J2EE

Tom Szymanski
Avaya Labs, Basking Ridge, NJ
tszymanski@avaya.Com

Flavors of Java

- Java comes in 3 editions
  - J2EE enterprise edition
  - J2SE standard edition (which you already know)
  - J2ME embedded edition
- Common J2 language, but J2EE and J2ME use different libraries and restrict the programming model in various ways

The J2EE Target Market

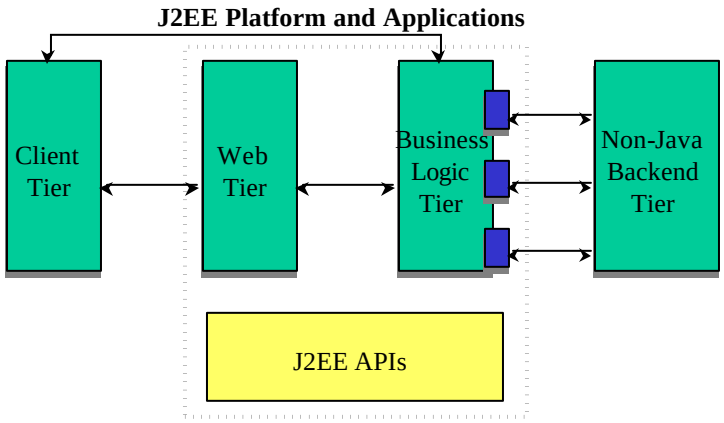
- E-commerce site is usual example
  - Browse through offerings
  - Select item, gather billing and shipping info
  - Check inventory (possibly trigger supply-chain)
  - Validate financial info
  - Arrange shipment & get tracking number
- Characteristics of a J2EE application
  - Distributed
  - Highly available
  - Reliable
  - Secure
  - Scalable

The J2EE Target Market (cont.)

- Bad news: enterprise systems are notoriously difficult to construct
  - Each of aforementioned characteristics is hard
  - Error handling and recovery permeates the code
- Good news: enterprise applications usually don't require much computation
- Trade off cycles for programming ease!
- Need 2 things
  - A toolkit of useful subsystems for building enterprise systems
  - *Help in integrating and packaging the pieces*

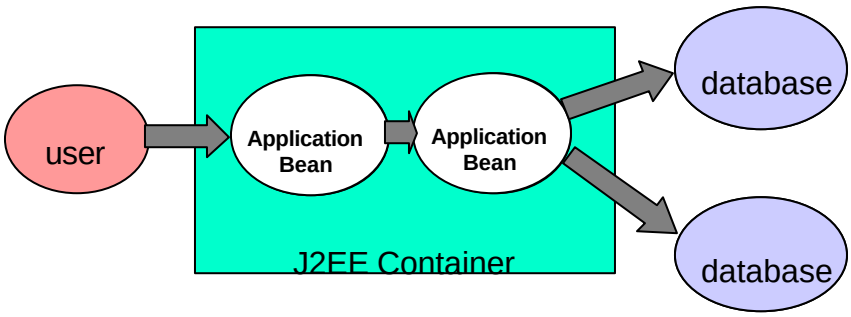
J2EE Application Model

- Multitier enterprise applications: a thin Web client connects to a backend (database, EIS, etc.) through a Web presentation and a business logic tier



Container Mediation in J2EE

- Components traditionally called “beans” and run in “containers”.
- All interactions are *mediated* by the “container” – e.g. enforcing security



Packaging Beans

- Everything in the J2EE world is packaged as a jar file containing classes, support files, and a *deployment descriptor*.
- The deployment descriptor is an XML document that *completely* defines a bean's interface to its environment
  - Names of bean in the jar file
  - Security requirements of each method
  - Resources (databases, URLs, references to other beans) used by the bean
  - Environment parameters used by the bean
  - Much, much, more.
- Goal is programming by configuration – code is not allowed to have external properties "hard wired"

Side Comments

- Players in the Enterprise Market
  - Corba
  - J2EE
  - .NET
- Component-based computing – style of computing using independent modules (possibly written in different languages and possibly running on different machines) connected by middleware
- Middleware – general term for "plumbing infrastructure" used to connect components
- Application Server – a server running J2EE or similar such software

Some Subsystems and Services in J2EE

- Naming and directory services
- Distributed objects and RMI
- Database access
- Concurrency control
- Transactional Integrity
- Life-cycle management & data persistence
- Messaging

- Note: some of the above are just libraries, others are features integrated throughout the platform

Naming and Directory Services

- Don't want hardwired "constants" in code, e.g., machine names, port numbers, database names, etc.
- Directories allow a level of indirection
  - Can move resources from one machine to another
  - Replace resources, etc
- J2SE has a simple directory service

```
Object Naming.lookup(string name);
Naming.bind(string name, Object);
```

- J2EE uses JNDI which supports a hierarchical name space in which each node has a set of attribute/value pairs (and an object) – like LDAP if you know what that is

Distributed Objects

- Remote method invocation (RMI) – allows an object on one machine to execute a method from an object residing on a different machines
- Part of J2SE
- Foundation of J2EE (slightly restricted)
- Implementing RMI requires two mechanisms
  - Transport (e.g., UDP, TCP, HTTP)
  - Marshalling/unmarshalling scheme

RMI in J2SE (similar in J2EE)

- Three pieces of code provided by user
  - Interface definition code
  - Client code
  - Server code
- The java platform produces additional code that does all the transport & marshalling
  - Stub code, which runs on the client
    - Called by the user's client code, marshals arguments, transmits them to server, unmarshals result
  - Skeleton (or Tie) code, which runs on the server
    - Waits for calls, unmarshals arguments, calls user's server code, marshals result

Java RMI – Interface Definition

```
package example;
import java.rmi.*;

interface Adder implements Remote {
        int byone(int i) throws RemoteException;
}
```

- the marker tag “Remote” is used to identify objects that might reside on another machine.

Java RMI – Client Code

```
import example.Adder;
mport java.rmi.Naming;

Adder a = (Adder) Naming.lookup(“rmi://mypc/sum”);
try {
    j = a.byone(j);
} catch (RemoteException e) {
    j += 1;
}
```

- The call on Naming.lookup actually returns an instance of class Adder_STUB (generated by the system)
  - knows how to invoke the corresponding object on mypc bound to the name “sum”
  - Implements Adder

Java RMI – Server Code

```
import example.Adder;
import java.rmi.*;

class AdderImpl extends UnicastRemoteObject implements Adder {
    int byone(int i) throws RemoteException {
        return I+1;
    }
}

Naming.bind("rmi://mypc/sum", new AdderImpl());
```

- The last line creates an instance of a system constructed AdderImpl_TIE class and starts it listening on a socket for incoming calls. It also binds an instance of Adder_STUB to the indicated name.

Java RMI Surprises

- Can't refer to fields in a remote object, only methods (because it's an interface!)
- Distributed garbage collection is necessary
- Sometimes you want to pass objects by value instead of by reference.
  - RMI parameters are passed by reference if they implement "Remote" else by value ("deep copy")
  - Same with return value.
- Inheritance implies code must move with calls; java security model makes this safe.
  - J2EE and other object-oriented systems are afraid to allow this so they require that all necessary code be locally available.

Transactional Integrity

- Enforces an “all or nothing” principle on a sequence of actions
- Example: transfer 5$ from brian to tom
  - Fetch brian’s wallet from database
  - Remove 5$
  - Return brian’s wallet to database
  - Fetch tom’s wallet from database
  - Add 5$
  - Return tom’s wallet to database
- Techniques exist for guaranteeing transactional integrity even across multiple databases and even across hardware crashes.

Transactional Integrity (cont.)

- General scheme
  - Get a transactionid (TID) from a transaction monitor
  - Associate the TID with the current thread, passing it as a hidden parameter during RMI
  - All database operations note the TID as they perform their operations
  - At the end of the computation, the transaction monitor is told to “commit” or “rollback”.
- Useful as a handy “undo” mechanism.
- Transactional integrity provides a nice set of properties commonly referred to as the “ACID” properties (see any database book)

Transactional Integrity in J2EE

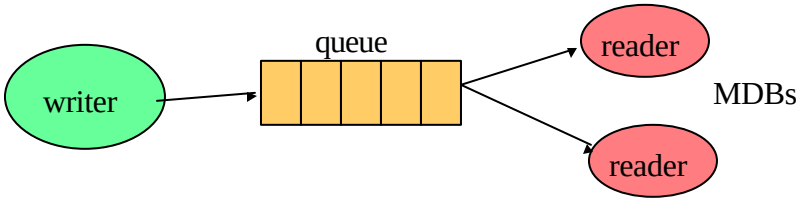
- In practice, you want to be able to switch transaction contexts (for a subtransaction) or turn it off for a while.
- J2EE allows deployer to specify for each method, what transactional properties should be applied to it.
  - "transaction required" – if there in no current TID, then start a new one and commit it when the method completes.
  - 5 other possibilites.

Life Cycle Management & Data Persistence

- J2EE entity beans
  - Treated as distributed objects
  - Persistence is achieved by storing each bean as a row in a database
  - Bean-managed persistence
    - Your bean's code must contain methods ejbLoad() and ejbStore() which are responsible for doing I/O to the database. Container calls these when it needs to.
  - Container-managed persistence
    - You supply an abstract bean, system generates a concrete subclass which includes all the necessary persistence code which it calls as needed.
- Platform typically caches a finite pool of entity beans in memory, "paging" them to the database as needed.

Messaging - JMS

- Reliable, persistent messages



- Useful paradigm for constructing concurrent systems especially when workloads fluctuate
- In J2EE, only way to implement concurrency since you can't create your own threads
- JMS supports transactional integrity

Effect of J2EE on Programming

- IDEs become essential
  - No one can remember all the details in the API's
  - Particularly "pluggable" IDE's like Eclipse
- Becomes natural to partition code team into specialists
- Essential to have a very wise architect overseeing the whole process
- Testing is hard – need to trigger all failure modes just to test that your system is properly configured
- Integration with non-J2EE system is hard

Things you give up to use J2EE

- full I/O control
- create subprocesses
- use classloaders
- load native libraries
- full control of AWT, Swing
- use reflection
- threading
- Listening on server sockets

J2EE Governance

- Sun
  - proposes specs
  - gets feedback
- Vendors (e.g., I.B.M., BEA, Oracle, etc)
  - Implement app servers
  - Get certified as spec compliant
  - Charge heavily for their platforms
  - Introduce proprietary features
  - Always one release behind the specs
- Issues
  - Interoperability
  - Avoiding the proprietary trap

J2EE Advantages & Disadvantages

- Advantages
  - Much leverage is derived from the platform
  - Fast application prototyping and development
  - Can use programmers of varying skill levels
  - Interoperability (in principle)
- Disadvantages
  - A long learning curve to master the platform
    - Training costs
  - Restrictions on the programmer
    - Can't do anything that interferes with the App Server's control of the system, e.g., writing your own control threads or periodically scheduled tasks
  - Temptation to use proprietary vendor extensions