DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

Lecture Notes

Course: Advance Java Programming
Course Code: OE25
Faculty: Shwetha A N

SIDDAGANGA INSTITUTE OF TECHNOLOGY
TUMKUR-3

An Autonomous Institution, Affiliated to VTU, Belagavi & Recognised by AICTE and Accredited by NBA, New Delhi

Advance Java Programming
Unit 1 Notes
J2EE Multi tier Architecture

The two-tier architecture depends heavily on keeping client software updated, which is both difficult to maintain and costly to deploy in a large corporation. Web-based, multi-tier systems don't require client software to be upgraded whenever presentation and functionality of an application are changed.

Distributive Systems
The concept of multi-tier architecture has evolved over decades, following a similar evolutionary course as programming languages. The key objective of multi-tier architecture is to share resources amongst clients.

The first evolution in programming languages id Assembly language. Software services consist of subroutines written in assembly language that communicate with each other using machine registers, which are memory spaces within the CPU of a machine. Whenever a programmer required functionality provided by a software service, the programmer called the appropriate assembly language subroutine from within the program. The drawback of this is Assembly language subroutines were machine specific and couldn't be easily replicated on different machines. This meant that subroutines had to be rewritten for each machine.

The next evolution is FORTRAN and COBOL. Programs written in FORTRAN could share functionality by using functions instead of assembly language subroutines. The same was true of programs written in COBOL. Functions are not machine specific, so functions could run on different machines by recompiling the function.

At that time there was a drawback with data exchange that is magnetic tapes were used to transfer data, programs, and software services to another machine. There wasn't a real-time transmission system.

Real-Time Transmission
Real-time transmission came about with the introduction of the UNIX operating system. The UNIX operating system contains support for Transmission Control Protocol/Internet Protocol (TCP/IP), which is a standard that specifies how to create, translate, and control transmissions between machines over a computer network.

Remote Procedure Call (RPC) defined a way to share functions written in any procedural language such as FORTRAN, COBOL, and the C programming language. This meant that software services were no longer limited to a machine.

Another important development in the evolution of distributive systems came with the development of eXternal Data Representation (XDR) to exchange complex data structures between programs and functions.

Software Objects
The next evolutionary step in programming language gave birth to object-oriented languages such as C++ and Java. Procedural languages focused on functionality, where a program was organized into functions that contained statements and data that were necessary to execute a task.

Programs written in an object-oriented language were organized into software objects, not by functionality. A software object is a software service that can be used by a program. Although objects and programs could use RPC for communication, RPC was designed around software services being functionally centric and not software-object- centric. This meant it was unnatural for programs to call software objects using RPC.

A new protocol was needed that could naturally call software objects. Simultaneously two protocols were developed to access software objects. These were Common Object Request Broker Architecture (CORBA) and Distributed Common Object Model (DCOM).

Web Services
The next evolution of software services was born and was called web services. Three new standards were developed with the introduction of web services. These are Web Services Description Language (WSDL), Universal Description, Discovery, and Integration (UDDI), and Service Oriented Architecture Protocol (SOAP).

Programmers use WSDL to publish their web service, thereby making the web service available to other programmers over the network. A programmer uses UDDI to locate web services that have been published and uses SOAP to invoke a particular web service.

The Tier
A tier is an abstract concept that defines a group of technologies that provide one or more services to its clients. A good way to understand a tier structure’s organization is by understanding the organization of large corporation.

At the lowest level of a corporation are facilities services that consist of resources necessary to maintain the office building. Facilities services encompass a wide variety of resources that typically include electricity, ventilation, elevator services, computer network services, and telephone services.

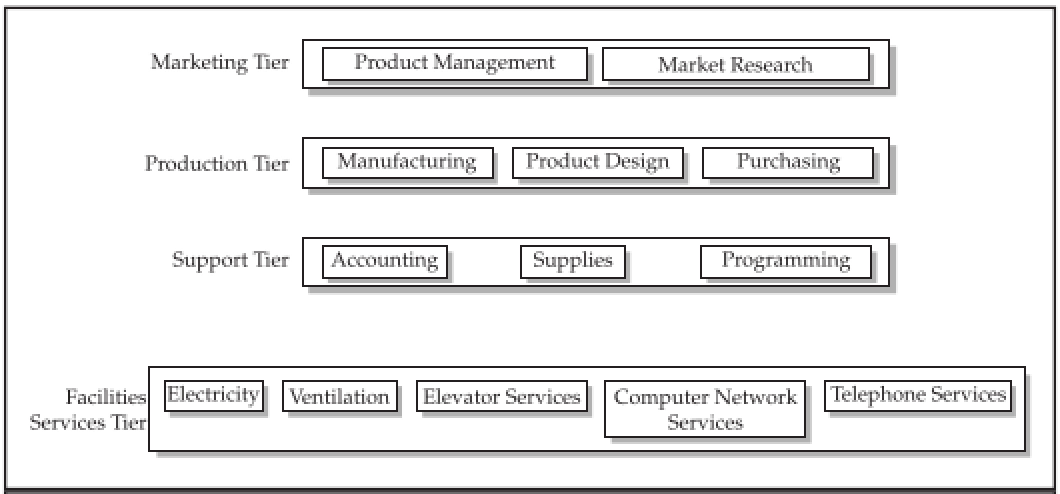


Figure 2-1. *Resources of a large organization are typically organized into a tier structure that operates similarly to the tier structure used in distributed systems.*

The next tier in the organization contains support resources such as accounting, supplies, computer programming, and other resources that support the main activity of the company. Above the support tier is the production tier. The production tier has the resources necessary to produce products and services sold by the company. The highest tier is the marketing tier, which consists of resources used to determine the products and services to sell to customers.

Any resource is considered a client when a resource sends a request for service to a service provider (also referred to as a service). A service is any resource that receives and fulfills a request from a client, and that resource itself might have to make requests to other resources to fulfill a client's request.

Let's say that a product manager working at the marketing tier decides the company could make a profit by selling customers a widget. The product manager requests an accountant to conduct a formal cost analysis of manufacturing a widget. The accountant is on the support tier of the organization. The product manager is the client and the accountant is the service.

However, the accountant requires information from the manufacturing manager to fulfill the product manager's request. The manufacturing manager works on the production tier of the organization. The accountant is the client to the manufacturing manager who is the service to the accountant.

A client is concerned about sending a request for service and receiving results from a service. A client isn't concerned about how a service provides the results. Services can be modified as changes occur in the functionality without affecting the client program.

Clients, Resources, and Components

Multi-tier architecture is composed of clients, resources, components, and containers. A client refers to a program that requests service from a component. A resource is anything a component needs to provide a service, and a component is part of a tier that consists of a collection of classes or a program that performs a function to provide the service. A container is software that manages a component and provides a component with system services.

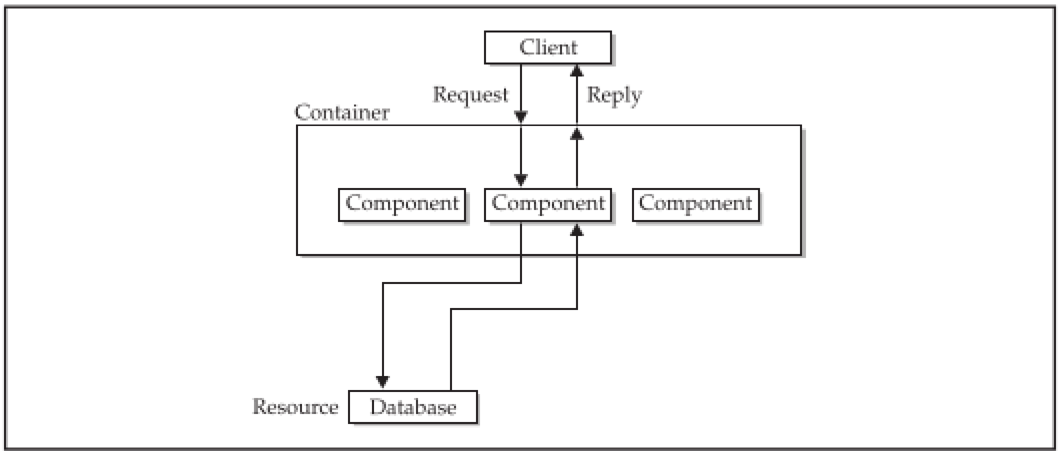


Figure 2-2. *A multi-tier architecture consists of clients, resources, components, and containers that are used by a programmer to create a distributive system.*

Accessing Services
A client uses a client protocol to access a service that is associated with a particular tier. There are a number of protocols that are used within a multi-tier infrastructure because each tier/component/resource could use different protocols.

One of the most commonly implemented multi-tier architectures is used in web-centric applications where browsers are used to interact with corporate online resources. A browser is a client and requests a service from a web server using HTTP.

J2EE Multi-Tier Architecture
J2EE is a four-tier architecture. These consist of the Client Tier (sometimes referred to as the Presentation Tier or Application Tier), Web Tier, Enterprise JavaBeans Tier (sometimes referred to as the Business Tier), and the Enterprise Information Systems Tier. Each tier is focused on providing a specific type of functionality to an application.

The Client Tier consists of programs that interact with the user. These programs prompt the user for input and then convert the user's response into requests that are forwarded to software on a component that processes the request and returns results to the client program. The component can operate on any tier, although most requests from clients are processed by components on the Web Tier. The client program also translates the server's response into text and screens that are presented to the user.

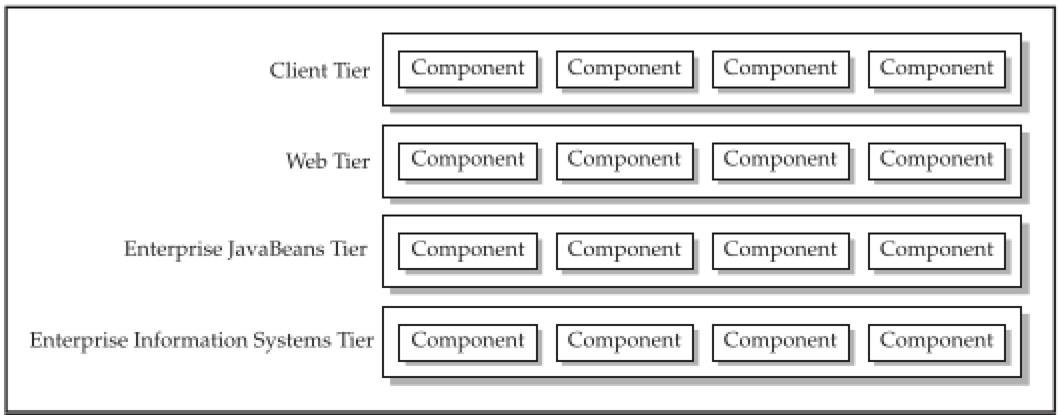

Client Tier
Component
Web Tier
Enterprise JavaBeans Tier
Enterprise Information Systems Tier

Figure 2-3. *J2EE consists of four tiers, each of which focuses on providing specific functionality to an application.*

The Web Tier provides Internet functionality to a J2EE application. Components that operate on the Web Tier use HTTP to receive requests from and send responses to clients that could reside on any tier.

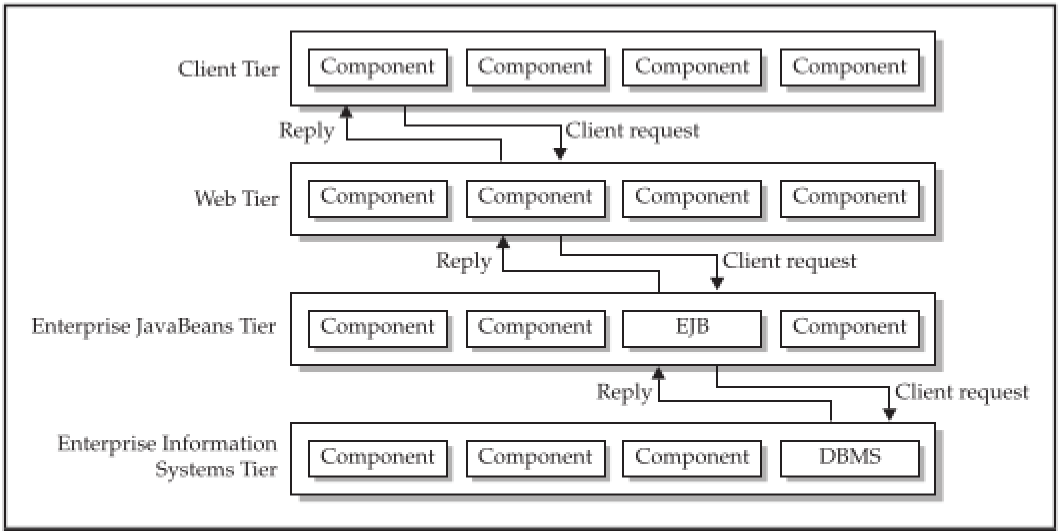

Client Tier
Component
Reply
Client request
Web Tier
Enterprise JavaBeans Tier
EJB
Enterprise Information Systems Tier
DBMS

Figure 2-4. *A request is typically passed from one tier to another before the request is fulfilled.*

For example a client's request for data that is received by a component working on the Web Tier is passed by the component to the Enterprise JavaBeans Tier where an Enterprise Java Bean working on the Enterprise JavaBeans Tier interacts with DBMS to fulfill the request. Requests are made to the Enterprise JavaBeans by using the Java Remote Method Invocation (RMI) API. The requested data is then returned by the Enterprise JavaBeans where the data is then forwarded to the Web Tier and then relayed to the Client Tier where the data is presented to the user.

The Enterprise JavaBeans Tier contains the business logic for J2EE applications. It's here where one or more Enterprise JavaBeans reside, each encoded with business rules that are called upon indirectly by clients. The Enterprise JavaBeans Tier is the keystone to every J2EE application because Enterprise JavaBeans working on this tier enable multiple instances of an application to concurrently access business logic and data so as not to impede the performance.

The EIS links a J2EE application to resources and legacy systems that are available on the corporate backbone network. It’s on the EIS where a J2EE application directly or indirectly interfaces with a variety of technologies, including DBMS and mainframes that are part of the mission-critical systems that keep the corporation operational.

Client Tier Implementation

The two components of client tier are applet clients and application clients. An applet client uses the browser as a user interface. An application client is a Java application which has its own user interface and is capable of accessing all the tiers in the multi-tier architecture. A rich client is a third type of client, but a rich client is not considered a component of the Client Tier because a rich client can be written in a language other than Java. A rich client also has its own user interface.

Classification of Clients

Besides defining clients as an applet client, application client, or a rich client, clients are also classified by the technology used to access components and resources that are associated with each tier. There are five classifications: a web client, Enterprise JavaBeans client, Enterprise Information System (EIS) client, web service peers, and a multi-tier client.

A web client consists of software, usually a browser, that accesses resources located on the Web Tier. These resources typically consist of web pages written in HTML or XML. Enterprise JavaBeans clients resides in client tier and only accesses one or more Enterprise JavaBeans that are located on the Enterprise JavaBeans Tier rather than resources on the Web Tier.

EIS clients are the interface between users and resources located on the EIS Tier. These clients use Java connectors, appropriate APIs, or proprietary protocols to utilize resources such as DBMS and legacy data sources.

A web service peer is a unique type of client because it’s also a service that works on the Web Tier. Technically, a web service peer forms a peer-to-peer relationship with other components on the Web Tier rather than a true client/server relationship.

Multi-tier clients are conceptually similar to a web service peer except a multi-tier client accesses components located on tiers other than the tier where the multi-tier client resides. Multi-tier clients typically use the Java Message Service (JMS) to communicate asynchronously with other tiers.

Web Tier Implementation

The Web Tier has several responsibilities in the J2EE multi-tier architecture, all of which is provided to the Client Tier using HTTP. A servlet is a Java class that resides on the Web Tier and is called by a request from a browser client that operates on the Client Tier.

A request for a servlet contains the servlet’s URL and is transmitted from the Client Tier to the Web Tier using HTTP. The request generates an instance of the servlet or reuses an existing instance, which receives any input parameters from the Web Tier that are necessary for the servlet to perform the service. Input parameters are sent as part of the request from the client.

An instance of a servlet fulfills the request by accessing components/resources on the Web Tier or on other tiers as is necessary based on the business logic that is encoded into the servlet. The servlet typically generates an HTML output stream that is returned to the web server. The web server then transmits the data to the client. This output stream is a dynamic web page.

JSP is similar to a servlet in that a JSP is associated with a URL and is callable from a client. However, JSP is different than a servlet in several ways, depending on the container that is used. Some containers translate the JSP into a servlet the first time the client calls the JSP, which is then compiled and the compiled servlet loaded into memory. The servlet remains in memory. Subsequent calls by the client to the JSP cause the web server to recall the servlet without translating the JSP and compiling the resulting code. Other containers precompile a JSP into a .java file that looks like a servlet file, which is then compiled into a Java class.

Enterprise JavaBeans Tier Implementation

The Enterprise JavaBeans Tier is a vital element in the J2EE multi-tier architecture because this tier provides concurrency, scalability, lifecycle management, and fault tolerance. The Enterprise JavaBeans Tier automatically handles concurrency issues that assure multiple clients have simultaneous access to the same object. The Enterprise JavaBeans Tier is the tier where some vendors include features that provide fault-tolerant operation by making it possible to have multiple Enterprise JavaBeans servers available through the tier. This means backup Enterprise JavaBeans servers can be contacted immediately upon the failure of the primary Enterprise JavaBeans server.

Collectively the Enterprise JavaBeans server and Enterprise JavaBeans container are responsible for low-level system services that are required to implement business logic of an Enterprise Java Bean. These system services are

- Resource pooling
- Distributed object protocols
- Thread management
- State management
- Process management
- Object persistence
- Security
- Deploy-time configuration

A key benefit of using the Enterprise JavaBeans server and Enterprise JavaBeans container technology is that this technology makes proper use of a programmer’s expertise. That is, a programmer who specializes in coding business logic isn’t concerned about coding system services. Likewise, a programmer whose specialty is system services can focus on developing system services and not be concerned with coding business logic.

Enterprise Information Systems Tier Implementation

The Enterprise Information Systems (EIS) Tier is the J2EE architecture’s connectivity to resources that are not part of J2EE. These include a variety of resources such as legacy systems, DBMS, and systems provided by third parties that are accessible to components in the J2EE infrastructure.

Developers can utilize off-the-shelf software that is commercially available in the marketplace because the EIS Tier provides the connectivity between a J2EE application and non-J2EE software. This connectivity is made possible through the use of CORBA and Java Connectors.

J2EE Best Practices

The old way of designing and building systems fails in meeting 24 hours, 7 days a week instant demand expected by thousands of concurrent users. To meet these demands new concepts in design were born. These concepts fall within two general categories: best practices and patterns. Best practices are proven design and programming techniques used to build advanced enterprise distributive systems. Patterns are routines that solve common programming problems that occur in such systems.

Enterprise Application Strategy

In olden days developers use to develop software for the existing requirements. Once the requirements are changed, they use to through the software and develop new software for modified requirements. This throwaway philosophy works well for applications that services a small group of users. However, this philosophy loses its economical and practical sense when applied to an enterprise application.

An enterprise application is a mission-critical system whose continual successful operation determines the corporation’s success. This means that an enterprise application is complex in nature and meets the needs of a diverse, constantly changing division of a corporation.

An enterprise application takes more time to develop. Once it is developed and operational, users may ask the changes after few days. Instead of changing the implementation, the changed requirements can be implemented as a separate module. We can call this module as hook. So this hook can be called from application. This minimizes the change in already developed application. But it has some disadvantages. That is if the number of hooks are increasing year by year, it is difficult to maintain the application.

To overcome these disadvantages, new approach has been adapted.

A New Strategy

In the corporate world, management sets the main objectives. These objectives are divided into sub objectives. Business units are formed to meet these sub objectives. All enterprise applications must interface with each other to assure that information can be shared amongst business units. That is, all applications and systems have to work together and have the flexibility built into the architecture so that an enterprise application is incrementally changed to meet new business demands without having to be reconstructed.

This eliminates the lag that occurs with the throwaway philosophy. No longer will business units need to suffer the pain that occurs when changes to the application are not made in a timely manner because the application is not maintainable. The new corporate information technology strategy is to employ an architecture within which enterprise applications can coexist and can be incrementally developed and implemented.

The Enterprise Application

Any application used by more than one person to conduct business could be considered an enterprise application. For the purpose of J2EE, consider an enterprise application to be one that

- Is concurrently used by more than a handful of users
- Uses distributive resources such as DBMS that are shared with other applications
- Delegates responsibility to perform functionality among distributive objects
- Uses web services architecture and J2EE technology to link together components

Corporate users having high expectations from an enterprise application. They want the enterprise application to

- Be available 24 hours a day, 7 days a week without any downtime
- Have an acceptable response time even in the face of increasing usage
- Have the flexibility to be modified quickly without requiring a redesign of the application
- Be vendor independent
- Be able to interact with existing systems
- Utilize existing system components

Clients

J2EE is organized into tiers, the first of which is the Client Tier. Software working on the Client Tier has several functions, many of which are easily developed by programmers. However, there are a few functions that pose a challenge to programmers. These functions are to

- Present the application's user interface to the person who is using the application
- Collect and validate information from the person using the application
- Control the client's access to resources
- Prevent clients from sending duplicate requests

Client Presentation

An enterprise application uses a thin client model where nearly all functionality associated with the application is contained on the server-side rather than with the client. Thin clients handle the user interface that presents information to the user and captures input from the user. There are two strategies for building presentation functionality into a client. These are to use a browser-based user interface or to create a rich client where a graphical user interface is programmed into the client. Each has its advantages and disadvantages.

Advantages of browser based strategy:
1. Implementation becomes easier.
2. Training is not required for a person to use browser based application.

Disadvantages of browser based strategy:
1. The developer doesn't have exact control over presentation, because the look and feel may change across different browsers.

2. The presentation is limited to interactions that can be implemented using a markup language.
3. It access the server each time an event is generated, which decreases the performance.
4. A browser-based strategy typically uses the HTTP protocol. HTTP is a stateless protocol, so the developer must have a strategy for maintaining session state on the server.

The richer client strategy gives a developer total control over elements of the presentation. That is, a developer can use WYSIWYG(what you see is what you get) to create the presentation and isn't limited to only events that are trapped by the browser. In addition, a richer client accesses a server only as needed and not in response to nearly every event that occurs in the presentation. This might result in fewer server interactions and increase response time.

***Best Practice** - Developing a Client for Enterprise Application:* The best practice when developing a client for an enterprise application is to use the strategy that is most appropriate for each aspect of the presentation. That is, both strategies can be used for pieces of the presentation of an enterprise application. Use the browser-based strategy for simple presentations and the richer client strategy whenever more complex presentations are necessary that cannot be adequately handled directly by a browser.

Client Input Validation

Information that is entered into a client by a user should be validated, depending on the nature of the information. Details of the validation process are application dependent; however, the developer has two places where validation can occur: with the client or on the server side.

A developer can implement three kinds of validation strategies: syntactic, lexical, and semantic. Syntactic validation determines if information that consists of several related values is well formed, such as time that is composed of hours and minutes. Lexical validation looks at a single value to assure that the type of value corresponds to the type of data that is expected by the application. For example, an hour value must be an integer to pass a lexical validation. Semantic validation examines the meaning of the information to determine if the information is likely to be correct. Semantic validation determines if the value of the hour is less than 24 and greater than or equal to 0.

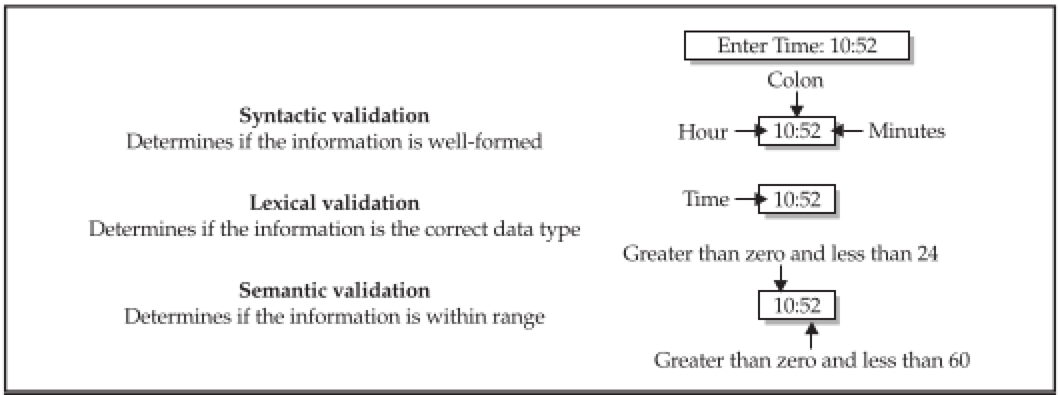

Enter Time: 10:52
Colon
Syntactic validation
Determines if the information is well-formed
Hour
10:52
Minutes
Lexical validation
Determines if the information is the correct data type
Time
10:52
Greater than zero and less than 24
Semantic validation
Determines if the information is within range
10:52
Greater than zero and less than 60

Figure 3-1. *Syntactic, lexical, and semantic are three validation strategies that can be used in an enterprise application.*

There are three fundamental factors that must be considered when designing validation procedures for an enterprise application. These are to avoid duplicating the validation procedure within the application, provide the user with immediate results from the validation process, and minimize the effect the validation process has on the server.

If validation procedure is written at client side, then the same code will repeat across all client machines. If there is any modification in validation procedure, it need to be modified in all client machines. This can be implemented by implementing validation procedure at server side.

Users require nearly instantaneous feedback as to the validity of the data whenever they enter information into a client. Feedback should indicate that the data is valid or invalid. Any delay providing feedback to the user can lead to a poor user experience with the application.

A common source of delay is when a developer uses a server-side validation process. This is because a conflict might arise if multiple applications concurrently call the object that validates data. A request to validate data might be queued until another application's data is validated.

Best Practice - *Developing a Validation Strategy:* The best practice when developing a validation strategy is to perform as much (if not all) of the validation on the server side. Avoid validating on the client side because client-side validation routines frequently must be updated when a new version of the browser is released.

Best Practice - *Reducing the Opportunity for User Error:* Another best practice is to reduce the opportunity for a user to enter incorrect information by using presentation elements that limit choices. These elements include radio buttons, check boxes, list boxes, and drop-down combo boxes. Information that is collected using those elements doesn't have to be validated because the application presents the user with only valid choices. There isn't any opportunity for the user to enter invalid data.

Client Control

In every enterprise application, clients are restricted to resources based on the client's needs. The scope of resources a client can access is commonly referred to as a client view. A client view might consist of specific databases, tables, or rows and columns of tables. There are two ways for a client view to be defined: through embedding logic to define the view into the application, or by using a controller component that is known as a resource guard.

The resource guard is a component that resides outside of the application that receives requests for resources from all applications. A request for a resource contains a client ID that is compared to the client's configuration. Access is either granted or rejected based upon access rights within the client's configuration. Another method that is useful whenever only database access is required is to group together users who have similar needs into a group profile, then assign permissions to the group. In this way, the DBMS manages security directly without you having to write your own security routines.

Best Practice - *Using Security Measures:* The best practice is to use security measures that exist in DBMS or networks, where feasible. However, if this isn't the best alternative, use a resource guard rather than embed the definition of a client view into the application, unless only one client uses the resource. This is because multiple clients and applications can use the resource guard to access shared resources. In this way, the logic to define the client view isn't duplicated in multiple applications.

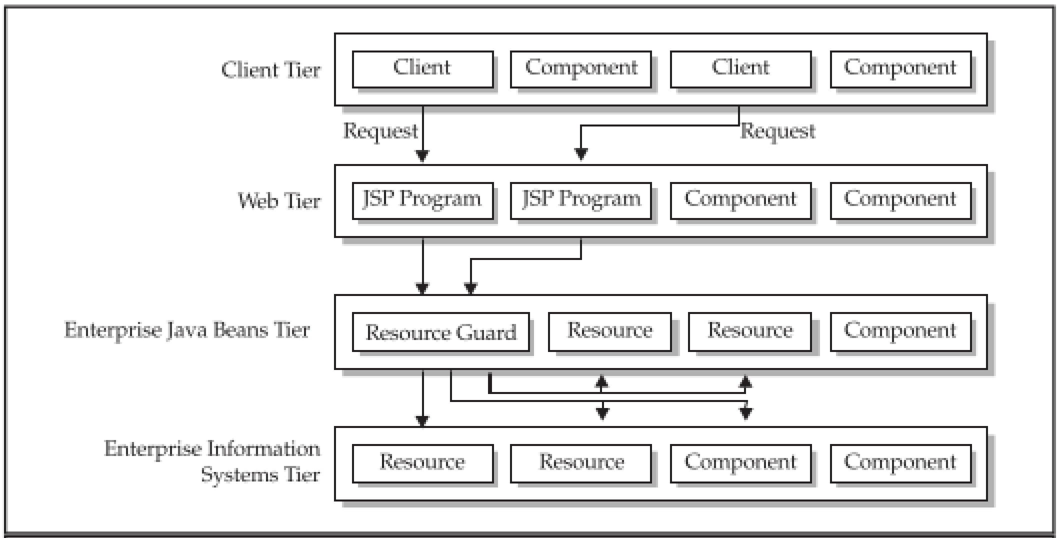


Figure 3-2. *A resource guard is a component outside the application that manages access to restricted resources.*

Best Practice - *Working with a Resource that Becomes Sharable:* The best practice to use when a resource becomes sharable is to remove the embedded logic that defines the client view from the application and place the logic into a resource guard. In this way, the logic is preserved while still providing flexibility to the application.

Best Practice - *Using Resource Guards :* The best practice is to use an Enterprise JavaBean as the resource guard and use security mechanisms that are already in the web container and resource, rather than build the logic for the resource guard from scratch.

Once the client view is defined, the developer must determine a strategy for implementing the client view. There are two commonly used strategies: the all-or-nothing strategy or the selective strategy. The all-or-nothing strategy requires the developer to write logic that enables or prevents a client from accessing the complete resources. Simply said, the client can either access all features of a resource or is prohibited from accessing the resource entirely.

In contrast, the selective strategy grants a client access to the resource, but restricts access to selected features of the resource based on the client's needs. For example, a client may have read access to the table that contains orders, but doesn't have rights to insert a new order or modify an existing order.

Best Practice - *Using the Selective Strategy :* The best practice is to use the selective strategy because this strategy provides the flexibility to activate or deactivate features of a resource as required by each client. The all-or-nothing strategy doesn't provide this flexibility. This means the developer will need to replace the all-or-nothing strategy with the selective strategy after the application is in production, should a client's needs change. In addition, the selective strategy can also be used to provide the same features as provided by the all-or-nothing strategy. That is, the developer can grant a client total access to a resource or prohibit a client from accessing the resource.

Duplicate Client Requests

A common problem with thin client applications is for the client to inadvertently submit a duplicate request for service, such as submitting a duplicate order. There are many ways a client can generate a duplicate request.

A browser is the user interface used in a thin client application. However, the browser contains elements that can lead to a duplicate request being sent. Namely, the Back and Stop buttons. The Back button causes the browser to recall the previously displayed web page from the web server. The Stop button halts the implementation of a request. This means the browser processes some, but not all, of the requests. Normally, the selection of these buttons has minimal consequence to the client because the browser either displays an unwanted page (previous page) or displays a partial page. In both cases, the user can easily correct the situation.

However, a problem might occur if the client is sending a web form such as an order form to the Web Tier. Let's say that the user submits the order form. The application generates a confirmation web page that the browser displays. The user then inadvertently selects the browser's Back button, which redisplays the order form. This might be confusing and cause the user to resubmit the order form. In reality, two orders are submitted.

Best Practice - *Flushing the Session:* The best practice is to flush the session explicitly and updating the session object without waiting for the page to complete. The session is still available and checks can be made on the server to see if the form was previously submitted.

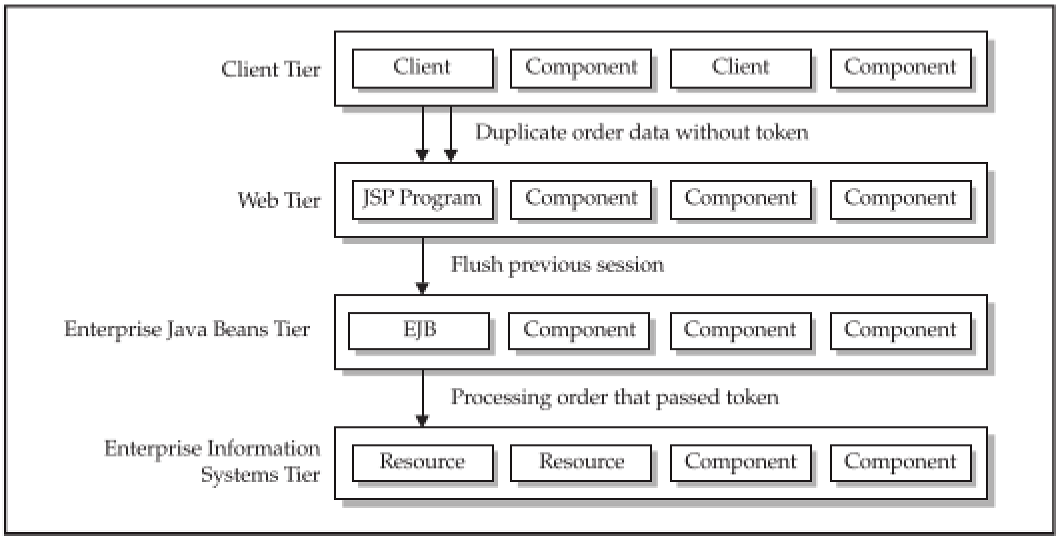


Figure 3-3. *Flush the previous session as the first action the program takes when a new form is submitted. This avoids duplicate sessions from occurring.*

Sessions Management

A web service-based enterprise application consists of distributive services (i.e., components) located on J2EE tiers that are shared amongst applications. A client accesses components by opening a session with the Web Tier. The session begins with an initial request for service and ends once the client no longer requests services.

During the session, a client and components exchange information, which is called session state. Practically any kind of information can be a session state, including a client’s ID, a client’s profile, or choices a client makes in a web form.

A component is an entity whose sole purpose is to receive information from a client, process that information, and return information to the client when necessary. Information used by a component is retained until the request from the client is fulfilled. Afterwards, information is destroyed. A component lacks persistence. Persistence is the inherent capability of retaining information (session state) between requests.

This means that it is up to the enterprise application to devise a way to maintain session state until the session is completed. There are two common ways to manage session state: on the client side or server side on the Enterprise JavaBeans Tier.

Client-Side Session State

Session state can be maintained at client side using three techniques. These are by using a hidden field, by rewriting URLs, and by using cookies.

Hidden Field

The component can include in the HTML form a field that isn’t displayed on the form. This field is called a hidden field. A hidden field is similar to other fields on the form in that a hidden field can hold a value. However, this value is assigned to the hidden field by the component rather than by the user.

A hidden field is treated the same as other fields on the form when a user selects the Submit button (see Listing 3-3). That is, the name of the hidden field and the value of the hidden field are extracted from the form along with the other fields by the browser and sent as a parameter to the component. This means that session state can be retained by assigning the session state as a value to one or more hidden fields on each form that is used during the session.

```
<INPUT  TYPE="HIDDEN" NAME="AccountNumber" VALUE="1234">
<INPUT  TYPE="TEXT" NAME="FIRSTNAME" SIZE="40">
<INPUT  TYPE="SUBMIT" VALUE="SUBMIT">
<INPUT  TYPE="RESET" VALUE="CLEAR">
```

Best Practice - *Using Hidden Fields* : The best practice for using a hidden field to maintain session state is to do so only when small amounts of string information need to be retained. Although using a hidden field is easy to implement, it can cause performance issues if large amounts session of state are required by the application. This is because the session state must be included with each page sent to the browser during the session regardless if the session state plays an active role on the page.

In addition, hidden fields contain only string values and not numeric, dates, and other data types. Furthermore, session state is exposed during transmission. This means session state that contains sensitive information such as credit card numbers must be encrypted to secure the information during transmission.

URL Rewriting
A URL is the element within the web page that uniquely identifies a component and is used by the browser to request a service. In this technique developer will append parameter name and parameter value to the URL. When the user clicks on this URL, it is sent to server. The server will read the parameter names and parameter values present in the URL and used them as session data.

```
<P> <A HREF = "http://www.mysite.com/jsp/myjsp.jsp?AccountNumber=1234>
Click to place a new order</A></P>
```

In the above example AccountNumber is parameter name and 1234 is parameter value.

Best Practice - *Using URL Rewriting :* The best practice is to use URL rewriting to retain session state whenever an HTML form is not used by the client. For example, the user might place an order using an HTML form. The order contains the user’s account number along with other information.

Cookies
A cookie is a small amount of data (maximum of 4KB) that the enterprise application stores on the client’s machine. A cookie can be used to store session state. The developer can create a JSP program that dynamically generates a page that writes and reads one or more cookies. In this way, session state persists between pages.

Best Practice - *Using Cookies :* The best practice is to use a cookie only to retain minimum data such as a client ID and use other techniques described in this section to retain large amounts of information. A developer must also implement a contingency routine should the user discard the cookie or deactivate the cookie feature.

There are two major disadvantages of using cookies to retain session state. First, cookies can be disabled, thereby prohibiting the enterprise application from using a cookie to manage session state. The other disadvantage is that cookies are shared amongst instances. This means a client might run two instances of a browser, creating two sessions. All instances access the same cookie and therefore share the same session state, which is likely to cause conflicts when processing information because the wrong session state might be processed.

Server-Side Session State

Storing session state on the client side has serious drawbacks, most of which center on the dependency on the client's machine. That is, session state is lost if a client's machine fails. An alternative to maintaining session state on the client side is to store session state on the server. Typically, the information technology department of a corporation goes to extremes to assure that the server and backup servers are available 24 hours a day, 7 days a week, which is not the treatment given to client machines.

Best Practice - *Using the HttpSession Interface :* The best practice is to maintain session state on the Enterprise JavaBeans Tier using an Enterprise JavaBean or on the Web Tier using the HttpSession interface. Each session state is assigned a session ID, which relates the session state with a particular client session. The session ID is used whenever the session state is written to or retrieved from the server.

This provides the most reliable way to save and access session state and is scalable, thereby able to handle multiple sessions and various sizes of session state. It also decreases the vulnerability to someone inadvertently or covertly gaining unauthorized access to the session state.

Replication Servers

It is not uncommon for an enterprise application to use a cluster of replication servers where each server has the full complement of components. Whenever a request is received from a client, the request is routed to the next available server within the cluster. In this way, the infrastructure can maintain acceptable performance even if hundreds of requests are received simultaneously.

However, this can easily result in a problem if session state is maintained on the server side. Which server has the session state for the client? There are two strategies that are used to manage this problem. These are to replicate session state across all servers within the cluster or to route a client to the same server for the duration of the session.

Best Practice - *Maintaining a Sticky User Experience :* The best practice is to maintain a sticky user experience, depending on your business needs. This means the client always uses the same server during the session and the session state is stored on one server within the cluster. This also means that the session is lost should the server go down.

Valid Session State

Another issue that is common with an enterprise application that stores session state on a server is whether or not the session state is valid. Session state automatically becomes invalid and removed when the session ends. However, there might be occasions when the session ungracefully terminates without removing the session state, such as during a communication failure that occurs during the session.

Best Practice - *Setting a Session Timeout :* The best practice is to always set a session timeout, which automatically invalidates session state after time has passed and the session state has not been accessed. The actual length of time before the session automatically terminates will vary depending on the nature of the application. However, once time has expired, the session ends and therefore the session state is removed.

Web Tier and JavaServer Pages
The web tier contains components that directly communicate with clients. The Web Tier is also the location where JavaServer Pages (JSP) programs reside.

Presentation and Processing
A JSP program can contain two components: the presentation component and the processing logic. The presentation component defines the content that is displayed by the client. Processing logic defines the business rules that are applied whenever the client calls the JSP program.

Placing both the presentation and processing logic components in the same component lead to nonmaintainable code. This is because of two reasons. First, presentation and processing logic components tend to become complex and difficult to understand.

The other reason is that programmers with different skill sets typically write each component. A programmer who is proficient in HTML writes the presentation component. A Java programmer writes the processing logic component. This means that two programmers must work on the same JSP program, which can be inefficient.

Best Practice - *Separating Code :* The best practice for writing a JSP program is to separate the presentation code and the processing code. Place the presentation code in the JSP program and place the processing code in Enterprise JavaBeans. Have the JSP program call the Enterprise JavaBeans whenever the JSP program is required to process information. An alternative best practice is to simply include files that contain code to hide the code from the graphic artist.

There are several benefits to using this strategy. First, the JSP program is easier for a programmer to understand. Another benefit is that the processing logic is sharable with other enterprise applications. This is because the Enterprise JavaBean that contains the processing logic can be called by other applications.

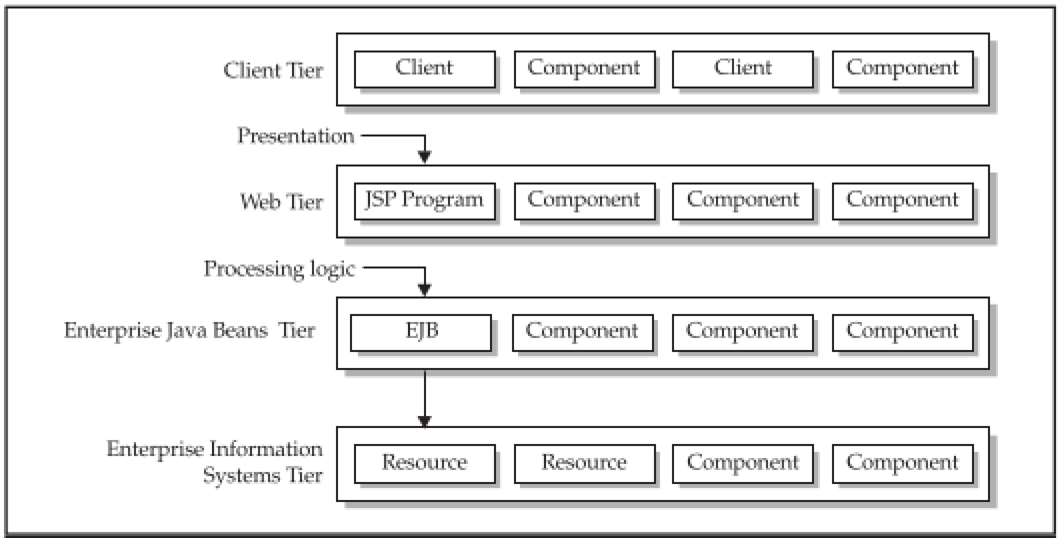


Figure 3-4. *A JSP program is used to create the presentation for an enterprise application while Enterprise JavaBeans are used to process information.*

The Inclusion Strategy

The designer of an enterprise application typically uses the same elements for all web pages of the user interface to provide continuity throughout the application. This means that the same code can appear in more than one web page, which is inefficient.

Best Practice - *Using the Inclusion Strategy :* The best practice to avoid redundant code is to use the inclusion strategy, which uses either the include directive or include action to insert commonly used code into a JSP program. The JSP program include directive is used within the JSP program to specify the file whose content must be included in the JSP program. This file should contain shared code such as HTML statements that define common elements of web pages. The include action calls a JSP program within another JSP program. Commonly used code is contained in the called JSP program.

The critical difference between the include action and the include directive is that the include action places the results generated by the called JSP program into the calling JSP program. In contrast, the include directive places source code into the JSP program rather than the results of running that source code.

Best practice – *using the Include Directive :* The best practice is to use the include directive whenever variables are used in the JSP program. The include directive places the variable name in the calling JSP program and lets the calling JSP program resolve the variable. In contrast, the include action places the value of the variable in the calling JSP program.

Style Sheets

It is important that an enterprise application's user interface is consistent throughout the application. Consistency helps the user become familiar with how to use the application. This means that the developer must write JSP programs that generate web pages having the same general appearance.

Best Practice - *Enforcing Continuity with CSS :* The best practice to enforce continuity among web pages that comprise an application's user interface is to use Cascading Style Sheet (CSS). CSS is a file that describes the style of elements that appear on the web page.

The developer tells the client (i.e., browser) to reference the CSS file by using a CSS directive in the web page. The browser then automatically applies the style defined in the CSS file to the web page that contains the CSS directive. The programmer can easily change the style of the user interface by modifying the CSS file. Those changes are automatically implemented as the client displays each web page.

Besides making the user interface maintainable, CSS also reduces the amount of storage space required for web pages. This is because code that is used to define the style is placed in a shared file rather than being replicated in all web pages that use the style.

Simplify Error Handling

An enterprise application should trap errors that can occur at runtime using techniques that are available in Java (see Java 2: The Complete Reference). However, it is important that the application translate raw error messages into text that is understood by the user of the application. Otherwise, the error message is likely to confuse.

Although errors can occur throughout the application, a client should present to the user errors generated by a JSP program or by a servlet. This is true even if an Enterprise JavaBean called by the JSP program catches the error. Once the error is trapped, the Enterprise JavaBeans forwards the error message to the JSP program. The JSP program translates the error message into a message easily understood by a user and then displays the translated error message in a dynamically generated a web page.

Best Practice - *Handling Errors :* A best practice for handling errors is to generate a user-friendly error message that reflects the processing that was executing when the error occurred, session state (where applicable), and the nature of the error.

Best Practice -*Saving Error Messages :* Another best practice when handling errors is to have either the JSP program or the Enterprise JavaBeans save all error messages and related information (i.e. session state) to an error file, then notify technical support that an error was detected.

Enterprise JavaBeans Tier

The Enterprise JavaBeans Tier contains Enterprise JavaBeans that provide processing logic to other tiers. Processing logic includes all code and data that is necessary to implement one or more business rules.

The purpose of creating an Enterprise JavaBean is to encapsulate code that performs one task very well and to make that code available to any application that needs that functionality. Although the concept of using Enterprise JavaBeans is easily understood, there can be confusion when designing Enterprise JavaBeans into an application's specification. Simply stated, the developer must determine what functionality should be built into an Enterprise JavaBean.

Best Practice—Making JavaBeans Self-Contained The best practice is make each Enterprise JavaBeans self-contained and minimize the interdependence of Enterprise JavaBeans where possible. That is, avoid having a trail of Enterprise JavaBeans calling each other. Instead, design an individual Enterprise JavaBean to complete a specific task.

Entity to Enterprise JavaBeans Relationship

There is a tendency for developers to create a one-to-one relationship between entities defined in an application's entity relationship diagram and with Enterprise JavaBeans. That is, each entity has its own entity Enterprise JavaBeans that contains all the processing logic required by the entity.

While the one-to-relationship seems a logical implementation of the entity relationship diagram, there are drawbacks that might affect the efficient running of the application. Each time an Enterprise JavaBean is created, there is increased overhead for the network and for the Enterprise JavaBeans container. Creating a one-to-one relationship, as such, tends to generate many Enterprise JavaBeans and therefore is likely to increase overhead, which results in a performance impact. In addition, this also limits the scalability of the application.

Best Practice - *Translating Entity Relationship Diagrams :* The best practice when translating an entity relationship diagram into Enterprise JavaBeans is to consolidate related processes that are associated with several entities into one session Enterprise JavaBeans. This results in the creation of fewer Enterprise JavaBeans while still maintaining the functionality required by the application.

Efficient Data Exchange

A JSP program and Enterprise JavaBeans frequently exchange information while the enterprise application is executing. The JSP program might pass information received from the client to the Enterprise JavaBeans. Likewise, the Enterprise JavaBeans might return values to the JSP program once processing is completed.

There are two common ways in which information is exchanged between a JSP program and Enterprise JavaBeans. These are by individually sending and receiving each data element or by using a value object to transfer data in bulk. Some developers intuitively use a value object only when bulk data needs to be transferred and send data individually when single values are exchanged.

Best Practice - *Exchanging Information Between JSP and Enterprise* : The best practice when exchanging information between a JSP program and Enterprise JavaBeans is to use a value object. In this way, there is less stress on the network than sending individual data and the value objects retain the association among data elements.

Enterprise JavaBeans Performance
While Enterprise JavaBeans provide an efficient way to process business rules in a distributed system, Enterprise JavaBeans remains vulnerable to bottlenecks that occur when Enterprise JavaBeans communicate with other components. Bottlenecks effectively decrease the efficiency of implementing Enterprise JavaBeans. The primary cause of bottlenecks is remote communication - that is, communication that occurs between components over the network.

Best Practice - *Placing Components in Communication :* The best practice is to keep remote communication to the minimum needed to exchange information and to minimize the duration and any communication. A common way to accomplish this objective is by placing components that frequently communicate with each other on the same server, where possible.

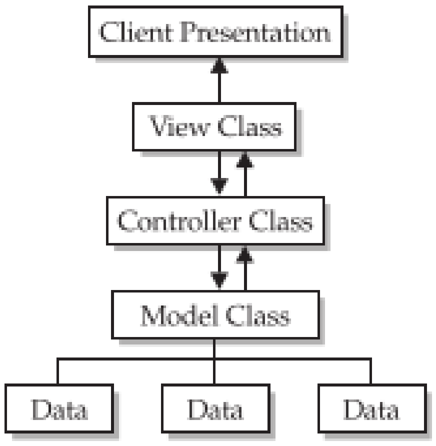
Consider Purchasing Enterprise JavaBeans
There are entities and workflow common to many businesses. Information and processes used for both of these are encapsulated into entities Enterprise JavaBeans and session Enterprise JavaBeans, respectively. An entity Enterprise JavaBeans is modeled after an entity in the enterprise application’s entity relationship diagram. A session Enterprise JavaBeans is modeled after processes common to multiple entities. Many corporations have entities that use the same or very similar functionality. For example, many corporations use the same credit card approval process. Therefore, the same basic Enterprise JavaBeans is re-created in each corporation. This means corporations waste dollars by building something that is already available in the marketplace.

Best Practice - *Surveying the Marketplace:* The best practice is to survey the marketplace for third-party Enterprise JavaBeans that meet some or all of the functionality that is required by an entity Enterprise JavaBeans or session Enterprise JavaBeans for an enterprise application. The Sun Microsystems, Inc. web site offers third-party Enterprise JavaBeans in their Solutions Marketplace.

The Model-View-Controller (MVC)
Best Practice - *Using the Model-View-Controller :* The best practice for simplifying the distribution of an application’s functionality is to use the Model-View-Controller (MVC) strategy that is endorsed by Sun Microsystems, Inc. and which has its roots in the decades-old technology of Smalltalk.

The MVC strategy basically divides applications into three broad components. These are the model class, the view class, and the controller class. The model class consists of components that control data used by the application. The view class is composed of components that present data to the client. And the controller class is responsible for event handling and coordinating activities between the model class and the view class.

Enterprise JavaBeans are used to build components of the model class. Likewise, JSP programs and servlets are used to create view class components. And session Enterprise JavaBeans are used for controller class components.



The Myth of Using Inheritance
There are three reasons for using inheritance in an enterprise application. First, inheritance enables functionality and data to be reused without having to rewrite the functionality and data several times in an application. Inheritance is also used to embellish both a functionality and data. That is, the class that inherits a functionality can modify the functionality without affecting the original functionality. Inheritance also provides a common interface based on functionality that is used by similar classes.

There are two kinds of classes used in inheritance: the base class and the derived class. The base class contains methods and data some or all of which are inherited by a derived class. An object of the derived class has access to some or all of the data and methods of the base class and all the data and methods of the derived class.

The relationship between a base class and derived class is referred to as coupling. That is, there is a derived class bonded to a base class. A base class must exist for the derived class to exist.

Developers of enterprise applications are concerned with how to efficiently translate an entity relationship diagram into an application's class model. A common error is for the developer to rely heavily on inheritance, which results in an application built on coupled classes.

Best Practice - *Using an Interface :* The best practice when translating an entity relationship diagram to an application's class model is to use an interface rather than use coupling, where possible. An interface is a class that adds functionality to a real-world object.

Interfaces and Inheritance
An interface contains functionality that is used across unlike real-world objects. This is different from a base class in that a base class provides functionality that is fundamental to like real-world objects.

For example, an acceleration interface provides acceleration functionality to any real-world object regardless if the real-world object is a motor vehicle, aircraft, or a baseball that is hit into the outfield. In this way, unrelated real-world objects that have the same functionality can share the same data and methods of an interface.

Best Practice - *Identifying Functionality* : The best practice is to identify functionality that is common among real-world objects that are used by an application. Place these features into a base class. The remaining features that are unique to each of these objects are placed into a derived class. And where there is a function that is common to unlike real-world objects, place that function into an interface class.

Composition and Inheritance

A misnomer when designing an application that uses an interface is that designers cannot reuse the implementation of the interface that is, the method that contains the interface signature can only be used with one object.

Best Practice - *Creating a Delegate Class:* The best practice is to create a delegate class. A delegate class contains implementations of interfaces that are commonly used by objects in an application. Let's say that the acceleration interface discussed previously in this section uses a standard formula to calculate the acceleration of an object. Therefore, the implementation of the acceleration interface is the same across unlike objects. A delegate class should be created that defines this implementation so the implementation can be used by other objects by calling the acceleration method of the delegate class.

Using a delegate class is a keystone to the composition strategy, which is a way of extending the responsibilities of an object without inheritance. The composition strategy delegates responsibility to another object without the object being a derived class. This is a subtle but critical factor that differentiates composition from inheritance.

Best Practice - *Using Composition :* The best practice is to use composition whenever functionality needs to expand the responsibilities of unlike objects. Instead of incorporating the function in each object or in each of the object's base classes, the developer should delegate the responsibility to a delegate class that performs the functionality as required by the object.

Potential Problems with Inheritance

One of the problem of inheritance is called the ripple effect. The ripple effect occurs whenever a change is made to the base class. Changes to a base class ripple down to all the derived classes and might negatively impact the implementation of attributes and functionality of the derived class.

This means that the developer who is responsible for maintaining a base class must examine the impact any change in the base class has on derived classes before making the change. Failure to assess the potential impact of a change could lead to errors in the derived class.

Another problem with inheritance happens as an application matures and requires frequent changes to both base and derived classes. These changes result in object transmute where data and methods of an existing class are moved to another class after the existing class is deleted.

Best Practice - *Minimizing the Use of Inheritance :* The best practice is to minimize the use of inheritance in an enterprise application. Only use inheritance when there is commonality among objects that is not functionality. Otherwise, use composition. That is, an object that is a "type of" should inherit commonality from a base class. An object that performs functionality "like" other classes should use composition to delegate responsibility.

Maintainable Classes

There are two factors that determine if classes are maintainable. These are coupling and cohesion. Coupling occurs when there is a class dependent on another class. Coupling also occurs when a class delegates responsibility to another class.

Before a change can be made to either the derived class or the base class, the developer must assess the impact on the coupled class. Changes to a base class might negatively impact a derived class. Changes to a derived class won't affect the base class, but could inadvertently modify functionality inherited from the base class. In either scenario, additional precautions must be taken by the developer to assure that the modification doesn't cause a negative impact.

Cohesion describes how well a class' functionality is focused. That is, a class that has broad functionality isn't as cohesive as a class that has a single functionality. For example, a class that validates account status and processes orders is less cohesive than a class that simply validates an account status.

Another design consideration is to avoid packages with cross dependencies. If package A has a class that depends on a class in package B, then the developer needs to ensure he or she doesn't introduce a class in B that depends on a class in A.

Best Practice - *Designing an Enterprise Application* : The best practice is to design an enterprise application with highly cohesive classes and minimum coupling. This strategy ensures that classes are optimally designed for maintenance. This is because a class that has few functions and isn't dependent on another class is less complicated to modify than a class with broad functionality that inherits from a base class.

Performance Enhancements

Although bytecode is optimized, bytecode still needs to be interpreted by the Java Virtual Machine (JVM). It is this overhead that detracts from the application's performance. There are two ways to reduce or practically eliminate the amount of bytecode that is interpreted at runtime: by using HotSpot, new in the Just In Time compiler from Sun Microsystems, Inc., or by using a native compiler.

HotSpot tunes the performance of an application at runtime so the application runs at optimal performance. HotSpot analyzes both client- and server-side programs and applies an optimization algorithm that has the greatest performance impact for the application. An application that uses HotSpot typically has a quick startup. The drawback of using HotSpot is that machine time and other resources are used to analyze and optimize bytecode while the application runs, rather than simply dedicating these resources purely to running the application. Another concern about using HotSpot is the complexity of debugging runtime errors. Runtime errors occur in the optimized code and not necessarily in the bytecode. Therefore, it can be difficult to re-create the error.

Best Practice - *Testing the Code :* The best practice for achieving top performance is to test both the native compiled code with the bytecode of the program. Code compiled with a native compiler such as that offered by Tower Technology optimizes translated bytecode into an executable similar to how source code written in C++ and other programming languages is compiled into an executable.

A new hybrid strategy is developing to increase performance of a Java application where an application is divided into static and dynamic modules. Static modules such as an Enterprise JavaBeans container are compiled into native libraries that are executables, and dynamic modules such as Enterprise JavaBeans are compiled into bytecode. Only dynamic modules are optimized at runtime.

The Power of Interfaces

Designing an enterprise application using interfaces provides built-in flexibility, known as pluggability, because an interface enables the developer to easily replace components. An interface is a collection of method signatures. A method signature consists of the name and the number and type of parameters for a method.

Let’s say a developer is building an enterprise application that processes orders. However, there are two different processes. One processes for bulk sales and another for retail sales. Two classes are defined, called retailOrder and bulkOrder. Also, an interface is created to make uniform the way in which components send orders. We’ll keep the interface simple for this example by requiring an order to have an account number, order number, product number, and quantity. The interface defines a method signature used to send an order.

```
sendOrder(int, int, int, int);
```

The developer must define a method in the retailOrder and bulkOrder classes that have the same signature as the sendOrder method signature. Likewise, the developer must place code within these methods to receive and process the order. The programmer who wants to write a routine that sends an order needs only to know the class name to use and that the class implements the interface. The programmer already knows the method to call to send the order because the method signature is defined in the interface.

If the programmer wants to send a retail order, the programmer creates an instance of the retailOrder class and then calls the sendOrder() method which passes it the order information. A similar process is followed to send a bulkOrder, except the programmer creates an instance of the bulkOrder class. The call is made to the same method.

Best Practice - *Handling Differing Behaviors :* The best practice is to create an interface whenever an application contains common behaviors where algorithms used to process the behaviors differ. Basically, the developer calls the method using the interface and passes the method information it needs and the method does everything else.

The Power of Threads

Proper use of threads can increase the efficiency of running an enterprise application because multiple operations can appear to perform simultaneously. A thread is a stream of executing program statements. More than one thread can be executed within an enterprise application. This means that multiple statements can run parallel.

The thread class is used to create a thread. Once the thread is created, the developer can specify the behavior of the thread (i.e., start, stop) and the point within the program where the thread begins execution.

A major benefit of using threads in an enterprise application is to be able to share processing time between multiple processes. Only one thread is processed at a time, although using multiple threads in an application gives the appearance of concurrent processing. Actually, the number of application threads being processed at one time is equal to or less than the number of CPUs on the machine.

The developer can assign a priority to each thread. A thread with a higher priority is processed before threads with lower priority. In this way, the developer is able to increase the response time of critical processes. The use of threads in an enterprise application can be risky if methods executed by threads are not synchronized, because more than one thread might run within a method. Multiple threads that execute the same method share values, which can cause unexpected results.

Best Practice - *Using Threads in an Enterprise Application :* There are several best practices to employ when using threads in an enterprise application. Avoid using threads unless multiple processes require access to the same resources concurrently. This is because using multiple threads incurs processing overhead that can actually decrease response time.

Keep the number of threads to a minimum; otherwise, you'll notice a gradual performance degradation. If your application experiences a decrease in performance, prioritize threads. Assign a higher priority to critical processes.

Another method to increase performance when using multiple threads in an application is to limit the size of methods that are synchronized. The objective is to minimize the amount of time that a synchronized method locks out other threads.

Careful analysis of a synchronized method might reveal that only a block of code within the method affects values that must not be accessed by another thread until the process is completed. Other code in the method could be executed without conflicting with other threads.

If this is the case, place the block of code that affects value into a separate synchronized method. Typically, this reduces the amount of code that is locked when the thread executes and therefore shortens the execution time of the method.

Be on the alert for possible deadlocks when using too many synchronized methods in an application. A deadlock can occur when a synchronized method calls other methods that are also synchronized. The call to the method is a thread that might be queued because another thread is executing in the called method. And the called method might also call another synchronized method, causing a similar situation to occur.

Best Practice - *Suspending or Stopping Threaded Objects :* The best practice is to design objects so that a request is made that a threaded object be either suspended or stopped, rather than directing the threaded object to suspend and stop. There is a subtle but important difference between a request and a directive. A directive causes an action to occur immediately. A request causes an action to occur at an appropriate time.

The Power of Notification

An enterprise application typically has many events that occur randomly. Each event might affect multiple objects within the application and therefore it is critical that a notification process be implemented within the application, so that changes experienced by an object can be transmitted to other objects that are affected by the change.

There are three ways in which objects are notified of changes: passive notification, timed notification, and active notification. Passive notification is the process whereby objects poll relative objects to determine the current state of the object. The current state is typically the value of one or more data members of the object.

Although passive notification is the easiest notification method to implement, this is also the notification method that has the highest processing overhead. This is because there could be many objects polling many other objects, and each poll consumes processing time.

An alternative to the passive notification method is timed notification. Timed notification suspends the thread that polls objects for a specific time period. Once time expires, the thread comes alive once more and polls relative objects to determine the current state of the object.

The drawback of both passive notification and timed notification is that each of these notification methods polls relative objects. This means that polling occurs even if there are no changes in status. This wastes processing time.

Best Practice - *Using Active Notification* : The best practice is to use active notification. Active notification requires the object whose status changes to notify other relative objects that a change occurred. Objects that are interested in the status of the object must first register with the object. This registration process basically identifies the objects that need to be notified when a change occurs.

The original active notification method was called the observable-observer method, which is also known as the publisher-scriber model. The publisher is the object that changes and the subscriber is the object that is interested in learning about these changes.

Best Practice - *Creating Different Threads :* The best practice is to create different threads for each notification process and assign a priority to each thread based on the importance of the notification. Assign a high priority to those notifications that are critical to running the application, and assign a low priority to those less critical to the success of the application.

Advance Java Programming

Unit 3 Notes

Java Server Pages

A Java Server Page(JSP) is a server side program that is similar in design and functionality to a Java Servlet.. A JSP is called by a client to provide web service. A JSP processes the request by using the logic built into it or it will call Java Servlet or Enterprise Javabeans to process the request. Once the request is processed, the JSP responds by sending the results to the client.

A JSP is simpler to create than Java Servlet, because JSP is written in HTML rather than java language. A JSP offers same features as Servlet because JSP is converted to servlet the first time a client requests a JSP.

There are three methods that are automatically called when a JSP is requested and terminated normally. They are jspInit(), jspDestroy() and service() method. The jspInit() method is called first when the JSP is requested and is used to initialize objects and variables that are used through out the life cycle of JSP.

The jspDestroy() method is called when JSP terminates normally. The jspDestroy() method is not called when JSP terminates abruptly due to system crash. The jspDestroy() method is used for clean up process like releasing database connection, closing file etc. The service() method is automatically called and retrieves a connection to HTTP.

JSP Tags

A JSP program consists of a combination of HTML tags and JSP tags. JSP tags define java code that is to be executed before the output of JSP program is sent to the browser. A JSP tag begins with a <%, which is followed by java code, and ends with %>. There is also an Extendable Markup Language(XML) version of JSP tags, which are formatted as <jsp:TagID> </jsp:TagID>.

JSP tags are embedded into the HTML component of a JSP program and are processed by a JSP virtual machine such as Tomcat. Tomcat reads the JSP program whenever a program is called by a browser and resolves JSP tags, then sends the HTML tags and related information to the browser.

Java code associated with JSP tags in the JSP program is executed when encountered by Tomcat, and the result of that process is sent to the browser. The browser knows how to display the result because the JSP tag is enclosed within an open and closed HTML tag.

There are five types of JSP tags which can be used in JSP program. Those are as follows:

1. Comment Tag: A comment tag opens with <%-- and closes with –%>, and is followed by a comment that usually describes the functionality of statements that follow the comment tag.

2. Declaration statement tag: A declaration statement tag opens with <%! and ends with %>. It consists of java declaration statement that define variables, objects and methods that are available to other components of the JSP program.

3. Directive tag: A directive tag opens with <%@ and commands the JSP virtual engine to perform a specific task, such as importing a java package required by objects and methods used in a declaration statement. The directive tag closes with %>. There are three commonly used directives. These are import, include and taglib. The import tag is used to import java packages into JSP program. The include tag inserts a specified file into the JSP program replacing the include tag. The taglib tag specifies a file that contains a tag library. For ex:
<#@ page import= “java.sql.*”; %>
<%@ include file=”C\books.html” %>
<%@ taglib uri=”mytags.tld” %>

4. Expression tag: An expression tag opens with <%= and is used for an expression statement whose result places the expression tag when the JSP virtual engine resolves JSP tags. An expression tag closes with %>.

5. Scriptlet tag: A scriptlet tag opens with <% and contains commonly used java control statements and loops. A scriptlet tag closes with %>.

Variables and Objects

Core Spring

Spring is a framework for development of J2EE applications. It is developed by Rod Johnson in 2003. Spring was developed to address the complexities of Enterprise application. Spring Framework is open source. Recent version of Spring Framework is 4.3.0.

Spring framework is popular because it makes the development of Enterprise applications easy by adapting four key strategies. They are
1. Development of Plain Old Java Objects
2. Achieving loose coupling through Dependency Injection(DI) technique
3. Implementing Aspect Oriented Programming
4. Avoiding boilerplate code by using templates

Development of Plain Old Java Objects
Normally an application contains n number of classes. If we develop Enterprise application using other frameworks like EJB3, struts, JSF etc, then while defining class, framework will force us to implement interfaces or extend classes of framework.

For example if we develop class of an application using EJB3 framework then it is mandatory to implement SessionBean interface.

```
public class HelloWorldBean implements SessionBean {
public void ejbActivate() {
}

public void ejbPassivate() {
}

public void ejbRemove() {
}

public void setSessionContext(SessionContext ctx) {
}

public String sayHello() {
return "Hello World";
}

public void ejbCreate() {
}
}
```

In the above example, the core functionality of a class is sayHello() method. But because of the class implementing SessionBean interface, the class has to implement all the abstract methods present in interface even though it is not required. So it makes the class definition lengthy and comples.

But if we develop class using Spring framework it doesn’t force to implement any interface or extend class specific to framework, which makes the class definition simple. Such classes are called Plain Old Java Objects(POJO).

```
public class HelloWorldBean {
public String sayHello() {
return "Hello World";
}
}
```

Achieving loose coupling through Dependency Injection(DI) technique
An application may consists of n number of classes. One class may depend on other class to achieve a task. If one class is depending on implementation of other class, then we can say both classes are tightly couples with each other. For example

Address.java
```
public class Address {
	private String city;
	private String state;
	private String country;

	public Address(String city, String state, String country) {
	this.city = city;
	this.state = state;
	this.country = country;
	}

	public String toString(){
	return city+" "+state+" "+country;
	}
}
```

Employee.java
```
public class Employee {
	private int id;
	private String name;
	Address addr; = new Address()
```

```
        public Employee() {System.out.println("def cons");}

        public Employee(int id, String name) {
        this.id = id;
        this.name = name;
        Address addr = new Address("Bangalore","Karnataka","India" );
        }

        void show(){
        System.out.println(id+" "+name);
        System.out.println(address.toString());
        }
  }
```

We have defined two classes, where in the Employee class we are creating an object of Address class. That means the class Employee is depending on implementation of Address class. So, these two classes are tightly coupled with each other.

There are some disadvantages if there is tightly coupling between classes, they are
1. It is difficult to test the application.
2. Difficult to understand the application.
3. Difficult to modify the applications

To avoid above problems, Spring Framework uses a technique called Dependancy Injection(DI). The DI will reduce the dependency between classes of an application. If the class is not depending on implementation of another class, then we can say the classes are loosely coupled with each other.

We will redefine Employee class to understand dependency injection technique.
Employee.java:

```
public class Employee {
        private int id;
        private String name;
        private Address address;

        public Employee() {System.out.println("def cons");}

        public Employee(int id, String name, Address address) {
        this.id = id;
        this.name = name;
        this.address = address;
        }

        void show(){
        System.out.println(id+" "+name);
        System.out.println(address.toString());
        }

}
```

In the above example, instead of creating object of Address class in the constructor of Employee class, we are passing it as a parameter to constructor. Through constructor parameter, the Employee class is able to access Address class. The is no direct dependency of Employee class on the implementation of Address class. So we can say both classes are loosely coupled with each other.

To establish relationship between one class to another class, Spring framework uses a third party i.e XML file.

```
<?xml version="1.0" encoding="UTF-8"?>
<beans
   xmlns="http://www.springframework.org/schema/beans"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xmlns:p="http://www.springframework.org/schema/p"
   xsi:schemaLocation="http://www.springframework.org/schema/beans
   http://www.springframework.org/schema/beans/spring-beans-3.0.xsd">

<bean id="a1" class="Address">
        <constructor-arg value="ghaziabad"></constructor-arg>
        <constructor-arg value="UP"></constructor-arg>
        <constructor-arg value="India"></constructor-arg>
</bean>


<bean id="e" class="Employee">
        <constructor-arg value="12" type="int"></constructor-arg>
        <constructor-arg value="Sonoo"></constructor-arg>
        <constructor-arg>
                <ref bean="a1"/>
        </constructor-arg>
</bean>

</beans>
```

<bean> tag is used to declare classes of an application in XML file along with id attribute. <constructor-arg> is used to pass values of data members to constructor of a class. If we consider constructor of Employee class, it contains two normal parameters, one is is and other is name. The third parameter is reference of Address class. So in XML file, for first two parameters value attribute is used to pass value. For third parameter, ref attribute is used to indicate it is a reference of other class with value a1, which is the id of Address class.

To test the above application, we need to write main() function.

```
public class Test {
    public static void main(String[] args) {

        Resource r=new ClassPathResource("ref.xml");

        Employee s=Context.getBean("e");
        s.show();

    }
}
```

ClassPathResource function is used to load XML file onto Spring container. GetBean() function is used to access Employee class. Then we can call any method which is defined in Employee class.

Implementing Aspect Oriented Programming(AOP)

The core functionality of any class is called Aspect or it is the requirement of an application. There will be some functionalities which are implemented across multiple classes of an application, then we will call such functionality as cross cutting concerns. For ex security functionality should be implemented in all classes, because all classes of application should be secure.

If there are cross cutting concerns in an application, then those will pose some problems. They are

1. If cross cutting concern need to modified then all the modules of application should be modified.
2. If core functionality of class need to be modified, then programmer should take care of cross cutting concern which is implemented in class.

To solve above problems, Spring framework separates the implementation cross cutting concerns from core classes of application. Instead cross cutting concerns are implemented in a separate class. This technique is called Aspect Oriented Programming.

So through Aspect oriented programming we can achieve high cohesion that means one class should implement only one functionality of appliaction.

Student.java:
```
public class Student {
  private Integer age;
  private String name;

  public void setAge(Integer age) {
    this.age = age;
  }

  public Integer getAge() {
        System.out.println("Age : " + age );
    return age;
  }

  public void setName(String name) {
    this.name = name;
  }

  public String getName() {
    System.out.println("Name : " + name );
    return name;
  }
}
```

Security1.java:
```
public class Security1 {

  public void SecFun(){
    System.out.println("I will do security checks");
  }
}
```

Student class is a class whose core functionality is maintaining student information. Security1 is the class where cross cutting concern is implemented. We will assume that Security check should be done before inserting new student details to avoid duplicate values. To achieve that Security1 class should be executed first before Student class.

To establish such dependency we will use XML file.
```
<?xml version="1.0" encoding="UTF-8"?>
```

```
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:aop="http://www.springframework.org/schema/aop"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
    http://www.springframework.org/schema/aop
    http://www.springframework.org/schema/aop/spring-aop-3.0.xsd ">


<bean id="SBean" class="Student">   </bean>
<bean id="securityAspect" class="Security1"></bean>

<aop:config>
  <aop:aspect id="myaspect" ref="securityAspect" >
     <!-- @Before -->
     <aop:pointcut id="pointCutBefore"   expression="execution(Student.*(..))" />
     <aop:before method="SecFun" pointcut-ref="pointCutBefore" />
  </aop:aspect>
</aop:config>
</beans>
```

<bean tag is used to declare above classes. <aop:config> is used to configure Aspect oriented programming. <aop:aspect> tag is used to declare Security1 class as crosscutting concern/Aspect. <aop:pointcut> tag is used to indicate Security1 class should be executed at Student class by using expression attribute. <aop:before> tag is used to indicate SecFun() method of Security1 class should be executed before Student class.

Avoiding boilerplate code by using templates

Boilerplate code is the set of instructions which are repeated across multiple classes of an application.

```
public Employee getEmployeeById(long id) {
        Connection conn = null;
        PreparedStatement stmt = null;
        ResultSet rs = null;
        try {
                conn = dataSource.getConnection();
                stmt = conn.prepareStatement("select id, firstname, lastname, salary from "
                 +"employee where id=?");
                rs = stmt.executeQuery();
                Employee employee = null;
```

```
            if (rs.next()) {
                employee = new Employee();
                employee.setId(rs.getLong("id"));
                employee.setFirstName(rs.getString("firstname"));
                employee.setLastName(rs.getString("lastname"));
                employee.setSalary(rs.getBigDecimal("salary"));
            }
            return employee;
        }
        catch (SQLException e) {
        }
        finally {
            if(rs != null) {
                try {
                    rs.close();
                }
                catch(SQLException e) {}
                }
                if(stmt != null) {
                    try {
                        stmt.close();
                    }
                     catch(SQLException e) {}
                    }
                    if(conn != null) {
                        try {
                            conn.close();
                        }
                        catch(SQLException e) {}
                    }
                }
        return null;
}
```

In the above example, to execute simple query like to display employee details having particular id, programmer needs to implement some steps to establish connection between application and a database. If we want to execute queries in multiple classes of an application, then same code will repeated in all classes to establish connection to database.

To avoid such repeated code, Spring framework uses Templates.

```
public Employee getEmployeeById(long id) {
        return jdbcTemplate.queryForObject("select id, firstname, lastname, salary " +
                                        "from employee where id=?",
                                        new RowMapper<Employee>() {
                                                public Employee mapRow(ResultSet rs,
                                                        int rowNum) throws SQLException {
                                                        Employee     employee     =     new
Employee();
                                                        employee.setId(rs.getLong("id"));

        employee.setFirstName(rs.getString("firstname"));

        employee.setLastName(rs.getString("lastname"));

        employee.setSalary(rs.getBigDecimal("salary"));
        return employee;
        }
        },
        id);
}
```

In the above example, it is the responsibility of jdbcTemplate to establish connection between database. As a programmer, it is not required to implement steps to establish connection, which will reduce size of application and makes the application simple.

Containing your beans

The spring is a container based framework. All classes should be loaded to container before executing the application. The container is responsible for creating association between classes.

In spring framework, there are two types of containers. They are

1. Bean Factory
2. Application Contexts

The only difference between above two containers is bean factory is so low lovel for some of the applications. So programmers preferably use application contexts.

Spring comes with several flavors of application context. They are

1. ClassPathXmlApplicationContext — Loads a context definition from an XML file located in the classpath, treating context definition files as classpath resources.
2. FileSystemXmlApplicationContext — Loads a context definition from an XML file in the file system.

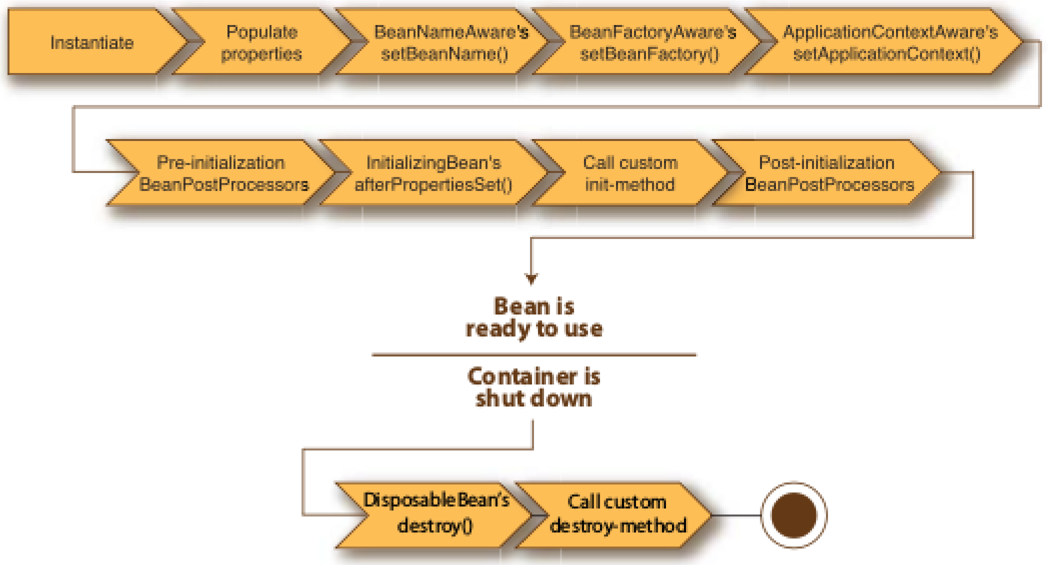
XmlWebApplicationContext — Loads context definitions from an XML file contained within a web application.

To load XML file using class path, we should use
ApplicationContext context = new ClassPathXmlApplicationContext("foo.xml");

To load XML file from file system, we should use
ApplicationContext context = new FileSystemXmlApplicationContext("c:/foo.xml");

In spring framework, a class or an object s called bean or javabean. In spring container the bean's life cycle has multiple stages as given in below diagram.

The stages of bean's

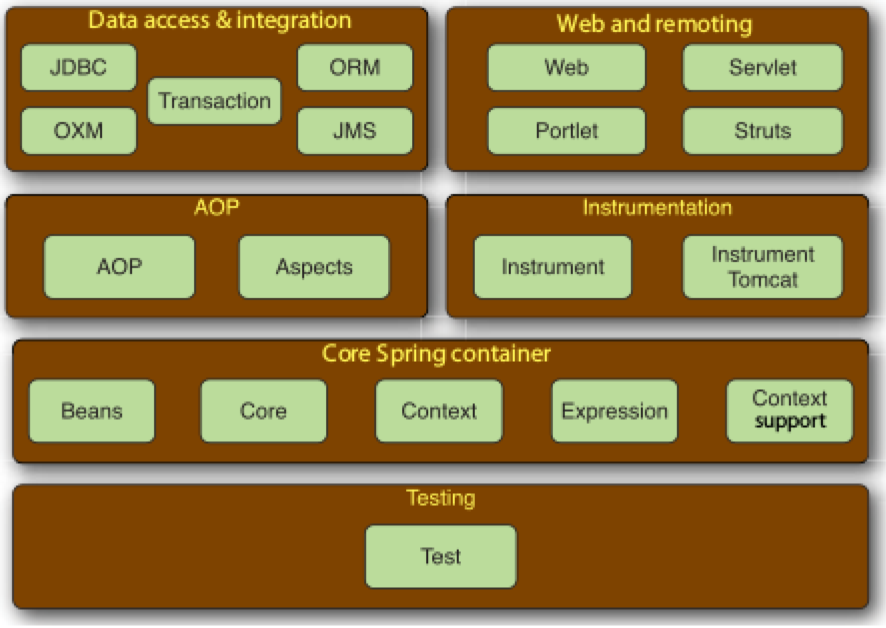

Instantiate
Populate properties
BeanNameAware's setBeanName()
BeanFactoryAware's setBeanFactory()
ApplicationContextAware's setApplicationContext()
Pre-initialization BeanPostProcessors
InitializingBean's afterPropertiesSet()
Call custom init-method
Post-initialization BeanPostProcessors
Bean is ready to use
Container is shut down
DisposableBean's destroy()
Call custom destroy-method

lifecycle is explained below.
1. Spring instantiates the bean.
2. Spring injects values and bean references into the bean's properties.
3. If the bean implements BeanNameAware , Spring passes the bean's ID to the setBeanName() method.
4. If the bean implements BeanFactoryAware , Spring calls the setBeanFactory() method, passing in the bean factory itself.
5. If the bean implements ApplicationContextAware , Spring will call the set-ApplicationContext() method, passing in a reference to the enclosing application context.
6. If any of the beans implement the BeanPostProcessor interface, Spring calls their postProcessBeforeInitialization() method.
7. If any beans implement the InitializingBean interface, Spring calls their afterPropertiesSet() method. Similarly, if the bean was declared with an init-method , then the specified initialization method will be called.
8. If there are any beans that implement BeanPostProcessor , Spring will call their postProcessAfterInitialization() method.
9. At this point, the bean is ready to be used by the application and will remain in the application context until the application context is destroyed.

10. If any beans implement the DisposableBean interface, then Spring will call their destroy() methods. Likewise, if any bean was declared with a destroy-method , then the specified method will be called.

Surveying the bean’s landscape

The Spring Framework is composed of several distinct modules. When we download and unzip the Spring Framework distribution, we can find 20 different JAR files in the dist directory. The 20 JAR files that make up Spring can be arranged in one of six different categories of functionality, as illustrated in figure below.



When taken as a whole, these modules give everything you need to develop enterprise-ready applications. The modules of spring framework are

1. Data access and integration: It consists of templates like jdbctemplate to reduce boilerplate code.
2. Web and Remoting: This provides MVC framework that promotes implementation of Spring’s loosely coupled techniques.
3. AOP: Spring provides rich support for aspect-oriented programming in its AOP module. This module serves as the basis for developing your own aspects for your Spring enabled application.
4. Core spring container: The centerpiece of the Spring Framework is a container that manages how the beans in a Spring-enabled application are created, configured, and managed. Within this module we can find the Spring bean factory, which is the portion of Spring that provides dependency injection. In addition to the bean factory and application context, this module also supplies many enterprise services such as email, JNDI access, EJB integration, and scheduling.

5. Testing: Within this module we can find a collection of mock object implementations for writing unit tests against code that works with JNDI , servlets, and portlets. For integration-level testing, this module provides support for loading a collection of beans in a Spring application context and working with the beans in that context.

The spring portfolio

The whole Spring portfolio includes several frameworks and libraries that build upon the core Spring Framework to give additional functionalities. The frameworks are

1. Spring web flow: It provide support for building conversational, flow-based web applications that guide users toward a goal.
2. Spring web services: Spring Web Services offers a contract-first web services model where service implementations are written to satisfy the service contract.
3. Spring security: It offers a declarative security mechanism for Spring-based applications.
4. Spring Integration: Spring Integration offers implementations of several common integration patterns in Spring's declarative style.
5. Spring batch: It is used for batch processing.
6. Spring social: It is used when application need to be integrated with social network.
7. Spring Mobile: Spring Mobile is a new extension to Spring to support development of mobile web applications.
8. Spring dynamic modules: Spring Dynamic Modules (Spring- DM) blends Spring's declarative dependency injection with OSG i's dynamic modularity.
9. Spring LDAP: Spring LDAP brings Spring-style template-based access to LDAP , eliminating the boilerplate code that's commonly involved in LDAP operations.
10. Spring .NET: Spring. NET offers the same loose-coupling and aspect-oriented features of Spring, but for the . NET platform.

Wiring Beans

In Spring, objects aren’t responsible for finding or creating the other objects that are dependent on. Instead, it is done by spring container using XML file.

Declaring beans

There are two ways to declare beans. They are
1. Using constructors
2. Using Factory methods

First we will see how to declare bean using constructors. To understand bean declaration, will consider the example of organizing a competition virtually. A competition may have n number of performers. So we will define an interface performer.

```
public interface Performer {
void perform() throws PerformanceException;
}
```

There will be a Juggler performer in the competition who will implement the interface.

```
public class Juggler implements Performer {
private int beanBags = 3;
public Juggler() {
}
public Juggler(int beanBags) {
this.beanBags = beanBags;
}
public void perform() throws PerformanceException {
System.out.println("JUGGLING " + beanBags + " BEANBAGS");
}
}
```

The Juggler will be able to play with 3 beanbags by default. We can change default value by using constructor. Now we have defined one class in an application. This class need to be declared in a XML configuaration file. The XML file will be

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-3.0.xsd">
<!-- Bean declarations go here -->
```

```
</beans>
```

Within the <beans> we can place all of your Spring configuration, including <bean> declarations. Spring comes with several XML namespaces through which you can configure the Spring. They are

1. aop: Provides elements for declaring aspects and for automatically proxying @AspectJ-annotated classes as Spring aspects.
2. beans: The core primitive Spring namespace, enabling declaration of beans and how they should be wired.
3. context: Comes with elements for configuring the Spring application context, including the ability to autodetect and autowire beans and injection of objects not directly managed by Spring.
4. jee: Offers integration with Java EE APIs such as JNDI and EJB.
5. JMS: Provides configuration elements for declaring message-driven POJOs.
6. lang: Enables declaration of beans that are implemented as Groovy, JRuby, or BeanShell scripts.
7. MVC: Enables Spring MVC capabilities such as annotation-oriented controllers, view controllers, and interceptors.
8. oxm: Supports configuration of Spring’s object-to-XML mapping facilities.
9. tx: Provides for declarative transaction configuration.

10: util: A miscellaneous selection of utility elements. Includes the ability to declare collections as beans and support for property placeholder elements.

Inside <beans> tag, the classes can be declared using <bean> tag. For ex

```
<bean id="duke" class="Juggler" />
```

The id attribute specifiy name of object and class attribute specify name of class. The above line indicates the spring container that duke object needs to be created which is of type Juggler. While creating duke object, it will call default constructor because we are not passing any parameters while declaration.

The above application can be executed using

```
ApplicationContext ctx = new ClassPathXmlApplicationContext("spring-idol.xml");
Performer performer = (Performer) ctx.getBean("duke");
performer.perform();
```

It will display the output

```
JUGGLING 3 BEANBAGS
```

Declaring bean through parameterized constructors

We can also declare above bean by using parameterized constructor. For ex

```
<bean id="duke" class="Juggler">
```

```
<constructor-arg value="15" />
</bean>
```

The < constructor-arg> tag is used to pass parameter value to a constructor. The Juggler class constructor takes one parameter, so we are passing value 15 using value attribute. Now the Juggler is able to perform with 15 beanbags.

Injecting Object references with constructors
We will define one more class in an application called PoeticJuggler. He is a performer of competition, he can tell poems while playing with the beanbags.

```
public class PoeticJuggler extends Juggler {
private Poem poem;
public PoeticJuggler(Poem poem) {
super();
this.poem = poem;
}
public PoeticJuggler(int beanBags, Poem poem) {
super(beanBags);
this.poem = poem;
}
public void perform() throws PerformanceException {
super.perform();
System.out.println("While reciting...");
poem.recite();
}
}
```

When we consider the above class, the constructor of a class is taking one reference parameter poem. To complete the definition of above class we will create an interface Poem.

```
public interface Poem {
void recite();
}
```

The Sonnet29 is a shakespear’s sonnet which will implement poem interface.

```
public class Sonnet29 implements Poem {
private static String[] LINES = {
"When, in disgrace with fortune and men's eyes,",
"I all alone beweep my outcast state",
"And trouble deaf heaven with my bootless cries",
"And look upon myself and curse my fate,",
"Wishing me like to one more rich in hope,",
"Featured like him, like him with friends possess'd,",
"Desiring this man's art and that man's scope,",
"With what I most enjoy contented least;",
"Yet in these thoughts myself almost despising,",
"Haply I think on thee, and then my state,",
"Like to the lark at break of day arising",
"From sullen earth, sings hymns at heaven's gate;",
"For thy sweet love remember'd such wealth brings",
"That then I scorn to change my state with kings." };
public Sonnet29() {
}
public void recite() {
for (int i = 0; i < LINES.length; i++) {
System.out.println(LINES[i]);
}
}
}
```

Now we have defined two more classes in an application. They are PoeticJuggler and Sonnet29. Before declaring PoeticJuggler class we need to declare Sonnet29. Because PoeticJuggler class is taking reference of Sonnet29 class.
The declaration of Sonnet29 class is

```
<bean id="sonnet" class="Sonnet29" />
```

Now we can declare PoeticJuggler class.

```
<bean id="poeticDuke" class="PoeticJuggler">
<constructor-arg value="15" />
<constructor-arg ref="sonnet" />
</bean>
```

When we look at constructor of PoeticJuggler class, it is taking 2 parameters, one is beanbags which is of type int. For that we are passing value 15 using <constructor-arg> tag along with value attribute. The second parameter of constructor is reference of Poem interface. For that we are passing sonnet using ref attribute indicating that it is reference of other class instead of simple value.
This is how we can pass reference of one class to another class.

Creating beans through factory methods

If a class doesn't have a constructor, then we can declare that class in XML file using factory method.

```
public class Stage {
public static Stage getInstance() {
}
}
```

The above class doesn't have a constructor, so the declaration of above class will be

```
<bean id="theStage" class="Stage" factory-method="getInstance" />
```

The factory-method attribute indicates the class need to be declared using one of the member function of a class instead of constructor.

Bean Scoping

Whenever we need instance of a bean after declaration, always same copy of the instance will get loaded. To get different copies of an instance of bean we can use scope attribute while declaring a bean.

```
<bean id="ticket" class="Ticket" scope="prototype" />
```

By using scope attribute with a value prototype, each time a different copy of ticket can be accessed.

Initializing and destroying beans

When a bean is instantiated, it may be necessary to perform some initialization to get it into a usable state. Likewise, when the bean is no longer needed and is removed from the container, some cleanup may be in order. To accommodate setup and tear-down of beans, Spring provides hooks into the bean lifecycle.

To define setup and teardown for a bean, simply declare the <bean> with init-method and/or destroy-method parameters. The init-method attribute specifies a method that is to be called on the bean immediately upon instantiation. Similarly, destroy-method specifies a method that is called just before a bean is removed from the container.

To illustrate, imagine that we have a Java class called Auditorium which represents the performance hall where the talent competition will take place. Auditorium will likely do a lot of things, but for now let's focus on two things that are important at the beginning and the end of the show: turning on the lights and then turning them back off.

To support these essential activities, the Auditorium class might have turnOn-Lights() and turnOffLights() methods:

```
public class Auditorium {
public void turnOnLights() {
...
}
public void turnOffLights() {
...
}
}
```

Let’s use the init-method and destroy-method attributes when declaring the auditorium bean:

```
<bean id="auditorium"
class="com.springinaction.springidol.Auditorium"
init-method="turnOnLights"
destroy-method="turnOffLights"/>
```

When declared this way, the turnOnLights() method will be called soon after the auditorium bean is instantiated, allowing it the opportunity to light up the performance venue. And, just before the bean is removed from the container and discarded, turnOffLights() will be called to turn the lights off.

Defaulting init-method and destroy-method
If many of the beans in a context definition file will have initialization or destroy methods with the same name, then it is not required to declare init-method or destroy-method on each individual bean. Instead you can take advantage of the default-init-method and default-destroy-method attributes on the <beans> element:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-3.0.xsd"
default-init-method="turnOnLights"
default-destroy-method="turnOffLights">
</beans>
```

The default-init-method attribute sets an initialization method across all beans in a given context definition. Likewise, default-destroy-method sets a common destroy method for all beans in the context definition.

Injecting into bean properties

To demonstrate Spring’s other form of DI that is through factory method/accessor methods, we will define a class called Instrumentalist. He is performer in a competition where he can play song using an instrument.

```
public class Instrumentalist implements Performer {

private String song;

public void setSong(String song) {
this.song = song;
}

public String getSong() {
return song;
}

private Instrument instrument;

public void setInstrument(Instrument instrument) {
this.instrument = instrument;
}

public void perform() throws PerformanceException {
System.out.print("Playing " + song + " : ");
instrument.play();
}

}
```

The above class has 2 data members, one is song which is of type String and other is instrument which is a reference of other class. To complete the definition of above class, we need to create interface called Instrumnet.

```
public interface Instrument {
public void play();
}
```

There is an instrument called Saxophone which will implement Instrument interface.

```
public class Saxophone implements Instrument {
public Saxophone() {
}

public void play() {
System.out.println("TOOT TOOT TOOT");
}
}
```

Now we have defined 2 more classes in an application, they are Saxophone and Instrumentalist. The Saxophone class need to be declared first before declaring Instrumentalist class because Instrumentalist class has a parameter which is reference of Saxophone class. The declaration of Saxophone class will be

```
<bean id="saxophone" class="Saxophone" />
```

The declaration of Instrumentalist class will be

```
<bean id="kenny2" class="Instrumentalist">
<property name="song" value="Jingle Bells" />
<property name="instrument" ref="saxophone" />
</bean>
```

To pass parameters to accessor methods, we will use <property> tag. Name attribute indicates name of parameter and value attribute indicates value of a parameter. Jingle Bells is the value for parameter song. The next parameter will be instrument, which is a reference of another class that is indicated using ref attribute.

While implementing dependency injection, it is advisable to use interfaces. Because while passing reference, implementation of any class of interface can be passed. There is no direct dependency of one class on another class. Instead they are related through interface.

For example if we want to pass piano as a instrument, we need to define piano class.

```
public class Piano implements Instrument {
public Piano() {
}
public void play() {
System.out.println("PLINK PLINK PLINK");
}
}
```

The declaration of piano class will be

```
<bean id="piano" class="Piano" />
```

The declaration of Instrumentalist class can be modified as

```
<bean id="kenny" class="Instrumentalist">
<property name="song" value="Jingle Bells" />
<property name="instrument" ref="piano" />
</bean>
```

Injecting inner beans
If value of any parameter is specific to an object, then such parameter is declared as inner bean. For ex

```
<bean id="kenny" class="Instrumentalist">
<property name="song" value="Jingle Bells" />
<property name="instrument">
<bean class="Saxophone" />
</property>
</bean>
```

The above declaration indicates kenny does not want to share Saxophone with other objects. Saxophone instrument is specific for a kenny object.

Inner bean can be declared while using <constructor-arg> tag also. For ex

```
<bean id="duke" class="PoeticJuggler">
<constructor-arg value="15" />
<constructor-arg>
<bean class="Sonnet29" />
</constructor-arg>
</bean>
```

Wiring properties with Spring’s p namespace
Spring’s p namespace offers a way to wire bean properties that doesn’t require so many angle brackets. The p namespace has a schema URI of http://www.springframework.org/schema/p. To use it, simply add a declaration for it in the Spring XML configuration:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
xmlns:p="http://www.springframework.org/schema/p"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
```

```
xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-3.0.xsd">
```

With it declared, you can now use p: -prefixed attributes of the <bean> element to wire properties.

```
<bean id="kenny" class="Instrumentalist"
p:song = "Jingle Bells"
p:instrument-ref = "saxophone" />
```

The p:song attribute is set to "Jingle Bells" , wiring the song property with that value. Meanwhile, the p:instrument-ref attribute is set to "saxophone" , effectively wiring the instrument property with a reference to the bean whose ID is saxophone. The -ref suffix serves as a clue to Spring that a reference should be wired instead of a literal value.

The advantage of p namespace over ways of declaring bean is it reduces number of angle brackets.

Wiring collections
If a data member of a class is a multi valued parameter means if it takes collection of values, then spring offers four types of collection configuration elements to pass values to such data members. They are

1. <list> : Wiring a list of values, allowing duplicates
2. <set> : Wiring a set of values, ensuring no duplicates
3. <map> : Wiring a collection of name-value pairs where name and value can be of any type
4. <props> : Wiring a collection of name-value pairs where the name and value are both Strings

To understand above four lements, we will define a class called OneManBand. He is a performer in competition, where he can play multiple instruments at a time.

```
import java.util.Collection;
public class OneManBand implements Performer {
public OneManBand() {
}
public void perform() throws PerformanceException {
for (Instrument instrument : instruments) {
instrument.play();
}
}
private Collection<Instrument> instruments;
}
public void setInstruments(Collection<Instrument> instruments) {
this.instruments = instruments;
}
}
```

A OneManBand iterates over a collection of instruments when it performs. The most important here is that the collection of instruments is injected through the setInstruments() method.

Wiring lists, sets, and arrays
To give Hank a collection of instruments to perform with, let’s use the <list> configuration element:

```
<bean id="hank"
class="OneManBand">
<property name="instruments">
<list>
<ref bean="guitar" />
<ref bean="cymbal" />
<ref bean="harmonica" />
</list>
</property>
</bean>
```

The <list> element contains one or more values. Here <ref> elements are used to define the values as references to other beans in the Spring context, configuring Hank to play a guitar, cymbal, and harmonica.

The instruments in OneManBand class can also be declared as
java.util.List<Instrument> instruments;
or
Instrument[] instruments;

Likewise, we can also use <set> to pass values like

```
<bean id="hank"
class="OneManBand">
<property name="instruments">
<set>
<ref bean="guitar" />
```

```
<ref bean="cymbal" />
<ref bean="harmonica" />
<ref bean="harmonica" />
</set>
</property>
</bean>
```

Wiring map collections
To understand map collection we can redefine OneManBand class as

```
public class OneManBand implements Performer {
public OneManBand() {
}

public void perform() throws PerformanceException {
for (String key : instruments.keySet()) {
System.out.print(key + " : ");
Instrument instrument = instruments.get(key);
instrument.play();
}
}
private Map<String, Instrument> instruments;
}
public void setInstruments(Map<String, Instrument> instruments) {
this.instruments = instruments;
}
}
```

In the new version of OneManBand , the instruments property is a java.util.Map where each member has a String as its key and an Instrument as its value. Because a Map 's members are made up of key-value pairs, a simple <list> or <set> configuration element cannot be used.

Instead, the following declaration of the hank bean uses the <map> element to configure the instruments property:

```
<bean id="hank" class="OneManBand">
<property name="instruments">
<map>
<entry key="GUITAR" value-ref="guitar" />
<entry key="CYMBAL" value-ref="cymbal" />
<entry key="HARMONICA" value-ref="harmonica" />
```

```
</map>
</property>
</bean>
```

Each <entry> element defines a member of the Map. The key attribute specifies the key of the entry whereas the value-ref attribute defines the value of the entry as a reference to another bean within the Spring context.

Wiring properties collections

In properties collection both key and value should be of type string

Minimizing XML configuration in Spring

Spring offers two techniques to reduce the amount of XML configuration. They are

1. Autowiring helps reduce or even eliminate the need for <property> and <constructor-arg> elements by letting Spring automatically figure out how to wire bean dependencies.
2. Autodiscovery takes autowiring a step further by letting Spring figure out which classes should be configured as Spring beans, reducing the need for the <bean> element.

Automatically wiring bean properties
The four kinds of autowiring

Spring provides 4 kinds of autowiring. They are

1. byName - Attempts to match all properties of the autowired bean with beans that have the same name (or ID) as the properties. Properties for which there's no matching bean will remain unwired.
2. ByType - Attempts to match all properties of the autowired bean with beans whose types are assignable to the properties. Properties for which there's no matching bean will remain unwired.
3. Constructor - Tries to match up a constructor of the autowired bean with beans whose types are assignable to the constructor arguments.
4. Autodetect - Attempts to apply constructor autowiring first. If that fails, byType will be tried.

Autowiring by name

If the name of a property matches with the name of a bean, then spring will wire that bean into a matched property. For example we have a bean declaration like

```
<bean id="kenny2"  class="Instrumentalist">
<property name="song" value="Jingle Bells" />
<property name="instrument" ref="saxophone" />
</bean>
```

Here we are explicitly initializing song and instrument data members. Assume that we have declared saxophone class with an id instrument like

```
<bean id="instrument" class="Saxophone" />
```

The id of saxophone class and property name is matched, I.e instrument. So the spring will automatically initialize instrument property with saxophone reference. So the declaration of Instrumentalist class can be rewritten as

```
<bean id="kenny" class="Instrumentalist" autowire="byName">
<property name="song" value="Jingle Bells" />
</bean>
```

byName autowiring establishes a convention where a property will automatically be wired with a bean of the same name. In setting the autowire property to byName, you're telling Spring to consider all properties of kenny and look for beans that are declared with the same names as the properties. The disadvantage of using byName autowiring is that it assumes that you'll have a bean whose name is the same as the name of the property of another bean.

Autowiring by type
Autowiring using byType works in a similar way to byName , except that instead of considering a property's name, the property's type is examined. When attempting to autowire a property by type, Spring will look for beans whose type is assignable to the property's type.

For example, suppose that the kenny bean's autowire property is set to byType instead of byName . In the above example, the instrument property is of type Instrument and Saxophone is also of type Instrument. Then the type of instrument property and Saxophone is matching. So the spring will automatically initialize instrument property with saxophone reference.

But there's a limitation to autowiring by type. What happens if Spring finds more than one bean whose type is assignable to the autowired property? In such a case, Spring isn't going to guess which bean to autowire and will instead throw an exception.

To overcome ambiguities with autowiring by type, Spring offers two options: you can either identify a primary candidate for autowiring or you can eliminate beans from autowiring candidacy.

To identify a primary autowire candidate, you'll work with the <bean> element's primary attribute. If only one autowire candidate has the primary attribute set to true , then that bean will be chosen in favor of the other candidates.

But by default for all the beans the primary attribute value is set to true. So we need to set primary attribute of all the beans to false, except the one we need to use for autowiring when there is ambiguity.

For example, to establish that the saxophone bean isn't the primary choice when autowiring Instrument's:
<bean id="saxophone" class="Saxophone" primary="false" />

The primary attribute is only useful for identifying a preferred autowire candidate. If you'd rather eliminate some beans from consideration when autowiring, then you can set their autowire-candidate attribute to false , as follows:
<bean id="saxophone" class="Saxophone" autowire-candidate="false" />

Autowiring constructors
If the bean is configured using constructor injection, then it is not required to use <constructor-arg> attribute. For example, consider the following redeclaration of the duke bean:
<bean id="duke" class="PoeticJuggler" autowire="constructor" />

In this above declaration of duke , the <constructor-arg> attributes are not there and the autowire attribute has been set to constructor . This tells Spring to look at Poetic Juggler's constructors and try to find beans in the Spring configuration to satisfy the arguments of one of the constructors.

Autowiring by constructor shares the same limitations as byType . Spring won’t attempt to guess which bean to autowire when it finds multiple beans that match a constructor’s arguments. Furthermore, if a class has multiple constructors, any of which can be satisfied by autowiring, Spring won’t attempt to guess which constructor to use.

Best-fit autowiring
If you want to autowire the beans, but you can’t decide which type of autowiring to use. You can set the autowire attribute to autodetect to let Spring make the decision for you. For example:

```
<bean id="duke" class="PoeticJuggler" autowire="autodetect" />
```

When a bean has been configured to autowire by autodetect , Spring will attempt to autowire by constructor first. If a suitable constructor-to-bean match can’t be found, then Spring will attempt to autowire by type.

Default autowiring
If we want to use same autowiring technique for every bean of application, then instead of declaring autowiring in all the beans, we can use default-autowire attribute in <beans> tag.

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-3.0.xsd"
default-autowire="byType">
</beans>
```

By default, default-autowire is set to none , indicating that no beans should be autowired unless they’re individually configured for autowiring with the autowire attribute. Here we’ve set it to byType to indicate that we want the properties of every bean to be automatically wired using that style of autowiring. But you can set default-autowire to any of the valid autowiring types to be applied to all beans in a Spring configuration file.

Once we set default-autowire attribute, we can still override autowiring technique by using autowire attribute.

Mixing auto with explicit wiring
Just because you choose to autowire a bean, that doesn’t mean you can’t explicitly wire
some properties. You can still use the <property> element on any property. For example, to explicitly wire Kenny’s instrument property even though he’s set
to autowire by type, use this code:

```
<bean id="kenny" class="Instrumentalist" autowire="byType">
<property name="song" value="Jingle Bells" />
```

```
<property name="instrument" ref="saxophone" />
</bean>
```

As illustrated here, mixing automatic and explicit wiring is also a great way to deal with ambiguous autowiring that might occur when autowiring using byType .

We can use <null/> to force an autowired property to be null . This is a special case of mixing autowiring with explicit wiring. For example, if you wanted to force Kenny’s instrument to be null , you’d use the following configuration:

```
<bean id="kenny" class="Instrumentalist" autowire="byType">
<property name="song" value="Jingle Bells" />
<property name="instrument"><null/></property>
</bean>
```

Wiring with annotations

Autowiring with annotations isn’t much different than using the autowire attribute in XML. Annotation wiring isn’t turned on in the Spring container by default. So, before we can use annotation-based autowiring, we need to enable it in our Spring configuration. The simplest way to do that is with the <context:annotation-config> element from Spring’s context configuration namespace:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:context="http://www.springframework.org/schema/context"
xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
http://www.springframework.org/schema/context
http://www.springframework.org/schema/context/spring-context-3.0.xsd">
<context:annotation-config />
<!-- bean declarations go here -->
</beans>
```

<context:annotation-config> tells Spring that you intend to use annotation-based wiring in Spring.

Spring supports a 3 different annotations for autowiring:

1. @Autowired
2. @Inject
3. @Resource

Using @Autowired

If we want to autowire the instrument parameter using @autowired, then we can write

```
@Autowired
public void setInstrument(Instrument instrument) {
this.instrument = instrument;
}
```

Now it is not required to use <property> tag in XML to assign value to instrument parameter. Spring will do that job.

The @Autowired annotation can even be used on constructors:

```
@Autowired
public Instrumentalist(Instrument instrument) {
this.instrument = instrument;
}
```

When used with constructors, @Autowired indicates that the constructor should be autowired when creating the bean, even if no <constructor-arg> elements are used to configure the bean in XML .

@Autowired can also be used for private data members of class like

```
@Autowired
private Instrument instrument;
```

Actually, there are a couple of circumstances that could keep @Autowired from getting its job done. Specifically, there must be exactly one bean that's applicable for wiring into the @Autowired property or parameter. If there are no applicable beans or if multiple beans could be autowired, then @Autowired will run into some trouble.

Optional autowiring

If no bean can be wired to a @Autowired parameter, then spring will generate exceptions. To avoid that we can configure optional autowiring by setting @Autowired 's required attribute to false . For example:

```
@Autowired(required=false)
private Instrument instrument;
```

Here, Spring will try to wire the instrument property. But if no bean of type Instrument can be found, then no problem. The property will be left null. Note that the required attribute can be used anywhere @Autowired can be used. But when used with constructors, only one constructor can be annotated with @Autowired and required set to true . All other @Autowired -annotated constructors must have required set to false . Moreover, when multiple constructors are annotated with @Autowired , Spring will choose the constructor which has the most arguments that can be satisfied.

Qualifying ambiguous dependencies

If there are more than one bean matched to autowire for a particular parameter, then spring will be in ambuiguity, to decide which bean to wire for a parameter. To resolve such ambiguity we can use Spring's @Qualifier annotation. For example

```
@Autowired
@Qualifier("guitar")
```

```
private Instrument instrument;
```

The above example says if there is any ambiguity in initializing instrument parameter, then spring should assign guitar as a reference(value) to instrument.

Applying standards-based autowiring with @Inject
Instead of using the Spring-specific @Autowired annotation, you might choose to use @Inject on the instrument property:

```
@Inject
private Instrument instrument;
```

Just like @Autowired , @Inject can be used to autowire properties, methods, and constructors. Unlike @Autowired , @Inject doesn't have a required attribute.

Qualifying @injected properties
If there is an ambiguity in autowiring, we can use @Named annotation similar to @Qualifier annotation. The @Named annotation works much like Spring's @Qualifier.

```
@Inject
@Named("guitar")
private Instrument instrument;
```

The @Qualifier helps narrow the selection of matching beans, @Named specifically identifies a selected bean by its ID.

Using expressions with annotation injection
As long as you're using annotations to autowire bean references into your Spring beans, you may want to also use annotations to wire simpler values. Spring introduced @Value , a new wiring annotation that lets you wire primitive values such as int, boolean, and String using annotations.

To use @Value annotation, annotate a property, method, or method parameter with @Value and pass in a String expression to be wired into the property. For example:

```
@Value("Eruption")
private String song;
```

Here we're wiring a String value into a String property. But the String parameter passed into @Value is just an expression—it can evaluate down to any type and thus @Value can be applied to just about any kind of property.

Along with hardcoded values, @Value annotation can also be used with SpEL expressions like

```
@Value("#{systemProperties.myFavoriteSong}")
private String song;
```

Automatically discovering beans

The <context:component-scan> element does everything that <context:annotation-config> does, plus it configures Spring to automatically discover beans and declare them for you. What this means is that most (or all) of the beans in your Spring application can be declared and wired without using <bean>.

To configure Spring for autodiscovery, use <context:component-scan> instead of <context:annotation-config> :

```
<beans xmlns="http://www.springframework.org/schema/beans"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:context="http://www.springframework.org/schema/context"
xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
http://www.springframework.org/schema/context
http://www.springframework.org/schema/context/spring-context-3.0.xsd">
<context:component-scan
base-package="com.springinaction.springidol">
</context:component-scan>
</beans>
```

The <context:component-scan> element works by scanning a package and all of its subpackages, looking for classes that could be automatically registered as beans in the Spring container. The base-package attribute tells <context:component-scan> the package to start its scan from.

Annotating beans for autodiscovery

By default, <context:component-scan> looks for classes that are annotated with one of a handful of special stereotype annotations:

- @Component —A general-purpose stereotype annotation indicating that the class is a Spring component
- @Controller —Indicates that the class defines a Spring MVC controller
- @Repository —Indicates that the class defines a data repository
- @Service —Indicates that the class defines a service

  Any custom annotation that is itself annotated with @Component

For example, suppose that our application context only has the eddie and guitar beans in it. We can eliminate the explicit <bean> declarations from the XML configuration by using <context:component-scan> and annotating the Instrumentalist and Guitar classes with @Component.

First, let’s annotate the Guitar class with @Component :

```
@Component
public class Guitar implements Instrument {
public void play() {
System.out.println("Strum strum strum");
}
}
```

When Spring scans the package which contains Guitar class, it'll find that Guitar is annotated with @Component and will automatically register it in Spring. By default, the bean's ID will be generated by camel-casing the class name. In the case of Guitar that means that the bean ID will be guitar.

```
@Component("eddie")
public class Instrumentalist implements Performer {
// ...
}
```

In this case, we've specified a bean ID as a parameter to @Component .

Filtering component-scans
To enable autodiscovery, we need to annotate every class with @Component annotation. But when we use third party application, we didn't have access to source code. In such cases, we can use component scan to enable auto discovery by using <context:include-filter> and/or <context:exclude-filter> subelements to <context:component-scan> as follows

```
<context:component-scan
base-package="com.springinaction.springidol">
<context:include-filter type="assignable" expression="Instrument"/>
</context:component-scan>
```

The type and the expression attributes of <context:include-filter> work together to define a component-scanning strategy. In this case, we're asking for all classes that are assignable to Instrument to be automatically registered as Spring beans.

Component scanning can be customized using any of five kinds of filters. They are

1. annotation - Filters scan classes looking for those annotated with a given annotation at the type level. The annotation to scan for is specified in the expression attribute.
2. Assignable - Filters scan classes looking for those that are assignable to the type specified in the expression attribute.
3. Aspectj - Filters scan classes looking for those that match the AspectJ type expression specified in the expression attribute.
4. Custom - Uses a custom implementation of org.springframework.core.type.TypeFilter, as specified in the expression attribute.
5. Regex - Filters scan classes looking for those whose class names match the regular expression specified in the expression attribute.

Just as <context:include-filter> can be used to tell <context:component-scan> what it should register as beans, you can use <context:exclude-filter> to tell it what not to register.

Using Spring's Java-based configuration
In spring we have the option of configuring application without using XML, using pure Java.

Setting up for Java-based configuration

Even though Spring's Java configuration option enables you to write most of your Spring configuration without XML , you'll still need a minimal amount of XML to bootstrap the Java configuration:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:context="http://www.springframework.org/schema/context"
xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
http://www.springframework.org/schema/context
http://www.springframework.org/schema/context/spring-context-3.0.xsd">
<context:component-scan

base-package="com.springinaction.springidol" />
</beans>
```

We've already seen how <context:component-scan> automatically registers beans. But it also automatically loads in Java-based configuration classes that are annotated with @Configuration. In this case, the base-package attribute tells Spring to look in com.springinaction.spring-idol to find classes that are annotated with @Configuration.

Defining a configuration class

The Java-based equivalent to the XML <beans> element is @Configuaration. For eample

```
@Configuration
public class SpringIdolConfig {
// Bean declaration methods go here
}
```

The @Configuration annotation serves as a clue to Spring that this class will contain one or more Spring bean declarations. Those bean declarations are just methods that are annotated with @Bean . Let's see how to use @Bean to wire beans using Spring's Java-based configuration.

Declaring a simple bean

The Java-based equivalent to the XML <bean> element is @Bean. For eample

```
@Bean
public Performer duke() {
return new Juggler();
}
```

This simple method is the Java configuration equivalent of the <bean> element we created earlier. The @Bean tells Spring that this method will return an object that should be registered as a bean in the Spring application context. The bean will get its ID from the method name. Everything that happens in the method ultimately leads to the creation of the bean.

In this case, the bean declaration is simple. The method creates and returns an instance of Juggler . That object will be registered in the Spring application context with an ID of duke.

Injecting with Spring’s Java-based configuration

Earlier, we saw how
to create a Juggler bean that juggles 15 beanbags by using the <constructor-arg> element in XML configuration. In the Java-based configuration, we can just pass the number directly into the constructor:

```
@Bean
public Performer duke15() {
return new Juggler(15);
}
```

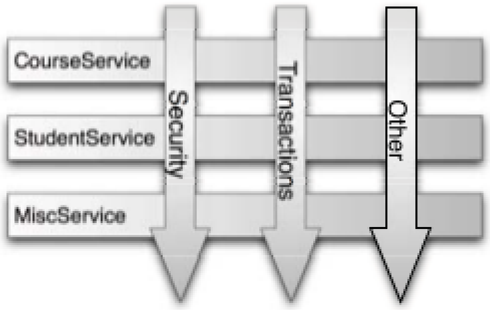
To pass reference of one class to other class, we can pass reference as parameter to constructor of class. For example

```
@Bean
public Performer poeticDuke() {
return new PoeticJuggler(sonnet29());
}
```

Aspect-oriented Spring

In software development, functions that span multiple points of an application are called cross-cutting concerns. Typically, these cross-cutting concerns are conceptually separate from (but often embedded directly within) the application's business logic. Separating these cross-cutting concerns from the business logic is where aspect-oriented programming (AOP) goes to work.

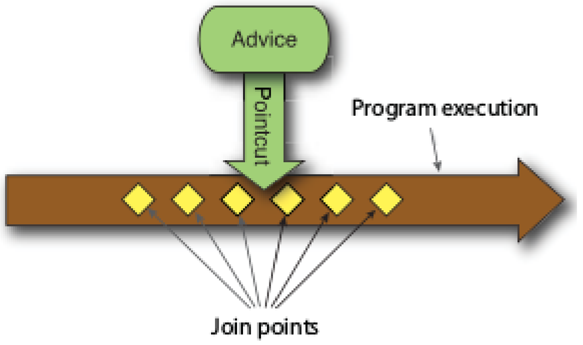
What's aspect-oriented programming?
A cross-cutting concern can be described as any functionality that affects multiple points of an application. Security, for example, is a cross-cutting concern, in that many methods in an application can have security rules applied to them. The figure given below gives a visual depiction of cross-cutting concerns.



The above figure represents a typical application that's broken down into modules. Each module's main concern is to provide services for its particular domain. But each module also requires similar ancillary functionalities, such as security and transaction management.

Cross-cutting concerns can be modularized into special classes called aspects. This has two benefits. First, the logic for each concern is now in one place, as opposed to being scattered all over the code base. Second, our service modules are now cleaner since they only contain code for their primary concern (or core functionality) and secondary concerns have been moved to aspects.

Defining AOP terminology
ADVICE
In AOP terms, the job of an aspect is called advice.



Advice defines both the what and the when of an aspect. In addition to describing the job that an aspect will perform, advice addresses the question of when to perform the job. Should it be applied before a method is invoked? After the method is invoked? Both before and after method invocation? Or should it only be applied if a method throws an exception?
Spring aspects can work with five kinds of advice:

1. Before - The advice functionality takes place before the advised method is invoked.
2. After - The advice functionality takes place after the advised method completes, regardless of the outcome.
3. After-returning - The advice functionality takes place after the advised method successfully completes.
4. After-throwing - The advice functionality takes place after the advised method throws an exception.
5. Around - The advice wraps the advised method, providing some functionality before and after the advised method is invoked.

JOIN POINTS
The application may have thousands of opportunities for advice to be applied. These opportunities are known as join points. A join point is a point in the execution of the application where an aspect can be plugged in. This point could be a method being called, an exception being thrown, or even a field being modified. These are the points where your aspect's code can be inserted into the normal flow of your application to add new behavior.

POINTCUTS
An aspect doesn't necessarily advise all join points in an application. Pointcuts help narrow down the join points advised by an aspect. If advice defines the what and when of aspects, then pointcuts define the where. A pointcut definition matches one or more join points at which advice should be woven.

ASPECTS
An aspect is the merger of advice and pointcuts. Taken together, advice and point-cuts define everything there is to know about an aspect—what it does and where and when it does it.

INTRODUCTIONS
An introduction allows you to add new methods or attributes to existing classes. The new method and instance variable can then be introduced to existing classes without having to change them, giving them new behavior and state.

WEAVING
Weaving is the process of applying aspects to a target object to create a new proxied object. The aspects are woven into the target object at the specified join points. The weaving can take place at several points in the target object's lifetime:

1. Compile time - Aspects are woven in when the target class is compiled. This requires a special compiler.

2. Classload time - Aspects are woven in when the target class is loaded into the JVM. This requires a special ClassLoader that enhances that target class's byte-code before the class is introduced into the application.
3. Runtime - Aspects are woven in sometime during the execution of the application. Typically, an AOP container will dynamically generate a proxy object that will delegate to the target object while weaving in the aspects. This is how Spring AOP aspects are woven.

Spring's AOP support
Not all AOP frameworks are created equal. They may differ in how rich their joinpoint models are. Some allow you to apply advice at the field modification level, whereas others only expose the join points related to method invocations. They may also differ in how and when they weave the aspects. Whatever the case, the ability to create pointcuts that define the join points at which aspects should be woven is what makes it an AOP framework.

Spring's support for AOP comes in four flavors:

1. Classic Spring proxy-based AOP
2. @AspectJ annotation-driven aspects
3. Pure- POJO aspects
4. Injected AspectJ aspects

Spring advice is written in java
All of the advice you create within Spring is written in a standard Java class. That way, you get the benefit of developing your aspects in the same integrated development environment (IDE) you'd use for your normal Java development.

Spring advises objects at runtime
In Spring, aspects are woven into Spring-managed beans at runtime by wrapping them with a proxy class. Between the time when the proxy intercepts the method call and the time when it invokes the target bean's method, the proxy performs the aspect logic.

Spring only supports method join points
Multiple join point models are available through various AOP implementations. Because it's based on dynamic proxies, Spring only supports method join points. This is in contrast to some other AOP frameworks, such as AspectJ and JBoss, which provide field and constructor join points in addition to method pointcuts.

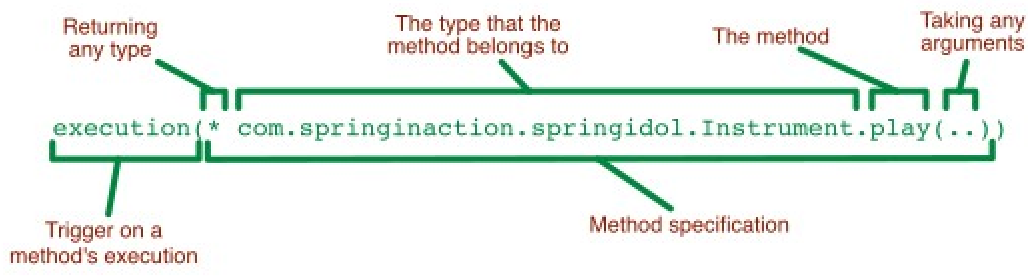
Selecting join points with pointcuts
pointcuts are used to pinpoint where an aspect's advice should be applied. Along with an aspect's advice, pointcuts are among the most fundamental elements of an aspect. Therefore, it's important to know how to write pointcuts. In Spring AOP , pointcuts are defined using AspectJ's pointcut expression language. The list of AspectJ pointcut designators that are supported in Spring AOP are:

- args() - Limits join point matches to the execution of methods whose arguments are instances of the given types.

- @args() - Limits join point matches to the execution of methods whose arguments are annotated with the given annotation types.
- Execution() - Matches join points that are method executions
- this() - Limits join point matches to those where the bean reference of the AOP proxy is of a given type
- target() - Limits join point matches to those where the target object is of a given type.
- @target() - Limits matching to join points where the class of the executing object has an annotation of the given type.
- Within() - Limits matching to join points within certain types
- @within() - Limits matching to join points within types that have the given annotation.
- @annotation - Limits join point matches to those where the subject of the join point has the given annotation

Writing pointcuts

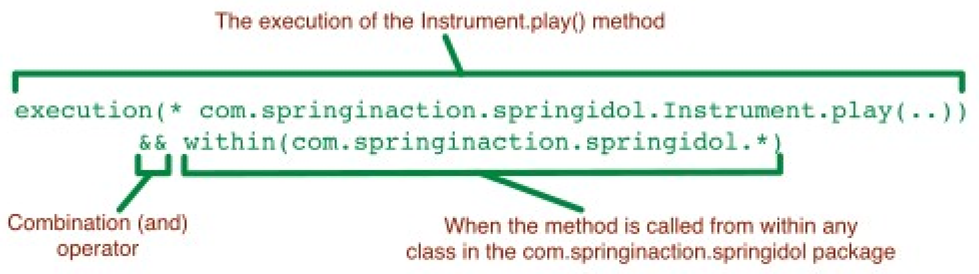
For example, the pointcut expression shown in figure can be used to apply advice whenever an Instrument 's play() method is executed:



```
execution(* com.springinaction.springidol.Instrument.play(..))
```

The execution() designator is used to select the Instrument 's play() method. The method specification starts with an asterisk, which indicates that we don't care what type the method returns. Then we specify the fully qualified class name and the name of the method we want to select. For the method's parameter list, we use the double- dot (..), indicating that the pointcut should select any play() method, no matter what the argument list is.

For example if we want to restrict the pointcut to single package like com.springinaction.springidol, then we can specify it as



```
execution(* com.springinaction.springidol.Instrument.play(..))
    && within(com.springinaction.springidol.*)
```

The execution of the Instrument.play() method
Combination (and) operator
When the method is called from within any class in the com.springinaction.springidol package

In this case, we could limit the match by tacking on a within() designator. Note that we used the && operator to combine the execution() and within() designators in an “and” relationship (where both designators must match for the point-cut to match). Similarly, we could’ve used the || operator to indicate an “or” relationship. And the ! operator can be used to negate the effect of a designator.

Using Spring’s bean() designator

In addition to the designators listed above, introduced a new bean() designator that lets you identify beans by their ID within a pointcut expression. bean() takes a bean ID or name as an argument and limits the pointcut’s effect to that specific bean.

For example, consider the following pointcut:
execution(* com.springinaction.springidol.Instrument.play())
and bean(eddie)

Here we’re saying that we want to apply aspect advice to the execution of an Instrument s play() method, but limited to the bean whose ID is eddie.

Narrowing a pointcut to a specific bean may be valuable in some cases, but we can also use negation to apply an aspect to all beans that don’t have a specific ID :
execution(* com.springinaction.springidol.Instrument.play())
and !bean(eddie)

In this case, the aspect’s advice will be woven into all beans whose ID isn’t eddie.

Declaring aspects in XML

The Spring development team come out with Spring’s aop configuration namespace. The AOP configuration elements are

- <aop:advisor> - Defines an AOP advisor.
- <aop:after> - Defines an AOP after advice
- <aop:after-returning> - Defines an AOP after-returning advice.
- <aop:after-throwing> - Defines an AOP after-throwing advice.
- <aop:around> - Defines an AOP around advice.
- <aop:aspect> - Defines an aspect.
- <aop:aspectj-autoproxy> - annotation-driven aspects using @AspectJ.
- <aop:before> - Defines an AOP before advice.
- <aop:config> - The top-level AOP element. Most <aop:*> elements must be contained within <aop:config>.
- <aop:declare-parents> - Introduces additional interfaces to advised objects that are transparently implemented.
- <aop:pointcut> - Defines a pointcut.

To illustrate Spring AOP , let’s create an Audience class for our talent show. The following class defines the functions of an audience.

```
public class Audience {
public void takeSeats() {
System.out.println("The audience is taking their seats.");
}
public void turnOffCellPhones() {
System.out.println("The audience is turning off their cellphones");
}
public void applaud() {
System.out.println("CLAP CLAP CLAP CLAP CLAP");
}
public void demandRefund() {
System.out.println("Boo! We want our money back!");
}
}
```

As you can see, there's nothing remarkable about the Audience class. It's a basic Java class with a handful of methods. And we can register it as a bean in the Spring application context like any other class:

```
<bean id="audience"
class="com.springinaction.springidol.Audience" />
```

Declaring before and after advice

Using Spring's AOP configuration elements, we can convert the audience bean into an aspect.

```
<aop:config>
<aop:aspect ref="audience">
<aop:before pointcut= "execution(* com.springinaction.springidol.Performer.perform(..))"
method="takeSeats" />
<aop:before pointcut= "execution(* com.springinaction.springidol.Performer.perform(..))"
method="turnOffCellPhones" />
<aop:after-returning pointcut=
"execution(* com.springinaction.springidol.Performer.perform(..))"
method="applaud" />
<aop:after-throwing pointcut=
"execution(* com.springinaction.springidol.Performer.perform(..))"
method="demandRefund" />
</aop:aspect>
</aop:config>
```

Within <aop:config> you may declare one or more advisors, aspects, or pointcuts. In the above example we have declared a single aspect using the <aop:aspect> element. The ref attribute references the POJO bean that will be used to supply the functionality of the aspect - in this case, audience . The bean that's referenced by the ref attribute will supply the methods called by any advice in the aspect.

The aspect has four different bits of advice. The two <aop:before> elements define method before advice that will call the takeSeats() and turnOffCellPhones() methods of the Audience bean before any methods matching the pointcut are executed. The <aop:after-returning> element defines an after-returning advice to call the applaud() method after the pointcut. Meanwhile, the <aop:after-throwing> element defines an after-throwing advice to call the demand-Refund() method if any exceptions are thrown. Figure 4.6 shows how the advice logic is woven into the business logic.

In the above example, the value of the pointcut attribute is the same for all of the advice elements. That's because all of the advice is being applied to the same pointcut. To avoid duplication of the pointcut definition, you may choose to define a named pointcut using the <aop:pointcut> element. The following XML shows how the <aop:pointcut> element is used within the <aop:aspect> element to define a named pointcut that can be used by all of the advice elements.

```
<aop:config>
<aop:aspect ref="audience">
<aop:pointcut id="performance" expression=
"execution(* com.springinaction.springidol.Performer.perform(..))" />
<aop:before pointcut-ref="performance" method="takeSeats" />
<aop:before pointcut-ref="performance" method="turnOffCellPhones" />
<aop:after-returning pointcut-ref="performance" method="applaud" />
<aop:after-throwing pointcut-ref="performance" method="demandRefund" />
</aop:aspect>
</aop:config>
```

Now the pointcut is defined in a single location and is referenced across multiple advice elements. The <aop:pointcut> element defines the pointcut to have an id of performance . Meanwhile, all of the advice elements have been changed to reference the named pointcut with the pointcut-ref attribute.

Declaring around advice

Similar to before and after configuaration elements, we can also use around configuration element. The combination of before and after configuration elements is around configuration element. For example applaud() method need to be executed before the performance starts and at the end when the performance completes. So instead of using both before and after configuration elements to accomplish this scenario, we can use around configuration element as follows

```
<aop:config>
<aop:aspect ref="audience">
<aop:pointcut id="performance" expression=
"execution(* com.springinaction.springidol.Performer.perform(..))" />
<aop:around pointcut-ref="performance" method="applaud" />
</aop:aspect>
</aop:config>
```

Passing parameters to advice

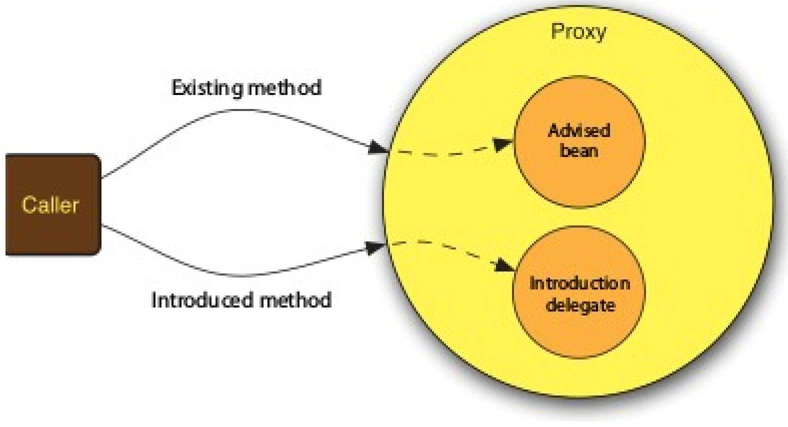
There are times when it may be useful for advice to not only wrap a method, but also inspect the values of the parameters passed to that method. For example perform method may take one parameter which indicates the the id(integer) of particular performance. Using XML we can also pass parameters to advices like

```
<aop:config>
<aop:aspect ref="audience">
<aop:pointcut id="performance" expression=
"execution(* com.springinaction.springidol.Performer.perform(int))"
and args(id)" />

<aop:around pointcut-ref="performance" method="applaud"  arg-names="id"/>
</aop:aspect>
</aop:config>
```

Introducing new functionality with aspects

If an aspect can wrap existing methods with additional functionality, why not add new methods to the object? In fact, using an AOP concept known as introduction, aspects can attach all new methods to Spring beans.



To implement AOP introductions, must use the <aop:declare-parents> element:

```
<aop:aspect>
<aop:declare-parents
types-matching="com.springinaction.springidol.Performer+"
implement-interface="com.springinaction.springidol.Contestant"
default-impl="com.springinaction.springidol.GraciousContestant"
/>
</aop:aspect>
```

As its name implies, <aop:declare-parents> declares that the beans it advises will have new parents in its object hierarchy. Specifically, in this case we're saying that the beans whose type matches the Performer interface should have Contestant in their parentage.

There are two ways to identify the implementation of the introduced interface. In this case, we're using the default-impl attribute to explicitly identify the implementation by its fully-qualified class name. Alternatively, we could've identified it using the delegate-ref attribute:

```
<aop:declare-parents
types-matching="com.springinaction.springidol.Performer+"
implement-interface="com.springinaction.springidol.Contestant"
delegate-ref="contestantDelegate"
/>
```

The delegate-ref attribute refers to a Spring bean as the introduction delegate.

Annotating aspects
With @AspectJ annotations, we can turn the Audience class into an aspect without the need for any additional classes or bean declarations.

```
@Aspect
public class Audience {
@Pointcut(
"execution(* com.springinaction.springidol.Performer.perform(..))")
public void performance() {
}
pointcut
@Before("performance()")
public void takeSeats() {
System.out.println("The audience is taking their seats.");
}
@Before("performance()")
public void turnOffCellPhones() {
System.out.println("The audience is turning off their cellphones");
}
@AfterReturning("performance()")
public void applaud() {
System.out.println("CLAP CLAP CLAP CLAP CLAP");
}
@AfterThrowing("performance()")
public void demandRefund() {
System.out.println("Boo! We want our money back!");
}
}
```

The @Pointcut annotation is used to define a reusable pointcut within an

@AspectJ aspect. The value given to the @Pointcut annotation is an AspectJ pointcut expression—here indicating that the pointcut should match the perform() method of a Performer . The name of the pointcut is derived from the name of the method to which the annotation is applied. Therefore, the name of this pointcut is performance().

Each of the audience's methods has been annotated with advice annotations. The @Before annotation has been applied to both takeSeats() and turnOffCellPhones() to indicate that these two methods are before advice. The @AfterReturning annotation indicates that the applaud() method is an after-returning advice method. And the @AfterThrowing annotation is placed on demandRefund() so that it'll be called if any exceptions are thrown during the performance. The name of the performance() pointcut is given as the value parameter to all of the advice annotations. This tells each advice method where it should be applied.

Because the Audience class contains everything that's needed to define its own point-cuts and advice, there's no more need for pointcut and advice declarations in the XML configuration.

Annotating around advice

Just as with Spring's XML -based AOP , you're not limited to before and after advice types when using @AspectJ annotations. You may also choose to create around advice. For that, you must use the @Around annotation, as in the following example:

```
@Around("performance()")
public void applaud() {
System.out.println("CLAP CLAP CLAP CLAP CLAP");
}
```

Here the @Around annotation indicates that the applaud() method is to be applied as around advice to the performance() pointcut.

Passing arguments to annotated advice

Supplying parameters to advice using @AspectJ annotation isn't much different than
how we did it with Spring's XML -based aspect declaration.

```
@Aspect
public class Musician implements performer {
private int id;

@Pointcut("execution(* com.springinaction.springidol.Performer.perform(int))"
and args(id)")
public void p1(int id) {
}
@Before("p1(id)")
public void perform(int id) {
System.out.println("playing music");
}
}
```

The <aop:pointcut> element has become the @Pointcut annotation and the <aop:before> element has become the @Before annotation. The only significant change here is that @AspectJ can lean on Java syntax to determine the details of the parameters passed into the advice. Therefore, there's no need for an annotation-based equivalent to the <aop:before> element's arg-names.

Annotating introductions

The annotation equivalent of <aop:declare-parents> is @AspectJ's @Declare-Parents . @DeclareParents works almost exactly like its XML counterpart when used inside of an @Aspect -annotated class.

```
@Aspect
public class ContestantIntroducer {
@DeclareParents(
value = "com.springinaction.springidol.Performer+",
defaultImpl = GraciousContestant.class)
public static Contestant contestant;
}
```

As you can see, ContestantIntroducer is an aspect. But unlike the aspects we've created so far, it doesn't provide before, after, or around advice. Instead, it introduces the Contestant interface onto Performer beans. Like <aop:declare-parents>, @DeclareParents annotation is made up of three parts:

- The value attribute is equivalent to <aop:declare-parents> 's types-matching attribute. It identifies the kinds of beans that should be introduced with the interface.
- The defaultImpl attribute is equivalent to <aop:declare-parents> 's default-impl attribute. It identifies the class that will provide the implementation for the introduction.
- The static property that is annotated by @DeclareParents specifies the interface that is to be introduced.

Injecting AspectJ aspects

Although Spring AOP is sufficient for many applications of aspects, it's a weak AOP solution when contrasted with AspectJ. AspectJ offers many types of pointcuts that aren't possible with Spring AOP.

For the most part, AspectJ aspects are independent of Spring. Although they can be woven into any Java-based application, including Spring applications, there's little involvement on Spring's part in applying AspectJ aspects.

But any well-designed and meaningful aspect will likely depend on other classes to assist in its work. If an aspect depends on one or more classes when executing its advice, you can instantiate those collaborating objects with the aspect itself. Or, better yet, you can use Spring's dependency injection to inject beans into AspectJ aspects.

To illustrate, let's create a new aspect for the Spring Idol competition. A talent competition needs a judge. So, let's create a judge aspect in AspectJ. JudgeAspect is such an aspect.

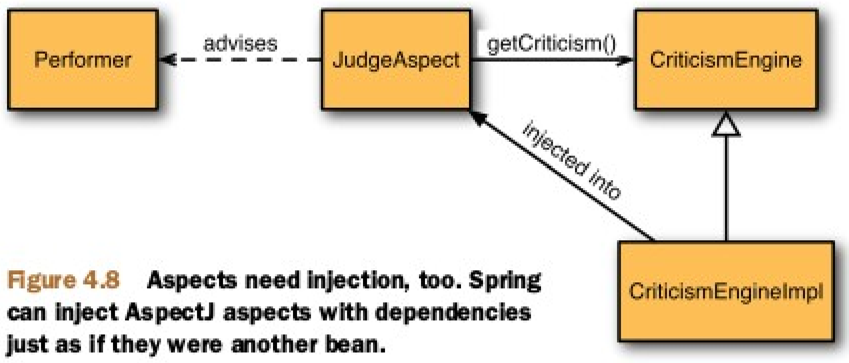


Figure 4.8 Aspects need injection, too. Spring can inject AspectJ aspects with dependencies just as if they were another bean.

```
public aspect JudgeAspect {
public JudgeAspect() {}
pointcut performance() : execution(* perform(..));
after() returning() : performance() {
System.out.println(criticismEngine.getCriticism());
}
// injected
private CriticismEngine criticismEngine;
public void setCriticismEngine(CriticismEngine criticismEngine) {
this.criticismEngine = criticismEngine;
}
}
```

The chief responsibility for JudgeAspect is to make commentary on a performance after the performance has completed. the judge doesn’t make commentary on its own. Instead, JudgeAspect collaborates with a CriticismEngine object, calling its getCriticism() method, to produce critical commentary after a performance. To avoid unnecessary coupling between JudgeAspect and the CriticismEngine, the JudgeAspect is given a reference to a CriticismEngine through setter injection.

CriticismEngine itself is an interface that declares a simple getCriticism() method. Here’s the implementation of CriticismEngine .

```
public class CriticismEngineImpl implements CriticismEngine {
public CriticismEngineImpl() {}
public String getCriticism() {
int i = (int) (Math.random() * criticismPool.length);
return criticismPool[i];
}
// injected
private String[] criticismPool;
public void setCriticismPool(String[] criticismPool) {
```

```
this.criticismPool = criticismPool;
}
}
```

CriticismEngineImpl implements the CriticismEngine interface by randomly choosing a critical comment from a pool of injected criticisms. This class can be declared as a Spring <bean> using the following XML:

```
<bean id="criticismEngine"
class="com.springinaction.springidol.CriticismEngineImpl">
<property name="criticisms">
<list>
<value>I'm not being rude, but that was appalling.</value>
<value>You may be the least talented
person in this show.</value>
<value>Do everyone a favor and keep your day job.</value>
</list>
</property>
</bean>
```

All that’s left is to wire CriticismEngineImpl into JudgeAspect . The following <bean> declaration injects the criticismEngine bean into JudgeAspect :

```
<bean class="com.springinaction.springidol.JudgeAspect"
factory-method="aspectOf">
<property name="criticismEngine" ref="criticismEngine" />
</bean>
```

Advance Java Programming
Unit 5 notes
Java Mail API

Web services built using J2EE components fulfill a variety of services based on the needs of a J2EE application. One of those needs might be to send and recieve email messages. The J2EE application probably sends the request along with an email message and recipient to a JSP or servlet. It is the responsibility of JSP or servlet to create and generate those email messages.

Likewise, a JSP or servlet might be required to automatically retrive email messages and based on business rules, those email messages are forwarded to a client or responded automatically by JAP or servlet.

JavaMail
A J2EE application is able to send and recieve email messages through the use of the JavaMail API. The JavaMail API is protocol independent and can send messages created by a J2EE application via email using existing email protocols. The JavaMail API can also receive email messages and make those messages available to a J2EE application for future processing.

A developer can create a Mail User Agent(MUA) as part of a J2EE application. A MUA is a program that enables a person or component to compose, send and receive email messages. A Mail Transfer Agent(MTA) uses one of several email protocols to transport email messages.

JavaMail API and Java Activation Framework
Java Activation Framework(JAF) contains services to determine the type of data that is associated with an email. This enables the program to call the appropriate EJB to process the data.

Protocols

In email, protocols provide a standard way to format an email message. There are four protocols used for emails. These are Multipurpose Internet Mail Extensions(MIME), Simple Mail Transfer Protocol(SMTP), Post Office Protocol(POP) and Internet Message Access Protocol(IMAP).

MIME is the protocol used to send multipart emails where each part of the multipart email defines its own formatting such as an email message that contains an attachment. Similarly the secure Multipurpose Internet Mail Extensions(S/MIME) protocol is used to encode multipart email messages for secure transmission.

The SMTP is the protocolused to deliver email messages. The MUA sends an email message to an SMTP server that is provided by an Internet Service Provider(ISP). The SMTP server is responsible for forwarding the message to the reciepient’s SMTP server.

The reciepient’s MUA retrives the email message from the reciepient’s SMTP server and then processes the message bymaking the message available to the reciepient the next time the email account is checked.

POP3 is store and forward service where email messages are stored on the mail server until the client logs in to recieve them. The email messages are then downloaded to the client and deleted from server. IMAP functions similar to a remote file server where a client views email on the server. Deleting a local copy of the email message does not delete the email message from the server.

Exceptions

There are 11 exceptions need to be implemented in JavaMail. Among them three of these exceptions are related to the folder that is used to store the email message. They are ReadOnlyFolderException, FolderClosedException and FolderNotFoundException. The ReadOnlyFolderException is thrown when the program attempts to open a folder for read-write access, but the folder is marked for read only. The FolderClosedException is thrown whenever the messaging object and the folder that owns the messaging object are terminated and the program attempts to open the folder. The FolderNotFoundException is thrown whenever method is called to interact with a folder that does not exist.

Next is general exceptions. They are MessagingException, AuthenticationFailedException and MethodNotSupportedException. The MessagingException is the base class for all messaging exceptions. The AuthenticationFailedException is thrown whenever an authentication fails. The MethodNotSupportedException occurs whenever a method that is invoked by the program is not supported by the implementation.

The other category of exceptions are SendFailedException, IllegalWriteException and NoSuchProviderException. The SendFailedException happens whenever the email message is unable to be set. The IllegalWriteException is thrown whenever the program attempts to write a read only messaging object. The NoSuchProviderException is thrown when attempts are made to instantiate a provider that does not exist.

The final two exceptions happens during storage. They are MessageRemovedException and StoreClosedException. The MessageRemovedException is thrown whenever an invalid method is called. The StoreClosedException occurs when the store that owns a messaging object is terminated.

Send Email Message

A J2EE component that provides email service to a client can send an email by using the JavaMail API. There are three steps required to send an email. They are obtain a session, create an email and send the email.

In the example given below, three string objects are declared. These are host, which is assigned the SMTP host. From – which is assigned the sender’s email address and to is assigned to the email address of the recipient.

The getProperties() is called to return the system properties. The mail.smtp.host property is then changed to the new smtp host using the put() method. Once the smtp host is set, the program must open a session using the getDefaultInstance() method.

The email message is formatted using MIME. Therefore, the program must create a new MimeMessage object, which is used to form the email. The from and to string objects are then converted to InternetAddtess objects, which are then passes as arguments to the setFrom() method and the addRecipient() method respectively. These set the internet address for the email. The setSubject() method is passed the subject line of the email and the body of the email message is passed to the setText() method. The final step is to pass the MimeMessage object to the send() method of Transport object, which transmits the email.

```
import java.util.Properties;
import javax.mail.*;
import javax.mail.internet.*;
public class sendMail{
	public static void main(String args[]) throws Exception{
		String host = "smtp.mydomain.com";
		String from = "me@myweb.com";
		String to = "you@myweb.com";
		Properties prop = System.getProperties();
		prop.put("mail.smtp.host", host);
		Session ses1 = Session.getDefaultInstance(prop, null);
		MimeMessage msg = new MimeMessage(ses1);
		msg.setFrom(new InternetAddress(from));
		msg.addRecipient(Message.RecipientType.TO, new InternetAddress(to));
		msg.setSubject("Urgent Message");
		msg.setText("You won the lottery");
		Transport.send(msg);
	}
}
```

Retrieving Email Messages
A J2EE component retrieves an email message by logging in to the SMTP host and then requesting reference to the INBOX folder that contains incoming email messages.

In the example given below, the smtp host, username and password are assigned to string objects. Next, the program creates an instance of the Property object. The property object is passed to the getDefaultInstance() method of the session object to create a new session.

Once logged in, the program retrieves reference to the INBOX folder using the getFolder() method. The INBOX folder is then opened for read only. The program creates an instance of the BufferedReaderobject called bReader, which is an input stream.

An array of messages called msg is then retrieved from the folder using the getMessage() method. Then, the program steps tohrough each message displaying the sender's name and subject line of each email message. These are retrieved using the getFrom() and getSubject() methods respectively.

In this example, the sender's name and subject line of each email message are displayed and the client is asked if he or she wants to read the email message. The response is assigned to the line1 string object and is evaluated by the if statement. The body of the email is retrieved and displayed if the response is yes.

```
import java.io.*;
import java.util.Properties;
import javax.mail.*;
import javax.mail.internet.*;
public class retrieveEmail {
  public static void main (String args[]) throws Exception {
   String host = "smtp.mydomain.com";
   String username = "userName";
   String password = "password";
   Properties prop = new Properties();
   Session ses1 = Session.getDefaultInstance(prop, null);
   Store store1 = ses1.getStore("pop3");
   store1.connect(host, username, password);
   Folder folder1 = store1.getFolder("INBOX");
   folder1.open(Folder.READ_ONLY);
   BufferedReader bReader = new BufferedReader (new InputStreamReader(System.in));
   Message msg[] = folder1.getMessages();
   for (int i=0, n= msg.length; i<n; i++)
   {
    System.out.println(i + ": " + msg [i].getFrom()[0] "\t"
        + msg [i].getSubject());
    System.out.println("Do you want to read message? [Y/N]");
    String line1 = bReader.readLine();
    if ("Y".equals(line1))
    {
     System.out.println(msg [i].getContent());
    }
    else if ("N".equals(line1))
    {
     break;
    }
   }
   folder1.close(false);
   store1.close();
  }
}
```

Deleting Email Messages
A J2EE component requests that an email message be deleted by setting the delete flag that is associated with the message to true. Each email message has a variety of flags that are used to indicate the status of the email message. The commonly used flags are answered, draft, seen and delete. Some flags are unique to a particular system and users define others. The JavaMail API has commonly used flags predefined in the Flags.Flag inner class.

In the given example, the Folder object called folder1 is opened in READ_WRITE mode. This is necessary because after the email message sender and subject are read, the program might change the setting of the delete flag, which is written to the folder.

Next, the program retrieves and display the sender and subject of each email message and prompts the user to delete the email message. If the client selects Yes, then DELETE flag is set to true by calling the setFlag() method and passing it the name of the flag and the setting. Passing a false value to the setFlag() method unsets a flag. The deleteStatus variable is passed to the close() method, which causes the email message to be deleted.

```
import java.io.*;
import java.util.Properties;
import javax.mail.*;
import javax.mail.internet.*;
public class deleteEmail {
  public static void main (String args[]) throws Exception {
   String host = "pop3.mydomain.com";
   String username = "userName";
   String password = "password";
   boolean deleteStatus = false;
   Properties prop = new Properties();
   Session ses1 = Session.getDefaultInstance(prop, null);
   Store store1 = ses1.getStore("pop3");
   store1.connect(host, username, password);
   Folder folder1 = store1.getFolder("INBOX");
   folder1.open(Folder.READ_WRITE);
   BufferedReader bReader = new BufferedReader (new InputStreamReader(System.in));
   Message msg[] = folder1.getMessages();
   for (int i=0, n= msg.length; i<n; i++)
   {
     System.out.println(i + ": " + msg [i].getFrom()[0] + "\t"
         + msg [i].getSubject());
     System.out.println("Do you want to delete message? [Y/N]");
     String line1 = bReader.readLine();
     if ("Y".equals(line1))
     {
       msg[i].setFlag(Flags.Flag.DELETED, true);
       deleteStatus = true;
     }
   }
   folder1.close(deleteStatus);
   store1.close();
  }
}
```

Replying to and Forwarding an Email Message
A J2EE component can reply to an email by using the reply() method of the message object. The reply() method copies the from address of the email message to the to address. We can indicate whether or not to reply to the sender or all recipients by passing the reply() method a boolean value where true replies to all recipients and false replies to the sender.

In this example, program begins by initializing string objects with information needed to retrieve and reply to email messages. Next, the program creates an instance of the Properties object called prop, by calling getProperties(). This object is passes to the getDefautInstance() to return a Session object called session. The getStore() method is called and passed the protocol used tostore email messages. In this example, the POP3 protocol is being used. The getStore() method returns a Store object called store1.

Login information is then passed to the connect() method to log in to the mail server. Once the login is successfull, the program calls the getFolder() method, passing the name of the folder that the program wants to reference. In this case, the INBOX folder is being retrieved. The getFolder() method returns a Folder object called folder1. The open() is then called to open the folder in READ_ONLY mode.

The program requires a buffered reader to read email messages from the INBOX folder and therefore creates an instance of the BufferedReader object, which is called reader. Then the sender and subject line of each email message are retrieved and displayed, and the client is asked if a reply is necessary.

In this scenario, the reply is sent only to the sender by passing a boolean false to the reply() method. The program proceeds to set the from header of the replay using the setFrom() method. The email message is retrieved as a MimeMessage object called body. Creating a StringBuffer object called buffer1, which is assigned the reply message, follows this.

Only if the original message is a MIME type, the program creates and sends the reply. In this case, the program retrieves the body of the original email message and assigns it to the String object content. The String object is used to create a StringReader object, which is used to create a BufferedReader object.

The program then reads each line of the original email message and appends the line to the buffer1, which is the StringBuffer objectthat already contains the reply message. Each line from the original email message begins with the '>'. This is a commonly used character to indicate the line was from the original email message.

The full reply message is passed to the setText() method, which places the message in the reply object. The reply object is then passed to the send() method for transmission.

```
import java.io.*;
import java.util.Properties;
import javax.mail.*;
import javax.mail.internet.*;
public class replyEmail {
  public static void main (String args[]) throws Exception {
    String host = "pop3.mydomain.com";
    String sendHost = "smtp.mydomain.com";
    String username = "userName";
    String password = "password";
    String from = "me@myweb.com";
    Properties prop = System.getProperties();
    prop.put("mail.smtp.host", sendHost);
    Session session = Session.getDefaultInstance(prop, null);
    Store store1 = session.getStore("pop3");
    store1.connect(host, username, password);
    Folder folder1 = store1.getFolder("INBOX");
    folder1.open(Folder.READ_ONLY);
    BufferedReader reader = new BufferedReader (new InputStreamReader(System.in));
    Message msg[] = folder1.getMessages();
    for (int i=0, n= msg.length; i<n; i++)
    {
     System.out.println(i + ": " + msg[i].getFrom()[0]
        + "\t" + msg[i].getSubject());
     System.out.println("Do you want to reply to the message? [Y/N]");
     String line = reader.readLine();
     if ("Y".equals(line))
     {
       MimeMessage reply = (MimeMessage)msg[i].reply(false);
       reply.setFrom(new InternetAddress(from));
       MimeMessage body = (MimeMessage) msg[i];
       StringBuffer buffer1 = new StringBuffer("My reply goes here\n");
       if (body.isMimeType("text/plain"))
       {
         String content = (String)body.getContent();
         StringReader cReader = new StringReader(content);
```

```
          BufferedReader bReader = new BufferedReader(cReader);
          String cLine;
          while ((cLine = bReader.readLine()) != null)
          {
            buffer1.append("> ");
            buffer1.append(cLine);
            buffer1.append("\r\n");
          }
        }
        reply.setText(buffer1.toString());
        Transport.send(reply);
      }
      else if ("N".equals(line))
      {
       break;
      }
    }
    folder1.close(false);
    store1.close();
  }
}
```

Forwarding an Email Message
A J2EE component can forward an email message using the basic framework used to reply to an email. In the below example, when the client responds positively to forwarding an email message, first the program creates an instance of Message object called fwd and then subject line for the new email message. The subject begins with “Fwd:”, which is the standard abbreviation used to indicate that the message is a forwarded message. The getSubject() method is called to retrieve the subject of the original email message.

The program proceeds to set the “from” and the “to” headers of the new email message. These use the from and to String objects declared at the beginning of the program. Next, the program adds a new part to the forwarded messages by creating a BodyPart object.

The setText() method is used to place the J2EE component generated text that describes the forwarded message to the recipient. The program must create a MultiPart object that will recieve the parts of the forwarded email message. This is mpart. The text of the J2EE component’s message is then added to the Multipart object by calling the addBodyPart() method.

Another part is created to contain the original message, which is set by calling the setDataHandler() method and passing the method to the DataHandler of the original message. The part is then added to the Multipart object by calling the addBodyPart() method. The Multipart object is then placed in the forward message by calling the setContent() method, which is then passed to the send() method for transmission.

```
import java.io.*;
import java.util.Properties;
import javax.mail.*;
import javax.mail.internet.*;
public class forwardEmail {
  public static void main (String args[]) throws Exception {
    String host = "smtp host";
    String sendHost = "smtp host";
    String username = "userName";
    String password = "password";
    String from = "me@myweb.com";
    String to = "forward_to@myweb.com";
    Properties prop = System.getProperties();
    prop.put("mail.smtp.host", sendHost);
    Session session = Session.getDefaultInstance(prop, null);
    Store store1 = session.getStore("pop3");
    store1.connect(host, username, password);
    Folder folder1 = store1.getFolder("INBOX");
    folder1.open(Folder.READ_ONLY);
    BufferedReader reader = new BufferedReader (new InputStreamReader(System.in));
    Message msg[] = folder1.getMessages();
    for (int i=0; i<msg.length; i++)
    {
     System.out.println(i + ": " + msg[i].getFrom()[0]
        + "\t" + msg[i].getSubject());
     System.out.println("Do you want to forward to the message? [Y/N]");
     String line = reader.readLine();
     if ("Y".equals(line))
     {
       Message fwd = new MimeMessage(session);
       fwd.setSubject("Fwd: " + msg[i].getSubject());
       fwd.setFrom(new InternetAddress(from));
       fwd.addRecipient(Message.RecipientType.TO,new InternetAddress(to));
       BodyPart msgBP = new MimeBodyPart();
       msgBP.setText("Your message goes here. \n");
       Multipart mPart = new MimeMultipart();
       mPart.addBodyPart(msgBP);
       msgBP = new MimeBodyPart();
       msgBP.setDataHandler(msg[i].getDataHandler());
       mPart.addBodyPart(msgBP);
       fwd.setContent(mPart);
       Transport.send(fwd);
     }
     else if ("N".equals(line))
     {
      break;
     }
    }
    folder1.close(false);
    store1.close();
  }
}
```

Sending Attachments

An attachment is a resource, typically a file, that is external to the email message but is sent along with an email message. A J2EE component can send an email message with an attachment using basically the same technique that is used to forward an email message.

In the below example, one JPG file is attached to the email message. The program is similar to the forward program and begins by assigning the host, to and from and attachment information to String objects that are later used to form the email message. Next, the program uses a Properties object called prop to set up the mail server for the email message by calling the put() method. The Properties() object is also used to create an instance of the Session object, which is called ses1.

The email message called msg is then associated with the session. The setFrom(), addRecipient() and setSubject() methods are called to set the sender, recipient and subject of the email message.

As with the program that forwarded an email message, sending an email message with an attachment requires that the program create two message parts. The first message part is the body of the email message and the second is the attachment.

First, the setText() method is called and passed the text of the body of the email message. Next, the program needs to create the second part of the email message by initially creating a Multipart object call mpart, to which the Bodypart called msgBP is added by calling the addBodyPart() method.

The second Multipart object is created to create the second body for the email message. The data handler to the attachment must be associated with the second Multipart object. This is accomplished by creating a DataSource object called src and passing the DataSource object to the setDataHandler() method. And then the filename must be associated the second Multipart object by calling the setFileName() method. The filename is displayed to the recipient, which is usually the original filename without the directory path – but the filename can be any string and does not have to match the original filename.

The parts are then combined by calling the addBodyPart() method, and the consolidated parts become the body of the email message when the setContent() is called and passed the Multipart object. The last step is to call the send() method to send the email message with the attachment.

```
import java.util.Properties;
import javax.mail.*;
import javax.mail.internet.*;
import javax.activation.*;
public class sendAttachment {
  public static void main (String args[]) throws Exception {
    String host = "smtp.mydomain.com";
    String from = "me@myweb.com";
    String to = "you@myweb.com";
    String filename = "myPhoto.jpg";
    Properties prop = System.getProperties();
    prop.put("mail.smtp.host", host);
    Session ses1 = Session.getInstance(prop, null);
    Message msg = new MimeMessage(ses1);
    msg.setFrom(new InternetAddress(from));
    msg.addRecipient(Message.RecipientType.TO, new InternetAddress(to));
    msg.setSubject("Photo Attached");
    BodyPart msgBP = new MimeBodyPart();
    msgBP.setText("Take a look at this!");
    Multipart mPart = new MimeMultipart();
    mPart.addBodyPart(msgBP);
    msgBP = new MimeBodyPart();
    DataSource src = new FileDataSource(filename);
    msgBP.setDataHandler(new DataHandler(src));
    msgBP.setFileName(filename);
    mPart.addBodyPart(msgBP);
    msg.setContent(mPart);
    Transport.send(msg);
  }
}
```

Receiving attachments
whenever a J2EE component receives an email message that contains an attachment, the J2EE component must decompose the email message into parts and then save the part of the email message that contains the attachment.

The program creates a Multipart object called mPart, which is assigned the content of the email message by calling the getContent() method. Next, the program steps through the parts of the email message and extracts each part. The getDisposition() method is called to determine if the current part is an attachment or inline. If so, then the saveFile() method is called and passed the filename and the input stream.

```
import java.io.*;
import java.util.Properties;
import javax.mail.*;
import javax.mail.internet.*;
public class receiveAttachment{
  public static void main (String args[]) throws Exception {
    String host = "pop3.mydomain.com";
    String sendHost = "smtp.mydomain.com";
    String username = "userName";
    String password = "password";
    String from = "me@myweb.com";
    Properties prop = System.getProperties();
    prop.put("mail.smtp.host", sendHost);
    Session session = Session.getDefaultInstance(prop, null);
    Store store1 = session.getStore("pop3");
    store1.connect(host, username, password);
    Folder folder1 = store1.getFolder("INBOX");
    folder1.open(Folder.READ_ONLY);
    BufferedReader reader = new BufferedReader (new InputStreamReader(System.in));
    Message msg[] = folder1.getMessages();
    for (int i=0; i<msg.length; i++)
    {
     System.out.println(i + ": " + msg[i].getFrom()[0]
       + "\t" + msg[i].getSubject());
     System.out.println("Do you want to save attachment? [Y/N]");
     String line = reader.readLine();
     if ("Y".equals(line))
     {
       Multipart mPart = (Multipart)msg[i].getContent();
       for (int j=0; j<mPart.getCount(); j++)
       {
         Part part1 = mPart.getBodyPart(j);
         String disp1 = part1.getDisposition();
         if (disp1 != null && (disp1.equals(Part.ATTACHMENT) ||  disp1.equals(Part.INLINE)))
         {
          saveFile(part1.getFileName(), part1.getInputStream());
         }
       }
     }
     else if ("N".equals(line))
     {
      break;
     }
    }
    folder1.close(false);
    store1.close();
  }
}
```

Searching an Email Folder
A J2EE component is able to search an email folder for an email message that either contains or does not contain search criteria in any component of the email message. The J2EE component begins the search by creating a logical expression, called a search term, using predefined classes. The search term is then passed to the search() method for processing.

Assume that in this example, the folder1 is already identified. The code segment begins by creating a SearchTerm object call searchCriteria. The SearchTerm object is assigned an expression that looks for matches to the subject or from headers of the emial message. That is, all email messages that have “welcome” as the subject or “personnel@mycompany.com” in the from header are considered a match.

Conditions are created within the SearchTerm object by usingthe AndTerm(), OrTerm() and NotTerm() methods. In this example, the OrTerm() method is used to set the condition to OR. Components are identified in the SearchTerm by calling the appropriate method. The SubjectTerm() method is used to set the value of the subject header in the SearchTerm and the FromStringTerm() method is used to set the sender’s address in the SearchTerm.

Once the SearchTerm object is assigned the search sriteria, the code segment calls the search() method and passes the searchCriteria. The search() methos returns an array of message objects that meet the search criteria.

```
SearchTerm searchCriteria = new OrTerm(new SubjectTerm(“welcome”),
new FromStringTerm(“personnel@mycompany.com”));
Message[] msg = folder1.search(searchCriteria);
```

Security

J2EE security concepts
A J2EE application is made secure by features found within the core java classes, within the JVM, and within the java programming language. There are many levels of a J2EE a J2EE application – from java statements used within classes, to low level programming that occurs at the JVM. Security features can be implemented at each level.

Security is a balance between ultimate protection and utilization. The more the security scale moves towards ultimate protection, the less utilization is realized. The proper amount of security to apply is determined by perceived risk/utilization factors. In a high risk environment, users accept tighter security measures and less utilization. In a low risk environment, users accept loose security measures and high utilization.

For example, air travelers accept delays at the airport caused by increased security measures whenever there is high security risk. However, such delays are intolerablein a low security risk environment. The same concept holds true with users of J2EE applications.

The java security model is an open specification, which is a major advantage java has over other languages and environments.

JVM security
security measures are strictly enforced by the JVM, as the JVM verifies incoming classes before classes are run and performs runtime checks to assure that classes perform only valid operations.

The JVM does not rely on security measures imposed by the java programming language to assure that classes are safe to run. Instead, JVM examines bytecode contained in classes before running the classes.

Bytecode undergoes a three step examination by the class loader component of the JVM before the bytecode is executed. As each class is loaded, the JVM conducts security checks that are possible to perform on the first reading of the class, such as verifying that the first four bytes are 0xCAFEBABE. Next, the JVM performs security validations that are possible only on a second pass through the bytecode, such as examining superclasses.

Still another security examination occurs at runtime, where the java performs type checking and examines array boundaries to assure no security violations exist because of late binding involving dynamic memory allocation. Late binding is not validated at compile time.

Security Management
A security manager object can be implemented within a J2EE applicationto take advantage of security features available from the security manager installed in the JVM. The purpose of a security manager is to control the operations of runtime programs on the JVM. In addition, a security manager enables a J2EE application to create a security policy that limits the operation of a J2EE application.

A security manager plays a crucial role in defending against security violations such as preventing a remotely loaded java applet from accessing the local file system. Likewise, a security manager requires that a remote connection from a java applet be limited to the server that provided the java applet.

A security manager contains default security rules that are enforced on every program that runs within the JVM. These rules are called permissions and are defined in the <java installation directory /jre/lib/security/java.policy file. Each user has a .java.policy file in the user's home directory that contains permissions specific to the user.

Additional security policies can be inserted into either security policy file by using the policytool. The policytool is used to grant permissions to access a resource. Start the policytool by entering policytool at the command line.

The policytool modifies the .java.policy file in the home directory and displays an error if the .java.plicy file is not found. The add policy entry is used once the policytool's screen is displayed to see existing policies.

The Add Permission option is used to display a screen that contains three options. These are a drop down list box that contains available permissions, the property permission box that contains specific properties to choose from and there is a text field that contains the corresponding java class name that is affected by the permission.

Java API Security
There are several java packages that are used to incorporate additional security measures within a J2EE application. These are the Java Cryptography Architecture(JCA) in the java.security.* package and the Java Cryptography Extension(JCE) found in the javax.crypto.* package. These are used to integrate cryptographic algorithms, ciphers, secure streams, key generation, certificates, digital fingerprints and signatures.

A digital fingerprint, also known as a message digest, consists of a calculation based on the content of the message. The message sender performs the calculation and the result is sent along with the message to the message receiver. The message receiver performs the same calculation and compares the result with the result received in the message.

A digital fingerprint is not efficient since the message can be intercepted and replaces with another message that has the same digital fingerprint. A digital signature is introduced as a solution for this problem. A digital signature requires that the digital fingerprint be encrypted by the message sender using the message sender's private key. The message receiver uses the message sender's public key to decipher the digital fingerprint. Certificates are used to manage the identities of message senders who use digital signatures.

The disadvantage of digital signature is the public key that is used to decipher the digital fingerprint. The message sender sends the public key to the message receiver. However, the message receiver still must be authenticated for the message sender to be sure the message sender is the person who actually sent the message.

A certificate is used to address the problem of digital signature. The certificate contains the message sender's email address, name and public key along with other information used to uniquely identify the message sender. The integrity of the certificate is attested by a well known and trusted organization called a certification authority such as VeriSign. The certification authority provides a message sender with a private key used to encrypt the digital fingerprint. Likewise, the certification authority provides the message receiver with the certificate that contains the certification authority's public key.

Browser Security

By its nature, a java applet is untrustworthy because a java applet is downloaded from a remote computer. The java applet's security manager restricts a java applet to secure operations such as preventing access to local computer resources.

For example, a java applet is prevented from accessing files on a local disk drive. Typically, local files contain passwords and other sensitive information including emails and address books.

A java applet cannot open a socket to an IP address other than to the IP address from where the java applet was downloaded, because hackers commonly use local computer to launch an attack against remote computers. The attack appears to be coming from the local computerwhen, in reality, the hacker's remote computer is using the local computer for misdirection.

Also, a java applet is denied access to monitoring incoming ports on local computer. An incoming port is used to receive transmissions such as email from a remote computer. A java applet could intercept and redirect incoming communication if the java applet security manager did not plug this potential security gap.

This leaves the local computer vulnerable to a security violation by a java applet by consuming too much CPU time, denying service to other programs.

A browser executes a plug-in whenever a browser encounters reference to the plug-in with an HTML page. This reference typically in the form of a URL to a file that contains a file extension that is associated with a plug-in, such as PDF for an acrobat file. A security manager does not restrain a plug-in has all the capabilities of any executable program running locally on the computer.

The security manager informs the user when an untrusted java applet generates a message on the screen. The warning message varies according to the browser that is used by the local computer.

Web Services Security
security is vital to a J2EE application that is implemented in a web services environment, because requests for service and responses from components are vulnerable during transmission over a network. The java community address security concerns with the introduction of the declarative security mechanism contained in the java Web Services Development Pack(WSDP).

The declarative security mechanism requires application providers to declare the application's security requirements, so that the security requirements can be addressed when the application is configured.

The application provider uses a declarative syntax deployment descriptor to define the security requirements for the appliaction. The deployment descriptor is used by the application deployer to implement the security requirements of the application. This is accomplished by mapping the security requirements to the security features of the web services container using a tool provided by the container vendor.

WSDP is designed to address security concerns through the web services in a multi-tier architecture. Each tier contains components and resources used by components to respond to requests from a J2EE application. Resources are divided into two groups: protected and unprotected resources. Protected resources are those that require that a component be authorized before access is granted. No authorization is required for a component to use unprotected resources.

WSDP security requirements address authentication and access control to web services resources. An authentication process must verify the identity and access rights of a component that needs access to a protected resource before the component is granted access to the resource. Components seeking access to unprotected resources are granted anonymous access to the resource without having to undergo the authentication process.

Web Services Security Classifications
Web services security is similar to security found on many operating systems in that access rights can be assigned to an individual user ID, to a particular functional role performed by a user, or to a group of users.

A user is a person or application that is assigned a user ID and password that can be authenticated by the security manager. A user can be associated with a role, such as a system administrator. Also, a user can be associated with a group of users, such as the accounts payable department.

A role is a name that identifies a function that is performed by one or more users. One or more users are assigned to a role if they perform the same function. A group is a name that identifies an association of users who have common characterstics.

A user is assigned a user ID and password, which are used to authenticate the user to web services. The administrator of web services uses the web services server security management tool to register the user ID and password with the server. The security management tool is also used to assign a user ID to either a role or group.

Access rights to web services resources are the rights assigned directly to a user ID or to a role or group. Administrators find it more efficient to assign access rights to resources to a role or group rather than to each user ID, because a single change to access rights of role or group affects multiple user ID’s.

User ID’s, roles and groups along with related access rights are stored in a database commonly referred to as a “realm”.

Security Within a Web Services Tier
Web services is organized into tiers. Each tier contains a web resource collection, which is a list of URL patterns and HTTP methods. A web resource collection is protected by security constraints that authorize access to members of a web resource collection.security constraints are defined in the deployment descriptor.

Security constraints are used by the web container to authenticate a user who requests service from a member of the resource collection. The web container doesn’t process a service request until the user is authenticated.

No further web container security measures are taken once a user is authenticated. This means that the user can access any resource in the collection without requiring additional permissions.

There are four kinds of authentication that can be implemented to protect a resource collection: basic authentication, form based authentication, client certificate authentication and digest authentication.

Basic authentication also known as HTTP basic authentication requires the web server to use the user ID and password to authenticate a user. Form based authentication uses a customized login screen that is displayed by an HTTP browser and is used to capture a user ID and password.

Client certificate authentication requires the use of a certificate to authenticate a requestor and web server. Transmission occurs over Secure Socket Layer(SSL), which encrypts data and uses a private and public key to hinder decoding if a hacker intercepts the transmission.

Digest authentication requires that the user password be encoded before the password id transmitted to a web server. Transmission occurs over non-SSL or SSL. A hacker can still intercept the transmission, but the hackers attempt to penetrate the web services server is impeded by the ciphered password.

Avoid using non-SSL for transmissions during a session. A session consists of a series of web services requests made by the same client once the client is authenticated by the web services server. Each session is uniquely identified with a session ID, which is then in each request to associate requests with a session.

Authentication occurs at the beginning of a session and does not continue with each request during the session. A session ID can be intercepted if the requests for the session are transmitted using non-SSL. It can then be used by a hacker in subsequent requests to spoof the web services server into thinking the hacker is the previously authenticated web services server.

Programmatic Security
J2EE applications that require a high level of security must implement programmatic security to embellish declarative security provided by the deployment descriptor. Programmatic security is implemented by using the getRemoteUser(), isUserInRole() and getUserPrincipal() methods of the HttpServletRequest interface.

These methods provide the remote username and the security role of the user and obtain a user principal object.