>  
    <protocol>JRMP</protocol>  
</method>
```

- We might **annotate** the class itself.

```
@Pooled(50) class X  
{  
    @Remote(JRMP) public void foo ();  
}
```

Java EE as an Aspect-Oriented Platform

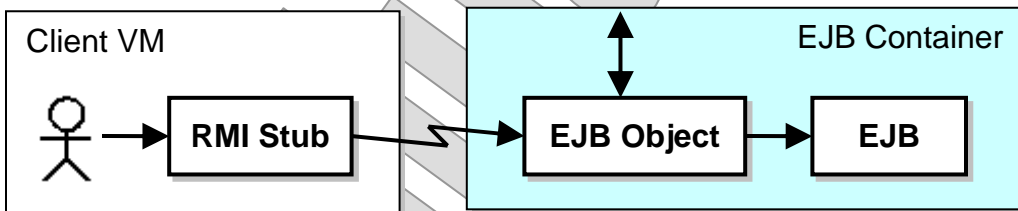
- Java EE containers provide enterprise-class features for free.
- We can see these features as aspects in the AOP sense of the term.
 - The fact that an object should be remotely available, secure, transactional, etc., is unrelated to its nature as a class in an OO model of a particular business domain.
 - These sorts of facts about components should be coded in Java.
 - We need ... metadata!
- J2EE was in fact an early expression of what later became known as AOP.
- Originally, J2EE took only the “descriptive” approach, using external metadata expressed in XML – a format that
 - Is **easy to read and write** without specialized tools
 - Is **highly expressive**, able to capture information sets of many shapes and sizes
 - **Requires no compilation** or other transformation to be shared between all Java EE roles
- Only the component provider (and maybe the application assembler) are likely to write Java code; but everyone can work with XML.
- As of Java EE 5, the “annotation” model is also supported thoroughly, meaning that the Java source file itself can express metadata about the component.
 - Java SE 5 adds syntax to support these annotations.

Deployment Descriptors vs. Annotations

- So Java EE 5 supports both approaches, and they are usually mixed to some degree, capturing many kinds of information:
 - What Java classes are to be treated as **what kinds of components**
 - Where components should be **published**: a URL for a servlet, or a unique name for an EJB
 - Who should be able to use what components, i.e. **security** policies
 - Various other values that are **volatile** or deployment-specific: local threshold values or network names, for instance
- Each approach is more appropriate for a given “metadatum.”
- The XML files are called **deployment descriptors**.
 - They are rather **verbose** and exacting to author and to maintain.
 - But they are **independent** of the Java source file and so can be changed without rebuilding the application.
- Java annotations can replace external XML for almost all of the metadata about a component.
 - They are much **easier to write**, and maintenance is nearly trivial as the annotation lives right next to the class declaration, method, field, or whatever is its target.
 - They are better suited to values that are really **inherent** to the component, because if they need to change, the application must be rebuilt for those changes to be expressed.
 - At the time of this writing there is still an excess of enthusiasm over the annotations feature; they are somewhat overused.

Remote Connectivity

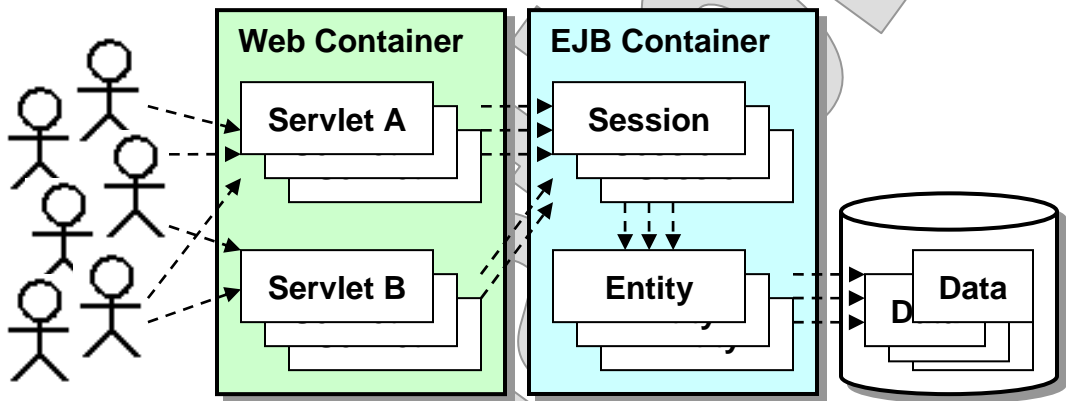
- Java EE components enjoy remote connectivity, implemented for them by the container.
- Servlets and JSPs are available through HTTP requests, thanks to interactions between the web server and a Java EE web container.
- For EJBs, Java SE defines a **remote method invocation (RMI)** standard.
 - A subset of RMI that supports CORBA, **RMI over IIOP** or **RMI/IIOP**, is the default transport for EJBs.



- All Java EE components thus enjoy remote connectivity and **location transparency**, which means that components and their clients are blissfully ignorant of their true proximity:
 - They could be in the same process space.
 - They could be in different processes on one machine.
 - They could be flung halfway around the world and talking via the Internet.
- No code in the component or client has to change to adapt to these different proximities.

Scalability

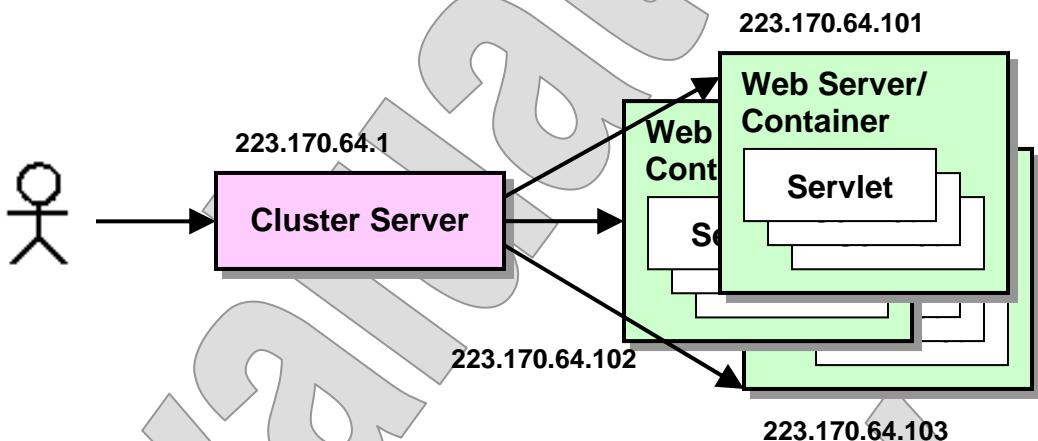
- Containers can provide excellent scalability on behalf of their hosted components.
- The key technique in achieving scalability is **pooling**.



- The container will manage a pool of instances of a given component type.
 - The size of the pool is controlled by the administrator.
 - To the outside world, it looks as though there is one component, or N components; but in fact there may be M components in memory, with $M < N$.
 - Components must be able to **switch contexts** to live in pools; this places a few constraints on their implementation.
 - Especially, they cannot make assumptions about their own **identity** that would be safe in the J2SE context.
- By this arrangement, the container is able to serve hundreds or thousands more clients than it keeps service objects in memory – or to manage much more persistent data than it holds in memory at one time.

Availability

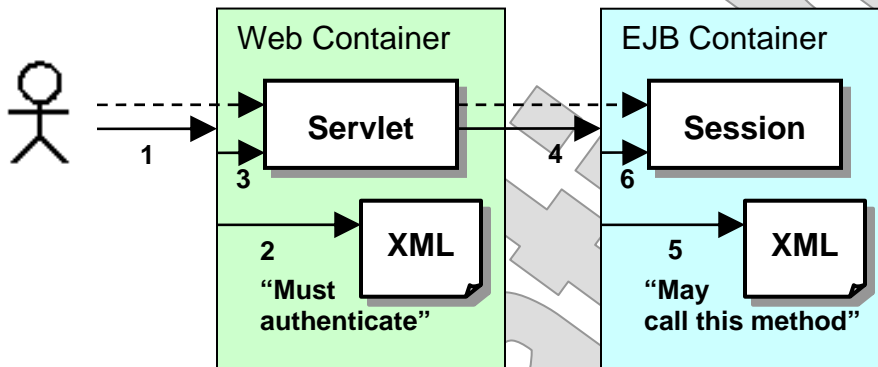
- Closely related to scalability is availability – the property of a component that it is always available for client use.
- But servers crash, and network connections fail.
- Java EE application servers can support **clustering**, a technique by which multiple processes on multiple networked hosts run redundantly.



- A cluster is far more fault-tolerant than a single machine.
- The trick is making the cluster of server processes, and the pools of objects within them, look like one component at one address to the client.
- A **cluster server** accomplishes this.
- Clustering is one of the least-specified Java EE features; it is supported as a standard, but clustering strategies may be proprietary.

Security

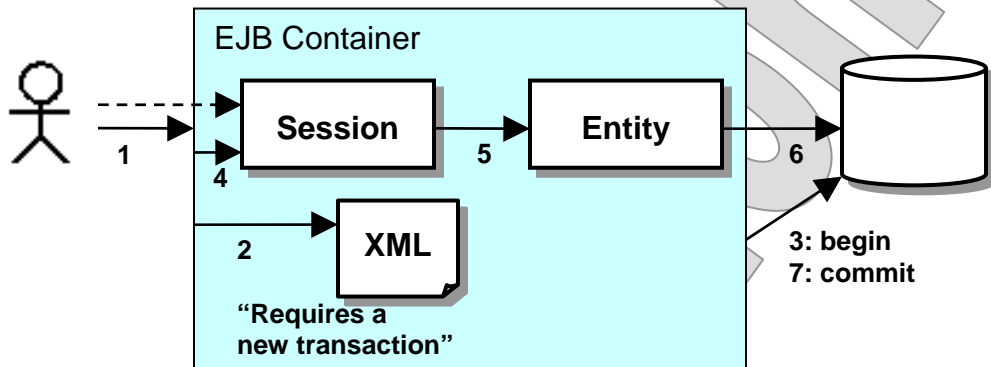
- Java EE defines security mechanisms that assure that components will not be used by unauthorized clients.



- **Authentication** mechanisms are standardized primarily at the HTTP front door to an application.
 - A web application can declare a requirement for HTTP basic or even certificate-based authentication.
- **Authorization** policies exist at many levels.
 - An authenticated client may be granted or denied privileges to see a certain **web page**.
 - An EJB may authorize callers at **component** or even individual **method** level.
- Java EE deployment descriptors can define **security roles** to which users and groups can be assigned.
 - These roles would be defined by an application assembler.
 - But since the actual users and groups will vary by locale, they are assigned to these roles by an administrator.

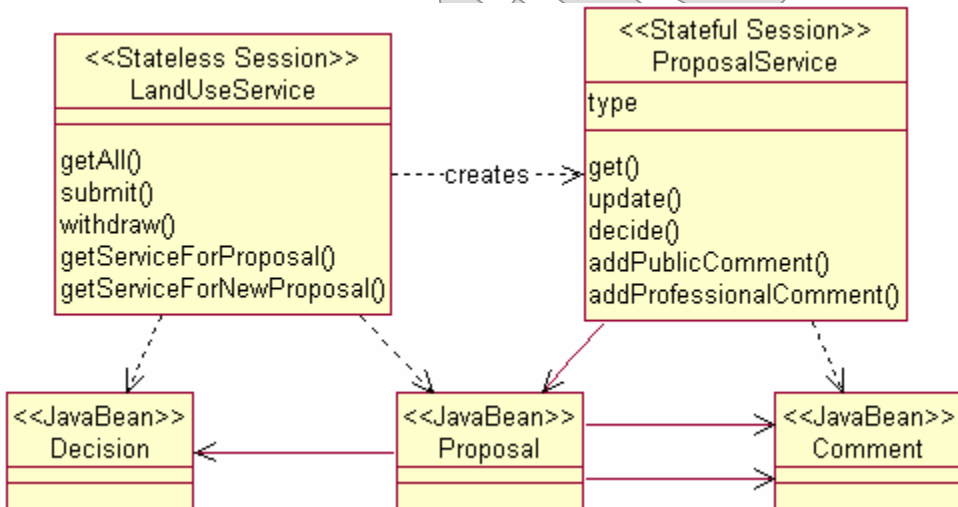
Transactionality

- For EJBs only, there is also declarative support for ACID transactions.



- The EJB container reads declarations in the deployment descriptor that inform it as to how to start and end transactions as calls come in to an EJB at runtime.
 - A given method may require a transaction to be in force; require its own, new transaction; allow enlistment in any existing transaction, refuse to participate in a transaction, etc.
 - The container, as it watches calls come in and responses go out, interacts with one or more external **resource managers** to assure that transactions are started, ended, and rolled back appropriately.
- This is a massive savings of development time and trouble.

- We'll look at specific Java EE technologies in the following chapter; but just to get a feel for how containers and metadata work, let's take an early peek at an EJB application.
- We'll focus on two EJBs that form a **service layer** in the application, capturing expected use cases:



- **LandUseService** is a so-called **stateless session bean**.
 - This means that it is an enterprise-class Java component that does not “remember” its caller from one method to the next – it has no sense of long-term connection to a client.
 - It serves as the entry point into the business functionality, allowing a caller to retrieve existing data records, create new ones, etc.
- **ProposalService** is a **stateful session bean**.
 - One can hold a reference to this bean and make a series of calls that build upon one another.
 - This service is responsible for editing a single proposal.

- There's much more here than we have time to study, but let's look at these two session beans, and try to understand:
 - How does the container know that they're EJBs in the first place?
 - How is one able to interact with the other, when both of them live and die at the pleasure of the EJB container?
- Just a few lines of Java answer the first question.
 - See `src/gov/usda/usfs/landuse/ejb/LandUseService.java`:

```
@Stateless
public class LandUseServiceImpl
    implements LandUseService
```

- The **@Stateless** annotation expresses the most fundamental element of metadata, identifying this class as a certain sort of EJB.
- `src/gov/usda/usfs/landuse/ejb/ProposalService.java` does something very similar:

```
@Stateful
public class ProposalServiceImpl
    implements ProposalService
```

- So these are examples of internal metadata, expressed using native Java annotations.
 - It's appropriate to take this approach, because the aspect of being a stateless session bean is so basic to the Java class that it's hard to imagine it changing without having to re-code the class anyway.

- To answer the second question, look at the deployment descriptor for the EJB module.
 - This is found in `ejb/META-INF/ejb-jar.xml`:

```
<enterprise-beans>
  <session>
    <ejb-name>LandUseServiceImpl</ejb-name>
    <ejb-local-ref>
      <ejb-ref-name>proposalService</ejb-ref-name>
      <ejb-ref-type>Session</ejb-ref-type>
      <local>
        gov.usda.usfs.landuse.ejb.ProposalService
      </local>
      <ejb-link>ProposalServiceImpl</ejb-link>
    </ejb-local-ref>
  </session>
</enterprise-beans>
```

- This fragment of XML informs the EJB container that a field defined in one EJB should implicitly refer to another EJB.
 - The name “proposalService” appears in the Java source code for the `LandUseService` – this is an **abstract name** defined by the component provider.
 - The EJB container maps this name to a concrete entity based on the external metadata.
 - This too could have been expressed in the Java source files; but XML is appropriate here because the link between the two is not fundamental to one or the other, and is more a matter of application assembly.
 - The link may be re-mapped without rebuilding either bean.

SUMMARY

- **The basic value proposition of Java – portable, object-oriented software – provides the foundation for Java EE.**
 - It also provides the foundation for aspect-oriented programming, with its annotation syntax.
- **Then, Java EE brings several brilliant inventions (or discoveries!) of its own:**
 - The **container** and contained component
 - **Context and lifecycle** relationships
 - **Declarative development** using XML and annotations
 - **Roles** as a way to define distinct responsibilities and to allow multiple parties to collaborate smoothly over the complete lifecycle of an enterprise component or application