KARPAGAM ACADEMY OF HIGHER EDUCATION
(Deemed to be University)
(Established Under Section 3 of UGC Act, 1956)

DEPARTMENT OF CS, CA & IT

SYLLABUS

SUBJECT NAME: COMPUTER FUNDAMENTALS **SUBJECT CODE: 18CSU103**

SEMESTER: I **CLASS: I BSc CS -A**

Instruction Hours / week: L: 4 T: 0 P: 0 **Marks:** Int : **40** Ext : **60** Total: **100**

COURSE OBJECTIVES

- To identify types of computers, how they process information and how individual computers interact with other computing systems and devices.
- To identify the function of computer hardware components.
- To identify the factors that goes into an individual or organizational decision on how to purchase computer equipment.
- To identify how to maintain computer equipment and solve common problems relating to computer hardware.
- To identify how software and hardware work together to perform computing tasks and how software is developed and upgraded.
- To identify different types of software, general concepts relating to software categories, and the tasks to which each type of software is most suited or not suited.
- To identify fundamental concepts relating to database applications.
- To identify what an operating system is and how it works, and solve common problems related to operating systems.
- To manipulate and control the Windows desktop, files and disks.
- To Identify how to change system settings, install and remove software.

COURSE OUTCOMES

- Understand the meaning and basic components of a computer system,
- Define and distinguish Hardware and Software components of computer system,
- Explain and identify different computing machines during the evolution of computer system,
- Gain knowledge about five generations of computer system,
- Explain the functions of a computer,
- Identify and discuss the functional units of a computer system,
- Identify the various input and output units and explain their purposes
- Understand the role of CPU and its components,
- Understand the concept and need of primary and secondary memory,
- Discuss the advantages, limitations and applications of computers,
- Understand the classification of computers,
- Distinguish the computers on the basis of purpose, technology and size

UNIT I

Introduction: Introduction to computer system, uses, types. **Data Representation:** Number systems and character representation, binary arithmetic. **Human Computer Interface:** Types of software, Operating system as user interface, utility programs.

UNIT II

Devices: Input and output devices (with connections and practical demo), keyboard, mouse, joystick, scanner, OCR, OMR, bar code reader, web camera, monitor, printer, plotter.

UNIT III

Memory: Primary, secondary, auxiliary memory, RAM, ROM, cache memory, hard disks, optical disks.

UNIT IV

Computer Organisation and Architecture: CPU, registers, system bus, main memory unit, cache memory, Inside a computer, SMPS, Motherboard, Ports and Interfaces, expansion cards, ribbon cables, memory chips, processors.

UNIT V

Overview of Emerging Technologies: Bluetooth, cloud computing, big data, data mining, mobile computing and embedded systems.

SUGGESTED READINGS

1. Goel, A. (2010). Computer Fundamentals. New Delhi: Pearson Education.
2. Aksoy, P., & DeNardis, L. (2006).Introduction to Information Technology. New Delhi: Cengage Learning
3. Sinha, P. K., & Sinha, P. (2007). Fundamentals of Computers. New Delhi: BPB Publishers.

KARPAGAM ACADEMY OF HIGHER EDUCATION
(Deemed to be University)
(Established Under Section 3 of UGC Act, 1956)
DEPARTMENT OF CS, CA & IT
LESSON PLAN

STAFF NAME : K.BANUROOPA
SUBJECT NAME : COMPUTER FUNDAMENTALS SUBJECT CODE : 18CSU103
SEMESTER : I CLASS : I BSc CS-A

S. No	Lecture Duration Period	Topics to be Covered	Support Materials / Page No.
UNIT I			
1.	1	**Introduction to Computer System**	T1: 1-4
2.	1	Characteristics of Computer- History of Computer	T1: 1-4
3.	1	Generation of Computer	T1: 4-5
4.	1	Types of Computer	T1: 5-6
5.	1	**Data Representation**	T1: 6-8
6.	1	Number systems and character representation	T1: 8-10
7.	1	Binary arithmetic	T1: 2-12
8.	1	Human Computer Interface: Types of Software	T2: 173
9.	1	Operating System as User Interface	T2: 251
10.	1	Recapitulation and discussion of possible questions	T1: 1-4
Total No .of Hours Planed For Unit I : 10 Hrs			
UNIT II			
1.	1	**Devices - Input and Output**	T2: 149
2.	1	Keyboard	T2: 149
3.	1	Mouse, Joystick	T2: 149
4.	1	Scanner	T2: 153
5.	1	OCR, OMR	T2: 154
6.	1	Bar code Reader, Web camera	T2: 155
7.	1	Monitor, Printer	T2: 159
8.	1	Plotter	T2: 160
9.	1	Recapitulation and discussion of possible questions	
Total No .of Hours Planed For Unit II : 9 Hrs			
UNIT III			
1.	1	**Memory: Primary,** Secondary	T2: 108
2.	1	Auxiliary Memory	T2: 112
3.	1	RAM	T2: 112
4.	1	ROM	T2: 112
5.	1	Cache Memory	T2: 113
6.	1	Hard Disks	T2: 124
7.	1	Optical Disks	T2: 134
8.	1	Recapitulation and discussion of possible questions	
Total No .of Hours Planed For Unit III : 8 Hrs			

UNIT IV			
1.	1	**Computer Organisation and Architecture :** CPU	T2: 101
2.	1	Registers	T2: 103
3.	1	System Bus, Main memory Unit	T2: 108
4.	1	Cache Memory	T2: 113
5.	1	Inside a Computer, SMPS	T1: 1.28
6.	1	Motherboard, ports and Interfaces	T1: 1.28
7.	1	Expansion Cards, Ribbon cables	T1: 1.28
8.	1	Memory Chips	T1: 1.31
9.	1	Processor	T1: 1.32
10.	1	Recapitulation and discussion of possible questions	
Total No .of Hours Planed For Unit IV : 10 Hrs			
UNIT V			
1.	1	**Overview of Emerging Technologies**	W1
2.	1	Bluetooth	W2
3.	1	Cloud Computing	W3
4.	1	Big Data	W4
5.	1	Data Mining	W5
6.	1	Mobile Computing	W6
7.	1	Embedded Systems	W7
8.	1	Recapitulation and discussion of possible questions	
9.	1	Discussion of previous ESE question papers	
10.	1	Discussion of previous ESE question papers	
11.	1	Discussion of previous ESE question papers	
Total No .of Hours Planed For Unit V : 11 Hrs			
Total No. of Hours Planned for this Course: 48 Hrs			

SUGGESTED READINGS:

T1. Goel, A. (2010). Computer Fundamentals. New Delhi: Pearson Education.
T2. Sinha, P. K., & Sinha, P. (2007). Fundamentals of Computers. New Delhi: BPB Publishers.
T3. Aksoy, P., & DeNardis, L. (2006).Introduction to Information Technology. New Delhi: Cengage Learning

WEB SITES

W1. https://en.wikipedia.org/wiki/List_of_emerging_technologies
W2. https://www.bluetooth.com/
W3. https://aws.amazon.com/what-is-cloud-computing/
W4. https://www.oracle.com/in/big-data/guide/what-is-big-data.html
W5. https://www.techopedia.com/definition/1181/data-mining
W6. https://www.tutorialspoint.com/mobile_computing/mobile_computing_overview.htm
W7. https://www.techopedia.com/definition/3636/embedded-system

Unit I

SYLLABUS:

J2EE Overview: Beginning of Java – Java Byte code – Advantages of Java –J2EE and J2SE. J2EE Multi Tier Architecture – Distributive Systems – The Tier – Multi Tier Architecture – Client Tier Web Tier Enterprise Java Beans Tier Enterprise Information Systems Tier Implementation.

INTRODUCTION

Today, more and more developers want to write distributed transactional applications for the enterprise and leverage the speed, security, and reliability of server-side technology. J2EE is Java, optimized for enterprise computing. J2EE stands for Java 2 Platform, Enterprise Edition. Unlike the traditional Java, which is often used to build client enhancements, J2EE is designed to build server applications. As an enterprise platform, the J2EE environment extends basic Java with tools that "provide a complete, stable, secure, and fast Java platform to the enterprise level". J2EE reduces the cost and complexity of creating large-scale solutions. The J2EE platform offers a Multitiered -distributed application model, the ability to reuse components, integrated Extensible Markup Language (XML)-based data interchange, a unified security model, and flexible transaction control.

J2EE OVERVIEW

J2EE is Java, optimized for enterprise computing. Officially J2EE stands for Java 2 Platform, Enterprise Edition. J2EE is an open, standard-based, development and deployment platform for building n-tier, web-based and server-centric and component-based enterprise applications. As an enterprise platform, the J2EE environment extends basic Java with tools that "provide a complete, stable, secure, and fast Java platform to the enterprise level." One goal of using J2EE is reducing the cost and complexity of creating large-scale solutions. Because Java is a strongly typed language, use of the language is often inherently more secure in Web applications than Web applications built with less strong typing

BEGINNING OF JAVA

Java was created in 1991. It was developed by James Gosling et al. of Sun Microsystems. Initially called Oak, in honor of the tree outside Gosling's window, its name was changed to Java because there was already a language called Oak. The original motivation for Java is the need for platform independent language that could be embedded in various consumer electronic products like toasters and refrigerators. As a programming language, Java can create all kinds of applications that you could create using any conventional programming language

JAVA BYTE CODE

Java bytecode is the form of instructions that the Java virtual machine executes. Each byte code opcode is one byte in length, although some require parameters, resulting in some multi-byte instructions. Not all of the possible 256 opcodes are used. Java byte code is designed to be executed in a Java virtual machine. There are several virtual machines available today, both free and commercial products.

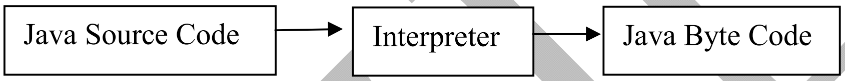


Figure 1.4.1 Converting Source code to bytecode

ADVANTAGES OF JAVA

JAVA offers a number of advantages to developers.

- **Java is simple**: Java was designed to be easy to use and is therefore easy to write, compile, debug, and learn than other programming languages. The reason that why Java is much simpler than C++ is because Java uses automatic memory allocation and garbage collection where else C++ requires the programmer to allocate memory and to collect garbage.
- **Java is object-oriented**: Java is object-oriented because programming in Java is centered on creating objects, manipulating objects, and making objects work together. This allows you to create modular programs and reusable code.
- **Java is platform-independent**: One of the most significant advantages of Java is its ability to move easily from one computer system to another. The ability to run the same program on many different systems is crucial to World Wide Web software, and Java succeeds at this by being platform-independent at both the source and binary levels.
- **Java is distributed**: Distributed computing involves several computers on a network working together. Java is designed to make distributed computing easy with the networking capability that is inherently integrated into it. Writing network programs in Java is like sending and receiving data to and from a file. For example, the diagram

below shows three programs running on three different systems, communicating with each other to perform a joint task.

- **Java is interpreted**: An interpreter is needed in order to run Java programs. The programs are compiled into Java Virtual Machine code called bytecode. The bytecode is machine independent and is able to run on any machine that has a Java interpreter. With Java, the program need only be compiled once, and the bytecode generated by the Java compiler can run on any platform.
- **Java is secure**: Java is one of the first programming languages to consider security as part of its design. The Java language, compiler, interpreter, and runtime environment were each developed with security in mind.
- **Java is robust**: Robust means reliable and no programming language can really assure reliability. Java puts a lot of emphasis on early checking for possible errors, as Java compilers are able to detect many problems that would first show up during execution time in other languages.
- **Java is multithreaded**: Multithreaded is the capability for a program to perform several tasks simultaneously within a program. In Java, multithreaded programming has been smoothly integrated into it, while in other languages, operating system-specific procedures have to be called in order to enable multithreading. Multithreading is a necessity in visual and network programming

J2EE AND J2SE

J2SE is considered the foundation edition of the Java platform and programming environment in which all other editions are based. J2EE is the edition of the Java 2 platform targeted at developing multi-tier enterprise applications.J2EE consists of a set of specifications, APIs and technologies defining enterprise application development. J2EE technology providers expose tools, frameworks and platforms that handle a good deal of the details of enterprise application infrastructure and behavior. J2EE implementations enjoy all of the features of the Java 2 Standard Edition (J2SE) platform with additional frameworks and libraries added to support distributed/Web development

J2EE MULTI TIER ARCHITECTURE

The J2EE platform uses a multitiered distributed application model. Application logic is divided into components according to function, and the various application components that make up a J2EE application are installed on different machines depending on the tier in the multitiered J2EE environment to which the application component belongs. Figure 1.7.1 shows two multitiered J2EE applications divided into the tiers described in the following list.

- Client-tier components run on the client machine.
- Web-tier components run on the J2EE server
- Enterprise JavaBean tier components run on the J2EE server.
- Enterprise information system (EIS)-tier software runs on the EIS server.

Although a J2EE application can consist of the three or four tiers shown in Figure 1.7.1, J2EE multitiered applications are generally considered to be three tiered applications because they are distributed over three different locations: client machines, the J2EE server machine, and the database or legacy machines at the back end. Three-tiered applications that run in this way extend the standard two-tiered client and server model by placing a multithreaded application server between the client application and back-end storage.

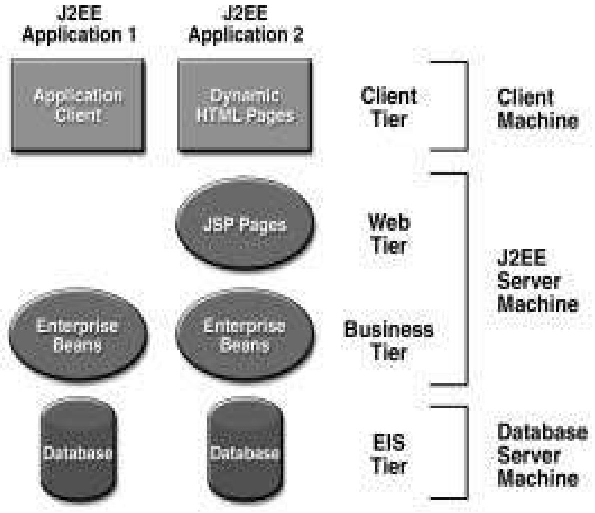


Figure1.7.1 J2EE Multitiered Applications

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, which are the fundamental design philosophy used to develop programs. In earlier days programmers originally used assembly language to create programs. These programs employed the concept of software services that were shared with the program running on the machine. 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.

Although the technique of using software services made creating programs efficient by reusing code, there was a drawback. 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 introduction of FORTRAN and COBOL brought the next evolution of programming languages and with it the next evolution of software services. Programs written in FORTRAN could share functionality by using functions instead of assembly language subroutines. The same was true of programs written in COBOL. A function is conceptually similar to a Java method, which is a group of statements that perform a specific functionality. The group is named, and is callable from within a program. Although both assembly language subroutines and functions are executed in a single memory space, functions had a critical advantage over assembly language subroutines.

A function could run on different machines by recompiling the function. However, software services were restricted to a machine. This meant programs and functions that comprise software services had to reside on the same machine. A program couldn’t call a software service that was contained on a different machine. Programs and software services were saddled with the same limitations that affected data exchange at that time. Magnetic tapes were used to transfer data, programs, and software services to another machine. There wasn’t a real-time transmission system

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 to draw a parallel to a typical large corporation (see Figure 1.9.1).

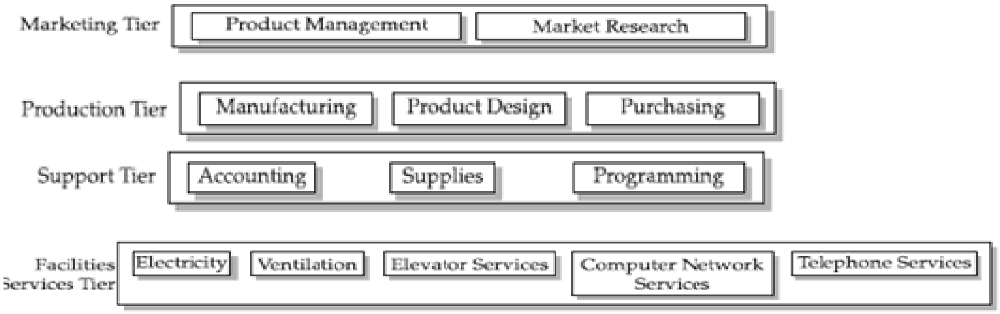


Figure 1.9.1 Multitier Architecture

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

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. 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. For Example 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. In multi-tier architecture, each tier contains services that include software objects, database management systems (DBMS), or connectivity to legacy systems.

Information technology departments of corporations employ multi-tier architecture because it's a cost-efficient way to build an application that is flexible, scalable, and responsive to the expectations of clients. This is because the functionality of the application is divided into logical components that are associated with a tier. Each component is a service that is built and maintained independently of other services. Services are bound together by a communication protocol that enables a service to receive and send information from and to other services.

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. This means that a programmer can quickly develop a system by creating a client program that formulates requests for services that already exist in the multi-tier architecture. These services already have the functionality built into them to fulfill the request made by the client program.

Services can be modified as changes occur in the functionality without affecting the client program. For example, a client might request the tax owed on a specific order. The request is sent to a service that has the functionality to determine the tax. The business logic for calculating the tax resides within the service. A programmer can modify the business logic in the service to reflect the latest changes in the tax code without having to modify the client program. These changes are hidden from the client program.

J2EE Multi-Tier Architecture

J2EE is four-tier architecture (see Figure1.10.1). 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. It's important to delineate between physical location and functionality. Two or more tiers can physically reside on the same Java Virtual Machine (JVM) although each tier provides a different type of functionality to a J2EE application. And since the J2EE multi-tier architecture is functionally centric, a J2EE application accesses only tiers whose functionality is required by the J2EE application. It's also important to disassociate a J2EE API with a particular tier. That is, some APIs (i.e., XML API) and J2EE components can be used on more than one tier, while other APIs (i.e., Enterprise JavaBeans API) are associated with a particular tier. 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.

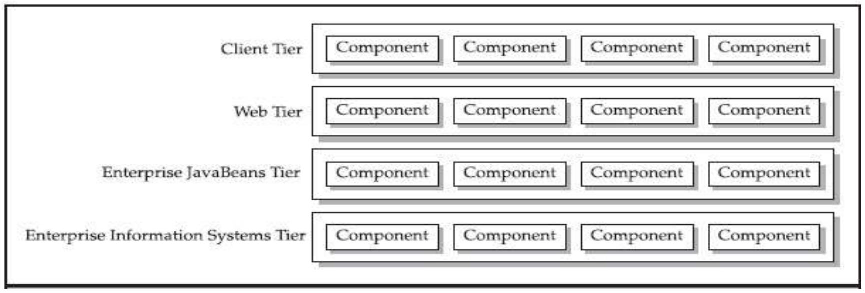


Figure1.10.1 J2EE consists of four tiers, each of which focuses on providing specific functionality to an application.

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. A client is any component that initiates a request, as explained previously in this chapter. For example (see Figure 2-4), 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

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. A client is any component that initiates a request, as explained previously in this chapter. For example (see Figure 2-4), 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

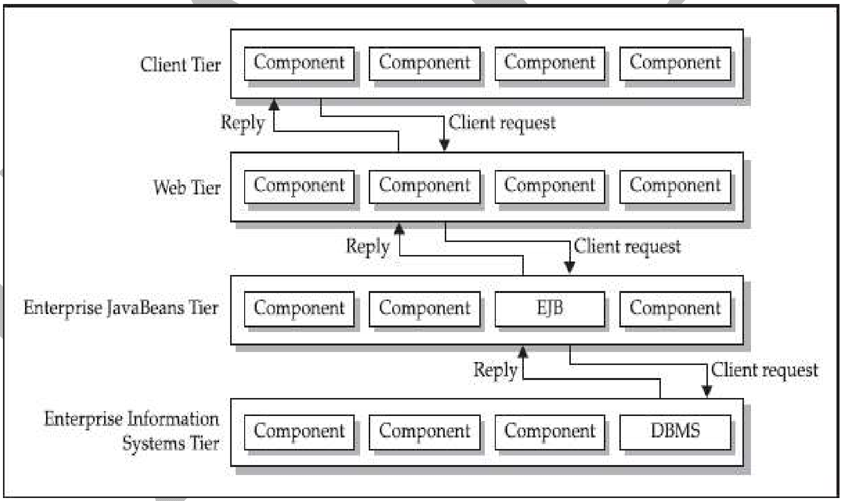


Figure 10.10.2 J2EE consists of four tiers

J2EE consists of four tiers, each of which focuses on providing specific functionality to an application. A request is typically passed from one tier to another before the 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. Enterprise JavaBeans are contained on the Enterprise JavaBeans server, which is a distributed object server that works on the Enterprise JavaBeans Tier and manages transactions and security, and assures that multithreading and persistence are properly implemented whenever an Enterprise JavaBean is accessed. Although an Enterprise JavaBean can access components on any tier, typically an Enterprise JavaBean accesses components and resources such as DBMS on the Enterprise Information System (EIS) Tier.

Access is made using an Access Control List (ACL) that controls communication between tiers. The ACL is a critical design element in the J2EE multi-tier architecture because ACL bridges tiers that are typically located on different virtual local area networks and because ACL adds a security level to web applications. Hackers typically focus their attack on the Web Tier to try to directly access DBMS. ACL prevents direct access to DBMS and similar resources. 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. Components that work on the EIS communicate to resources using CORBA or Java connectors, referred to as J2EE Connector Extensions.

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. A client is any component that initiates a request, as explained previously in this chapter. For example (see Figure 2-4), 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

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. A client is any component that initiates a request, as explained

previously in this chapter. For example (see Figure 2-4), 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

CLIENT TIER IMPLEMENTATION

There are two components on the Client Tier that are described in the J2EE specification. These are applet clients and application clients. An applet client is a component used by a web client that operates within the applet container, which is a Java-enabled browser. An applet uses the browser as a user interface.

An application client is a Java application that operates within the application client container, which is the Java 2 Runtime Environment, Standard Edition (JRE). An application has its own user interface and is capable of accessing all the tiers in the multi-tier architecture depending how the ACLs are configured, although typically an application has access to only the web layer. 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 and therefore J2EE doesn't define a rich client container.

A rich client is similar to an application client in that both are applications that contain their own user interface. And as with an application client, a rich client can access any tier in the environment, depending on the ACLs configuration, using HTTP, SOAP, ebXML, or an appropriate protocol.

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. These responsibilities are to act as an intermediary between components working on the Web Tier and other tiers and the Client Tier.

Intermediary activities include:

- Accepting requests from other software that was sent using POST, GET, and PUT operations, which are part of HTTP transmissions
- Transmit data such as images and dynamic content

There are two types of components that work on the Web Tier. These are servlets and Java Server Pages (JSP), although many times they are proxied to the Application or EJB Tier. 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 servlet is associated with a URL that is mapped by the servlet container.

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 s 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.

Business logic used by JSP and servlet's is contained in one or more Enterprise JavaBeans that are callable from within the JSP and servlet. The code is the same for both JSP and servlet, although the format of the code differs. JSP uses custom tags to access an Enterprise JavaBeans while servlet's are able to directly access Enterprise JavaBeans.

ENTERPRISE JAVABEANS TIER IMPLEMENTATION

J2EE uses distributive object technology to enable Java developers to build portable, scalable, and efficient applications that meet the 24-7 durability expected from an enterprise system. The Enterprise JavaBeans Tier contains the Enterprise JavaBeans server, which is the object server that stores and manages Enterprise JavaBeans. 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 enable scalability of an application, because the tier is designed to work in a clustered environment. This assumes that vendor components that are used support clustering. If not, a Local Director is typically used for horizontal load balancing

The Enterprise JavaBeans Tier manages instances of components. This means component containers working on the Enterprise JavaBeans Tier create and destroy instances of components and also move components in and out of memory. Fault-tolerance is an important consideration in mission-critical applications. 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. The Enterprise JavaBeans server has an Enterprise JavaBeans container within which is a collection of Enterprise JavaBeans. As discussed in previous sections of this chapter, an Enterprise Java Bean is a class that contains business logic and is callable from a servlet or JSP.

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.

Any component, regardless of the tier where the component is located, can use Enterprise JavaBeans. This means that an Enterprise Java Bean client can reside outside the Client Tier. The protocol used to communicate between the Enterprise JavaBeans Tier and other tiers is dependent on the protocol used by the other tier. Components on the Client Tier and the Web Tier communicate with the Enterprise JavaBeans Tier using the Java RMI API and either IIOP or JRMP. Sometimes software on other tiers, usually the middle tier, uses JMS to communicate with the Enterprise JavaBeans Tier.

This communication isn't exclusively used to send and receive messages between machines. JMS is also used for other communication, such as decoupling tiers using the queue mechanism. However, the Enterprise Java Bean that is used must be a message-driven bean (MDB). MDBs are commonly used to process messages on a queue that may or may not reside on the local machine.

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. This tier provides flexibility to developers of J2EE applications because developers can leverage existing systems and resources currently available to the corporation and do not need to replicate them in J2EE. Likewise, 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 or through proprietary protocols. Java Connector technology enables software developers to create a Java Connector for legacy systems and for third-party software. The connector defines all the elements that are needed to communicate between the J2EE application and the non-J2EE software. This includes rules for connecting to each other and rules for conducting secured transactions.

J2EE COMPONENTS

J2EE applications are made up of components. A J2EE component is a self-contained functional software unit that is assembled into a J2EE application with its related classes and files and that communicates with other components.

The J2EE specification defines the following J2EE components:

- Application clients and applets are components that run on the client.
- Java Servlet and Java Server Pages (JSP) technology components are Web components that run on the server.
- Enterprise JavaBeans (EJB) components (enterprise beans) are business components that run on the server.

J2EE components are written in the Java programming language and are compiled in the same way as any program in the language. The difference between J2EE components and "standard" Java classes is that J2EE components are assembled into a J2EE application, verified to be well formed and in compliance with the J2EE specification, and deployed to production, where they are run and managed by the J2EE server.

J2EE Clients

A J2EE client can be a Web client or an application client.

Web Clients

A Web client consists of two parts: dynamic Web pages containing various types of markup language (HTML, XML, and so on), which are generated by Web Components running in the Web tier, and a Web browser, which renders the pages received from the server.

A Web client is sometimes called a thin client. Thin clients usually do not do things like query databases, execute complex business rules, or connect to legacy applications. When you use a thin client, heavyweight operations like these are off-loaded to enterprise beans executing on the J2EE server where they can leverage the security, speed, services, and reliability of J2EE server-side technologies.

Applets

A Web page received from the Web tier can include an embedded applet. An applet is a small client application written in the Java programming language that executes in the Java virtual machine installed in the Web browser. However, client systems will likely need the Java Plug-in and possibly a security policy file in order for the applet to successfully execute in the Web browser. Web components are the preferred API for creating a Web client program because no plug-ins or security policy files are needed on the client systems. Also, Web components enable cleaner and more modular application design because they provide a way to separate applications programming from Web page design. Personnel involved in Web page design thus do not need to understand Java programming language syntax to do their jobs.

Application Clients

A J2EE application client runs on a client machine and provides a way for users to handle tasks that require a richer user interface than can be provided by a markup language. It typically has a graphical user interface (GUI) created from Swing or Abstract Window Toolkit (AWT) APIs, but a command-line interface is certainly possible. Application clients directly access enterprise beans running in the business tier. However, if application requirements warrant it, a J2EE application client can open an HTTP connection to establish communication with a servlet running in the Web tier.

J2EE SERVER COMMUNICATIONS

Figure 1.16-1 shows the various elements that can make up the client tier. The client communicates with the business tier running on the J2EE server either directly or, as in the case of a client running in a browser, by going through JSP Pages or servlet's running in the Web tier. The J2EE application uses a thin browser-based client or thick application client. In deciding which one to use, you should be aware of the trade-offs between keeping functionality on the client and close to the user (thick client) and offloading as much functionality as possible to the server (thin client). The more functionality you off-load to the server, the easier it is to distribute, deploy, and manage the application; however, keeping more functionality on the client can make for a better perceived user experience

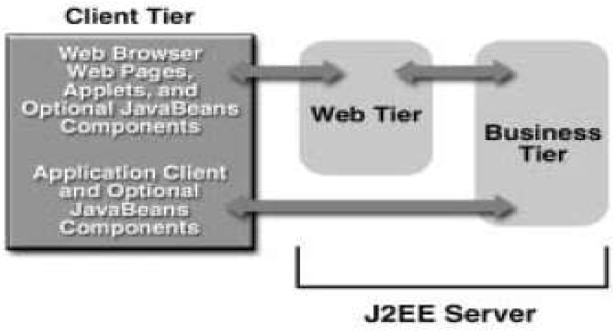

Client Tier
Web Browser Web Pages, Applets, and Optional JavaBeans Components
Application Client and Optional JavaBeans Components
Web Tier
Business Tier
J2EE Server

Figure 1.16-1 Server Communications

Web Components

J2EE Web components can be either servlets or JSP pages. Servlets are Java programming language classes that dynamically process requests and construct responses. JSP pages are text-based documents that execute as servlets but allow a more natural approach to creating static content. Static HTML pages and applets are bundled with Web components during application assembly, but are not considered Web components by the J2EE specification. Server-side utility classes can also be bundled with Web components and, like HTML pages, are not considered Web components. Like the client tier and as shown in Figure 1.17–1, the Web tier might include a JavaBeans component to manage the user input and send that input to enterprise beans running in the business tier for processing.

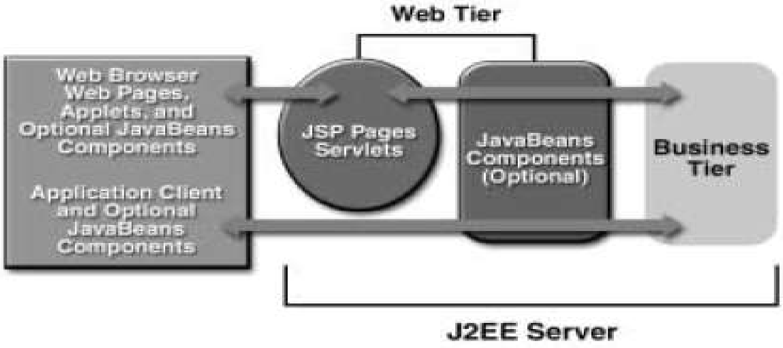

Web Tier
Web Browser Web Pages, Applets, and Optional JavaBeans Components
Application Client and Optional JavaBeans Components
JSP Pages Servlets
JavaBeans Components (Optional)
Business Tier
J2EE Server

Figure 1.17-1 Web Tier and J2EE Application

BUSINESS COMPONENTS

Business code, which is logic that solves or meets the needs of a particular business domain such as banking, retail, or finance, is handled by enterprise beans running in the business tier. Figure 1.18-1 shows how an enterprise bean receives data from client programs, processes it (if necessary), and sends it to the enterprise information system tier for storage. An enterprise bean also retrieves data from storage, processes it (if necessary), and sends it back to the client program.

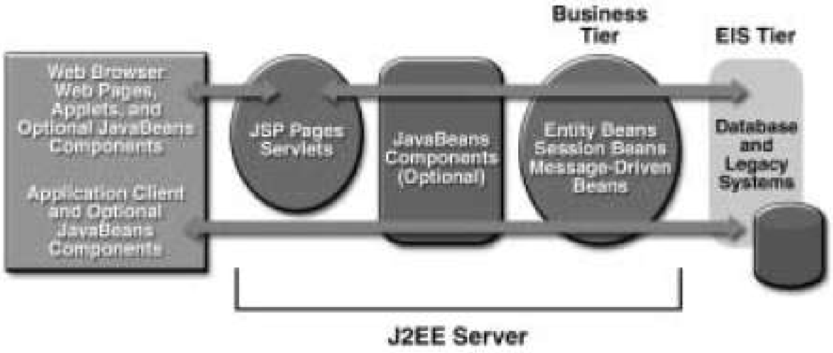

Business Tier
EIS Tier
Web Browser Web Pages, Applets, and Optional JavaBeans Components
Application Client and Optional JavaBeans Components
JSP Pages Servlets
JavaBeans Components (Optional)
Entity Beans Session Beans Message-Driven Beans
Database and Legacy Systems
J2EE Server

Figure 1.18-1 Business and EIS Tiers

There are three kinds of enterprise beans: session beans, entity beans, and message- driven beans. A session bean represents a transient conversation with a client. When the client finishes executing, the session bean and its data are gone. In contrast, an entity bean represents persistent data stored in one row of a database table. If the client terminates or if the server shuts down, the underlying services ensure that the entity bean data is saved. A message-driven bean combines features of a session bean and a Java Message Service ("JMS") message listener, allowing a business component to receive JMS messages asynchronously. This tutorial describes entity beans and session beans.

J2EE CONTAINERS

Normally, thin-client multitiered applications are hard to write because they involve many lines of intricate code to handle transaction and state management, multithreading, resource pooling, and other complex low-level details. The component based and platform-independent J2EE architecture makes J2EE applications easy to write because business logic is organized into reusable components. In addition, the J2EE server provides underlying services in the form of a container for every component type. Because it is not necessary to develop these services yourself, you are free to concentrate on solving the business problem at hand.

Container Services

Containers are the interface between a component and the low-level platform specific functionality that supports the component. Before a Web, enterprise bean, or application client component can be executed, it must be assembled into a J2EE application and deployed into its container.

The assembly process involves specifying container settings for each component in the J2EE application and for the J2EE application itself. Container settings customize the underlying support provided by the J2EE server, which includes services such as security, transaction management, Java Naming and Directory Interface (JNDI) lookups, and remote connectivity. Here are some of the highlights:

- The J2EE security model lets you configure a Web component or enterprise bean so that system resources are accessed only by authorized users.
- The J2EE transaction model lets you specify relationships among methods that make up a single transaction so that all methods in one transaction are treated as a single unit.

- JNDI lookup services provide a unified interface to multiple naming and directory services in the enterprise so that application components can access naming and directory services.

The J2EE remote connectivity model manages low-level communications between clients and enterprise beans. After an enterprise bean is created, a client invokes methods on it as if it were in the same virtual machine. The fact that the J2EE architecture provides configurable services means that application components within the same J2EE application can behave differently based on where they are deployed. For example, an enterprise bean can have security settings that allow it a certain level of access to database data in one production environment and another level of database access in another production environment.

The container also manages non configurable services such as enterprise bean and servlet life cycles, database connection resource pooling, data persistence, and access to the J2EE platform APIs. Although data persistence is a non configurable service, the J2EE architecture lets you override container-managed persistence by including the appropriate code in your enterprise bean implementation when you want more control than the default container-managed persistence provides. For example, you might use bean-managed persistence to implement your own finder (search) methods or to create a customized database cache

Container Types

The deployment process installs J2EE application components in the J2EE containers illustrated in Figure .1.19–1.

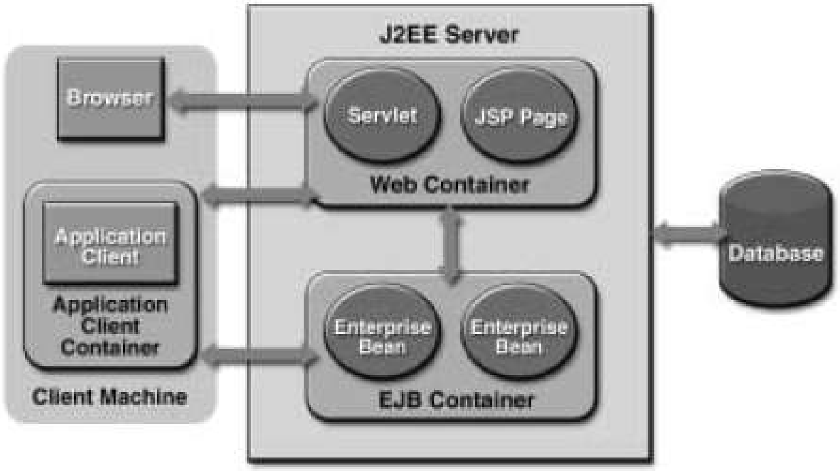


Figure .1.19–1 J2EE Server and Containers

- **J2EE server:** The runtime portion of a J2EE product. A J2EE server provides EJB and Web containers.

- **Enterprise JavaBeans (EJB) container:** Manages the execution of enterprise beans for J2EE applications. Enterprise Beans and their container run on the J2EE server.
- **Web container:** Manages the execution of JSP page and servlet components for J2EE applications. Web components and their container run on the J2EE server.
- **Application client container:** Manages the execution of application client components. Application clients and their container run on the client.
- **Applet container:** Manages the execution of applets. Consists of a Web browser and Java Plug-in running on the client together

Possible Questions

PART-B

(Each Question carries 6 Marks)

1. State and explain the advantages of java.
2. Discuss about JVM and Java byte code.
3. Explain J2EE multitier architecture with a neat sketch.
4. Give a detailed note on J2EE and J2SE.
5. Discuss about distributive systems in multi-tier architecture
6. Discuss about EJB tier implementation.
7. Explain the need of J2EE and why J2EE is important?
8. What are the classifications of client?
9. Give a detailed note on beginning of Java.
10. Explain the following

i) Web Tier Implementation ii) Client Tier Implementation

PART-C

(One Compulsory Question carries 10 Marks)

1. Discuss the concept of Distributive Systems in J2EE.
2. Discuss the concept of tier and how it is applied to J2EE architecture.
3. Write a program to create a sign in form using Servelts

J2EE (17CSP301)
Unit I -Multiple Choice Questions

S.no	Question	Option1	Option 2	Option 3	Option 4	Answer
1	The expansion of BCPL is ___________.	Basic Combined Programming Language	Beginners Combined Programming Language	Basic Control Programming Language	Beginners Control Programming Language	Basic Combined Programming Language
2	Programmers divide a program into functionality and create code segments called ____________.	programs	subprograms	macros	functions	functions
3	In the year ___________ the American National Standard Institute formally adopted a standard for the C Programming language.	1970	1980	1990	2000	1990
4	Java is ____________ programming language that uses classes to create instances of objects.	object based	object oriented	procedure oriented	knowledge based	object oriented
5	___________ is a routine that recovers spent memory without the programmer having to write code to free previously reserved memory.	Memory release	Garbage collection	Memory management	Garbage compaction	Garbage collection
6	Java ___________converts java source code into byte code that is executed by the Java Virtual machine.	interpreter	compiler	assembler	preprocessor	compiler
7	Java compiler generates ____________.	binary code	octal code	byte code	hexadecimal code	byte code
8	Small amount of data stored on the client is called ____________.	cookie	servlet	images	applet	cookie
9	An ___________ is a small program that can be efficiently downloaded over the internet and is executed by a java compatible browser.	cookie	servlet	images	applet	applet
10	Request and execution occur on the user's computer called ___________.	server	client	client and server	JVM	client

11	Embedded in the web page might be a reference to run a small java program called an ______________.	cookie	applet	image	servlet	applet
12	The ___________ reads the reference to the applet, then requests that the web server download the applet.	cookie	browser	servlet	applet	browser
13	Once the applet is received, the browser requests the ________ to execute the applet automatically without any additional interaction by the user.	server	client	client and server	JVM	JVM
14	___________ could not offer the dynamics demanded by internet users and corporations.	Static web pages	Dynamic web pages	Browsers	Applets	Static web pages
15	Java was developed by ____________.	IBM	Microsoft	Sun Microsystems	Oracle Corporation	Sun Microsystems
16	Features found in ____________ were adopted in Java by the Java development team.	C only	C++ only	C and C++	Visual C++	C and C++
17	Java is a ___________ programming language.	multiuser	multitasking	multithreaded	procedure oriented	multithreaded
18	A __________ is a process that can work independently from other processes and permit multiple access to the same program simultaneously.	macro	procedure	function	thread	thread
19	The original edition of Java is called ____________.	J2ME	J2SE	J2EE	Core Java	J2SE
20	A ___________ program is automatically translated into a java servlet.	Java	EJB	JSP	HTML	JSP
21	___________ interfaces between commercial DBMS products and Java.	API	EJB	JSP	XML	EJB
22	____________ contains the API used to create wireless java applications.	J2ME	J2SE	J2EE	EJB	J2SE

23	During the evolutionary process, the java development team included more interfaces and libraries as programmers demanded new APIs. These new features to the JDK were called ___________.	SDK	Java Bean	BDK	Extensions	Extensions
24	___________ consists of specifications and API for developing reusable server-side business components designed to run on applications servers.	Java	EJB	JSP	Servlets	EJB
25	___________ is a program that resides on the server	. Servlet	Cookie	Applet	JSP	. Servlet
26	___________ consists of specifications and APIs for developing reusable server-side business components designed to run on applications servers.	EJB	JSP	Servlets	Java	EJB
27	A ___________ bean retains data accumulated during a session with a client.	session servlet	stateful session	stateless session	JMS container	stateful session
28	A ____________ bean does not maintain any state between method calls.	session servlet	stateless session	stateful session	JMS container	stateless session
29	A message-driven bean is called by the _____________.	sessionservlet	JMS container	message-oriented middleware	API	JMS container
30	The core components of J2EE are _____________.	Java Beans	Java servlets and Java beans	Java servlets and JSPs	Java beans, Java servlets and JSPs	Java beans, Java servlets and JSPs
31	The expansion of CORBA is ___________.	Combined Object Request Basic Architecture	Common Object Request Broker Architecture	Combined Object Request Broker Architecture	Common Object Request Basic Architecture	Common Object Request Broker Architecture
32	The expansion of XDR is ____________.	Exchange Data Representation	External Data Representation	External Digital Representation	Exchange Digital Representation	External Data Representation
33	__________ are the internal software services.	servlets	functions	RPCs	JSPs	functions
34	__________ are the external software services.	servlets	functions	RPCs	JSPs	RPCs

35	In multi-tier architecture, each tier contains __________ that include software objects, DBMS or connectivity to legacy systems.	services	java programs	servlets	requests	services
36	____________ is a part of a tier that consists of a collection of classes or a program that performs a function to provide the services.	container	component	resource	service	component
37	A __________ is anything a component needs to provide a service.	container	component	resource	service	resource
38	A ___________ is a software that manages a component and provides a component with system services.	container	component	resource	service	container
39	J2EE is a __________ tier architecture.	2	3	4	5	3
40	__________ tiers can physically reside on the same JVM although each tier provides a different type of functionality to a J2EE application.	1	2	3	4	2
41	The __________ tier consists of programs that interact with the user.	client	web	EJB tier	EIS	client
42	The _________ provides internet functionality to a J2EE application.	client	web	EJB tier	EIS	web
43	The _________ tier contains the business logic for J2EE applications.	client	web	EJB tier	EIS	EJB tier
44	The ________ tier links a J2EE application to resources and legacy systems that are available on the corporate backbone network.	client	web	EJB tier	EIS	EIS
45	The __________ tier is the keystone to every J2EE application.	client	web	EJB tier	EIS	EJB tier

46	___________ are contained on the EJB server which is a distributed object server that works on the EJB tier.	servlets	EJB	JSP	client programs	EJB
47	It is on the ___________ where a J2EE application directly or indirectly interfaces with a variety of technologies including DBMS and mainframes.	servlets	EJB	JSP	client programs	client programs
48	There are ____________ components on the client tier.	2	3	4	5	2
49	A _______ is a component used by a web client that operates within the applet container, which is a java-enabled browser.	application client	applet client	servlet	JSP	applet client
50	A _________ is a java application that operates within the application client container, which is the Java 2 Runtime Environment Standard Edition.	application client	applet client	servlet	JSP	application client
51	A _________ has its own interface and is capable of accessing all the tiers in the multi-tier architecture.	application client	applet client	application	servlet	application
52	A ___________ is not considered as the component of the client tier.	application client	applet client	rich client	server	rich client
53	A ___________ can access any tier in the environment depending on the ACLs configuration using HTTP, SOAP, etc.	application client	applet client	rich client	servlet	rich client
54	Clients are classified into ____________ types.	2	3	4	5	5
55	A ___________ consists of software usually a browser that accesses resources located on the web tier.	web client	EJB client	EIS client	multitier client	web client
56	_____________ only accesses one or more EJB that are located on the EJBs tier rather than resources on the web tier.	web client	EJB client	EIS client	multitier client	EJB client

57	___________ are the interface between users and resources located on the EIS tier.	web client	EJB client	EIS client	multitier client	EIS client
58	A ___________ is a unique type of client because it is also a service that works on the web tier.	web client	EJB client	EIS client	web service peer	web service peer
59	____________ are conceptually similar to a web service peer.	web client	EJB client	EIS client	multitier client	multitier client
60	____________ are similar to web clients.	web client	EJB client	EIS client	multitier client	EJB client

UNIT II

SYLLABUS:

J2EE Database Concepts: Data – Database – Database Schema. **JDBC Objects**: Driver Types – Packages – JDBC Process – Database Connection – Statement Objects – Result Set – Meta Data

DATA

The term data means groups of information that represent the qualitative or quantitative attributes of a variable or set of variables. Data (plural of "datum") are typically the results of measurements and can be the basis of graphs, images, or observations of a set of variables. Data are often viewed as the lowest level of abstraction from which information and knowledge are derived. In computer science, data is anything in a form suitable for use with a computer. Data is often distinguished from programs. Data is a collection of facts, figures and statistics related to an object. Data can be processed to create useful information. Data is a valuable asset for an organization. Data can be used by the managers to perform effective and successful operations of management. It provides a view of past activities related to the rise and fall of an organization. It also enables the user to make better decision for future. Data is very useful for generating reports, graphs and statistics.

DATABASE

A database is an integrated collection of logically related records or files consolidated into a common pool that provides data for one or more multiple uses. One way of classifying databases involves the type of content, for example: bibliographic, full-text, numeric, and image. Software organizes the data in a database according to a database model. A number of database architectures exist. Many databases use a combination of strategies. Databases consist of software-based "containers" that are structured to collect and store information so users can retrieve, add, update or remove such information in an automatic fashion. Database programs are designed for users so that they can add or delete any information needed. The structure of a database is the table, which consists of rows and columns of information.

DATABASE SCHEMA

The schema of a database system is its structure described in a formal language supported by the database management system (DBMS). In a relational database, the schema defines the tables, the fields, relationships, views, indexes, packages, procedures, functions, queues, triggers, types, sequences, materialized views, synonyms, database links, directories, Java, XML schemas, and other elements. Schemas are generally stored in a data dictionary. Although a schema is defined in text database language, the term is often used to refer to a graphical depiction of the database structure.

Levels of database schema

- Conceptual schema, a map of concepts and their relationships.
- Logical schema, a map of entities and their attributes and relations
- Physical schema, a particular implementation of a logical schema
- Schema object, Oracle database object

Conceptual schema

A conceptual schema or conceptual data model is a map of concepts and their relationships. This describes the semantics of an organization and represents a series of assertions about its nature. Specifically, it describes the things of significance to an organization (entity classes), about which it is inclined to collect information, and characteristics of (attributes) and associations between pairs of those things of significance (relationships).

Logical schema

A Logical Schema is a data model of a specific problem domain expressed in terms of a particular data management technology.

Without being specific to a particular database management product, it is in terms of relational tables and columns, object-oriented classes, or XML tags. This is as opposed to a conceptual data model, which describes the semantics of an organization without reference to technology, or a physical data model, which describe the particular physical mechanisms used to capture data in a storage medium. The next step in creating a database, after the logical schema is produced, is to create the physical schema.

Physical Schema

Physical Schema is a term used in relation to data management. In the ANSI four-schema architecture, the internal schema was the view of data that involved data management technology. This was as opposed to the external schema that reflected the view of each person in the organization, or the conceptual schema that was the integration of a set of external schemas.

Schema Object

A schema object is a logical data storage structure. Schema objects do not have a one-to-one correspondence to physical files on disk that store their information. However, Oracle stores a schema object logically within a table space of the database. The data of each object is physically contained in one or more of the table space's data files. For some objects such as tables, indexes, and clusters, you can specify how much disk space Oracle allocates for the object within the table space's data files.

There is no relationship between schemas and table spaces: a table space can contain objects from different schemas, and the objects for a schema can be contained in different table spaces.

Associated with each database user is a schema. A schema is a collection of schema objects. Examples of schema objects include tables, views, sequences, synonyms, indexes, clusters, database links, snapshots, procedures, functions, and packages.

DATABASE AND PLATFORM PORTABILITY

Data connectivity architecture can either simplify or radically complicate portability among databases, database versions, and platforms. Ideally, data connectivity components should share a common architecture that makes it easy to change or upgrade the underlying database infrastructure. Most software companies and enterprise IT organizations must support more than one database platform – and more than one version of every platform they support. This can mean also managing a myriad of data connectivity methods, driver versions, and client library packages.

Adding a new database or even upgrading to a new version of the same database can create substantial development, integration, and testing work. For example, data connectivity components designed to work with only one database will handle BLOB/CLOB data types (large binary or character objects) differently than a component designed to work exclusively with another database. Developers will spend significant time and effort on additional coding and testing for each new database that they need to support. Standardizing and simplifying data connectivity architecture dramatically reduces the cost and complexity associated with supporting multiple database back ends. For independent software vendors in particular, this is a significant business priority.

INTRODUCTION TO JDBC

An application programming Interface (API) is a set of classes, methods and resources that programs can use to do their work. APIs exist for windowing systems, file systems, database systems, networking systems, and others. JDBC is a Java API for database connectivity that is part of the Java API developed by Sun Microsystems. JDBC provides Java developers with an industry standard API for database-independent connectivity between java applications and a wide range of relational database management systems such as oracle. Informix, Microsoft SQL Server and Sybase.

The API provides a call level interface to the database.

- Connect to a database
- Execute SQL statements to query your database
- Generate query results
- Perform updates, inserts and deletions
- Execute stored procedures

The following figure 2.6.1 shows the components of the JDBC model. In its simplest form, JDBC makes it possible to do these basic things: The Java application calls JDBC classes and interfaces to submit SQL statements and retrieve results.

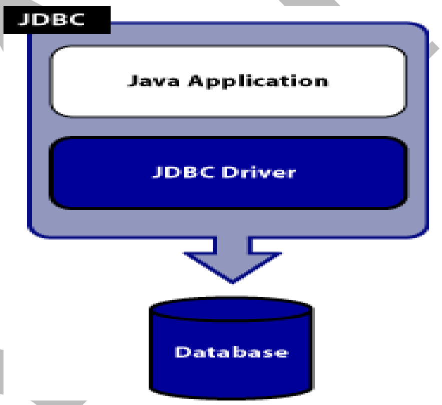


Figure 2.6.1 Components of the JDBC Model

The JDBC API is implemented through the JDBC driver. The JDBC Driver is a set of classes that implement the JDBC interfaces to process JDBC calls and return result sets to a Java application. The database (or data store) stores the data retrieved by the application using the JDBC Driver.

JDBC OBJECTS

The main objects of the JDBC API include:

- A Data Source object is used to establish connections. Although the Driver Manager can also be used to establish a connection, connecting through a Data Source object is the preferred method.
- A Connection object controls the connection to the database. An application can alter the behavior of a connection by invoking the methods associated with this object. An application uses the connection object to create statements.
- Statement, Prepared Statement, and Callable Statement objects are used for executing SQL statements. A Prepared Statement object is used when an application plans to reuse a statement multiple times. The application prepares the SQL it plans to use. Once prepared, the application can specify values for parameters in the prepared SQL statement. The statement can be executed multiple times with different parameter values specified for each execution. A Callable Statement is used to call stored procedures that return values. The Callable Statement has methods for retrieving the return values of the stored procedure
- A ResultSet object contains the results of a query. A ResultSet is returned to an application when a SQL query is executed by a statement object. The ResultSet object provides methods for iterating through the results of the query

BENEFITS OF JDBC

The benefits of using JDBC include the following:

- A developer only needs to know one API to access any relational database
- There is no need to rewrite code for different databases.
- There is no need to know the database vendor’s specific APIs
- It provides a standard API and is vendor independent
- Almost every database vendor has some sort of JDBC driver
- JDBC is part of the standard Java 2 platform

JDBC ARCHITECTURE

The JDBC API contains two major sets of interfaces: the first is the JDBC API for application writers, and the second is the lower-level JDBC driver API for driver writers. JDBC technology drivers fit into one of four categories. Applications and applets can access databases via the JDBC API using pure Java JDBC technology-based drivers, as shown in the Figure 2.9.2 below

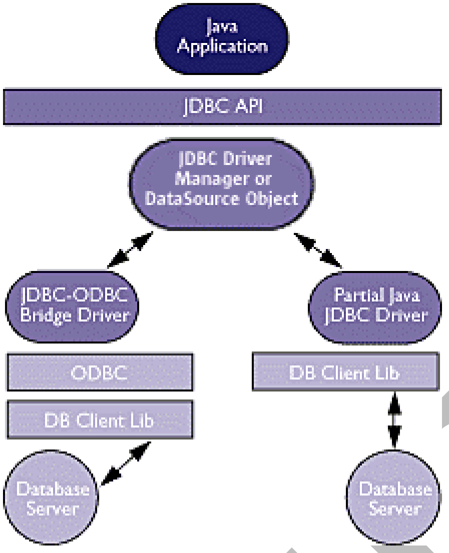
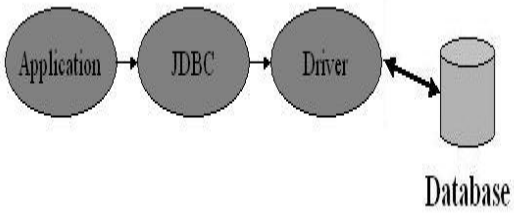


Figure 2.9.2 JDBC connectivity using ODBC drivers

Left side, Type 4: Direct-to-Database Pure Java Driver
This style of driver converts JDBC calls into the network protocol used directly by DBMS, allowing a direct call from the client machine to the DBMS server and providing a practical solution for intranet access.

Right side, Type 3: Pure Java Driver for Database Middleware
This style of driver translates JDBC calls into the middleware vendor's protocol, which is then translated to a DBMS protocol by a middleware server. The middleware provides connectivity to many different databases.

Java application calls the JDBC library. JDBC loads a driver which talks to the database. We can change database engines without changing database code. The Figure 2.9.1 shows the architecture of JDBC.



.

Figure 2.9.1 JDBC Architecture

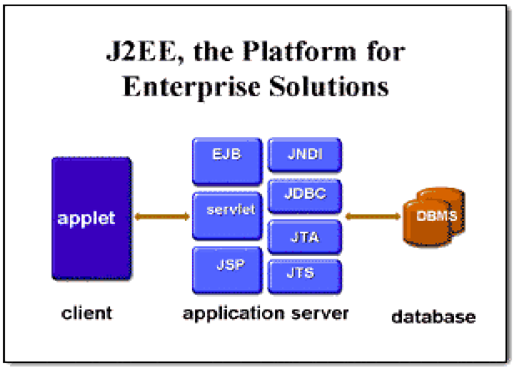
Left side, Type 1: JDBC-ODBC Bridge plus ODBC Driver
This combination provides JDBC access via ODBC drivers. ODBC binary code and in many cases, database client code must be loaded on each client machine that uses a JDBC-ODBC Bridge. Sun provides a JDBC-ODBC Bridge driver, which is appropriate for experimental use and for situations in which no other driver is available.

Right side, Type 2: A native API partly Java technology-enabled driver This type of driver converts JDBC calls into calls on the client API for Oracle, Sybase, Informix, DB2, or other DBMS. Note that, like the bridge driver, this style of driver requires that some binary code be loaded on each client machine.

JDBC IN J2EE

As a core part of the Java 2 Platform, the JDBC API is available anywhere that the platform is. This means that your applications can truly write database applications once and access data anywhere. The JDBC API is included in the Java 2 Platform, Standard Edition (J2SE) and the Java 2 Platform, Enterprise Edition (J2EE), providing server-side functionality for industrial strength scalability.

An example of a J2EE based architecture that includes a JDBC implementation:



Requirements

Software: The Java 2 Platform (either the Java 2 SDK, Standard Edition, or the Java 2 SDK, Enterprise Edition), an SQL database, and a JDBC technology-based driver for that database.

Hardware: Same as for the Java 2 Platform.

TWO-TIER AND THREE-TIER MODELS

The JDBC API supports both two-tier and three-tier models for database access Fig 2.11.1 illustrate two-tier architecture for data access.

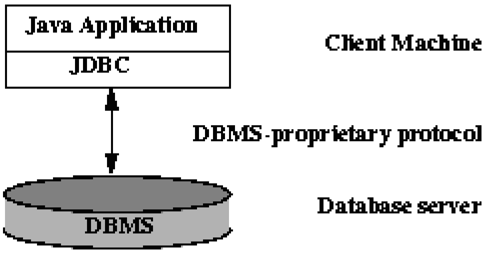


Fig 2.11.1 Two tier architecture for data access

In the two-tier model, a Java applet or application talks directly to the data source. This requires a JDBC driver that can communicate with the particular data source being accessed. A user's commands are delivered to the database or other data source, and the results of those statements are sent back to the user. The data source may be located on another machine to which the user is connected via a network. This is referred to as a client/server configuration, with the user's machine as the client, and the machine housing the data source as the server. The network can be an intranet, which, for example, connects employees within a corporation, or it can be the Internet**.**

In the three-tier model, commands are sent to a "middle tier" of services, which then sends the commands to the data source. The data source processes the commands and sends the results back to the middle tier, which then sends them to the user. MIS directors find the three-tier model very attractive because the middle tier makes it possible to maintain control over access and the kinds of updates that can be made to corporate data. Another advantage is that it simplifies the deployment of applications. Finally, in many cases, the three-tier architecture can provide performance advantages. In Figure 2.11.2: illustrates three-tier architecture for database access.

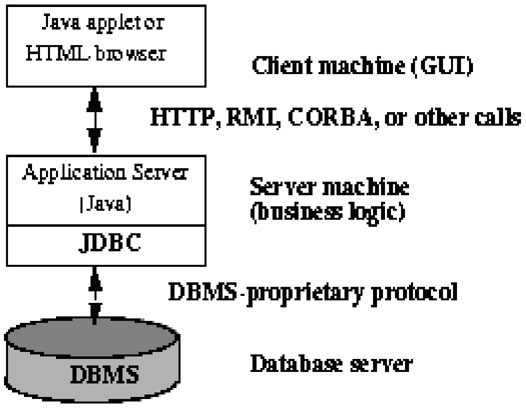


Fig 2.11.2 Three-tier architecture for database access

Until recently, the middle tier has typically been written in languages such as C or C++, which offer fast performance. However, with the introduction of optimizing compilers that translate Java bytecode into efficient machine-specific code and technologies such as Enterprise JavaBeans, the Java platform is fast becoming the standard platform for middle-tier development. This is a big plus, making it possible to take advantage of Java's robustness, multithreading, and security features.

With enterprises increasingly using the Java programming language for writing server code, the JDBC API is being used more and more in the middle tier of three-tier architecture. Some of the features that make JDBC a server technology are its support for connection pooling, distributed transactions, and disconnected rowsets.

JDBC DRIVER TYPES

JDBC technology-based drivers generally fit into one of four categories. In Figure 2.12.1 shows various driver implementation possibilities

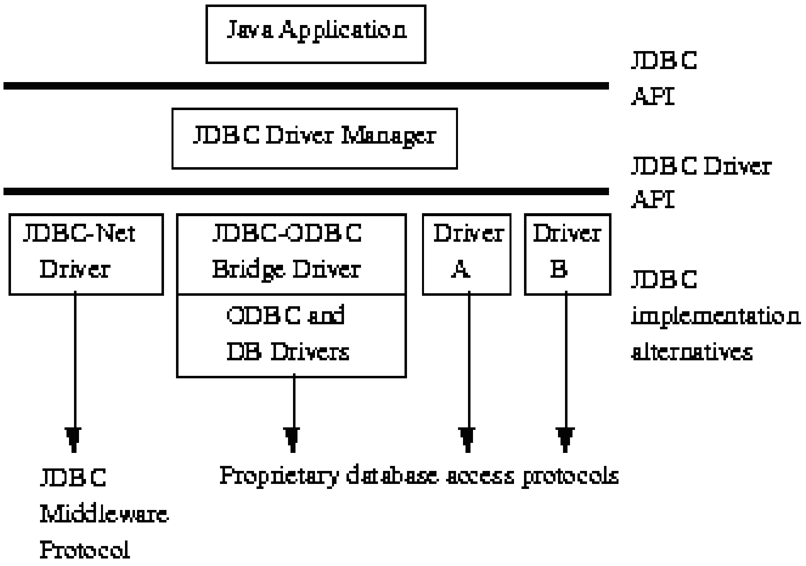


Figure 2.12.1 Various driver implementation possibilities

JDBC technology-based drivers generally fit into one of four categories. In Figure 2.12.1 shows various driver implementation possibilities

JDBC Drivers Types: Sun has defined four JDBC driver types. These are:

Type 1: JDBC-ODBC Bridge Driver

The first type of JDBC driver is JDBC-ODBC Bridge which provides JDBC access to any ODBC complaint databases through ODBC drivers. Sun's JDBC-ODBC bridge is example of type 1 driver.

Type 2: Native -API Partly - Java Driver

Type 2 drivers are developed using native code libraries, which were originally designed for accessing the database through C/C++. Here a thin code of Java wrap around the native code and converts JDBC commands to DBMS-specific native calls.

Type 3: JDBC-Net Pure Java Driver

Type 3 drivers are three-tier solutions. This type of driver communicates to a middleware component which in turn connects to database and provide database connectivity.

Type 4: Native-Protocol Pure Java Driver

Type 4 drivers are entirely written in Java that communicates directly with vendor's database through socket connection. Here no translation or middleware layer, are required which improves performance tremendously

<u>JDBC-ODBC Bridge driver</u> (Type 1 JDBC Driver)

The Type 1 driver translates all JDBC calls into ODBC calls and sends them to the ODBC driver. ODBC is a generic API. The JDBC-ODBC Bridge driver is recommended only for experimental use or when no other alternative is available. In figure 2.12.2 Type 1 JDBC – ODBC Bridge.

Advantage

The JDBC-ODBC Bridge allows access to almost any database, since the database's ODBC drivers are already available.

Disadvantages

1. Since the Bridge driver is not written fully in Java, Type1 drivers are not portable
2. A performance issue is seen as a JDBC call goes through the bridge to the ODBC driver, then to the database, and this applies even in the reverse process. They are the slowest of all driver types.
3. The client system requires the ODBC Installation to use the driver and Not good for the Web.

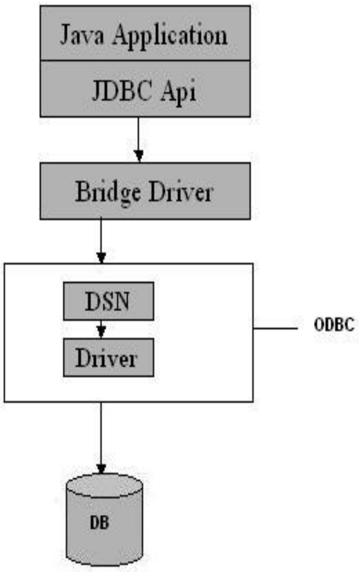


Figure 2.12.2 Type 1: JDBC-ODBC Bridge

<u>Native-API/partly Java driver</u> (Type 2 JDBC Driver)

The distinctive characteristic of type 2 jdbc drivers is that Type 2 drivers convert JDBC calls into database-specific calls i.e. this driver is specific to a particular database. Some distinctive characteristic of type 2 jdbc drivers are shown below. Example: Oracle will have oracle native api.

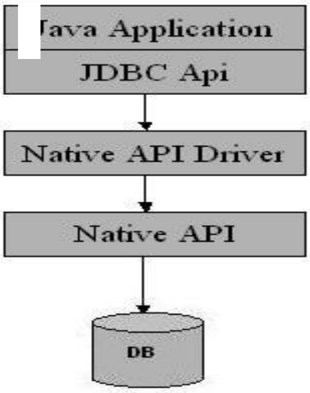


Figure 2.12.3 Type 2: Native API/ Partly Java Driver

<u>Advantage</u>

The distinctive characteristic of type 2 jdbc drivers are that they are typically offer better performance than the JDBC-ODBC Bridge as the layers of communication (tiers) are less than that of Type 1 and also it uses Native api which is Database specific.

Disadvantage

1. Native API must be installed in the Client System and hence type 2 drivers cannot be used for the Internet.
2. Like Type 1 drivers, it's not written in Java Language which forms a portability issue.
3. If we change the Database we have to change the native api as it is specific to a database
4. Mostly obsolete now
5. Usually not thread safe.

2.12.4 All Java/Net-protocol driver (Type 3 JDBC Driver)

Type 3 database requests are passed through the network to the middle-tier server. The middle-tier then translates the request to the database. If the middle-tier server can in turn use Type1, Type 2 or Type 4 drivers.

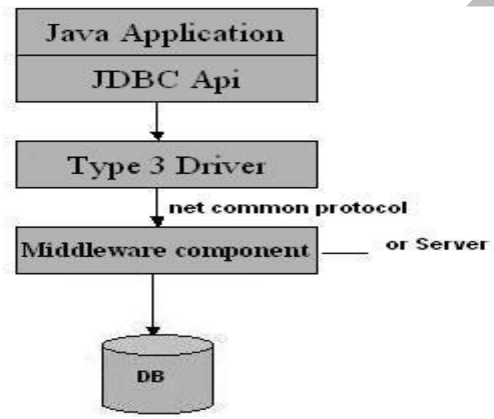


Figure 2.12.4 **Type 3: All Java/ Net-Protocol Driver**

Advantage

1. This driver is server-based, so there is no need for any vendor database library to be present on client machines.
2. This driver is fully written in Java and hence Portable. It is suitable for the web
3. There are many opportunities to optimize portability, performance, and scalability.
4. The net protocol can be designed to make the client JDBC driver very small and fast to load.
5. The type 3 driver typically provides support for features such as caching (connections, query results, and so on), load balancing, and advanced system administration such as logging and auditing.
6. This driver is very flexible allows access to multiple databases using one driver
7. They are the most efficient amongst all driver types.

Disadvantage

It requires another server application to install and maintain. Traversing the recordset may take longer, since the data comes through the backend server

Native-protocol/all-Java driver (Type 4 JDBC Driver)

The Type 4 uses java networking libraries to communicate directly with the database server.

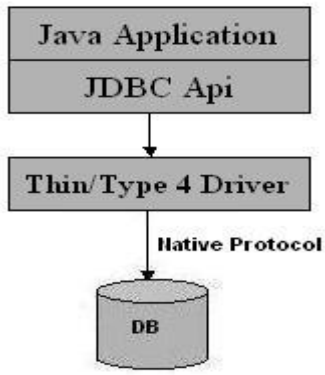


Figure 2.12.5 Type 4: Native-protocol/all-Java driver

Advantage

1. The major benefit of using a type 4 jdbc drivers are that they are completely written in Java to achieve platform independence and eliminate deployment administration issues. It is most suitable for the web.
2. Number of translation layers is very less i.e. type 4 JDBC drivers don't have to translate database requests to ODBC or a native connectivity interface or to pass the request on to another server, performance is typically quite good
3. You don’t need to install special software on the client or server. Further, these drivers can be downloaded dynamically.

Disadvantage

With type 4 drivers, the user needs a different driver for each database.

JDBC PACKAGE

The purpose of the JDBC package is to provide vendor-neutral access to relational databases. The implementation differences of the various databases used are abstracted from the user through the use of the JDBC API. Though the specification does not indicate that the API is to be used solely for relational databases, historically it has been used primarily for relational database access.

The developers of the JDBC API specification have tried to keep the API as simple as possible so that it can be a foundation upon which other APIs are built. For instance, the connector API can be implemented on top of an existing JDBC API using appropriate resource adapters. JDBC is composed of a number of interfaces. These interfaces are implemented by driver developers. The API is implemented by either a vendor or a third party to create a JDBC driver.

The Type 4 JDBC driver is considered the best driver to use for two reasons. One reason is that since the driver has been written completely in Java, it is extremely portable. Another reason is that the driver is not required to map JDBC calls to corresponding native CLI calls. This avoids the overhead of mapping logic required by the Type 1 or Type 2 driver, or the overhead of communicating with middleware required by the Type 3 driver.

Such improvements in efficiency should allow the driver to execute faster than the other types of JDBC drivers.

JDBC 2.0 API

The JDBC 2.0 API includes the complete JDBC API, which includes both core and Optional Package API, and provides industrial-strength database computing capabilities. It is not, however, limited to SQL databases; the JDBC 2.0 API makes it possible to access data from virtually any data source with a tabular format.

The JDBC 2.0 API includes two packages:

- java.sql package--the JDBC 2.0 core API
    - JDBC API included in the JDKTM 1.1 release (previously called JDBC 1.2). This API is compatible with any driver that uses JDBC technology.
    - JDBC API included in the Java 2 SDK, Standard Edition, version 1.2 (called the JDBC 2.0 core API). This API includes the JDBC 1.2 API and adds many new features.
- javax.sql package--the JDBC 2.0 Optional Package API. This package extends the functionality of the JDBC API from a client-side API to a server-side API, and it is an essential part of Java2 SDK, Enterprise Edition technology.
- Being an Optional Package, it is not included in the Java 2 Platform SDK, Standard Edition, version 1.2, but it is readily available from various sources.
    - Information about the JDBC 2.0 Optional Package API is available from the JDBC web page. The javax.sql package may also be downloaded from this web site.
    - Driver vendors may include the javax.sql package with their products.
    - The Java 2 SDK, Enterprise Edition, includes many Optional Package APIs, including the JDBC 2.0 Optional Package.

The java.sql Package

The java.sql package contains the entire JDBC API that sends SQL (Structured Query Language) statements to relational databases and retrieves the results of executing those SQL statements. Figure 18-1 shows the class hierarchy of this package. The JDBC 1.0 API became part of the core Java API in Java 1.1. The JDBC 2.0 API supports a variety of new features and is part of the Java 2 platform.

The Driver interface represents a specific JDBC implementation for a particular database system. Connection represents a connection to a database. The Statement, PreparedStatement, and CallableStatement interfaces support the execution of various kinds of SQL statements. ResultSet is a set of results returned by the database in response to a SQL query. The ResultSetMetaData interface provides metadata about a result set, while DatabaseMetaData provides metadata about the database as a whole.

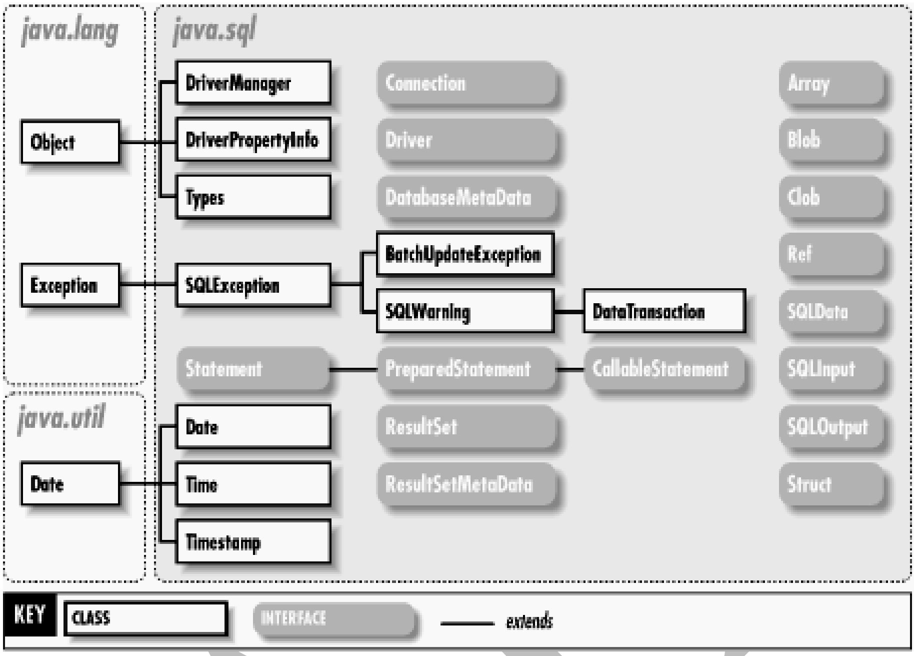


Figure 2.13.2 The java.sql Package

The java.sql package contains API for the following:

- Making a connection with a data source
  - DriverManager class
  - Driver interface
  - DriverPropertyInfo class
  - Connection interface
- Custom mapping an SQL user-defined type to a class in the Java programming language
  - SQLData interface
  - SQLInput interface
  - SQLOutput interface
  - Providing information about the database and the columns of a ResultSet object
  - DatabaseMetaData interface
  - ResultSetMetaData interface
- Throwing exceptions
  - SQLException thrown by most methods when there is a problem accessing data and by some methods for other reasons

  - SQLWarning thrown to indicate a warning
  - DataTruncation thrown to indicate that data may have been truncated
  - BatchUpdateException thrown to indicate that not all commands in a batch update executed successfully
- Providing security
  - SQLPermission interface

Package javax.sql

Provides the API for server side data source access and processing from the Java programming language

Table 2.13.3.1 Interface Summary

Interface Summary	
ConnectionEventListener	A ConnectionEventListener is an object that registers to receive events generated by a PooledConnection.
ConnectionPoolDataSource	A ConnectionPoolDataSource object is a factory for PooledConnection objects.
DataSource	A DataSource object is a factory for Connection objects.
PooledConnection	A PooledConnection object is a connection object that provides hooks for connection pool management.
RowSet	The RowSet interface adds support to the JDBC API for the JavaBeans(TM) component model.
RowSetInternal	A rowset object presents itself to a reader or writer as an instance of RowSetInternal.
RowSetListener	The RowSetListener interface is implemented by a component that wants to be notified when a significant event happens in the life of a RowSet
RowSetMetaData	The RowSetMetaData interface extends ResultSetMetaData with methods that allow a metadata object to be initialized.
RowSetReader	An object implementing the RowSetReader interface may be registered with a RowSet object that supports the reader/writer paradigm.
RowSetWriter	An object that implements the RowSetWriter interface may be registered with a RowSet object that supports the reader/writer paradigm.
XAConnection	An XAConnection object provides support for distributed transactions.
XADataSource	A factory for XAConnection objects.

Table 2.13.3.2 Class Summary

Class Summary	
ConnectionEvent	The ConnectionEvent class provides information about the source of a connection related event.
RowSetEvent	A RowSetEvent is generated when something important happens in the life of a rowset, like when a column value changes.

Package javax.sql Description

Provides the API for server side data source access and processing from the Java programming language. This package supplements the java.sql package and, as of the version 1.4 releases, is included in the Java 2 SDK, Standard Edition. It remains an essential part of the Java 2 SDK, Enterprise Edition (J2EE).

The javax.sql package provides for the following:

1. The DataSource interface as an alternative to the DriverManager for establishing a connection with a data source
2. Connection pooling
3. Distributed transactions
4. Rowsets

Applications use the DataSource and RowSet APIs directly, but the connection pooling and distributed transaction APIs are used internally by the middle-tier infrastructure.

Using a DataSource Object to Make a Connection

The javax.sql package provides the preferred way to make a connection with a data source. The DriverManager class, the original mechanism, is still valid, and code using it will continue to run. However, the newer DataSource mechanism is preferred because it offers many advantages over the DriverManager mechanism.

These are the main advantages of using a DataSource object to make a connection:

- Applications do not need to hard code a driver class.
- Changes can be made to a data source's properties, which means that it is not necessary to make changes in application code when something about the data source or driver changes.

Connection pooling and distributed transactions are available through a DataSource object that is implemented to work with the middle-tier infrastructure. Connections made through the DriverManager do not have connection pooling or distributed transaction capabilities

Driver vendors provide DataSource implementations. A particular DataSource object represents a particular physical data source, and each connection the DataSource object creates is a connection to that physical data source. A logical name for the data source is registered with a naming service that uses the Java Naming and Directory Interface (JNDI) API, usually by a system administrator or someone performing the duties of a system administrator. An application can retrieve the DataSource object it wants by doing a lookup on the logical name that has been registered for it. The application can then use the DataSource object to create a connection to the physical data source it represents.

A DataSource object can be implemented to work with the middle tier infrastructure so that the connections it produces will be pooled for reuse. An application that uses such a DataSource implementation will automatically get a connection that participates in connection pooling. A DataSource object can also be implemented to work with the middle tier infrastructure so that the connections it produces can be used for distributed transactions without any special coding.

Connection Pooling

Connections made via a DataSource object that is implemented to work with a middle tier connection pool manager will participate in connection pooling. This can improve performance dramatically because creating new connections is very expensive. Connection pooling allows a connection to be used and reused, thus cutting down substantially on the number of new connections that need to be created.

Connection pooling is totally transparent. It is done automatically in the middle tier of a J2EE configuration, so from an application's viewpoint, no change in code is required. An application simply uses the DataSource.getConnection method to get the pooled connection and uses it the same way it uses any Connection object.

The classes and interfaces used for connection pooling are:

- ConnectionPoolDataSource
- PooledConnection
- Connection Event
- ConnectionEventListener

The connection pool manager, a facility in the middle tier of a three-tier architecture, uses these classes and interfaces behind the scenes. When a ConnectionPoolDataSource object is called on to create a PooledConnection object, the connection pool manager will register as a ConnectionEventListener object with the new PooledConnection object. When the connection is closed or there is an error, the connection pool manager (being a listener) gets a notification that includes a ConnectionEvent object.

Distributed Transactions

As with pooled connections, connections made via a DataSource object that is implemented to work with the middle tier infrastructure may participate in distributed transactions. This gives an application the ability to involve data sources on multiple servers in a single transaction.

The classes and interfaces used for distributed transactions are:

- XADataSource
- XAConnection

These interfaces are used by the transaction manager; an application does not use them directly. The XAConnection interface is derived from the PooledConnection interface, so what applies to a pooled connection also applies to a connection that is part of a distributed transaction. A transaction manager in the middle tier handles everything transparently. The only change in application code is that an application cannot do anything that would interfere with the transaction manager's handling of the transaction. Specifically, an application cannot call the methods Connection.commit or Connection.rollback, and it cannot set the connection to be in auto-commit mode (that is, it cannot call Connection.setAutoCommit(true)).

An application does not need to do anything special to participate in a distributed transaction. It simply creates connections to the data sources it wants to use via the DataSource.getConnection method, just as it normally does. The transaction manager manages the transaction behind the scenes. The XADataSource interface creates XAConnection objects, and each XAConnection object creates an XAResource object that the transaction manager uses to manage the connection.

Rowsets

The RowSet interface works with various other classes and interfaces behind the scenes. These can be grouped into three categories.

Event Notification

RowSetListener:A RowSet object is a JavaBeans component because it has properties and participates in the JavaBeans event notification mechanism. The RowSetListener interface is implemented by a component that wants to be notified about events that occur to a particular RowSet object. Such a component registers itself as a listener with a rowset via the RowSet.addRowSetListener method.

When the RowSet object changes one of its rows, changes all of it rows, or moves its cursor, it also notifies each listener that is registered with it. The listener reacts by carrying out its implementation of the notification method called on it.

RowSetEvent: As part of its internal notification process, a RowSet object creates an instance of RowSetEvent and passes it to the listener. The listener can use this RowSetEvent object to find out which rowset had the event.

Metadata

RowSetMetaData: This interface, derived from the ResultSetMetaData interface, provides information about the columns in a RowSet object. An application can use RowSetMetaData methods to find out how many columns the rowset contains and what kind of data each column can contain. The RowSetMetaData interface provides methods for setting the information about columns, but an application would not normally use these methods. When an application calls the RowSet method execute, the RowSet object will contain a new set of rows, and its RowSetMetaData object will have been internally updated to contain information about the new columns.

3. **The Reader/Writer Facility**

A RowSet object that implements the RowSetInternal interface ca n call on the RowSetReader object associated with it to populate itself with data. It can also call on the RowSetWriter object associated with it to write any changes to its rows back to the data source from which it originally got the rows. A rowset that remains connected to its data source does not need to use a reader and writer because it can simply operate on the data source directly.

RowSetInternal: By implementing the RowSetInternal interface, a RowSet object gets access to its internal state and is able to call on its reader and writer. A rowset keeps track of the values in its current rows and of the values that immediately preceded the current ones, referred to as the *original* values. A rowset also keeps track of (1) the parameters that have been set for its command and (2) the connection that was passed to it, if any. A rowset uses the RowSetInternal methods behind the scenes to get access to this information. An application does not normally invoke these methods directly.

RowSetReader: A disconnected RowSet object that has implemented the RowSetInternal interface can call on its reader (the RowSetReader object associated with it) to populate it with data. When an application calls the RowSet.execute method, that method calls on the rowset's reader to do much of the work. Implementations can vary widely, but generally a reader makes a connection to the data source, reads data from the data source and populates the rowset with it, and closes the connection. A reader may also update the RowSetMetaData object for its rowset. The rowset's internal state is also updated, either by the reader or directly by the method RowSet.execute.

RowSetWriter: A disconnected RowSet object that has implemented the RowSetInternal interface can call on its writer (the RowSetWriter object associated with it) to write changes back to the underlying data source.

Implementations may vary widely, but generally, a writer will do the following:

- Make a connection to the data source
- Check to see whether there is a conflict, that is, whether a value that has been changed in the rowset has also been changed in the data source
- Write the new values to the data source if there is no conflict
- Close the connection

The RowSet interface may be implemented in any number of ways, and anyone may write an implementation. Developers are encouraged to use their imaginations in coming up with new ways to use rowsets

JDBC PROCESS

JDBC Data structure

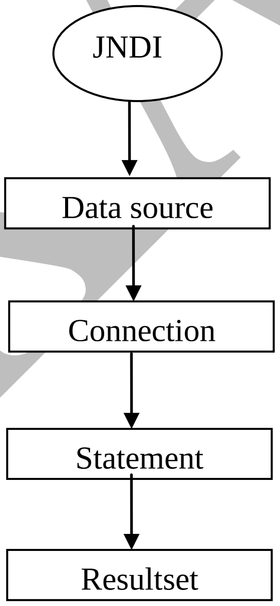


Figure 2.14.1 Data structure of JDBC

Steps involved in JDBC Process:

1. Load the driver

2. Define the Connection URL

3. Establish the Connection

4. Create a Statement object

5. Execute a query

6. Process the results

7. Close the connection

JDBC: Details of Process

1. **Load the driver**

```
try
{
Class.forName("connect.microsoft.MicrosoftDriver");
Class.forName("oracle.jdbc.driver.OracleDriver");
}
catch { ClassNotFoundException cnfe)
{
System.out.println("Error loading driver: " cnfe);
}
```

2. Define the Connection URL

```
String host = "dbhost.yourcompany.com";
String dbName = "someName";
int port = 1234;
String oracleURL = "jdbc:oracle:thin:@" + host + ":" + port + ":" +_
dbName;
String sybaseURL = "jdbc:sybase:Tds:" + host +":" + port + ":" + "?SERVICENAME=" +
dbName;
```

3. Establish the Connection

```
String username = "jay_debesee";
String password = "secret";
Connection connection =_
DriverManager.getConnection(oracleURL,username, password);
DatabaseMetaData dbMetaData = connection.getMetaData();
String productName = dbMetaData.getDatabaseProductName();
```

System.out.println("Database: " + productName);

String productVersion = dbMetaData.getDatabaseProductVersion();

System.out.println("Version: " + productVersion);

4. Create a Statement

Statement statement = connection.createStatement();

5. Execute a Query

String query = "SELECT col1, col2, col3 FROM sometable";

ResultSet resultSet = statement.executeQuery(query);

– To modify the database, use executeUpdate, supplying a string that uses UPDATE, INSERT, or DELETE

– Use setQueryTimeout to specify a maximum delay to wait for results

6. Process the Result

```
while(resultSet.next()) {
System.out.println(resultSet.getString(1) + " " +
resultSet.getString(2) + " " +
resultSet.getString(3));
}
```

First column has index 1, not 0

– ResultSet provides various get*Xxx* methods that take a column index *or column name* and returns the data

– You can also access result meta data (column names, etc.)

7. **Close the Connection**

connection.close();

– Since opening a connection is expensive, postpone this step if additional database operations are expected

Statement Objects

Through the Statement object, SQL statements are sent to the database.

– Three types of statement objects are available:

• **Statement** – For executing a simple SQL statement

• **PreparedStatement** – For executing a precompiled SQL statement passing in parameters

• **CallableStatement** – For executing a database stored procedure

Statement Methods

- executeQuery

– Executes the SQL query and returns the data in a table (ResultSet)

– The resulting table may be empty but never null

ResultSet results = statement.executeQuery("SELECT a, b FROM_ table");

• executeUpdate

– Used to execute for INSERT, UPDATE, or DELETE, SQL statements

– The return is the number of rows that were affected in the database

– Supports Data Definition Language (DDL) statements

CREATE TABLE, DROP TABLE and ALTER TABLE

int rows = statement.executeUpdate("DELETE FROM EMPLOYEES" + _ "WHERE STATUS=0");

- **execute**

– Generic method for executing stored procedures and prepared statements

– Rarely used (for multiple return result sets)

– The statement execution may or may not return a ResultSet (use statement.getResultSet). If the return value is true, two or more result sets were produced

• **getMaxRows/setMaxRows**

– Determines the maximum number of rows a ResultSet may contain

– Unless explicitly set, the number of rows is unlimited (return value of 0)

• **getQueryTimeout/setQueryTimeout**

– Specifies the amount of a time a driver will wait for a statement to complete before throwing a SQLException

RESULTSET

ResultSet and Cursors

The rows that satisfy a particular query are called the result set. The number of rows returned in a result set can be zero or more. A user can access the data in a result set using a cursor one row at a time from top to bottom. A cursor can be thought of as a pointer to the rows of the result set that has the ability to keep track of which row is currently being accessed. The JDBC API supports a cursor to move both forward and backward and also allowing it to move to a specified row or to a row whose position is relative to another row.

Types of Result Sets

The ResultSet interface provides methods for retrieving and manipulating the results of executed queries, and ResultSet objects can have different functionality and characteristics. These characteristics are result set type, result set concurrency, and cursor hold ability.

The type of a ResultSet object determines the level of its functionality in two areas: the ways in which the cursor can be manipulated, and how concurrent changes made to the underlying data source are reflected by the ResultSet object.

The sensitivity of the ResultSet object is determined by one of three different ResultSet types:

TYPE_FORWARD_ONLY — the result set is not scrollable i.e. the cursor moves only forward, from before the first row to after the last row.

TYPE_SCROLL_INSENSITIVE — the result set is scrollable; its cursor can move both forward and backward relative to the current position, and it can move to an absolute position.

TYPE_SCROLL_SENSITIVE — the result set is scrollable; its cursor can move both forward and backward relative to the current position, and it can move to an absolute position.

Before you can take advantage of these features, however, you need to create a scrollable ResultSet object. The following line of code illustrates one way to create a scrollable ResultSet object:

```
 Statement stmt = con.createStatement(ResultSet.TYPE_SCROLL_SENSITIVE,
ResultSet.CONCUR_READ_ONLY);
ResultSet srs = stmt.executeQuery(".....");
```

The first argument is one of three constants added to the ResultSet API to indicate the type of a ResultSet object: TYPE_FORWARD_ONLY, TYPE_SCROLL_INSENSITIVE, and TYPE_SCROLL_SENSITIVE. The second argument is one of two ResultSet constants for specifying whether a result set is read-only or updatable: CONCUR_READ_ONLY and CONCUR__UPDATABLE. If you do not specify any constants for the type and updatability of a ResultSet object, you will automatically get one that is TYPE_FORWARD_ONLY and CONCUR_READ_ONLY.

Result Set Methods

When a ResultSet object is first created, the cursor is positioned before the first row. To move the cursor, you can use the following methods:

- next() - moves the cursor forward one row. Returns true if the cursor is now positioned on a row and false if the cursor is positioned after the last row.
- previous() - moves the cursor backwards one row. Returns true if the cursor is now positioned on a row and false if the cursor is positioned before the first row.
- first() - moves the cursor to the first row in the ResultSet object. Returns true if the cursor is now positioned on the first row and false if the ResultSet object does not contain any rows.
- last() - moves the cursor to the last row in the ResultSet object. Returns true if the cursor is now positioned on the last row and false if the ResultSet object does not contain any rows.
- beforeFirst() - positions the cursor at the start of the ResultSet object, before the first row. If the ResultSet object does not contain any rows, this method has no effect.
- afterLast() - positions the cursor at the end of the ResultSet object, after the last row. If the ResultSet object does not contain any rows, this method has no effect.
- relative(int rows) - moves the cursor relative to its current position.
- absolute(int n) - positions the cursor on the n-th row of the ResultSet object

METADATA

Databases store user data, and they also store information about the database itself. Most DBMSs have a set of system tables, which list tables in the database, column names in each table, primary keys, foreign keys, stored procedures, and so forth. Each DBMS has its own functions for getting information about table layouts and database features. JDBC provides the interface DatabaseMetaData, which a driver writer must implement so that its methods return information about the driver and/or DBMS for which the driver is written. For example, a large number of methods return whether or not the driver supports a particular functionality. This interface gives users and tools a standardized way to get metadata. In general, developers writing tools and drivers are the ones most likely to be concerned with metadata.

Simple JDBC Code

The basic process for a single data retrieval operation using JDBC would be as follows.

- a JDBC driver would be loaded;
- a database Connection object would be created from using the DriverManager (using the database driver loaded in the first step);

- a Statement object would be created using the Connection object;
- a SQL Select statement would be executed using the Statement object, and a ResultSet would be returned;
- the ResultSet would be used to step through (or iterate through) the rows returned and examine the data.

The following JDBC code sample demonstrates this sequence of calls.

```
import java.sql.* ;
public class JDBCSample {
public static void main( String args[]) {
String connectionURL =
"jdbc:postgresql://localhost:5432/movies;user=java;password=samples";
// Change the connection string according to your db, ip, username and password
  try {     // Load the Driver class.
    Class.forName("org.postgresql.Driver");
    // If you are using any other database then load the right driver here.
    //Create the connection using the static getConnection method
    Connection con = DriverManager.getConnection (connectionURL);
    //Create a Statement class to execute the SQL statement
    Statement stmt = con.createStatement();
    //Execute the SQL statement and get the results in a Resultset
    ResultSet rs = stmd.executeQuery("select moviename, releasedate from movies");
    // Iterate through the ResultSet, displaying two values
    // for each row using the getString method
    while (rs.next())
     System.out.println("Name= " + rs.getString("moviename") + " Date= " +
rs.getString("releasedate");
}
catch (SQLException e) {
    e.printStackTrace(); }
catch (Exception e) {     e.printStackTrace();
}
finally {     // Close the connection
    con.close(); } } }
```

POSSIBLE QUESTIONS

PART-B

(Each Question carries 6 Marks)

1. List out the different types of Keys in a database.
2. Explain JDBC Driver Types
3. Elaborate the steps to create a database schema.
4. Discuss about reading, scrollable, updateable resultset with a program.
5. Describe the steps needed to execute a SQL query using JDBC.
6. Explain J2EE database packages.
7. Discuss about Normalization process with example.
8. Briefly discuss about the Statement objects with examples
9. Explain the following:

   i) Define data, database, and table
10. Explain about Database connection

PART-C

(One Compulsory Question carries 10 Marks)

1. Discuss the Working Process of JDBC.
2. What do mean by Meta data? Explain.
3. Write a java program to connect to an access database and display the contents.
4. Explain in detail about different type of statements with examples.

J2EE (17CSP301)
Unit II-Multiple Choice Questions

S.no	Question	Option 1	Option 2	Option 3	Option 4	Answer
1	A __________ is a collection of data.	field	record	database	DBMS	database
2	A database is managed by ____________.	SQL	DBMS	JAVA	J2EE	DBMS
3	____________ refers to an atomic unit.	field	data	record	DBMS	data
4	A ____________ is a component of a database that contains data in the form of rows and columns.	tuple	table	record	attribute	table
5	A ___________ is a document that defines all components of a database.	SQL	database schema	table	file	database schema
6	The best way to identify attributes of an entity is by analyzing __________ of the entity.	instances	fields	data	records	instances
7	The ___________ describes the number of characters used to store values of the attribute.	attribute range	attribute size	attribute format	attribute type	attribute size
8	The __________ uniquely identifies the attribute from other attributes of the same entity.	attribute name	attribute size	attribute format	attribute type	attribute name
9	A ___________ is nearly identical to the data type of a column in a table.	attribute name	attribute size	attribute format	attribute type	attribute type
10	The ___________ is minimum and maximum values that can be assigned to an attribute.	attribute name	attribute size	attribute format	attribute range	attribute range
11	The ___________ is the value that is automatically assigned to the attribute.	attribute name	attribute size	attribute definition value	attribute type	attribute definition value
12	The __________ consists of the way in which an attribute appears in the existing system.	attribute format	attribute size	attribute definition value	attribute type	attribute format
13	The ___________ identifies the origin of the attribute value.	attribute format	attribute source	attribute definition value	attribute type	attribute source

14	A ____________ is free form text that is used to describe an attribute.	acceptable values	required values	comments	attribute values	comments
15	_____________ must be reduced to data elements.	values	attributes	comments	information	attributes
16	The unique name given to the data element is called ____________.	data name	data type	data size	attribute	data name
17	A _____________ describes the kind of values that are associated with the data.	data name	data type	data size	attribute	data type
18	The ____________ is the maximum number of characters required to represent values of the data.	data name	data type	data size	attribute	data size
19	The nature of the data provide a hint to the ____________.	data name	data type	data size	attribute	data name
20	_____________ should have as few characters as possible to identify the data.	data name	data type	data size	attribute	data name
21	A _____________ can be abbreviated using components of the name.	data name	data type	data size	attribute	data name
22	A _____________ describes the characteristics of data associated with a data element.	data name	data type	data size	attribute	data type
23	_____________ data stores alphabetical characters and punctuations.	Character	Alpha	Alphanumeric	Numeric	Character
24	____________ data stores only alphabetical characters.	Character	Alpha	Alphanumeric	Numeric	Alpha
25	_____________ data stores alphabetical characters, punctuations, and numbers.	Character	Alpha	Alphanumeric	Numeric	Alphanumeric
26	_____________ data stores numbers only.	Character	Alpha	Alphanumeric	Numeric	Numeric
27	___________ data stores date and time values.	Character	Alpha	Date/Time	Numeric	Date/Time
28	_____________ data stores one of two values – yes or no.	Character	Alpha	Alphanumeric	Logical	Logical
29	_____________ data stores large text fields, images, and other binary data.	Character	Alpha	Alphanumeric	Large Object	Alphanumeric

30	___________ is the process of organizing data elements into related groups to minimize redundant data and to assure data integrity.	Transaction	Normalization	Grouping	Creation	Normalization
31	There are ____________ normal forms.	2	3	4	5	5
32	A common way to organize data elements into __________ is to first assemble a list of all data elements.	groups	text	objects	class	groups
33	A __________ requires the information to be atomic.	1 NF	2 NF	3 NF	4 NF	1 NF
34	The ___________ requires the data to be in the first normal form.	1 NF	2 NF	3 NF	4 NF	2 NF
35	The ___________ requires that data elements to be in second normal form.	1 NF	2 NF	3 NF	4 NF	3 NF
36	A ___________ is a data element that uniquely identifies a row of data elements within a group.	primary key	secondary key	tertiary key	foreign key	primary key
37	A __________ occurs when data depends on other data such as when nonkey data is dependent on a primary key.	redundancy	normalization	functional dependency	transitive dependency	functional dependency
38	A _________ is a functional dependency between two or more nonkey data elements.	redundancy	normalization	functional dependency	transitive dependency	transitive dependency
39	A ___________ is a primary key of another group used to draw a relationship between two groups of data elements.	primary key	secondary key	tertiary key	foreign key	foreign key
40	The relationship between primary keys and foreign keys of data groups is called ___________.	functional dependency	referential integrity	transitive dependency	operational dependency	referential integrity
41	There are __________ types of JDBC drivers.	2	3	4	5	4
42	The JDBC process is divided into ___________ routines.	2	3	4	5	5

43	The ___________ method is used to load the JDBC driver.	Class.forName()	Results.next()	System.out.println()	DB.createStatement()	Class.forName()
44	The _________ method returns a connection interface that is used throughout the process to reference the database.	Class.forName()	Results.next()	DriverManager.getConnection()	DB.createStatement()	DriverManager.getConnection()
45	The _________ method is used to create a statement object.	Class.forName()	Results.next()	DriverManager.getConnection()	Connect.createStatement()	Connect.createStatement()
46	The _________ method is called to terminate the statement.arameter.	Class.forName()	Results.next()	Db.close()	Connect.createStatement()	Db.close()
47	The _________ method of the ResultSet object is used to copy the value of a specified column in the current row of the ResultSet to a string object.	Class.forName()	Results.next()	Db.close()	getString()	getString()
48	The URL consists of _________ parts.	2	3	4	5	3
49	The statement object contains the method, which is passed the query as an argument.	Results.next()	Class.forName()	executeQuery()	Db.createStatement()	executeQuery()
50	The __________ method of the statement object is used when there may be multiple results returned.	Results.next()	Class.forName()	executeQuery()	execute()	execute()
51	The __________ method of the connection object is called to return a statement object.	createStatement()	Class.forName()	executeQuery()	execute()	createStatement()
52	The __________ method of the connection object is called to return the PreparedStatement object	createStatement()	Class.forName()	executeQuery()	preparedStatement()	preparedStatement()
53	The _________ object is used to call a stored procedure from within a J2EE object.	statement	preparedstatement	callableStatement	ResultSet	callableStatement
54	The CallableStatement object used __________ types of parameter when calling a stored procedure.	2	3	4	5	3

55	The __________ parameter contains any data that needs to be passed to the stored procedure.processed by the CPU?	IN	OUT	INOUT	IO	IN
56	The ____________ object is used whenever a J2EE component needs to immediately execute a query without first having the query compiled.	statement	preparedstatement	callableStatement	ResultSet	statement
57	A SQL query can be preempted and executed using the ____________ object.	statement	preparedstatement	callableStatement	ResultSet	preparedstatement
58	The __________ object contains methods that are used to copy data from the ResultSet into a java collection object or variable for further processing.	statement	preparedstatement	callableStatement	ResultSet	ResultSet
59	The ____________ parameter is a single parameter that is used to both pass information to the stored procedure and retrieve information from a stored procedure.	IN	OUT	INOUT	IO	INOUT
60	The ___________ parameter contains a value returned by the stored procedures. The future generation of computers?	IN	OUT	INOUT	IO	OUT

UNIT 3
JAVA SERVLETS

SYLLABUS:

Java Servlets: Benefits – Anatomy – Reading Data from Client –Reading HTTP Request Headers – Sending Data to client – Working with Cookies.

INTRODUCTION

Java Servlets are making headlines these days, claiming to solve many of the problems associated with CGI and proprietary server APIs. Servlets provide the means through which Web Applications can be written in Java. What are Web Applications? Web Applications allow a web site to be dynamic rather than straight static pages. In other words, Servlets transform a web site with plain text and images into a rich interactive environment for the user. Servlets are snippets of Java programs which run inside a Servlet Container. A Servlet Container is much like a Web Server which handles user requests and generates responses. Servlet Container is different from a Web Server because it can not only serve requests for static content like HTML page, GIF images, etc., it can also contain Java Servlets and JSP pages to generate dynamic response. Servlet Container is responsible for loading and maintaining the lifecycle of the a Java Servlet. Servlet Container can be used standalone or more often used in conjunction with a Web server. Example of a Servlet Container is Tomcat and that of Web Server is Apache.

OVERVIEW OF SERVLET

Servlets are Java programming language objects that dynamically process requests and construct responses. The **Java Servlet API** allows a software developer to add dynamic content to a Web server using the Java platform. The generated content is commonly HTML, but may be other data such as XML. Servlets are the Java counterpart to non-Java dynamic Web content technologies such as PHP, CGI and ASP.NET, and as such some find it easier to think of them as 'Java scripts'. Servlets can maintain state across many server transactions by using HTTP cookies, session variables or URL rewriting.

The servlet API, contained in the Java package hierarchy javax.servlet, defines the expected interactions of a Web container and a servlet. A Web container is essentially the component of a Web server that interacts with the servlets. The Web container is responsible for managing the lifecycle of servlets, mapping a URL to a particular servlet and ensuring that the URL requester has the correct access rights.

A Servlet is an object that receives a request and generates a response based on that request. The basic servlet package defines Java objects to represent servlet requests and responses, as well as objects to reflect the servlet's configuration parameters and execution environment. The package javax.servlet.http defines HTTP-specific subclasses of the generic servlet elements, including session management objects that track multiple requests and responses between the Web server and a client. Servlets may be packaged in a WAR file as a Web application.

Servlets are server side components. These components can be run on any platform or any server due to the core java technology which is used to implement them. Servlets augment the functionality of a web application. They are dynamically loaded by the server's Java runtime environment when needed. On receiving an incoming request from the client, the web server/container initiates the required servlet. The servlet processes the client request and sends the response back to the server/container, which is routed to the client.

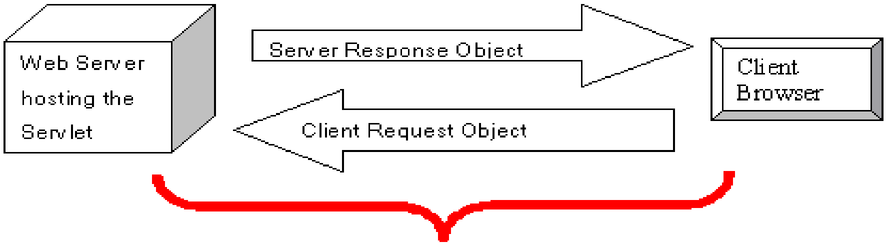


Figure 3.2.1: HTTP request response model.

Web based Client/server interaction uses the HTTP (hypertext transfer protocol). HTTP is a stateless protocol based on a request and response model with a small, finite number of request methods like GET, POST, HEAD, OPTIONS, PUT, TRACE, DELETE, CONNECT, etc. The response contains the status of the response and meta information describing the response. Most of the servlet-based web applications are built around the framework of the HTTP request/response model (Figure 3.2.1).

CGI VERSUS SERVLET

When a CGI program (or script) is invoked what typically happens is that a new process is spawned to handle the request. This process is external to that of the web server and as such you have the overhead of creating a new process and context switching etc. If you have many requests for a CGI script then you can imagine the consequences! Of course this is a generalization and there are wrappers for CGI that allow them to run in the same process space as the web server. Java Servlets on the other hand actually run inside the web server (or Servlet engine).

The developer writes the Servlet classes compiles them and places them somewhere that the server can locate them. The first time a Servlet is requested it is loaded into memory and cached. From then on the same Servlet instance is used with different requests being handled by different threads. The below table 3.3.1 depicts the difference between CGI and Servlet

Table 3.3.1 Difference Between CGI And Servlet

CGI	Servlet
Written in C, C++, Visual Basic and Perl	Written in Java
Difficult to maintain, non-scalable, non-manage	Powerful, reliable, and efficient
Prone to security problems of programming	Improves scalability, reusability (

language	component based)
Resource Intensive and inefficient	Leverages Built-in security of Java programming language
Platform and application-specific	Platform independent and portable

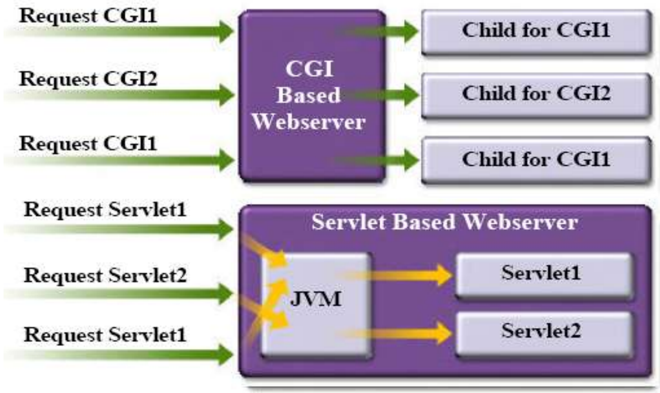


Figure:3.3.1 CGI vs. Servlet

The figure 3.3.1 shows difference between CGI and servlet-based model. In CGI, for every HTTP request, a new process has to be created while in servlet model, it is the thread that gets created in the same Java VM (Virtual Machine) and that thread can stay there for servicing other requests. Also in CGI, every time a new request comes, the program image of CGI has to be loaded in memory, which results in many redundant load of the same program. In the case of Servlet, a single class is loaded for serving many requests. This results in efficient memory usage.

BENEFITS OF JAVA SERVLETS

There are so many servers-side software development are available, But Java servlet technology is superior to others. Server-side Java technologies give us platform independence, efficiency, access to other enterprise Java APIs, reusability, and modularity.

As Java Servlets enjoy the same benefits of a regular Java program, they are platform-independent and can be ported across various platforms that support Java. As compared with predecessors of Java servlet technology, Java Servlets performs quite exceedingly well. Due to its distinct and unique multi-threading capability, it is possible for hundreds and even thousands of users can access the same servlet simultaneously without affecting the load of the web server.

Since Servlets are an extension of the Java platform, they can access all of the Java APIs. A Java servlet can send and receive email, invoke methods of objects using Java RMI or CORBA, object directory information using the JNDI package, make use of an Enterprise Java Bean (EJB), or may other part of the ever-growing platform.

Software reusability is the essential for software development. As Servlets are web components, they can be reused easily. Also Java, as an Object Oriented language, helps to encapsulate shared functionality to be reused. Thus Java Servlets are reusable.

When developing server-side software applications, its size becomes larger and automatically complexity intrudes in. It is always helpful if such a large application gets broken into discreet modules that are each responsible for a specific task. This divide and conquer principle helps to maintain and understand easily. Java Servlets provide a way to modularize user application.

Advantages of Servlets

1. No CGI limitations
2. Abundant third-party tools and Web servers supporting Servlet
3. Access to entire family of Java APIs
4. Reliable, better performance and scalability
5. Platform and server independent Secure
6. Most servers allow automatic reloading of Servlet's by administrative action.

SERVLET REQUEST AND RESPONSE

There are three different players in figure 3.5.1. They are browser, web server, and servlet container. In many cases, a web server and a servlet container are running in a same machine even in a single virtual machine. So they are not really distinguished in many cases. The role of the web server is to receive HTTP request and then passes it to the web container or servlet container which then creates Java objects that represent “HTTP request” and a “session” and then dispatches the request to the servlet by invoking service() method defined in the servlet

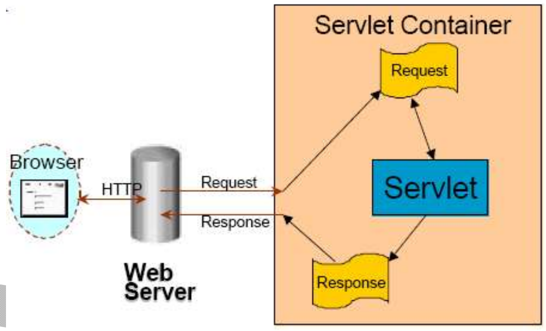


Fig 3.5.1 Servlet Request and Response

And once the servlet handles the request, it creates a HTTP response, which is then sent to the client through the web server.

- HTTPServletRequest object
  - Information about an HTTP request
    - Headers
    - Query String
    - Session
    - Cookies
- HTTPServletResponse object
  - Used for formatting an HTTP response

- Headers
- Status codes
- Cookies

First Servlet:

```
import javax.servlet.*;
import javax.servlet.http.*;
import java.io.*;
Public class HelloServlet extends HttpServlet
{
public void doGet(HttpServletRequest request,
HttpServletResponse response) throws ServletException, IOException {
response.setContentType("text/html");
PrintWriter out = response.getWriter();
out.println("<title>First Servlet</title>");
out.println("<big>Hello Code Camp!</big>");
}
}
```

So in this simple servlet program, the request comes in the form of HTTPServetRequest object and response is created in the form of HttpServletResponse object

SERVLET CLASSES AND INTERFACES

Servlet Request Interface

public interface **ServletRequest:** Defines an object to provide client request information to a servlet. The servlet container creates a ServletRequest object and passes it as an argument to the servlet's service method. A ServletRequest object provides data including parameter name and values, attributes, and an input stream. Interfaces that extend ServletRequest can provide additional protocol specific data (for example, HTTP data is provided by **HttpServletRequest**)

ServletResponse Interface

public interface **ServletResponse:** Defines an object to assist a servlet in sending a response to the client. The servlet container creates a ServletResponse object and passes it as an argument to the servlet's service method.

A subclass of HttpServlet must override at least one method, usually one of these:

- doGet, if the servlet supports HTTP GET requests
- doPost, for HTTP POST requests
- doPut, for HTTP PUT requests
- doDelete, for HTTP DELETE requests
- init and destroy, to manage resources that are held for the life of the servlet

Web clients usually activate a servlet in one of two ways:

- Get – Sends data as part of a URL

http://rmyers.com/servlet/Hello?name="john"

- Post – Sends data down the data stream following the request

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
public class HelloWorldServlet extends HttpServlet
{
public void doGet(HttpServletRequest req,
HttpServletResponse res)
throws ServletException, IOException
{
res.setContentType("text/html");
PrintWriter out = res.getWriter();
out.println("<HTML>");
out.println("<HEAD>");
out.println("<TITLE>Hello World Sample </TITLE>");
out.println("</HEAD>");
out.println("<BODY>");
out.println("<CENTER><H1>Hello World!</H1></CENTER>");
out.println("</BODY>");
out.println("</HTML>");
out.close();
}
}
```

JAVA SERVLET ANATOMY AND LIFE CYCLE

Anatomy of Java Servlets:

init()

– Invoked once when the servlet is first instantiated

– Perform any set-up in this method and Setting up a database connection

destroy()

– Invoked before servlet instance is removed.

– Perform any clean-up and Closing a previously created database connection

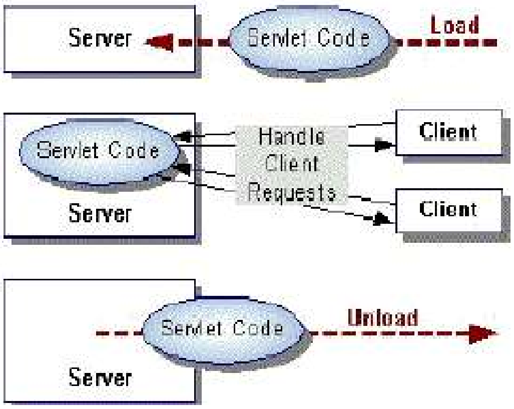


Figure 3.7.1.1 Function of doGet()

doGet()

– the doGet() function is called when the servlet is called via an HTTP GET

doPost()

– the doPost() function is called when the servlet is called via an HTTP POST POST
good way to get input from HTML forms

J2EE Overview

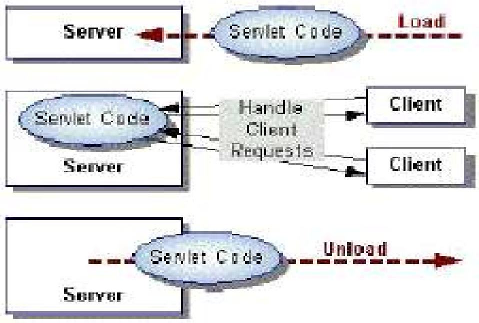


Figure 3.7.1.2 Function of doPost()

Life Cycle of Java Servlets:

The life cycle of a servlet is controlled by servlet-container in which the servlet has been deployed. When a HTTP request is mapped to a servlet, the container performs the following steps.

- If an instance of the servlet does not exist, the Web container
  - Loads the servlet class
  - Creates an instance of the servlet class
  - Initializes the servlet instance by calling the init() method
- Invokes the service method, passing HttpServletRequest and HttpServletResponse objects as parameters.

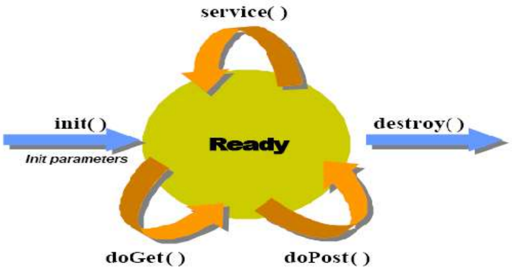


Figure 3.7.2.1 Methods used in Java Servlets

The init() method gets called once when a servlet instance is created for the first time. And then service() method gets called every time there comes a new request. Now service() method in turn calls doGet() or doPost() methods for incoming HTTP requests. And finally when the servlet instance gets removed, the destroy() method gets called. So init() and destroy() methods get called only once while service(),

Java Servlets

Self- instr uctional

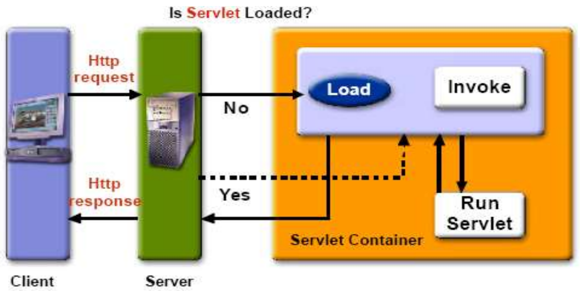


Figure 3.7.2.2 Httprequest and Httpresponse

doGet(), and doPost() methods are called a number of times depending on how many HTTP requests are received. As it was mentioned before, init () and destroy () methods are called only once, init() at the time service instance is created while destroy() gets called at the time servlet instance gets removed. And init() can be used to perform some set up operation such as setting up a database connection and destroy() method is used to perform any clean up, for example, removing a previously created database connection.

Example for init():

```
public class CatalogServlet extends HttpServlet {
private BookDB bookDB;
// Perform any one-time operation for the servlet,
// like getting database connection object.
// Note: In this example, database connection object is assumed
// to be created via other means (via life cycle event mechanism)
// and saved in ServletContext object. This is to share a same
// database connection object among multiple servlets.
public void init() throws ServletException {
bookDB = (BookDB)getServletContext().
getAttribute("bookDB");
if (bookDB == null) throw new
UnavailableException("Couldn't get database.");
}
... }
```

Example: destroy()

```
public class CatalogServlet extends HttpServlet {
private BookDB bookDB;
public void init() throws ServletException {
bookDB = (BookDB)getServletContext().
getAttribute("bookDB");
if (bookDB == null) throw new
UnavailableException("Couldn't get database.");
}
public void destroy() {
bookDB = null;
}
```

This is destroy example code again from CatalogServlet code. Here destroy() method nulling the local variable that contains the reference to database connection.

service() methods take generic requests and responses:

- service(ServletRequest request, ServletResponse response)
- doGet() or doPost() take HTTP requests and responses:
  - doGet(HttpServletRequest request, HttpServletResponse response)

doPost(HttpServletRequest request, HttpServletResponse response)

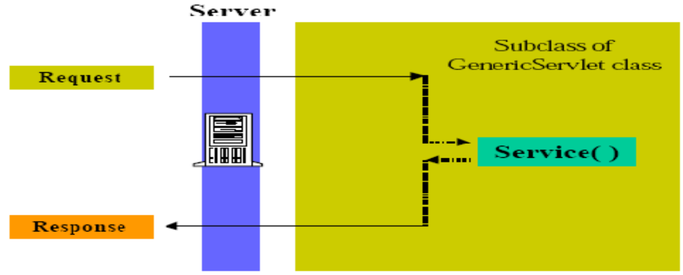


Figure 3.7.2.3 using service() method to invoke GenericServlet class

This Figure 3.7.2.3 shows how service () method of a subclass of GenericServlet class is invoked.

doGet() and doPost() Methods

Using doGet() and doPost() it is possible to do the following functions:

- Can able to extract client sent information such as user-entered parameter values that were sent as query string.
- To set and get attributes to and from scope objects.
- Perform some business logic or access the database.
- Optionally include or forward your requests to other web components.
- Populate HTTP response message and then send it to client.

Example: Simple doGet()

```
import javax.servlet.*;
import javax.servlet.http.*;
import java.io.*;
Public class HelloServlet extends HttpServlet {
public void doGet(HttpServletRequest request,
HttpServletResponse response)
throws ServletException, IOException {
// Just send back a simple HTTP response
response.setContentType("text/html");
PrintWriter out = response.getWriter();
out.println("<title>First Servlet</title>");
out.println("<big>Hello J2EE Programmers! </big>");
}
}
```

This is a very simple example code of doGet() method. In this example, a simple HTTP response message is created and then sent back to client

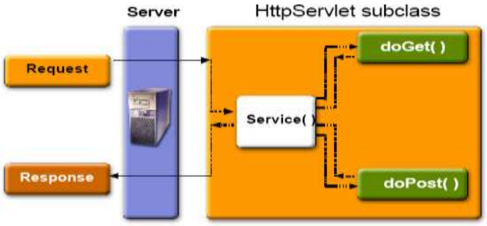


Figure 3.7.2.3 HttpServlet subclass

READING DATA FROM A CLIENT

A Client uses either the GET or POST Method to pass information to a java servlet. The doGet() or doPost() netgid us called in the Java Servlet depending on the method used by the client.

Data sent by a client is read into java servlet by calling the getParameters() method of the HttpservletRequest object that instantiated in the argument list of the doGet() and dopost() methods. The getParameters() method requires one argument, which is the name of the parameter that contains the data sent by the client. The getParameters() method returns a String object. The String object contains the value of the parameter, if the client assigns a value to the parameter. An empty string object is returned if the client didn't assign a value to the parameter. Also, a null is returned if the parameter isn't received from the client.

A HTML form can contain a set of check boxes or other form objects that have the same data name but different values. This means that data received from a client might have multiple occurrences of the same parameter name.

The user can read a set of parameters that have the same name by calling the getParameterValues() method. The getParameterValues() method has one argument which is the name of the parameter, and returns an array of string objects. Each element of the array contains a value of the set of parameters. The getParameterValues( ) method returns a null if data received from the client doesn't contain the parameter named in the argument.

User can retrieve all the parameters by calling the getParameterNames() method. The getParameterNames() method does not require an argument and returns an Enumeration. Parameter names appear in any order and can be cast to String object and used with the getParameter() and getParameterValues() methods.

Figure conatins an HTML form that prompts a user to enter their name , when the user selects the Submit button, the browser calls the URL /servlet/HelloServlet Java Servlet and sends the username as data. Figure illustrates the HelloServlet.class Java Servlet that reads data sent by this form. In this example the getParameter() method returns a string that is assigned to the email String object called email. The value of the email String object is then returned to the browser in the form of an HTML page.

```
<HTML>
<HEAD><TITLE>Greetings Form</TITLE></HEAD>
<BODY>
<FORM METHOD=GET ACTION="/servlet/HelloServlet">
What is your name?
<INPUT TYPE=TEXT NAME=username SIZE=20>
<INPUT TYPE=SUBMIT VALUE="Introduce Yourself">
</FORM>
```

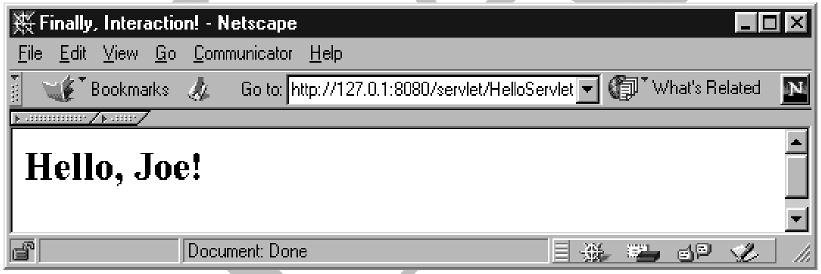
```
</BODY>
</HTML>
```

This form submits a form variable named username to the URL /servlet/HelloServlet.
The HelloServlet itself does little more than create an output stream, read the username form variable, and print a nice greeting for the user.

Here’s the code:

```
import javax.servlet.*;
import javax.servlet.http.*;
import java.io.*;
public class HelloServlet extends HttpServlet {
public void doGet(HttpServletRequest req, HttpServletResponse resp)
throws ServletException, IOException {
String name;
name= req.getParameter("username");
resp.setContentType("text/html");
PrintWriter out = resp.getWriter();
out.println("<HTML>");
out.println("<HEAD><TITLE>Finally, interaction!</TITLE></HEAD>");
out.println("<BODY><H1>Hello, " + name+"!</H1>");
out.println("</BODY></HTML>");
}
}
```

Result:


Finally, Interaction! - Netscape
File Edit View Go Communicator Help
Bookmarks
Go to: http://127.0.1:8080/servlet/HelloServlet
What's Related
Document: Done

Hello, Joe!

READING HTTP REQUEST HEADERS

When an HTTP client (e.g. a browser) sends a request, it is required to supply a request line (usually GET or POST). If it wants to, it can also send a number of headers, all of which are optional except for Content-Length, which is required only for POST requests. Here are the most common headers:

- Accept The MIME types the browser prefers.
- Accept-Charset The character set the browser expects.
- Accept-Encoding The types of data encodings (such as gzip) the browser knows how to decode. Servlets can explicitly check for gzip support and return gzipped HTML pages to browsers that support them, setting the Content-Encoding response header to

indicate that they are gzipped. In many cases, this can reduce page download times by a factor of five or ten.

- Accept-Language The language the browser is expecting, in case the server has versions in more than one language.
- Authorization Authorization info, usually in response to a WWW-Authenticate header from the server
- Connection Use persistent connection? If a servlet gets a Keep-Alive value here, or gets a request line indicating HTTP 1.1 (where persistent connections are the default), it may be able to take advantage of persistent connections, saving significant time for Web pages that include several small pieces (images or applet classes). To do this, it needs to send a Content-Length header in the response, which is most easily accomplished by writing into a ByteArrayOutputStream, then looking up the size just before writing it out.
- Content-Length (for POST messages, how much data is attached)
- Cookie (one of the most important headers; see separate section in this tutorial on handling cookies)
- From (email address of requester; only used by Web spiders and other custom clients, not by browsers)
- Host (host and port as listed in the original URL)
- If-Modified-Since (only return documents newer than this, otherwise send a 304 "Not Modified" response)
- Pragma (the no-cache value indicates that the server should return a fresh document, even if it is a proxy with a local copy)
- Referer (the URL of the page containing the link the user followed to get to current page)
- User-Agent (type of browser, useful if servlet is returning browser-specific content)
- UA-Pixels, UA-Color, UA-OS, UA-CPU (nonstandard headers sent by some Internet Explorer versions, indicating screen size, color depth, operating system, and cpu type used by the browser's system)

SENDING DATA TO A CLIENT

A java Servlet responds to a client request by reading client data and the HTTP request headers, then processing information based on the nature of the request. For example, a client request for information about merchandise in an online product catalog requires the Java Servlet to search the product database to retrieve product information and then format the product information into a web page which is returned to the client.

There are two ways in which a java Servlet replied to a client request. These are by sending information to the response stream and by sending information in the HTTP response header.

The HTTP response header is similar to the HTTP request header except the contents of the HTTP response header are generated by the web server that responds to the client’s request. Information is sent to the response stream by creating an instance of the PrintWriter object and then using the println() method to transmit the information to the client.

An Http response header contains a status line, response headers, and a blank line, followed by the document. There are three components to the status line these are the HTTP version number, a status code and a brief message associated with the status code.
example :

```
HTTP/1.1 200 OK
Content-type : text/plain
My response
```

In the above example The HTTP Version number is 1.1 and the status code is 200, indicating that everything is fine with the request that was received from the client. OK is the message that is associated with the status code. This example contains HTTP response Header, which is Content-Type that identifies the document Mime type as plain text. The document contains the expression My response.

HTTP Headers form the core of an HTTP request, and are very important in an HTTP response. They define various characteristics of the data that is requested or the data that has been provided. The headers are separated from the request or response body by a blank line. HTTP headers can be near-arbitrary strings, but only some are commonly understood.

An HTTP servlet can return three kinds of things to the client: a single status code, any number of HTTP headers, and a response body. A status code is an integer value that describes, as you would expect, the status of the response. The status code can indicate success or failure, or it can tell the client software to take further action to finish the request. The numerical status code is often accompanied by a "reason phrase" that describes the status in prose better understood by a human. Usually, a status code works behind the scenes and is interpreted by the browser software. Sometimes, especially when things go wrong, a browser may show the status code to the user. The most famous status code is probably the "404 Not Found" code, sent by a web server when it cannot locate a requested URL.

The response body is the main content of the response. For an HTML page, the response body is the HTML itself. For a graphic, the response body contains the bytes that make up the image. A response body can be of any type and of any length; the client knows what to expect by reading and interpreting the HTTP headers in the response.

A generic servlet is much simpler than an HTTP servlet--it returns only a response body to its client. It's possible, however, for a subclass of GenericServlet to present an API that divides this single response body into a more elaborate structure, giving the appearance of returning multiple items. In fact, this is exactly what HTTP servlets do. At the lowest level, a web server sends its entire response as a stream of bytes to the client. Any methods that set status codes or headers are abstractions above that.

It's important to understand this because even though a servlet programmer doesn't have to know the details of the HTTP protocol, the protocol does affect the order in which a servlet can call its methods. Specifically, the HTTP protocol specifies that the status code and headers must be sent *before* the response body. A servlet, therefore, should be careful to always set its status codes and headers before returning any of its response body. Some servers, including the Java Web Server, internally buffer some length of a servlets response body.

A Java servlet can write to the HTTP response header by calling the setStatus() method of the HttpServletResponse objkect. The setStatus() method requires one argument, which is an integer that represents the status code. The table 3.10.1 contains a list of HTTP status codes. The argument can take the form of the int itself or a predefined constant that represents the int. The setStatus() method automatically inserts the status code and the short message associated with the status code into the HTTP response header.

For example, the following statement writes the status code 100 to the HTTP response header informing the client to send a follow-up request:

Response.setSatus(100);

There are two methods that developes use in a Java servlet to write the HTTP response header. These are sendError() and sendRedirect(). Both of these methods throw an IOException exception and return a void. An Http response header is used to give the client directions on how to process the response document.

Table HTTP1.1 Status Codes

Status Code	**Reason Phrase**	**Description**
100	Continue	Client should continue sending its request. This is a special status code; see below for details.
200	OK	Generic successful request message response. This is the code sent most often when a request is filled normally.
202	Accepted	The request was accepted by the server but has not yet been processed. This is an intentionally “non-commital” response that does not tell the client whether or not the request will be carried out; the client determines the eventual disposition of the request in some unspecified way. It is used only in special circumstances
204	No Content	The request was successful, but the server has determined that it does not need to return to the client an entity body.
205	Reset Content	The request was successful; the server is telling the client that it should reset the document from which the request was generated so that a duplicate request is not sent. This code is intended for use with forms
301	Moved Permanently	The resource requested has been moved to a new URL permanently. Any future requests for this resource should use the new URL.
302	Found	The resource requested is temporarily using a different URL. The client should continue to use the original URL. See code 307.
303	See Other	This code is the same as 302, except that the client should make a GET request for the new URI, regardless of the original request method

304	Not Modified	The client sent a conditional *GET* request, but the resource has not been modified since the specified date/time, so the server has not sent it.
305	Use Proxy	This code allows origin servers to redirect requests through a caching proxy. The proxy's address is given in the Location header
307	Temporary Redirect	The resource is temporarily located at a different URL than the one the client specified. 307 was created to clear up some confusion related to 302 that occurred in earlier versions of HTTP
400	Bad Request	Generic response when the request cannot be understood or carried out due to a problem on the client's end. Bad syntax in the client request
401	Unauthorized	The client is not authorized to access the resource. Often returned if an attempt is made to access a resource protected by a password or some other means without the appropriate credentials
403	Forbidden	The request has been disallowed by the server. This is a generic "no way" response that is not related to authorization. For example, if the maintainer of Web site blocks access to it from a particular client, any requests from that client will result in a 403 reply.
404	Not Found	The most common HTTP error message, returned when the server cannot locate the requested resource. Usually occurs due to either the server having moved/removed the resource, or the client giving an invalid URL (misspellings being the most common cause.)
405	Method Not Allowed	The requested method is not allowed for the specified resource. The response includes an Allow header that indicates what methods the server will permit.
415	Unsupported Media Type	The request cannot be processed because it contains an entity using a media type the server does not support.
500	Internal Server Error	Generic error message indicating that the request could not be fulfilled due to a server problem.
501	Not Implemented	The server does not know how to carry out the request, so it cannot satisfy it.
503	Service Unavailable	The server is temporarily unable to fulfill the request for internal reasons. This is often returned when a server is overloaded or down for maintenance.
505	HTTP Version Not Supported	The request used a version of HTTP that the server does not understand.

WORKING WITH COOKIES

A cookie is a bit of information sent by a web server to a browser that can later be read back from that browser. When a browser receives a cookie, it saves the cookie and thereafter sends the cookie back to the server each time it accesses a page on that server, subject to certain rules. Because a cookie's value can uniquely identify a client, cookies are often used for session tracking. Version 2.0 of the Servlet API provides the javax.servlet.http.Cookie class for working with cookies. The HTTP header details for the cookies are handled by the Servlet API.

Create a cookie with the Cookie() constructor:

```
public Cookie(String name, String value)
```

This creates a new cookie with an initial name and value. The rules for valid names and values are given in Netscape's Cookie Specification and RFC 2109.

A servlet can send a cookie to the client by passing a Cookie object to the addCookie() method of HttpServletResponse:

```
public void HttpServletResponse.addCookie(Cookie cookie)
```

This method adds the specified cookie to the response. Additional cookies can be added with subsequent calls to addCookie() . Because cookies are sent using HTTP headers, they should be added to the response before you send any content. Browsers are only required to accept 20 cookies per site, 300 total per user, and they can limit each cookie's size to 4096 bytes.

The code to set a cookie looks like this:

```
Cookie cookie = new Cookie("ID", "123");
res.addCookie(cookie);
```

A servlet retrieves cookies by calling the getCookies() method of HttpServlet- Request:

```
public Cookie[] HttpServletRequest.getCookies()
```

This method returns an array of Cookie objects that contains all the cookies sent by the browser as part of the request or null if no cookies were sent.

The code to fetch cookies looks like this:

```
Cookie[] cookies = req.getCookies();
if (cookies != null) {
  for (int i = 0; i < cookies.length; i++) {
    String name = cookies[i].getName();
    String value = cookies[i].getValue();
  }
}
```

The following methods are used to set these attributes:

- **public void Cookie.setVersion(int v) :** Sets the version of a cookie. Servlets can send and receive cookies formatted to match either Netscape persistent cookies (Version 0) or the newer, somewhat experimental, RFC 2109 cookies (Version 1). Newly constructed cookies default to Version to maximize interoperability.
- **public void Cookie.setDomain(String pattern):** Specifies a domain restriction pattern. A domain pattern specifies the servers that should see a cookie. By default, cookies are returned only to the host that saved them. Specifying a domain name pattern overrides this. The pattern must begin with a dot and must contain at least two dots. A pattern

matches only one entry beyond the initial dot. For example, ".foo.com" is valid and matches www.foo.com and upload.foo.combut not www.upload.foo.com. For details on domain patterns, see Netscape's Cookie Specification and RFC 2109.

- **public void Cookie.setMaxAge(int expiry):**Specifies the maximum age of the cookie in seconds before it expires. A negative value indicates the default, that the cookie should expire when the browser exits. A zero value tells the browser to delete the cookie immediately.

public void Cookie.setPath(String uri): Specifies a path for the cookie, which is the subset of URIs to which a cookie should be sent. By default, cookies are sent to the page that set the cookie and to all the pages in that directory or under that directory. For example, if /servlet/CookieMonster sets a cookie, the default path is "/servlet". That path indicates the cookie should be sent to /servlet/Elmo and to /servlet/subdir/BigBird--but not to the /Oscar.html servlet alias or to any CGI programs under /cgi-bin. A path set to "/" causes a cookie to be sent to all the pages on a server. A cookie's path must be such that it includes the servlet that set the cookie.

- **public void Cookie.setSecure(boolean flag)**:Indicates whether the cookie should be sent only over a secure channel, such as SSL. By default, its value is false.
- **public void Cookie.setComment(String comment):**Sets the comment field of the cookie. A comment describes the intended purpose of a cookie. Web browsers may choose to display this text to the user. Comments are not supported by Version cookies.
- **public void Cookie.setValue(String newValue):**Assigns a new value to a cookie. With Version cookies, values should not contain the following: whitespace, brackets and parentheses, equals signs, commas, double quotes, slashes, question marks, at signs, colons, and semicolons. Empty values may not behave the same way on all browsers

Read cookies from Servlets.

The Cookie Class provides an easy way to read Cookies. Use **getCookies()** method to retrieve all the cookies in the servlet program. This getCookies() method returns an array of cookie objects. In this example we will show you how you can retrieve all the cookies and display using Servlets

```
import javax.servlet.ServletException;
import javax.servlet.http.Cookie;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
public class ReadCookies extends HttpServlet {
public void doGet(HttpServletRequest request, HttpServletResponse response)
throws ServletException,  java .io.IOException {
 Cookie cookie = null;
   Cookie[] cookies = request.getCookies();
   boolean newCookie = false;
   if (cookies != null) {
```

```
   for (int i = 0; i < cookies.length; i++) {
    if (cookies[i].getName().equals("Cookies")) {
     cookie = cookies[i];
    }
   }
  }
  if (cookie == null)
 {
  newCookie = true;
  int maxAge;
   try {
    maxAge = new Integer(getServletContext().getInitParameter("cookie-age")).
intValue();
   }
  catch (Exception e) {
    maxAge = -1;
   }
 cookie = new Cookie("Cookies", "" + getNextCookieValue());
   cookie.setPath(request.getContextPath());
   cookie.setMaxAge(100);
   response.addCookie(cookie);
  }
  response.setContentType("text/ html ");
  java.io.PrintWriter out = response.getWriter();

  out.println("<html>");
  out.println("<head>");
  out.println("<title>Read Cookie</title>");
  out.println("</head>");
  out.println("<body>");

  out.println("<h2> Our Cookie named \"Cookies\"information</h2>");
 out.println("Cookie value: " + cookie.getValue() + "<br>");
  if (newCookie) {
  out.println("Cookie Max-Age: " + cookie.getMaxAge() + "<br>");
  out.println("Cookie Path: " + cookie.getPath() + "<br>");
  }
 out.println("</body>");
  out.println("</html>");
 out.close();
 }
```

```
  private long getNextCookieValue() {
 return new java.util.Date().getTime();  }
public void doPost(HttpServletRequest request,
                          HttpServletResponse response)
  throws ServletException, java.io.IOException
 {
  doGet(request, response);
  }
}
```

Writing a cookie

```
        import java.io.*;
        import javax.servlet.*;
        import javax.servlet.http.*;
        public class setcookies extends Httpservelt
        {
            public void doGet(HttpserveltRequest request,
                HttpservletResponse response) throws servletException, IOException
            {
            Cookie mycookie = new Cookie("userID", 222);
            response.addCookie(mycookie);
            response.setContentType("text/html");
            PrintWriter out = response.getWriter();
            out.println("<! DOCTYPE HTML PUBLIC \"-//W3C//DTD HTML 4.0"+
                "Transitional//EN\">\n"+
            "<HTML>\n"+
            "<HEAD><TITLE>My Cookie</TITLE></HEAD>\n"+
            "<BODY>\n"+
            "<H1>"+MY COOKIE + "</H1>/n"+
            "<BODY>\n"+
            "<p> cookie written</p>\n"+
            "</BODY></HTML>");
            }
            }
```

Setting and Reading Cookies

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
public class SettingandReadingCookies extends HttpServlet
{
   public void doGet(HttpServletRequest request, HttpServletResponse response)
   throws IOException, ServletException
   {
    response.setContentType("text/html");
    PrintWriter out = response.getWriter();
    out.println("<HTML>");
    out.println("<HEAD>");
    out.println("<TITLE>");
    out.println("A Web Page");
    out.println("</TITLE>");
    out.println("</HEAD>");
    out.println("<BODY");
    Cookie[] cookies = request.getCookies();
    boolean foundCookie = false;
    for(int loopIndex = 0; loopIndex < cookies.length; loopIndex++) {
       Cookie cookie1 = cookies[loopIndex];
       if (cookie1.getName().equals("color")) {
          out.println("bgcolor = " + cookie1.getValue());
          foundCookie = true;
       }
    }
    if (!foundCookie) {
       Cookie cookie1 = new Cookie("color", "cyan");
       cookie1.setMaxAge(24*60*60);
       response.addCookie(cookie1);
}
   out.println(">");
    out.println("<H1>Setting and Reading Cookies</H1>");
    out.println("This page will set its background color using a cookie when reloaded.");
    out.println("</BODY>");
    out.println("</HTML>");
}
}
```

TRACKING SESSIONS

There are a number of problems that arise from the fact that HTTP is a "stateless" protocol. In particular, when you are doing on-line shopping, it is a real annoyance that the Web server can't easily remember previous transactions. This makes applications like shopping carts very problematic: when you add an entry to your cart, how does the server know what's already in your cart? Even if servers did retain contextual information, you'd still have problems with e-commerce. When you move from the page where you specify what you want to buy (hosted on the regular Web server) to the page that takes your credit card number and shipping address (hosted on the secure server that uses SSL), how does the server remember what you were buying?

There are three typical solutions to this problem.

- **Cookies.** You can use HTTP cookies to store information about a shopping session, and each subsequent connection can look up the current session and then extract information about that session from some location on the server machine. This is an excellent alternative, and is the most widely used approach.
- **URL Rewriting.** You can append some extra data on the end of each URL that identifies the session, and the server can associate that session identifier with data it has stored about that session. This is also an excellent solution, and even has the advantage that it works with browsers that don't support cookies or where the user has disabled cookies. However, it has most of the same problems as cookies, namely that the server-side program has a lot of straightforward but tedious processing to do. In addition, you have to be very careful that every URL returned to the user (even via indirect means like Location fields in server redirects) has the extra information appended. And, if the user leaves the session and comes back via a bookmark or link, the session information can be lost.
- **Hidden form fields.** HTML forms have an entry that looks like the following: <INPUT TYPE="HIDDEN" NAME="session" VALUE="...">. This means that, when the form is submitted, the specified name and value are included in the GET or POST data. This can be used to store information about the session. However, it has the major disadvantage that it only works if every page is dynamically generated, since the whole point is that each session has a unique identifier.

Servlets provide an outstanding technical solution: the HttpSession API. This is a high-level interface built on top of cookies or URL-rewriting. In fact, on many servers, they use cookies if the browser supports them, but automatically revert to URL-rewriting when cookies are unsupported or explicitly disabled. But the servlet author doesn't need to bother with many of the details, doesn't have to explicitly manipulate cookies or information appended to the URL, and is automatically given a convenient place to store data that is associated with each session.

The Session Tracking API

Using sessions in servlets is quite straightforward, and involves looking up the session object associated with the current request, creating a new session object when necessary, looking up information associated with a session, storing information in a session, and discarding completed or abandoned sessions.

Looking up the HttpSession object associated with the current request.

This is done by calling the getSession method of HttpServletRequest. If this returns null, you can create a new session, but this is so commonly done that there is an option to automatically create a new session if there isn't one already. Just pass true to getSession. Thus, your first step usually looks like this:

```
 HttpSession session = request.getSession(true);
```

Looking up Information Associated with a Session.

HttpSession objects live on the server; they're just automatically associated with the requester by a behind-the-scenes mechanism like cookies or URL-rewriting. These session objects have a builtin data structure that let you store any number of keys and associated values. In version

2.1 and earlier of the servlet API, you use getValue("key") to look up a previously stored value. The return type is Object, so you have to do a typecast to whatever more specific type of data was associated with that key in the session. The return value is null if there is no such attribute. In version 2.2, getValue is deprecated in favor of getAttribute, both because of the better naming match with setAttribute (the match for getValue is putValue, not setValue), and because setAttribute lets you use an attached HttpSessionBindingListener to monitor values, while putValue doesn't and because setAttribute lets you use an attached HttpSessionBindingListener to monitor values, while putValue doesn't. Nevertheless, since few commercial servlet engines yet support version 2.2, I'll use getValue in my examples. Here's one representative example, assuming ShoppingCart is some class you've defined yourself that stores information on items being purchased.

```
HttpSession session = request.getSession(true);
ShoppingCart previousItems =
  (ShoppingCart)session.getValue("previousItems");
if (previousItems != null) {
  doSomethingWith(previousItems);
} else {
  previousItems = new ShoppingCart(...);
  doSomethingElseWith(previousItems);
}
```

In most cases, you have a specific attribute name in mind, and want to find the value (if any) already associated with it. However, you can also discover all the attribute names in a given session by calling getValueNames, which returns a String array. In version 2.2, use getAttributeNames, which has a better name and which is more consistent in that it returns an Enumeration, just like the getHeaders and getParameterNames methods of HttpServletRequest.

Although the data that was explicitly associated with a session is the part you care most about, there are some other pieces of information that are sometimes useful as well.

- **getId.** This method returns the unique identifier generated for each session. It is sometimes used as the key name when there is only a single value associated with a session, or when logging information about previous sessions.
- **isNew.** This returns true if the client (browser) has never seen the session, usually because it was just created rather than being referenced by an incoming client request. It returns false for preexisting sessions.
- **getCreationTime.** This returns the time, in milliseconds since the epoch, at which the session was made. To get a value useful for printing out, pass the value to the Date constructor or the setTimeInMillis method of GregorianCalendar.
- **getLastAccessedTime.** This returns the time, in milliseconds since the epoch, at which the session was last sent from the client.
- **getMaxInactiveInterval**. This returns the amount of time, in seconds, that a session should go without access before being automatically invalidated. A negative value indicates that the session should never timeout.

Associating Information with a Session

As discussed in the previous section, you read information associated with a session by using getValue (or getAttribute in version 2.2 of the servlet spec). To specify information, you use putValue (or setAttribute in version 2.2), supplying a key and a value. Note that putValue replaces any previous values. Sometimes that's what you want (as with the referringPage entry in the example below), but other times you want to retrieve a previous value and augment it (as with the previousItems entry below).

Here's an example:

```
 HttpSession session = request.getSession(true);
 session.putValue("referringPage", request.getHeader("Referer"));
 ShoppingCart previousItems =
   (ShoppingCart)session.getValue("previousItems");
 if (previousItems == null) {
   previousItems = new ShoppingCart(...);
 }
 String itemID = request.getParameter("itemID");
 previousItems.addEntry(Catalog.getEntry(itemID));
 // You still have to do putValue, not just modify the cart, since
 // the cart may be new and thus not already stored in the session.
 session.putValue("previousItems", previousItems);
```

Example: Showing Session Information

Here is a simple example that generates a Web page showing some information about the current session. You can also download the source or try it on-line.

```
package hall;
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
import java.net.*;
import java.util.*;
/** Simple example of session tracking. See the shopping
 *  cart example for a more detailed one. *  <P>  */
public class ShowSession extends HttpServlet {
  public void doGet(HttpServletRequest request, HttpServletResponse response)
      throws ServletException, IOException {
    HttpSession session = request.getSession(true);
    response.setContentType("text/html");
    PrintWriter out = response.getWriter();
    String title = "Searching the Web";
    String heading;
    Integer accessCount = new Integer(0);;
    if (session.isNew()) {
      heading = "Welcome, Newcomer";
  } else {
   heading = "Welcome Back";
   Integer oldAccessCount =
    // Use getAttribute, not getValue, in version
    // 2.2 of servlet API.
    (Integer)session.getValue("accessCount");
   if (oldAccessCount != null) {
    accessCount =
     new Integer(oldAccessCount.intValue() + 1);
   }
  }
  // Use putAttribute in version 2.2 of servlet API.
  session.putValue("accessCount", accessCount);
     out.println(ServletUtilities.headWithTitle(title) +
        "<BODY BGCOLOR=\"#FDF5E6\">\n" +
        "<H1 ALIGN=\"CENTER\">" + heading + "</H1>\n" +
        "<H2>Information on Your Session:</H2>\n" +
        "<TABLE BORDER=1 ALIGN=CENTER>\n" +
        "<TR BGCOLOR=\"#FFAD00\">\n" +
        "  <TH>Info Type<TH>Value\n" +
        "<TR>\n" +
        "  <TD>ID\n" +
        "  <TD>" + session.getId() + "\n" +
        "<TR>\n" +
        "  <TD>Creation Time\n" +
        "  <TD>" + new Date(session.getCreationTime()) + "\n" +
        "<TR>\n" +
        "  <TD>Time of Last Access\n" +
        "  <TD>" + new Date(session.getLastAccessedTime()) + "\n" +
        "<TR>\n" +
        "  <TD>Number of Previous Accesses\n" +
        "  <TD>" + accessCount + "\n" +
        "</TABLE>\n" +
        "</BODY></HTML>");
 }
 public void doPost(HttpServletRequest request,
          HttpServletResponse response)
   throws ServletException, IOException {
  doGet(request, response);
 }
}
```

POSSIBLE QUESTIONS

PART-B

(Each Question carries 6 Marks)

1. What is a cookie? Explain its working with example.
2. i) List the benefits of using a Java servlet.
   ii) Discuss about reading data from a client.
3. Explain the following
   i) Request headers ii) Working with cookies
4. What is Java servlet? Discuss java servlet in detail
5. Explain in detail about all the headers on a HTTP request.
6. Write a note on a simple java servlet and explain its anatomy.
7. Discuss about java servlets and common gateway interface programming and benefits of using java servlet.
8. Write a servlet Program to lock a server.
9. What is a cookie? Explain its working with example.
10. Write a servlet program that returns list of information in table format.

PART-C

(One Compulsory Question carries 10 Marks)

1. Discuss the concept of Distributive Systems in J2EE.
2. Discuss the Working Process of JDBC.
3. Explain the concept of Cookies.
4. Differentiate Entity Java Bean and Session Java Bean.
5. Discuss about Java Server Pages

J2EE (17CSP301)
Unit III-Multiple Choice Questions

S.no	Question	Option 1	Option 2	Option 3	Option 4	Answer
1	There are ________ types of data.	2	3	4	5	2
2	___________ is information received from the client that is typically either entered by the user into the user interface or generated by the user interface itself databases is called	Explicit data	Implicit data	CGI	Browser	Explicit data
3	___________ is HTTP information that is generated by the client rather than the user.	Explicit data	Implicit data	CGI	Browser	Implicit data
4	The result of processing a request is returned to the client as ___________.	Explicit data	Implicit data	CGI	Browser	Explicit data
5	A __________ is a server side program.	servlet	JSP	EJB	Java	servlet
6	Java servlet remains alive after the request is fulfilled. This is called __________.	persistence	reliability	Integrity	robustness	persistence
7	A __________ is a java class that reads requests sent from a client and responds by the sending information to the client.	servlet	JSP	EJB	EIS	servlet
8	The doGet() method requires _________ arguments.	2	3	4	5	2
9	The doPost() method requires __________ arguments.	2	3	4	5	2
10	Incoming data includes __________ data.	implicit	explicit	implicit and explicit	meta	implicit and explicit
11	The _________ method is used in conjunction with a PrintWriter to send outgoing explicit data such as text that appears on a webpage.	println()	setContentType()	doGet()	doPost()	println()
12	The __________ method is used to set the value for the ContentType HTTP header information.	println()	setContentType()	doGet()	doPost()	setContentType()

13	The _________ method is called automatically when the java servlet is created.	init()	setContentType()	doGet()	doPost()	init()
14	The __________ method is called whenever a request for the java servlet is made to the web server.	init()	service()	doGet()	doPost()	service()
15	The _________ method is called when an instance of a java servlet is removed from memory.	init()	service()	destroy()	doPost()	service()
16	The __________ method is not called when an abnormal occurrence such as a system malfunction causes the java servlet to abruptly terminate.	init()	service()	destroy()	doPost()	service()
17	The web-app element should contain a servlet element with __________ subelements.	2	3	4	5	3
18	The ____________ contains the name used to access the java servlet.	servlet-name	servlet-class	init-param	servlet-id	servlet-name
19	A client uses the __________ method to pass information to a java servlet.	GET only	POST only	either GET or POST	PUT	either GET or POST
20	Data sent by a client is read into a java servlet by calling the __________ method.		doGet()	doPost()	getParameterValues()	getParameter()
21	The _________ method returns a null if data received from the client doesnot contain the parameter named in the argument.	getParameter()	doGet()	doPost()	getParameterValues()	getParameterValues()
22	The _________ method does not require an argument and returns an enumeration.	getParameter()	getParameterNames()	doPost()	getParameterValues()	getParameterNames()
23	A request from a client contains __________ components.	2	3	4	5	2
24	The HTTP Request Header ________ identifies the MIME type of data that can be handled by the browser that made the request.	Accept	Accept_Charset	Accept_Language	Authorization	Accept

25	The HTTP Request Header __________ identifies the character sets that can be used by the browser that made the request.	Accept	Accept_Charset	Accept_Language	Authorization	Accept_Charset
26	The HTTP Request Header __________ specifies the preferred languages that are used by the browser.	Accept	Accept_Charset	Accept_Language	Authorization	Accept_Language
27	The HTTP Request Header __________ is used by a browser to identify the client to the java servlet whenever a protected web page is being processed.	Accept	Accept_Charset	Accept_Language	Authorization	Authorization
28	The HTTP Request Header __________ identifies whether a browser can retrieve multiple files using the same socket, which is referred to as persistence.	Connection	Content-length	Cookie	Host	Connection
29	The HTTP Request Header __________ contains the size of the data in bytes that are transmitted using the POST method.	Connection	Content-length	Cookie	Host	Content-length
30	The HTTP Request Header __________ contains the host and port of the original URL	Connection	Content-length	Cookie	Host	Host
31	The HTTP Request Header ________ signifies that the browser's requests should be fulfilled only if the data has changed since a specified date.	If-Modified-Since	If-Unmodified-Since	Referer	User-Agent	If-Modified-Since
32	The HTTP Request Header __________ signifies that the browser's requests should be fulfilled only if the data is older than a specified date.	If-Modified-Since	If-Unmodified-Since	Referer	User-Agent	If-Unmodified-Since
33	The HTTP Request Header ___________ contains the URL of the web page that is currently displayed in the browser.	If-Modified-Since	If-Unmodified-Since	Referer	User-Agent	Referer
34	The HTTP Request Header _________ identifies the browser that made the request.	If-Modified-Since	If-Unmodified-Since	Referer	User-Agent	User-Agent

35	HTTP _________ version uses the Keep-Alive message to keep a connection open.	1.1	1.2.	1.3	1.4	1.1
36	There are __________ ways in which a java servlet replies to a client request.	2	3	4	5	2
37	A java servlet can write to the HTTP response header by calling the __________ method of the HttpServlet Response object.	setStatus()	sendError()	sendRedirect()	setServerStatus()	setStatus()
38	The __________ method is used to notify the client that an error has occurred.	setStatus()	sendError()	sendRedirect()	setServerStatus()	sendError()
39	The ___________ method transmits a location header to the browser.	setStatus()	sendError()	sendRedirect()	setServerStatus()	sendRedirect()
40	The HTTP Response Header ___________ is a parameter for the connection header.	close	Content-Encoding	Content-Language	Content-Length	close
41	The HTTP Response Header ___________ indicates page encoding .	close	Content-Encoding	Content-Language	Content-Length	Content-Encoding
42	The HTTP Response Header __________ indicates the language of the document.	close	Content-Encoding	Content-Language	Content-Length	Content-Language
43	The HTTP Response Header ____________ indicates the number of bytes in the message before any character encoding is applied.	close	Content-Encoding	Content-Language	Content-Length	Content-Length
44	The HTTP Response Header ___________ indicates the MIME type of the response document.	Content-Type	Expires	Last-Modified	Location	Content-Type
45	The HTTP Response Header ___________ specifies the time in milliseconds when document is out of date.use	Content-Type	Expires	Last-Modified	Location	2
46	The HTTP Response Header ___________ indicates the last time the document was changed.	Content-Type	Expires	Last-Modified	Location	3

47	The HTTP Response Header ___________ indicates the location of the document.	Content-Type	Expires	Last-Modified	Location	4
48	183. The HTTP Response Header ___________ indicates the number of seconds to wait before asking for a page update.	Refresh	Retry-After	Set-Cookie	WWW-Authenticate	Refresh
49	The HTTP Response Header ___________ indicates the number of seconds to wait before requesting service, if the service is unavailable.	Refresh	Retry-After	Set-Cookie	WWW-Authenticate	Retry-After
50	The HTTP Response Header __________ identifies the cookie for the page.	Refresh	Retry-After	Set-Cookie	WWW-Authenticate	Set-Cookie
51	The HTTP Response Header __________ indicates the authorization type.	Refresh	Retry-After	Set-Cookie	WWW-Authenticate	WWW-Authenticate
52	A cookie is composed of __________ pieces.	2	3	4	5	2
53	The __________ is used to identify a particular cookie from among other cookies stored at the client.	cookie name	cookie value	cookie API	cookie Id	cookie name
54	The __________ is associated with the cookies.	cookie name	cookie value	cookie API	cookie Id	cookie value
55	A java servlet writes a cookie by passing the construction of the cookie object _________ arguments.	2	3	4	5	2
56	The _________ method returns an array of cookie objects.	getCookie()	addCookie()	setValue()	getvalue()	getCookie()
57	A java servlet can modify the value of an existing cookies by using the __________ method of the cookie object.	getCookie()	addCookie()	setValue()	getvalue()	setValue()

UNIT 4
ENTERPRISE JAVABEAN

SYLLABUS:

Enterprise Java Beans: Deployment Descriptors – Session Java Bean –Entity Java Bean Message Driven Bean.

INTRODUCTION TO ENTERPRISE JAVA BEAN

The Enterprise JavaBeans (EJB) 1.1 specification defines architecture for the development and deployment of transactional, distributed object applications-based, server-side software components. Organizations can build their own components or purchase components from third-party vendors. These server-side components, called enterprise beans, are distributed objects that are hosted in Enterprise JavaBean containers and provide remote services for clients distributed throughout the network. Enterprise JavaBeans (EJB) is Sun Microsystems solution to the portability and complexity of CORBA. EJB introduces a much simpler programming model than CORBA, allowing developers to create portable distributed components called enterprise beans. The EJB programming model allows developers to create secure, transactional, and persistent business objects (enterprise beans) using a very simple programming model and declarative attributes. Unlike CORBA, facilities such as access control (authorization security) and transaction management are extremely simple to program. Where CORBA requires the use of complex APIs to utilize these services, EJB applies these services to the enterprise bean automatically according to declarations made in a kind of property file called a deployment descriptor. This model ensures that bean developers can focus on writing business logic while the container manages the more complex but necessary operations automatically.

OVERVIEW OF EJB

Enterprise beans are Java EE components that implement Enterprise JavaBeans (EJB) technology. Enterprise beans run in the EJB container, a runtime environment within the Application Server (see Container Types). Although transparent to the application developer, the EJB container provides system-level services such as transactions and security to its enterprise beans. These services enable you to quickly build and deploy enterprise beans, which form the core of transactional Java EE applications. Written in the Java programming language, an **enterprise bean** is a server-side component that encapsulates the business logic of an application. The business logic is the code that fulfills the purpose of the application. In an inventory control application, for example, the enterprise beans might implement the business logic in methods called checkInventoryLevel and orderProduct. By invoking these methods, clients can access the inventory services provided by the application.

BENEFITS OF ENTERPRISE BEANS

For several reasons, enterprise beans simplify the development of large, distributed applications. First, because the EJB container provides system-level services to enterprise beans, the bean developer can concentrate on solving business problems. The EJB container, rather than the bean developer, is responsible for system-level services such as transaction management and security authorization.

Second, because the beans rather than the clients contain the application's business logic, the client developer can focus on the presentation of the client. The client developer does not have to code the routines that implement business rules or access databases. As a result, the clients are thinner, a benefit that is particularly important for clients that run on small devices.

Third, because enterprise beans are portable components, the application assembler can build new applications from existing beans. These applications can run on any compliant Java EE server provided that they use the standard APIs.

WHEN TO USE ENTERPRISE BEANS

You should consider using enterprise beans if your application has any of the following requirements:

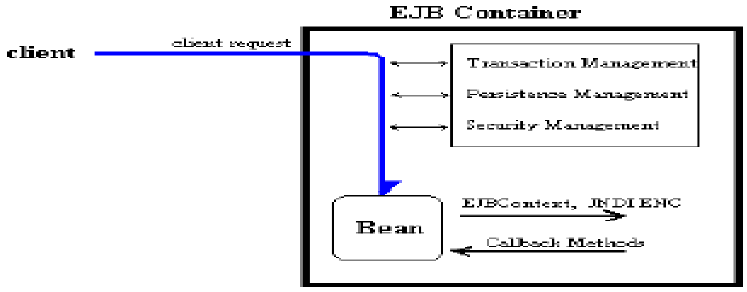
The application must be scalable. To accommodate a growing number of users, you may need to distribute an application's components across multiple machines. Not only can the enterprise beans of an application run on different machines, but also their location will remain transparent to the clients.

Transactions must ensure data integrity. Enterprise beans support transactions, the mechanisms that manage the concurrent access of shared objects.

The application will have a variety of clients. With only a few lines of code, remote clients can easily locate enterprise beans. These clients can be thin, various, and numerous

THE EBJ CONTAINER

Enterprise beans are software components that run in a special environment called an EJB container. The container hosts and manages an enterprise bean in the same manner that the Java Web server hosts a servlet or an HTML browser hosts a Java applet. An enterprise bean cannot function outside of an EJB container. The EJB container manages every aspect of an enterprise bean at run time including remote access to the bean, security, persistence, transactions, concurrency, and access to and pooling of resources. The container isolates the enterprise bean from direct access by client applications. When a client application invokes a remote method on an enterprise bean, the container first intercepts the invocation to ensure persistence, transactions, and security are applied properly to every operation a client performs on the bean. The container manages security, transactions, and persistence automatically for the bean, so the bean developer doesn't have to write this type of logic into the bean code itself. The enterprise bean developer can focus on encapsulating business rules, while the container takes care of everything else.



EJB Containers manage enterprise beans at runtime

Containers will manage many beans simultaneously in the same fashion that the Java WebServer manages many servlets. To reduce memory consumption and processing, containers pool resources and manage the life cycles of all the beans very carefully. When a bean is not being used, a container will place it in a pool to be reused by another client, or possibly evict it from memory and only bring it back when it's needed. Because client applications don't have direct access to the beans -- the container lies between the client and bean -- the client application is completely unaware of the container's resource management activities. A bean that is not in use, for example, might be evicted from memory on the server, while its remote reference on the client remains intact. When the client invokes a method on the remote reference, the container simply re-incarnates the bean to service the request. The client application is unaware of the entire process.

An enterprise bean depends on the container for everything it needs. If an enterprise bean needs to access a JDBC connection or another enterprise bean, it does so through the container; if an enterprise bean needs to access the identity of its caller, obtain a reference to itself, or access properties it does so through the container.

The enterprise bean interacts with its container through one of three mechanisms: callback methods, the EJBContext interface, or JNDI.

- Callback Methods: Every bean implements a subtype of the EnterpriseBean interface which defines several methods, called callback methods. Each callback method alerts the bean of a different event in its lifecycle and the container will invoke these methods to notify the bean when it's about to pool the bean, persist its state to the database, end a transaction, remove the bean from memory, and so on. The callback methods give the bean a chance to do some housework immediately before or after some event.
- EJBContext: Every bean obtains an EJBContext object, which is a reference directly to the container. The EJBContext interface provides methods for interacting with the container so that that bean can request information about its environment, like the identity of its client or the status of a transaction, or can obtain remote references to itself.
- Java Naming and Directory Interface (JNDI): JNDI is a standard extension to the Java platform for accessing naming systems like LDAP, NetWare, files systems, etc. Every bean automatically has access to a special naming system called the Environment Naming Context (ENC). The ENC is managed by the container and accessed by beans using JNDI. The JNDI ENC allows a bean to access resources like JDBC connections, other enterprise beans, and properties specific to that bean.

EJB CLASSES

There are three kinds of EJB types. These are the Entity JavaBean class, the session Java bean class and the message-driven java bean class. These are commonly referred to as an entity bean ,a session bean and a message driven bean. The table 4.6.1 lists the three enterprise bean type.

Table 4.6.1 Enterprise Bean Types

Enterprise Bean Type	Purpose
Session	Performs a task for a client; optionally may implement a web service
Entity Bean	Used to represent Business Data
Message-Driven	Acts as a listener for a particular messaging type, such as the JavaMessage Service API

EJB INTERFACES

Interface in java means a group of related methods with empty bodies. EJB have generally 4 interfaces. Interface in java means a group of related methods with empty bodies. EJB have generally 4 interfaces. These are as follows

1) Remote interface:- Remote interface are the interface that has the methods that relate to a particular bean instance. In the Remote interface we have all get methods as given below in the program. This is the interface where all of the business method go.**javax.ejb.Remote** package is used for creating Remote interface.

```
package ejb;
import javax.ejb.Remote;
@Remote
public interface Bean30Remote {
String getMessage();
String getAddress();
String getCompanyname();
}
```

2) Local Interface:-Local interface are the type of interface that are used for making local connections to EJB.**@Local** annotation is used for declaring interface as Local. **javax.ejb.Local** package is used for creating Local interface.

```
package ejb;
import javax.ejb.Local;
@Local
public interface NewSessionLocal {
}
```

3) Home Interface:-Home interface is the interface that has methods that relate to all EJB of a certain class as a whole. The methods which are defined in this Interface are create() methods and find() methods . The create() method allows us to create beans. find() method is used in Entity beans.

```
import javax.ejb.EJBHome;
import javax.ejb.CreateException;
import java.rmi.RemoteException;
public interface Bean30RemoteHome extends EJBHome{
public Bean30Remote create() throws CreateException, RemoteException;
}
```

4) Localhome Interface:-The local interfaces extend the following interfaces. These interfaces are generally for use by clients

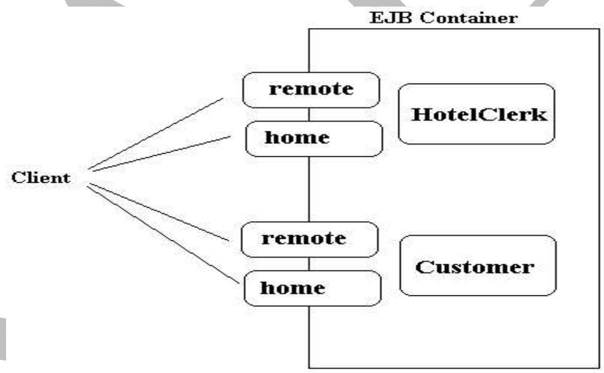
javax.ejb.EJBLocalObject - for the Object interface
javax.ejb.EJBLocalHome - for the Home interface

```
package ejb;
import javax.ejb.EJBLocalHome;
import javax.ejb.CreateException;
public interface NewSessionLocalHome extends EJBLocalHome {
public NewSessionLocal create() throws CreateException;
}
```

ENTERPRISE BEANS

To create an EJB server-side component, an enterprise bean developer provides two interfaces that define a bean's business methods, plus the actual bean implementation class. The client then uses a bean's public interfaces to create, manipulate, and remove beans from the EJB server. The implementation class, to be called the bean class, is instantiated at run time and becomes a distributed object. Enterprise beans live in an EJB container and are accessed by client applications over the network through their remote and home interfaces. The remote and home interfaces expose the capabilities of the bean and provide all the methods needed to create, update, interact with, and delete the bean. A bean is a server-side component that represents a business concept like a Customer or a HotelClerk.

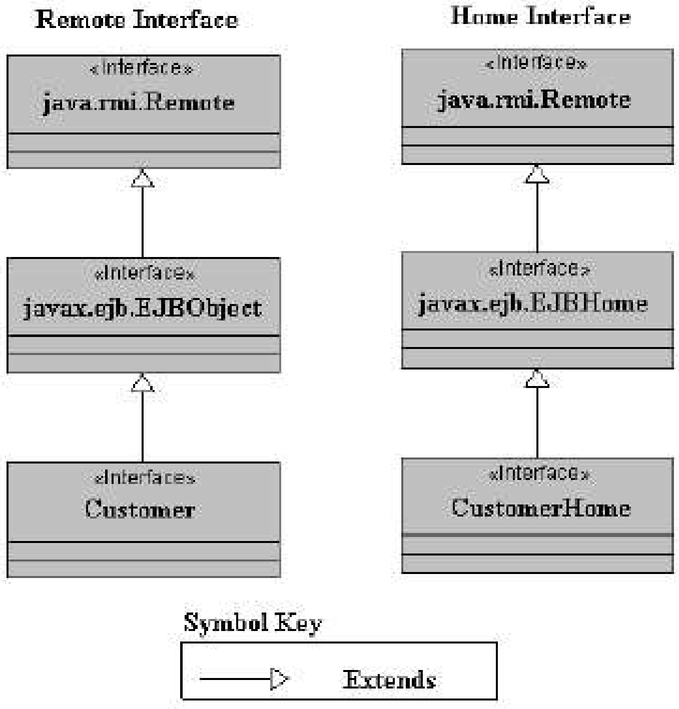
4.9 REMOTE AND HOME INTE



Conceptual View of EJB Architecture

The remote and home interfaces represent the bean, but the container insulates the beans from direct access from client applications. Every time a bean is requested, created, or deleted, the container manages the whole process.

The home interface represents the life-cycle methods of the component (create, destroy, find) while the remote interface represents the business method of the bean. The remote and home interfaces extend the javax.ejb.EJBObject and javax.ejb.EJBHome interfaces respectively. These EJB interface types define a standard set of utility methods and provide common base types for all remote and home interfaces.



Class Diagram of Remote and Home Interfaces

Clients use the bean's home interface to obtain references to the bean's remote interface. The remote interface defines the business methods like accessor and mutator methods for changing a customer's name, or business methods that perform tasks like using the HotelClerk bean to reserve a room at a hotel. Below is an example of how a Customer bean might be accessed from a client application. In this case the home interface is the CustomerHome type and the remote interface is the Customer type.

```
CustomerHome home = // ... obtain a reference that
// implements the home interface.
// Use the home interface to create a
// new instance of the Customer bean.
Customer customer = home.create(customerID);
// using a business method on the Customer.
customer.setName(someName);
```

The remote interface defines the business methods of a bean, the methods that are specific to the business concept it represents. Remote interfaces are subclassed from the javax.ejb.EJBObject interface, which is a subclass of the java.rmi.Remote interface. The importance of the remote interfaces inheritance hierarchy is discussed later in Section 4. For now, focus on the business methods and their meaning. Below is the definition of a remote interface for a Customer bean:

```
import javax.ejb.EJBObject;
import java.rmi.RemoteException;
public interface Customer extends EJBObject {
```

```
public Name getName() throws RemoteException;
public void setName(Name name) throws RemoteException;
public Address getAddress() throws RemoteException;
public void setAddress(Address address) throws RemoteException;
}
```

The remote interface defines accessor and mutator methods to read and update information about a business concept. This is typical of a type of bean called an entity bean, which represents a persistent business object (business objects whose data is stored in a database). Entity beans represent business data in the database and add behavior specific to that data.

BUSINESS METHODS

Business methods can also represent tasks that a bean performs. Although entity beans often have task-oriented methods, tasks are more typical of a type of bean called a session bean. Session beans do not represent data like entity beans. They represent business processes or agents that perform a service, like making a reservation at a hotel. Below is the definition of the remote interface for a HotelClerk bean, which is a type of session bean:

```
import javax.ejb.EJBObject;
import java.rmi.RemoteException;
public interface HotelClerk extends EJBObject {
public void reserveRoom(Customer cust, RoomInfo ri,
Date from, Date to)
throws RemoteException;
public RoomInfo availableRooms(Location loc, Date from, Date to)
throws RemoteException;}
```

The business methods defined in the HotelClerk remote interface represent processes rather than simple accessors. The HotelClerk bean acts as an agent in the sense that it performs tasks on behalf of the user, but is not itself persistent in the database

EJB DEPLOYMENT DESCRIPTOR

Deployment descriptor is the file which tells the EJB server that which classes make up the bean implementation, the home interface and the remote interface. it also indicates the behavior of one EJB with other. The deployment descriptor is generally called as ejb-jar.xml and is in the directory META-INF of the client application. In the example given below our application consists of single EJB Here the node

```
<?xml version ="1.0" encoding="UTF-8"?>
<application-client version="5" xmlns="http://java
.sun.com/xml/ns/javaee"  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
http://java.sun.com/xml/ns/javaee/application-client_5.xsd">
<description>Accessing Database Application</description>
<display-name>Secure-app-client</display-name><enterprise-beans>
<session>
<ejb-name>secure</ejb-name>
<home>org.glassfish.docs.secure.secureHome</home>
```

```
<remote>org.glassfish.docs.secure.secure</remote>
<ejb-class>org.glassfish.docs.secure.secureBean</ejb-class>
<session-type>Stateless</session-type>
</session>
</enterprise-beans>
</application-client>
```

<ejb-name>secure</ejb-name>:-This is the node that assigns the name to the EJB.

<description>Accessing Database Application</description>:-This node gives the brief description about the Ejb module created.

<session-type>Stateless</session-type>:-This node assigns the Session bean as stateless or stateful. Here stateless means to say accessing Remote interface.

DEPLOYING EJB TECHNOLOGY

The container handles persistence, transactions, concurrency, and access control automatically for the enterprise beans. The EJB specification describes a declarative mechanism for how these things will be handled, through the use of an XML deployment descriptor. When a bean is deployed into a container, the container reads the deployment descriptor to find out how transaction, persistence (entity beans), and access control should be handled. The person deploying the bean will use this information and specify additional information to hook the bean up to these facilities at run time. A deployment descriptor has a predefined format that all EJB-compliant beans must use and all EJB-compliant servers must know how to read. This format is specified in an XML Document Type Definition, or DTD. The deployment descriptor describes the type of bean (session or entity) and the classes used for the remote, home, and bean class. It also specifies the transactional attributes of every method in the bean, which security roles can access each method (access control), and whether persistence in the entity beans is handled automatically or is performed by the bean. Below is an example of a XML deployment descriptor used to describe the Customer bean:

```
<?xml version="1.0"?>
<!DOCTYPE ejb-jar PUBLIC "-//Sun Microsystems, Inc.//DTD Enterprise
JavaBeans 1.1//EN" "http://java.sun.com/j2ee/dtds/ejb-jar_1_1.dtd">
<ejb-jar>
<enterprise-beans>
<entity>
<description>
This bean represents a customer
</description>
<ejb-name>CustomerBean</ejb-name>
<home>CustomerHome</home>
<remote>Customer</remote>
<ejb-class>CustomerBean</ejb-class>
<persistence-type>Container</persistence-type>
<prim-key-class>Integer</prim-key-class>
<reentrant>False</reentrant>
```

```
<cmp-field><field-name>myAddress</field-name></cmp-field>
<cmp-field><field-name>myName</field-name></cmp-field>
<cmp-field><field-name>myCreditCard</field-name></cmp-field>
</entity>
</enterprise-beans>
<assembly-descriptor>
      <security-role>
  <description>
  This role represents everyone who is allowed full access to the Customer bean.
  </description>
  <role-name>everyone</role-name>
  </security-role>
  <method-permission>
  <role-name>everyone</role-name>
  <method>
  <ejb-name>CustomerBean</ejb-name>
  <method-name>*</method-name>
  </method>
  </method-permission>
  <container-transaction>
  <description>
  All methods require a transaction
  </description>
  <method>
  <ejb-name>CustomerBean</ejb-name>
  <method-name>*</method-name>
  </method>
  <trans-attribute>Required</trans-attribute>
  </container-transaction>
  </assembly-descriptor>
  </ejb-jar>
```

EJB-capable application servers usually provide tools that can be used to build the deployment descriptors; this greatly simplifies the process. When a bean is to be deployed, its remote, home, and bean class files and the XML deployment descriptor must be packaged into a jar file. The deployment descriptor must be stored in the jar under the special name META-INF/ejb-jar.xml. This jar file, called an ejb-jar, is vendor neutral; it can be deployed in any EJB container that supports the complete EJB specification. When a bean is deployed in an EJB container, its XML deployment descriptor is read from the jar to determine how to manage the bean at run time. The person deploying the bean will map attributes of the deployment descriptor to the container's environment. This will include mapping access security to the environment's security system, adding the bean to the EJB container's naming system, and so on. Once the bean developer has finished deploying the bean, it will become available for client applications and other beans to use.

SESSION BEAN

A **session bean** represents a single client inside the Application Server. To access an application that is deployed on the server, the client invokes the session bean's methods. The session bean performs work for its client, shielding the client from complexity by executing business tasks inside the server.

As its name suggests, a session bean is similar to an interactive session. A session bean is not shared; it can have only one client, in the same way that an interactive session can have only one user. Like an interactive session, a session bean is not persistent. (That is, its data is not saved to a database.) When the client terminates, its session bean appears to terminate and is no longer associated with the client.

STATE MANAGEMENT MODES

There are two types of session beans: stateful and stateless.

1. Stateful Session Beans

The state of an object consists of the values of its instance variables. In a stateful session bean, the instance variables represent the state of a unique client-bean session. Because the client interacts ("talks") with its bean, this state is often called the conversational state. The state is retained for the duration of the client-bean session. If the client removes the bean or terminates, the session ends and the state disappears. This transient nature of the state is not a problem, however, because when the conversation between the client and the bean ends there is no need to retain the state.

As an example, the HotelClerk bean can be modified to be a stateful bean which can maintain conversational state between method invocations. This would be useful, for example, if you want the HotelClerk bean to be able to take many reservations, but then process them together under one credit card. This occurs frequently, when families need to reserve two or more rooms or when corporations reserve a block of rooms for some event.

Below the HotelClerkBean is modified to be a stateful bean:

```
import javax.ejb.SessionBean;
import javax.naming.InitialContext;
public class HotelClerkBean implements SessionBean {
InitialContext jndiContext;
//conversational-state
Customer cust;
Vector resVector = new Vector();
        public void ejbCreate(Customer customer) {}
cust = customer;
}
public void addReservation(Name name, RoomInfo ri,
Date from, Date to) {
ReservationInfo resInfo =
new ReservationInfo(name,ri,from,to);
resVector.addElement(resInfo);
}
```

```
public void reserveRooms() {
CreditCard card = cust.getCreditCard();
Enumeration resEnum = resVector.elements();
while (resEnum.hasMoreElements()) {
ReservationInfo resInfo =
(ReservationInfo) resEnum.nextElement();
RoomHome roomHome = (RoomHome)
getHome("java:comp/env/ejb/RoomEJB", RoomHome.class);
Room room =
roomHome.findByPrimaryKey(resInfo.roomInfo.getID());
double amount = room.getPrice(resInfo.from,restInfo.to);
CreditServiceHome creditHome = (CreditServiceHome)
getHome("java:comp/env/ejb/CreditServiceEJB",
CreditServiceHome.class);
CreditService creditAgent = creditHome.create();
creditAgent.verify(card, amount);
ReservationHome resHome = (ReservationHome)
getHome("java:comp/env/ejb/ReservationEJB",
ReservationHome.class);
Reservation reservation =
resHome.create(resInfo.getName(),
resInfo.roomInfo,resInfo.from,resInfo.to);
}
public RoomInfo[] availableRooms(Location loc,
Date from, Date to) {
// Make an SQL call to find available rooms
}
private Object getHome(String path, Class type) {
Object ref = jndiContext.lookup(path);
return PortableRemoteObject.narrow(ref,type);
}}
```

In the stateful version of the HotelClerkBean class, the conversational state is the Customer reference, which is obtained when the bean is created, and the Vector of ReservationInfo objects.

By maintaining the conversational state in the bean, the client is absolved of the responsibility of keeping track of this session state. The bean keeps track of the reservations and processes them in a batch when the serverRooms() method is invoked.

To conserve resources, stateful session beans may be passivated when they are not in use by the client. Passivation in stateful session beans is different than for entity beans. In stateful beans, passivation means the bean conversational-state is written to a secondary storage (often disk) and the instance is evicted from memory. The client's reference to the bean is not affected by passivation; it remains alive and usable while the bean is passivated.

When the client invokes a method on a bean that is passivated, the container will activate the bean by instantiating a new instance and populating its conversational state with the state written to secondary storage. This passivation/activation process is often accomplished using simple Java serialization but it can be implemented in other proprietary ways as long as the mechanism behaves the same as normal serialization. (One exception to this is that transient fields do not need to be set to their default initial values when a bean is activated.) Stateful session beans, unlike stateless beans, do use the ejbActivate() and ejbPassivate() methods. The container will invoke these methods to notify the bean when it's about to be passivated (ejbPassivate()) and immediately following activation ejbActivate()). Bean developers should use these methods to close open resources and to do other clean-up before the instance's state is written to secondary storage and evicted from memory.

The ejbRemove() method is invoked on the stateful instance when the client invokes the remove() method on the home or remote interface. The bean should use the ejbRemove() method to do the same kind of clean-up performed in the ejbPassivate() method.

2. Stateless Session Beans

A stateless session bean does not maintain a conversational state with the client. When a client invokes the methods of a stateless bean, the bean's instance variables may contain a state specific to that client, but only for the duration of the invocation. When the method is finished, the client-specific state should not be retained. Clients may, however, change the state of instance variables in pooled stateless beans, and this state is held over to the next invocation of the pooled stateless bean. Except during method invocation, all instances of a stateless bean are equivalent, allowing the EJB container to assign an instance to any client. That is, the state of a stateless session bean should apply accross all clients.Because stateless session beans can support multiple clients, they can offer better scalability for applications that require large numbers of clients. Typically, an application requires fewer stateless session beans than stateful session beans

to support the same number of clients. A stateless session bean can implement a web service, but other types of enterprise beans cannot.

An example of a stateless session bean is a CreditService bean, representing a credit service that can validate and process credit card charges. A hotel chain might develop a CreditService bean to encapsulate the process of verifying a credit card number, making a charge, and recording the charge in the database for accounting purposes. Below are the remote and home interfaces for the CreditService bean:

```
// remote interface
public interface CreditService extends javax.ejb.EJBObject {
public void verify(CreditCard card, double amount)
throws RemoteException, CreditServiceException;
public void charge(CreditCard card, double amount)
throws RemoteException, CreditServiceException;
}
// home interface
public interface CreditServiceHome extends java.ejb.EJBHome {
public CreditService create()
throws RemoteException, CreateException;
}
```

The remote interface, CreditService, defines two methods, verify() and charge(), which are used by the hotel to verify and charge credit cards. The hotel might use the verify() method to make a reservation, but not charge the customer. The charge() method would be used to charge a customer for a room. The home interface, CreditServiceHome provides one create() method with no arguments. All home interfaces for stateless session beans will define just one method, a no-argument create() method, because session beans do not have find methods and they cannot be initiated with any arguments when they are created. Stateless session beans do not have find methods, because stateless beans are all equivalent and are not persistent. In other words, there is no unique stateless session beans that can be located in the database. Because stateless session beans are not persisted, they are transient services. Every client that uses the same type of session bean gets the same service.

Below is the bean class definition for the CreditService bean. This bean encapsulates access to the Acme Credit Card processing services. Specifically, this bean accesses the Acme secure Web server and posts requests to validate or charge the customer's credit card.

```
import javax.ejb.SessionBean;
public class CreditServiceBean implements SessionBean {
URL acmeURL;
HttpURLConnection acmeCon;
public void ejbCreate() {
try {
InitialContext jndiContext = new InitialContext();
URL acmeURL = (URL)
jndiContext.lookup("java:comp/ejb/env/url/acme");
acmeCon = acmeURL.openConnection();
}
catch (Exception e) {
throws new EJBException(e);
} }
public void verify(CreditCard card, double amount) {
String response = post("verify:" + card.postString() +
":" + amount);
if (response.substring("approved")== -1)
throw new CreditServiceException("denied");
}
public void charge(CreditCard card, double amount)
throws CreditCardException {
String response = post("charge:" + card.postString() +
":" + amount);
if (response.substring("approved")== -1)
throw new CreditServiceException("denied");
}
private String post(String request) {
try {
acmeCon.connect();
acmeCon.setRequestMethod("POST "+request);
String response = acmeCon.getResponseMessage();
}
catch (IOException ioe) {
throw new EJBException(ioe);
}
}
public void ejbRemove() {
acmeCon.disconnect();
}
public void setSessionContext(SessionContext cntx) {}
public void ejbActivate() {}
public void ejbPassivate() {}
}
```

The CreditService stateless bean demonstrates that a stateless bean can represent a collection of independent but related services. In this case, credit card validation and charges are related but not necessarily interdependent.

Stateless beans might be used to access databases or unusual resources, as is the case with the CreditService bean, or to perform complex computations. The CreditService bean is used as an example in this tutorial to demonstrate the service nature of a stateless bean and to provide a context for discussing the behavior of the call back methods. The use of a stateless session beans is not limited to the behavior illustrated in this example; stateless beans can be used to perform any kind of service.

The CreditServiceBean class uses a URL resource factory (acmeURL()) to obtain and maintain a reference to the Acme Web server which exists on another computer very far away. The CreditServiceBean uses the acmeURL() to obtain a connection to the Web server and post requests for the validation and charge of credit cards. The CreditService bean is used by clients instead of a direct connection so that the service can be better managed by the EJB container, which will pool connections and managed transactions and security automatically for the EJB client.

The ejbCreate() method is invoked at the beginning of its lifetime and is invoked only once. The ejbCreate() method is a convenient method for initiating resource connections and variables that will be of use to the stateless bean for its lifetime. In the example above, the CreditServiceBean uses the ejbCreate() to obtain a reference to the HttpURLConnection factory, which it will use throughout its lifetime to obtain connections to the Acme Web server.

The CreditServiceBean uses the JNDI ENC to obtain a URL connection factory in the same way that the CustomerBean used the JNDI ENC to obtain a DataSource resource factory for JDBC connections. The JNDI ENC is a default JNDI context that all beans have access to automatically. The JNDI ENC is used to access static properties, other beans, and resource factories like the java.net.URL and JDBC javax.sql.DataSource . In addition, the JNDI ENC also provides access to JavaMail and Java Messaging Service resource factories.

The ejbCreate() and ejbRemove() methods are each only invoked once in its lifetime by the container; when the bean is first created and when it's finally destroyed. Invocations of the create() and remove() methods on its home and remote interfaces by the client do not result in invocations of the ejbCreate() and ejbRemove() methods on the bean instance. Instead, an invocation of the create() method provides the client with a reference to the stateless bean type and the remove() methods invalidate the reference. The container will decide when bean instances are actually created and destroyed and will invoke the ejbCreate() and ejbRemove() methods at these times. This allows stateless instances to be shared between many clients without impacting the clients' references. In the CreditServiceBean, the ejbCreate() and ejbRemove() methods are used to obtain a URL connection at the beginning the bean instance's life and to disconnect from it at the end of the bean instance's life.

The verify() and charge() methods delegate their requests to the post() method, a private helper method. The post() method uses an HttpURLConnection to submit the credit card information to the Acme Web server and return the reply to the verify() or charge() method. The HttpURLConnection may have been disconnected automatically by the container -- this might occur if, for example, a lot of time elapsed since its last use -- so the post() method always invokes the connect() method, which does nothing if the connection is already established. The verify() and charge() methods parse the return value looking for the substring "approved," which indicates that the credit card was not denied. If "approved" was not found, it's assumed that the card was denied and a business exception is thrown. The setSessionContext() method provides the bean instance with a reference to the SessionContext, which serves the same purpose as the EntityContext did for the CustomerBean in the panel on . The SessionContext is not used in this example. The ejbActivate() and ejbPassivate() methods are not implemented in the CreditService bean because passivation is not used in stateless session beans. These methods are defined in the javax.ejb.SessionBean interface for the stateful session beans, and so an empty implementation must be provided in stateless session beans.Stateless session bean will never provide anything but empty implementations of these methods.

WHEN TO USE SESSION BEANS

In general, you should use a session bean if the following circumstances hold:

- At any given time, only one client has access to the bean instance.
- The state of the bean is not persistent, existing only for a short period (perhaps a few hours).
- The bean implements a web service.

Stateful session beans are appropriate if any of the following conditions are true:

- The bean's state represents the interaction between the bean and a specific client.
- The bean needs to hold information about the client across method invocations.

The bean mediates between the client and the other components of the application, presenting a simplified view to the client.

To improve performance, choose a stateless session bean if it has any of these traits:

- The bean's state has no data for a specific client.
- In a single method invocation, the bean performs a generic task for all clients. For example, use a stateless session bean to send an email that confirms an online order

ENTITY BEANS

The entity bean is one of three primary bean types: entity, session and Message Driven. The entity Bean is used to represent data in the database. It provides an object-oriented interface to data that would normally be accessed by the JDBC or some other back-end API. More than that, entity beans provide a component model that allows bean developers to focus their attention on the business logic of the bean, while the container takes care of managing persistence, transactions, and access control.

There are two basic kinds of entity beans: container-managed persistence (CMP) and bean-managed persistence (BMP). With CMP, the container manages the persistence of the entity bean. With BMP, the entity bean contains database access code (usually JDBC) and is responsible for reading and writing its own state to the database.

CONTAINER-MANAGED PERSISTENCE

Container-managed persistence beans are the simplest for the bean developer to create and the most difficult for the EJB server to support. This is because all the logic for synchronizing the bean's state with the database is handled automatically by the container. This means that the bean developer doesn't need to write any data access logic, while the EJB server is supposed to take care of all the persistence needs automatically -- a tall order for any vendor. Most EJB vendors support automatic persistence to a relational database, but the level of support varies. Some provide very sophisticated object-to-relational mapping, while others are very limited.In this panel, you will expand the CustomerBean developed earlier to a complete definition of a Container-managed persistence bean. In the panel on bean-managed persistence, you will modify the CustomerBean to manage its own persistence.

BEAN CLASS

An enterprise bean is a complete component that is made up of at least two interfaces and a bean implementation class. All these types will be presented and their meaning and application explained, starting with the bean class, which is defined below:

```
      import javax.ejb.EntityBean;
public class CustomerBean implements EntityBean {
int customerID;
Address myAddress;
Name myName;
CreditCard myCreditCard;
// CREATION METHODS
public Integer ejbCreate(Integer id) {
customerID = id.intValue();
return null;
}
public void ejbPostCreate(Integer id) {
}
public Customer ejbCreate(Integer id, Name name) {
myName = name;
return ejbCreate(id);
}
public void ejbPostCreate(Integer id, Name name) {
}
// BUSINESS METHODS
public Name getName() {
return myName;
}
public void setName(Name name) {
myName = name;
}
public Address getAddress() {
return myAddress;
}
public void setAddress(Address address) {
myAddress = address;
}
public CreditCard getCreditCard() {
return myCreditCard;
}
public void setCreditCard(CreditCard card) {
myCreditCard = card;
}
// CALLBACK METHODS
public void setEntityContext(EntityContext cntx) {
}
public void unsetEntityContext() {
}
public void ejbLoad() {
}
        public void ejbStore() {
}
public void ejbActivate() {
}
public void ejbPassivate() {
}
public void ejbRemove() {
}
}
```

Notice that there is no database access logic in the bean. This is because the EJB vendor provides tools for mapping the fields in the CustomerBean to the database. The CustomerBean class, for example, could be mapped to any database providing it contains data that is similar to the fields in the bean. In this case, the bean's instance fields are composed of a primitive int and simple dependent objects (Name, Address,and CreditCard) with their own attributes Below are the definitions for these dependent objects:

```
// The Name class
public class Name implements Serializable {
public String lastName, firstName, middleName;
public Name(String lastName, String firstName,
String middleName) {
this.lastName = lastName;
this.firstName = firstName;
this.middleName = middleName;
}
public Name() {}
}
// The Address class
public class Address implements Serializable {
public String street, city, state, zip;
public Address(String street, String city,
String state, String zip) {
this.street = street;
this.city = city;
this.state = state;
this.zip = zip;
}
public Address() {}
}
// The CreditCard class
public class CreditCard implements Serializable {
public String number, type, name;
public Date expDate; public CreditCard(String number, String type,
String name, Date expDate) {
this.number = number;
this.type = type;
this.name = name;
this.expDate = expDate;
}
public CreditCard() {}
}
```

These fields are called container-managed fields because the container is responsible for synchronizing their state with the database; the container manages the fields. Container-managed fields can be any primitive data types or serializable type. This case uses both a primitive int (customerID) and serializable objects (Address, Name, CreditCard). To map the dependent objects to the database, a fairly sophisticated mapping tool would be needed. Not all fields in a bean are automatically container-managed fields; some may be just plain instance fields for the bean's transient use. A bean developer distinguishes container-managed fields from plain instance fields by indicating which fields are container-managed in the deployment descriptor. The container-managed fields must have corresponding types (columns in RDBMS) in the database either directly or through object-relational mapping. The CustomerBean might, for example, map to a CUSTOMER table in the database that has the following definition:

```
CREATE TABLE CUSTOMER
{
id INTEGER PRIMARY KEY,
last_name CHAR(30),
first_name CHAR(20),
middle_name CHAR(20),
street CHAR(50),
city CHAR(20),
state CHAR(2),
zip CHAR(9),
credit_number CHAR(20),
credit_date DATE,
credit_name CHAR(20),
credit_type CHAR(10)
}
```

With container-managed persistence, the vendor must have some kind of proprietary tool that can map the bean's container-managed fields to their corresponding columns in a specific table, CUSTOMER in this case.

Once the bean's fields are mapped to the database, and the Customer bean is deployed, the container will manage creating records, loading records, updating records, and deleting records in the CUSTOMER table in response to methods invoked on the Customer bean's remote and home interfaces.

A subset (one or more) of the container-managed fields will also be identified as the bean's primary key. The primary key is the index or pointer to a unique record(s) in the database that makes up the state of the bean. In the case of the CustomerBean, the id field is the primary key field and will be used to locate the bean's data in the database. Primitive single field primary keys are represented as their corresponding object wrappers. The primary key of the Customer bean for example is a primitive int in the bean class, but to a bean's clients it will manifest itself as the java.lang.Integer type. Primary keys that are made up of several fields, called compound primary keys, will be represented by a special class defined by the bean developer. Primary keys are similar in concept to primary keys in a relational database -- actually when a relational database is used for persistence, they are often the same thing.

HOME INTERFACE

To create a new instance of a CMP entity bean, and therefore insert data into the database, the create() method on the bean's home interface must be invoked. The Customer bean's home interface is defined by the CustomerHome interface; the definition is shown below:

```
public interface CustomerHome extends javax.ejb.EJBHome {
public Customer create(Integer customerNumber)
throws RemoteException,CreateException;
public Customer create(Integer customerNumber, Name name)
throws RemoteException,CreateException;
public Customer findByPrimaryKey(Integer customerNumber)
throws RemoteException, FinderException;
public Enumeration findByZipCode(int zipCode)
throws RemoteException, FinderException;
}
```

Below is an example of how the home interface would be used by an application client to create a new customer:

```
CustomerHome home = // Get a reference to the CustomerHome object
Name name = new Name("John", "W", "Smith");
Customer customer = home.create(new Integer(33), name);
```

A bean's home interface may declare zero or more create() methods, each of which must have corresponding ejbCreate() and ejbPostCreate() methods in the bean class.These creation methods are linked at run time, so that when a create() method is invoked on the home interface, the container delegates the invocation to the corresponding ejbCreate() and ejbPostCreate() methods on the bean class. When the create() method on a home interface is invoked, the container
delegates the create() method call to the bean instance's matching ejbCreate() method. The ejbCreate() methods are used to initialize the instance state before a record is inserted into the database. In this case, they initialize the customerID and Name fields. When the ejbCreate() method is finished (they return null in CMP) the container reads the container-managed fields and inserts a new record into the CUSTOMER table indexed by the primary key, in this case customerID as it maps to the CUSTOMER.ID column. In EJB, an entity bean doesn't technically exist until after its data has been inserted into the database, which occurs during the ejbCreate() method. Once the data has been inserted, the entity bean exists and can access its own primary key and remote references, which isn't possible until after the ejbCreate() method completes and the data is in the database. If a bean needs to access its own primary key or remote reference after it's created, but before it services any business methods, it can do so in the ejbPostCreate() method. The ejbPostCreate() allows the bean to do any post-create processing before it begins serving client requests. For every ejbCreate() there must be a matching (matching arguments) ejbPostCreate() method.

The methods in the home interface that begin with "find" are called the find methods. These are used to query the EJB server for specific entity beans, based on the name of the method and arguments passed. Unfortunately, there is no standard query language defined for find methods, so each vendor will implement the find method differently. In CMP entity beans, the find methods are not implemented with matching methods in the bean class; containers implement them when the bean is deployed in a vendor specific manner. The deployer will use vendor specific tools to tell the container how a particular find method should behave. Some vendors will use object-relational mapping tools to define the behavior of a find method while others will simply require the deployer to enter the appropriate SQL command.

There are two basic kinds of find methods: single-entity and multi-entity methods return a remote reference to the one specific entity bean that matches the find request. If no entity beans are found, the method throws an ObjectNotFoundException .Every entity bean must define the single-entity find method with the method name findByPrimaryKey(), which takes the bean's primary key type as an argument. (In the above example, you used the Integer type, which wrapped the int type of the id field in the bean class.) The multi-entity find methods return a collection (Enumeration or Collection type) of entities that match the find request. If no entities are found, the multi-entity find returns an empty collection. (Note that an empty collection is not the same thing as a null reference.)

POSSIBLE QUESTIONS

PART-B

(Each Question carries 6 Marks)

1. Discuss briefly the two basic kinds of entity beans.
2. Write about EJB deployment in detail
3. i) What are Enterprise Java Beans? Describe EJB interfaces.
   ii) Discuss about query element and relationship element.
4. Explain briefly about creating a session java bean.
5. Design a counter that counts number of times user has visited the site in current browsing session.
6. What is deployment descriptor? Discuss about different types of java bean
7. What are Enterprise JavaBeans? Discuss about Session Java Bean.
8. Explain different elements that are used in a typical deployment descriptor.
9. Discuss about message driven bean with example program.

PART-C

(One Compulsory Question carries 10 Marks)

1. Create a bean counter
2. Discuss about Session Java Bean with an example.
3. Design a bean to add components

J2EE (17CSP301)
Unit IV-Multiple Choice Questions

S.No	Question	Option 1	Option 2	Option 3	Option 4	Answer
1	The EJB __________ is a vendor provided entity located on the EJB server that manages system-level services for EJB.	container	classes	interfaces	packages	container
2	There are _________ kinds of EJB types. Information	2	3	4	5	3
3	The session and entity beans must have ____________ interfaces.	2	3	4	5	2
4	A ___________ is used to represent business data.	entity bean	session bean	message-driven bean	EIS bean	entity bean
5	A __________ bean is used to model a business process.	entity	session	message-driven	EIS bean	session
6	A __________ bean is used to receive messages from a JMS resource.	entity	session	message-driven	EIS bean	message-driven
7	The _________ handles communication between the EJB and other components in the EJB environment using the Home interface and the Remote interface.	EJB container	EJB classes	EJB interfaces	deployment descriptors	EJB container
8	A _________ describes how EJBs are managed at runtime and enables the customization of EJB behavior without modification to the EJB code.	EJB container	EJB classes	EJB interfaces	deployment descriptors	deployment descriptors
9	A _____________ is written in a file using XML syntax.	EJB container	EJB classes	EJB interfaces	deployment descriptors	deployment descriptors
10	The expansion of IDE is _____________.	Integral Development Environment	Integrated Development Environment	Integrity Development Environment	Internal Development Environment	Integrated Developmen t Environment
11	The _____________ file is packages in the Java Archive file along with the other files that are required to deploy the EJB.	EJB container	EJB classes	EJB interfaces	deployment descriptors	deployment descriptors

12	The _________ element is the root element of the deployment descriptor.	<ejb-jar>	<ejb-name>	<ejb-class>	<entity>	<ejb-jar>
13	There are __________ elements that are contained within the <enterprise-beans> element.	2	3	4	5	3
14	The first element within the <ejb-jar> element is the __________ element.	<enterprise-beans>	<home>	<local>	<ejb-class>	<enterprise-beans>
15	The _____________ element contains subelements that describe the entity EJB.	<enterprise-beans>	<home>	<local>	<entity>	<entity>
16	The _________ element describes the fully qualified class name of the Remote interface, which defines the entity EJB's business mthods to remote clients.	<remote >	<local-home>	<reentrant>	<persistence-type>	<remote >
17	The __________ element defines how the entity EJB manages persistence.	<remote >	<local-home>	<reentrant>	<persistence-type>	<persistence-type>
18	The _________ element declares whether or not an entity EJB can be looped back without throwing an exception.	<remote>	<reentrant>	<ejb-class>	<remote>	<reentrant>
19	The subelement ____________ describes the deployment descriptor.	<description>	<display-name>	<small-icon>	<large-icon>	<description >
20	The subelement _________ describes the JAR file and individual EJB components.	<description>	<display-name>	<small-icon>	<large-icon>	<display-name>
21	The subelement __________ describes one or more enterprise beans contained in the JAR file.	<enterprise-beans>	<ejb-client-jar>	<assembly-descriptor>	<description>	<enterprise-beans>
22	The subelement ____________ describes the path of the client JAR and is used by the client to access EJBs described in the deployment descriptor.	<enterprise-beans>	<ejb-client-jar>	<assembly-descriptor>	<description>	<ejb-client-jar>

23	The subelement ____________ describes how EJBs are used in the J2EE application.	<enterprise-beans>	<ejb-client-jar>	<assembly-descriptor>	<description>	<assembly-descriptor>
24	The subelement _________ describes a small icon within the jar file that is used to represent the JAR file.	<description>	<small-icon>	<display-name>	<large-icon>	<small-icon>
25	There subelement __________ describes the fully qualified class name of the session or entity EJB remote interface.	<remote>	<local-home>	<local >	<ejb-class >	<remote>
26	The subelement ___________ describes the primary key filed for entity beans that use container-managed persistence.	<primary-field>	<prim-key-class>	<persistence-type>	<local>	<primary-field>
27	The subelement _____________ specifies the version of container-managed persistence.	<reentrant >	<cmp-version>	<cmp-field>	<env-entry>	<cmp-version>
28	The _________ element is used to specify an EJB's security role.	<security-role-ref>	<role-name>	<role-link>	<description>	<security-role-ref>
29	A ___________ is used in a deployment descriptor to specify a query method and a QL statement that is used as the criteria for selecting data from a relational database.	<query>	<method-param>	<ejb-ql>	<query-method>	<query-method>
30	The __________ subelement itself has two subelements.	<query>	<method-param>	<ejb-ql>	<query-method>	<query-method>
31	The _____________ subelement specifies the name of the method.	<query>	<method-param>	<ejb-ql>	<method-name>	<method-name>
32	The _________ subelement of the <query> element contains a SQL statement that is used to retrieve information from the database.	<ejb-ql>	<query>	<query-method>	<method-param>	<ejb-ql>
33	There are __________ types of cardinality relationships.	2	3	4	5	4

34	The cardinality relationships has one of __________ directions.	2	3	4	5	2
35	A ______________ is to execute a unit of work that may involve multiple tasks.	transaction	method	assembly	attribute	transaction
36	The _________ method is called whenever the session bean is removed from the pool and is referenced by a client.	ejbActivate()	ejbPassivate()	ejbRemove()	ejbCreate()	ejbActivate( )
37	The __________ method is called before the instance enters the "passive" state when the session bean is returned to the object pool and should contain routines that release resources.	ejbActivate()	ejbPassivate()	ejbRemove()	ejbCreate()	ejbPassivate ()
38	The __________ method is called just before the bean is available for garbage collection.	ejbActivate()	ejbPassivate()	ejbRemove()	ejbCreate()	ejbRemove()
39	The _____________ method is a method that contains business logic that is customized to the service provided by the EJB.	ejbActivate()	ejbPassivate()	ejbRemove()	myMethod()	myMethod()
40	A _________ is considered the powerhouse of a J2EE application.	entity java bean	session java bean	message-driven bean	net bean	entity java bean
41	Data collected and managed by an entity bean is called _____________.	data	persistent data	information	net bean	persistent data
42	There are ___________ groups of methods that are typically contained in an entity bean.	2	3	4	5	3
43	There are _______ commonly used callback methods.	4	5	64	7	7
44	The _________ method is called immediately following the creation of the instance and sets the content that is associated with the entity.	setEntityContext()	unsetEntityContext()	ejbLoad()	ejbStore()	setEntityContext()

45	The __________ method is called whenever the instance of the entity bean is activated from its “passive” state.	setEntityContext()	unsetEntityContext()	ejbLoad()	ejbActivate()	ejbActivate()
46	A container invokes the __________ method to instruct the instance to synchronize its state by loading its state from the underlying database.	setEntityContext()	unsetEntityContext()	ejbLoad()	ejbActivate()	ejbLoad()
47	The _____________ method is invoked by a container to instruct the instance to synchronize its state by storing it to the underlying database.	setEntityContext()	unsetEntityContext()	ejbLoad()	ejbStore()	ejbStore()
48	The _________ method is called before the instance enters the “passive” state and should contain routines that release resources.	ejbPassivate()	ejbActivate()	ejbRemove()	ejbLoad()	ejbPassivate()
49	Thee __________ method is called immediately before the entity terminates by either the client or by the EJB container.	ejbPassivate()	ejbActivate()	ejbRemove()	ejbLoad()	ejbRemove()
50	There are __________ methods defined in a BMP bean.	2	3	4	5	5
51	In BMP bean, ___________ method must contain code that reads data from a database.	ejbLoad()	ejbstore()	ejbCreate()	ejbRemove()	ejbLoad()
52	In BMP bean, the ___________ method must have code that inserts a new record in a database.	ejbLoad()	ejbstore()	ejbCreate()	ejbRemove()	ejbCreate()
53	In BMP bean, the ___________ method writes data to a database.	ejbLoad()	ejbstore()	ejbCreate()	ejbRemove()	ejbstore()
54	The __________ method is where the MBD processes messages received indirectly from a client.	onMessage()	getText()	ejbRemove()	setMessageDrivenContext()	onMessage()

UNIT 5
JAVA SERVER PAGES & JAVA RMI

SYLLABUS:

JSP: What is Java Server Pages? - Evolution of Dynamic Content Technologies – JSP & Java 2 Enterprise ed. **JSP Fundamentals**: Writing your first JSP- Tag conversions- Running JSP. **Programming JSP Scripts**: Scripting Languages – JSP tags- JSP directives – Scripting elements – Flow of Control – comments; **Java Remote Method Invocation.**

INTRODUCTION TO JAVA SERVER PAGES

Java Server Pages (JSP) technology is the Java platform technology for delivering dynamic content to web clients in a portable, secure and well-defined way. The JavaServer Pages specification extends the Java Servlet API to provide web application developers with a robust framework for creating dynamic web content on the server using HTML, and XML templates, and Java code, which is secure, fast, and independent of server platforms. JSP has been built on top of the Servlet API and utilizes Servlet semantics. JSP has become the preferred request handler and response mechanism. Although JSP technology is going to be a powerful successor to basic Servlets, they have an evolutionary relationship and can be used in a cooperative and complementary manner.

Servlets are powerful and sometimes they are a bit cumbersome when it comes to generating complex HTML. Most servlets contain a little code that handles application logic and a lot more code that handles output formatting.

This can make it difficult to separate and reuse portions of the code when a different output format is needed. For these reasons, web application developers turn towards JSP as their preferred servlet environment. Java Server Pages (JSP) are Sun's solution to the generation of dynamic HTML. With JSPs, you can generate HTML on the fly, which is important because

- The content of the HTML may depend upon user submitted data
- The information represented by the HTML is dynamic by nature
- The HTML may be customizable on a per user basis
- JSPs are HTML-like pages that can contain a mixture of HTML tags, data, and Java code.

BENEFITS OF JSP

One of the main reasons why the Java Server Pages technology has evolved into what it is today and it is still evolving is the overwhelming technical need to simplify application design by separating dynamic content from static template display data. Another benefit of utilizing JSP is that it allows to more cleanly separate the roles of web application/HTML designer from a software developer. The JSP technology is blessed with a number of exciting benefits, which are chronicled as follows:

1. The JSP technology is platform independent, in its dynamic web pages, its web servers, and its underlying server components. That is, JSP pages perform perfectly without any

hassle on any platform, run on any web server, and web-enabled application server. The JSP pages can be accessed from any web server.

2. The JSP technology emphasizes the use of reusable components. These components can be combined or manipulated towards developing more purposeful components and page design. This definitely reduces development time apart from the At development time, JSPs are very different from Servlets, however, they are precompiled into Servlets at run time and executed by a JSP engine which is installed on a Web-enabled application server such as BEA WebLogic and IBM WebSphere

JSP ARCHITECTURE

JSP pages are high level extension of servlet and it enables the developers to embed java code in html pages. JSP files are finally compiled into a servlet by the JSP engine. Compiled servlet is used by the engine to serve the requests.

javax.servlet.jsp package defines two interfaces:

- JSPPage
- HttpJspPage

These interfaces define the three methods for the compiled JSP page. These methods are:

- jspInt() – Called when JSP in requested
- jspDestroy() - Called when JSP is terminated
- jspService(HttpServletRequest request , HttpServletResponse response)-

The figure 5.3.1 shows the architecture of the JSP with its methods

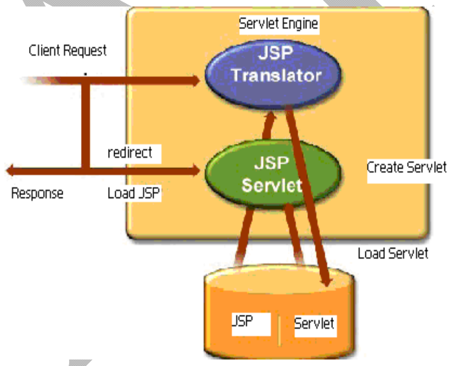


Figure 5.3.1 JSP Architecture

The jspInt() method is identical to the init() method in a Java servlet and in an applet. The jspInt() method is called first when the JSP is requested and is used to initialize objects and variables that are used throughout the life of the JSP.

The jspDestroy() method is identical to the destroy method in a java servlet. The destroy() method is automatically called when the JSP terminates normally. The destroy() method is used for cleanup where resources used during the execution of the JSP are released, such as disconnecting from a database. The jspService() 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 the 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 and Extendable Markup Language (XML) version of JSP tags, which are formatted as <jsp:TagID></JSP:TagID>.

In JSP tags can be divided into 5 different types. These are:

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. **Directives tag:** In the directives we can import packages, define error handling pages or the session information of the JSP page.
3. **Declarations tag:**This tag is used for defining the functions and variables to be used in the JSP.
4. **Scriplets:** In this tag we can insert any amount of valid java code and these codes are placed in *_jspService* method by the JSP engine.
5. **Expressions:** An expression tag opens with <%= and is used for an expression statement whose result replaces the expression statement whose result replaces the expression tag when the JSP virtual engine resolves JSP tags. An expression tags close with %>

JSP Directives

Syntax of JSP directives is:

<%! //java codes   %>

JSP Declaratives begins with <%! and ends %> with .We can embed any amount of java code in the JSP Declaratives. Variables and functions defined in the declaratives are class level and can be used anywhere in the JSP page

<%@directive attribute="value" %>

Where **directive** may be:

- page: page is used to provide the information about it. Example: <%@page language="java" %>
- include: include is used to include a file in the JSP page. Example:<%@ include file="/header.jsp" %>
- taglib: taglib is used to use the custom tags in the JSP pages (custom tags allows us to defined our own tags)
  Example: <%@ taglib uri="tlds/taglib.tld" prefix="mytag" %>

and **attribute** may be:

- language="java"
  This tells the server that the page is using the java language. Current JSP specification supports only java language.
  Example: <%@page language="java" %>
- extends="mypackage.myclass"
  This attribute is used when we want to extend any class. We can use comma(,) to

import more than one packages.

Example:

<%@page language="java"import="java.sql.*,mypackage.myclass" %>

- session="true"

  When this value is true session data is available to the JSP page otherwise not. By default this value is true.

  Example: <%@page language="java" session="true" %>
- errorPage="error.jsp"

  errorPage is used to handle the un-handled exceptions in the page. Example: <%@page language="java" session="true" errorPage="error.jsp"%>
- contentType="text/html;charset=ISO-8859-1"

  Use this attribute to set the MIME type and character set of the JSP. Example:<%@page language="java" session="true" contentType="text/html; charset=ISO-8859-1" %>
- errorPage="error.jsp"

  errorPage is used to handle the un-handled exceptions in the page. Example: <%@page language="java" session="true" errorPage="error.jsp"%>
- contentType="text/html;charset=ISO-8859-1"

  Use this attribute to set the MIME type and character set of the JSP. Example:<%@page language="java" session="true" contentType="text/html; charset=ISO-8859-1" %>

Example:

```
<%@page contentType="text/html" %>
<html>
<body><%!
int cnt=0;
private int getCount(){
//increment cnt and return the value
cnt++;
return cnt;
}
%>
<p>Values of Cnt are:</p>
<p><%=getCount()%></p>
<p><%=getCount()%></p>
<p><%=getCount()%></p>
<p><%=getCount()%></p>
<p><%=getCount()%></p>
<p><%=getCount()%></p>
</body>
</html>
```

JSP SCRIPTLETS

```
 <%  //java codes   %>
```

JSP Scriptlets begins with <% and ends %> .We can embed any amount of java code in the JSP Scriptlets. JSP Engine places these code in the _jspService() method. Variables available to the JSP Scriptlets are:

- **request:** request represents the clients request and is a subclass of HttpServletRequest. Use this variable to retrieve the data submitted along the request.
  Example:

```
<%//java codes
String userName=null; serName=request.getParameter("userName");    %>
```

- response: response is subclass of HttpServletResponse.
- session: session represents the HTTP session object associated with the request.
- out: out is an object of output stream and is used to send any output to the client.

Other variable available to the scriptlets are pageContext, application,config and exception.

JSP EXPRESSIONS

Syntax of JSP Expressions are:

```
<%="Any thing"   %>
```

JSP Expressions start with Syntax of JSP Scriptles are with <%= and ends with %>. Between these this you can put anything and that will convert to the String and that will be displayed.

Example:

```
<%="Hello World!" %>
```

Above code will display 'Hello World!'.

Display current time using Date class

- Current time: <%= new java.util.Date() %>

Display random number using Math class

- Random number: <%= Math.random() %>

Use implicit objects

- Your hostname: <%= request.getRemoteHost() %>
- Your parameter: <%= request.getParameter(“yourParameter”) %>
- Server: <%= application.getServerInfo() %>
- Session ID: <%= session.getId() %>

VARIABLES AND OBJECTS

In JSP variable can be declared same as in java. But the declaration statement must appear as a JSP tag within the JSP program before the variable or object used in the program.

Declaring and using a variable

```
<HTML>  <HEAD>   <TITLE>Creating a Variable</TITLE>  </HEAD>
 <BODY>   <H1>Creating a Variable</H1>
  <%
    int days= 365;%>
  <<p> Number of days = <%= days %></p>
 </BODY>
</HTML>
```

The variable days is used in an expression tag that is embedded within the HTML paragraph tag <p>. A JSP expression tag begins with <%=, which is followed by the expression. The JSP virtual engine resolves the JSP expression before sending the output of the JSP program to the browser. That is, the JSP tag <%=days%> is replaced with the number 365, afterwards, the HTML paragraph tag and related information is sent to the browser. It is able to place multiple statements with in a JSP tag by extending the close JSP program. This is illustrated in the below example where three variables are declared.

```
  <HTML>
   <HEAD>
    <TITLE>Creating a Variables</TITLE>
   </HEAD>
   <BODY>
    <%
       int days= 365;
            int month=12;
               int weeks;
    %>
    <<p> Number of days = <%= days %></p>
   </BODY>
  </HTML>
```

Array is used to store similar type of data in series. E.g. fruits name. Fruits can be a mango, banana, and apple. Name of students in classroom denote to 10th Standard, Bachelor in science can have group of 30 to 40 students.
Arrays can be String array, int array, and dynamic arrays are ArrayList, vector.

The following program shows the JSP program create three String objects,

```
<%@ page contentType="text/html; charset=iso-8859-1" language="java" %>
<% String[] stArray={"bob","riche","jacky","rosy"};
%>
<html>
<body>
   <%
    int i=0;
    for(i=0;i<stArray.length;i++)
    {
     out.print("stArray Elements       :"+stArray[i]+"<br/>");
    }
   %>
</body>
</html>
```

This String Array has four elements. When we go through this array, have to use loop either for or while loop. We are using here for loop, First stArray.length give use total number of elements in array then we fetch one by one for loop iterator. Array starts from zero so here we have only 0,1,2,3 elements if we try to get stArray[4] it will throw

```
<%@ page contentType="text/html; charset=iso-8859-1" language="java" %>
<%
String[] stArray=new String[4];
stArray[0]="bob";
stArray[1]="riche";
stArray[2]="jacky";
stArray[3]="rosy";
%>
<html>
<body>
<%
    int i=0;
    for(i=0;i<stArray.length;i++)
    {
     out.print("stArray Elements       :"+stArray[i]+"<br/>");
    }
   %>
</body>
</html>
```

Integer Array in JSP

```
<%@ page contentType="text/html; charset=iso-8859-1" language="java" %>
<%
int[] intArray={23,45,13,54,78};
%>
<html>
<body>
   <%
    int i=0;
    for(i=0;i<intArray.length;i++)
    {
     out.print("intArray Elements       :"+intArray[i]+"<br/>");
    }
   %>
</body> </html>
```

Dynamic arrays are automatically growable and reduceable according to per requirement. We don’t need to define it size when declaring array. It takes extra ratio of capacity inside memory and keeps 20% extra

Vector ArrayList

vectorArray.jsp

```
<%@ page import="java.util.Vector" language="java" %>
<%
Vector vc=new Vector();
vc.add("bob");
vc.add("riche");
vc.add("jacky");
vc.add("rosy");
%>
<html> <body>
   <%
    int i=0;
    for(i=0;i<vc.size();i++)
    {
     out.print("Vector Elements       :"+vc.get(i)+"<br/>");
    }
   %>
</body> </html>
```

ArrayList: ArrayList also same just it is unsynchronized, unordered and faster than vector.

ArrayList.jsp

```
<%@ page import="java.util.ArrayList" language="java" %>
<%
ArrayList ar=new ArrayList();
ar.add("bob");
ar.add("riche");
ar.add("jacky");
ar.add("rosy");
%>
<html>
<body>
   <%      int i=0;
    for(i=0;i<ar.size();i++)
    {       out.print("ArrayList Elements       :"+ar.get(i)+"<br/>");      }
   %>
</body> </html>
```

JSP METHODS

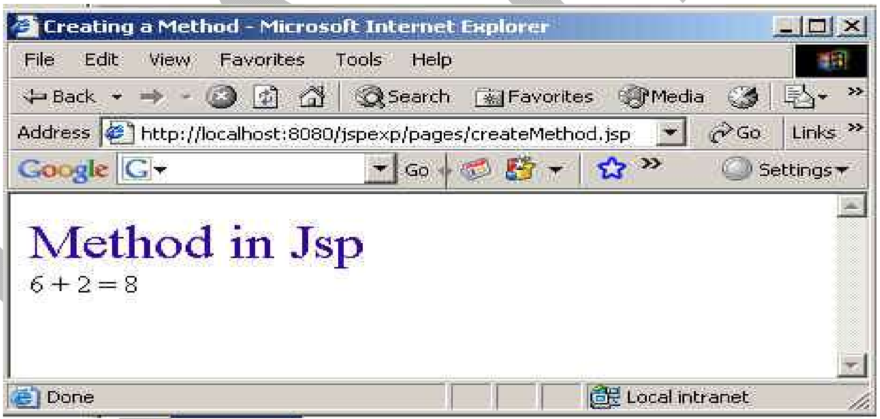
JSP offers the same versatility that have with JSP programs, such as defining methods that are local to the JSP program. A method is defined similar to how a method is defined in a java program except the method definition is place with in a JSP tag.Once the method is defined it can be called within the JSP tag.

In this below example it shows how to declare a method and how to use it. In this example making a method named as **addNum(int i, int b)** which will take two numbers as its parameters and return integer value. The method is declared inside a declaration directive i.e. <%! ----------- %> this is a declaration tag. This tag is used mainly for declaration the variables and methods. In the method adding to numbers is performed. To print the content of the method we are using scriptlet tag inside which we are going to use the out implicit object. <% ------- %> This tag is known as Scriptlets. The main purpose of using this tag is to embed a java code in the jsp page.

The code of the program is given below:

```
<% < HTML >
 <HEAD>   <TITLE>Creating a Method</TITLE>   </HEAD>
 <BODY>
<font  size="6" color ="#330099"> Method in Jsp </ font ><br>
   <%!
   int addNum(int n, int m)
   {
     return n + m;
   }
   %>
```

Output of the program is given below:


Creating a Method - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites Media
Address http://localhost:8080/jspexp/pages/createMethod.jsp Go Links
Google Go Settings
Done
Local intranet

```
Method in Jsp
6 + 2 = 8
```

A JSP program is capable of handling practically any kind of method that normally use in a Java program. The following example shows how to define and is an overloaded method.
Both methods are defined in the same JSP tags, although each follows Java Syntax structure for defining a method. One method uses a default value for the curve, while the overloaded method enables the statement that calls the method to provide the value of the curve.
Once again, these methods are called form an embedded JSP tag placed inside two HTML paragraph tags.

```
<HTML> <HEAD>
<TITLE> JSP Programming</TITLE> </HEAD>
<BODY>
<%! boolean curve (int grade)
```

```
        {               return 10 + grade;
        }
        boolean curve (int grade, int curveValue)
        {               return curveValue + grade;
        }
%>
<p> your curve grade is : <%=curve(80,100)%></p>
<p> your curve grade is : <%=curve(70)%></p>
</BODY> </HTML>
```

CONTROL STATEMENTS

One of the most powerful features available in JSP is the ability to change the flow of the program to truly create dynamic content for a web page based on conditions received form the browsers.

If Statement

There are two control statements used to change the flow of a JSP program. These are the if statement and the switch statement, both of which are also used to direct the flow of a java program. The if statement evaluates a condition statement to determine if one or more lines of code are to be executed or skipped.

The if statement requires three JSP tags. The first contains the beginning of the if statement, including the conditional expression. The second contains the else statement, and the third has the closed French brace used to terminate the else block.

Example of if-else condition

ifelse.jsp

```
  <%@ page language="java" import="java.sql.*" %>
<html> <head>
<title>while loop in JSP</title> </head>
<body>
<%
String sName="joe";
String sSecondName="noe";
   if(sName.equals("joe")){
     out.print("if condition check satisfied JSP count :"+sName+"<br>");
   }
   if(sName.equals("joe") && sSecondName.equals("joe"))
  {
     out.print("if condition check if Block <br>");
   }
   else
   {
     out.print("if condition check else Block <br>");
   } %>
</body> </html>
```

Using an if-else Ladder

```
<HTML>  <HEAD>
  <TITLE>Using an if-else Ladder</TITLE>  </HEAD>
 <BODY>
  <H1>Using an if-else Ladder</H1>
  <%
    String day = "Friday";
    if(day == "Monday")
      out.println("It\'s Monday.");
    else if (day == "Tuesday")
      out.println("It\'s Tuesday.");
    else if (day == "Wednesday")
      out.println("It\'s Wednesday.");
    else if (day == "Thurssday")
      out.println("It\'s Thursday.");
    else if (day == "Friday")
      out.println("It\'s Friday.");
    else if (day == "Saturday")
      out.println("It\'s Saturday.");
    else if (day == "Sunday")
      out.println("It\'s Sunday.");
  %>
 </BODY> </HTML>
```

Switch Statement

A switch statement compares a value with one or more other values associated with a case statement. The code segment that is associated wit the matching case statement is executed. Code segments associated with other case statements are ignored.

```
<HTML>
 <HEAD>   <TITLE>Using the switch Statement</TITLE>  </HEAD>
 <BODY>   <H1>Using the switch Statement</H1>
  <%
    int day = 3;
    switch(day) {
      case 0:
        out.println("It\'s Sunday.");
        break;
      case 1:
        out.println("It\'s Monday.");
        break;
      case 2:
        out.println("It\'s Tuesday.");
        break;
```

```
         case 3:
            out.println("It\'s Wednesday.");
            break;
         case 4:
            out.println("It\'s Thursday.");
            break;
         case 5:
            out.println("It\'s Friday.");
            break;
         default:
            out.println("It must be Saturday.");
      }
   %>
  </BODY> </HTML>
```

LOOPS

There are three kinds of loops commonly used in a JSP program. These are the for loop, while loop, and the do…while loop.

For Loop:

The for loop repeats usually a specified number of times

Example of for loop in JSP

for.jsp

```
<%@ page language="java" import="java.sql.*" %>
<html>
<head> <title>For loop in JSP</title> </head>
<body>
<%
for(int i=0;i<=10;i++)
{
  out.print("Loop through JSP count :"+i+"<br/>");
}
%>
</body> </html>
```

While Loop:

The while loop executes continually as long as a specified condition remains true. However, the while loop may not execute because the condition may never be true. In contrast the do…while loop executes at least once; then, the conditional expression in the do… while loop is evaluated to determine if the loop should be executed another time.

Example of while loop in JSP

while.jsp

```
  <%@ page language="java" import="java.sql.*" %>
<html> <head>
```

```
<title>while loop in JSP</title>
</head> <body>
<%
int i=0;
while(i<=10)
{
  out.print("While Loop through JSP count :"+i+"<br/>");
  i++;
}
%>
</body> </html>
```

Example of do-while loop in JSP

doWhile.jsp

```
  <%@ page language="java" import="java.sql.*" %>
<html>
<head> <title>do-while loop in JSP</title> </head>
<body>
<%
int i=0;
do{
  out.print("While Loop through JSP count :"+i+"<br/>");
  i++;
}
while(i<=10);
%>
</body> </html>
```

RESPONSE OBJECT IN JSP

Response is a process to responding against it request. Response Object in JSP is used to send information, or output from web server to the user. Response Object sends output in form of stream to the browser. This can be redirecting one file to another file, response object can set cookie, set ContentType, Buffer size of page, caching control by browser, CharSet, expiration time in cache.

Response object tells browser what output has to use or display on browser, and what stream of data contain PDF, html/text, Word, Excel.

Method	Return	Description
addCookie(Cookie cookie)	void	Add specified cookies to response object
addDateHeader(String name,long date)	void	Adds response header with given name and date value
addHeader(String name,String value)	void	Adds response header with given name and value

encodeRedirectURL(String URL)	String	Encode specified URL for sendRedirect method
encodeURL(String URL)	String	Encode specified URL with session ID, if not unchanged URL return
flushBuffer()	void	Forces any content in buffer to be written in client
getBufferSize()	int	Returns actual buffer size used for response
getCharacterEncoding()	String	Returns character encoding for page in response MIME type charset=iso-8859-1
getContentType()	String	Returns MIME type used for body in response text/html;
getOutputStream()	ServletOutputStream	Returns ServletOutputStream binary data stream to written in response
getWriter()	PrintWriter	Returns a PrintWriter object that can send character text to client
isCommitted()	boolean	Returns a Boolean, if response has been commited
resetBuffer()	void	Clear content in buffer of response without clearing header and status
sendRedirect(String location)	void	Sends a redirect response to client with redirect specified URL location. Contain Relative path
setBufferSize(**int** size)	void	Set a preferred buffer size for the body of response
setCharacterEncoding(String charset)	void	Set a character encoding for send to client response body charset=iso-8859-1
setContentLength(int length)	void	Set a length of content body in response
setContentType(String contentType)	void	Set a content Type of response being sent to client, if response is yet not committed
setHeader(String name, String value)	void	Set a response header with given name and value

Example of ContentType in JSP response object

setContentType.jsp

```
<%@ page language="java" %>
<%
response.setContentType("text/html");
%>
<html>
<head>
<title>Response object set Content type</title>
</head>
<body>
This is setting content type of JSP page
</body>
</html>
```

if documentation is in PDF format then setContentType as

```
<% response.setContentType("application/pdf"); %>
```

For Microsoft word document

```
<%
response.setContentType("application/msword");
%>
```

For Microsoft excel document

```
<%
response.setContentType("application/vnd.ms-excel");
%>
```

or

```
<%@ page contentType="text/html; charset=iso-8859-1" language="java" %>
```

Example of control page into cache

cacheControl.jsp

```
  <%@ page language="java" %>
<%
response.setHeader("Cache-Control","no-cache");
/*--This is used for HTTP 1.1 --*/
response.setHeader("Pragma","no-cache");
 /*--This is used for HTTP 1.0 --*/
response.setDateHeader ("Expires", 0);
/*---- This is used to prevents caching at the proxy server */
%>
<html>
<head>
<title>Response object in cache controlling</title>
</head>
<body>
```

This is no cache, Can test this page by open this page on browser and then open any other website after press back button on browser, if cacheControl.jsp is reloaded, it means it is not cached

Example of sendRedirect of Response Object

sendRedirect.jsp

```
  <%@ page  language="java" %>
<%
response.sendRedirect("http://yahoo.com");
///  response.sendRedirect("YouCanSpecifyYourPage.jsp");
%>
<html>
<head>
<title>Response object Send redirect</title>
</head>
<body>
This page redirects to specified URL location
</body>
</html>
```

REQUEST OBJECT

Request Object in JSP is most usable object. This request object take information from the user and send to server, then this request is proceed by server and response back to user. The process starts when we write address of any page in browser and press enter to page. Request object can be used in

Query String

When using query String, data is passed to server in form of URL string. This URL string is visible on browser and anyone can see it. E.g

http://www.domainName.com/concerts.jsp**?id=9&type=Exhibition**

This is query string after page name. id is key and 9 is value. This query String either self made or can send by form property. This can get with request.getParameter(“Key”) method. This will give us 9.

Query String self made example in JSP

queryString.jsp

```
<%@ page  language="java" import="java.sql.*" %>
<html>
<head><title>Query String in JSP</title></head>
<body>
<a href="queryString.jsp?id=9&name=joe">This is data inside query string</a>
<%
String queryString=request.getQueryString();
out.print("<br>"+queryString);
%>
</body> </html>
```

We can get single id and value instead of using getQueryString method of request.

Query String example with getParameter

queryStringParameter.jsp

```
  <%@ page  language="java" import="java.sql.*" %>
<html>
<head><title>Query String in JSP</title></head>
<body>
<a href="queryString.jsp?id=9&name=joe">This is data inside query string</a>
<%
String queryStringVariable1=request.getParameter("id");
String queryStringVariable2=request.getParameter("name");
out.print("<br>"+queryStringVariable1);
out.print("<br>"+queryStringVariable2);
%>
</body> </html>
```

FORM

Form is most important part of request object in JSP. It takes value from user and sends it to server for further processing. Suppose we have a registration form, and value in field have to save in database. We need to get these values from browse and get in server side; once we get this value in java, we can save in database through JDBC or any database object. Form in html has two

properties get or post method, get method send information to server in query string. Post method doesn't send data in query string in to server. When data is sent to server it is not visible on browser. Post method as compare to get method is slower. Get method has a limit of string length in some kilobytes.

Form Example in JSP with request object

formGetMethod.jsp

```
<%@ page  language="java" import="java.sql.*" %>
<%
String queryStringVariable1=request.getParameter("name");
String queryStringVariable2=request.getParameter("className");
%>
<html>
<head><title>Query String in JSP</title> </head>
<body>
<%
out.print("<br>Field 1 :"+queryStringVariable1);
out.print("<br>Field 2 :"+queryStringVariable2);
%>
<form method="get" name="form1">
Name <input type="text" name="name"> <br><br>
Class <input type="text" name="className"> <br><br>
```

```
<input type="submit" name="submit" value="Submit">
</form></body></html>
```

Request object with Check box form in JSP

formGetParameterValues.jsp

```
  <%@ page  language="java" import="java.sql.*" %>
<%
String queryStringVariable1=request.getParameter("name");
String[] queryStringVariable2=request.getParameterValues("className");
%>
<html>
<head><title>Query String in JSP</title></head>
<body>
<%
try{
    out.print("<br>Field 1 :"+queryStringVariable1);

    for(int i=0;i<=queryStringVariable2.length;i++){
     out.print("<br>Check box Field "+i+" :"+queryStringVariable2[i]);
    }
}
catch(Exception e)
{
 e.printStackTrace();
}
%>
<br>
<br>
<form method="get" name="form1">
Name <input type="text" name="name"> <br><br>
 class 1
 <input type="checkbox" name="className" value="c1">
 class 2 <input type="checkbox" name="className" value="c2"><br><br>
<input type="submit" name="submit" value="Submit">
</form></body></html>
```

Request object with radio button form in JSP

formGetRadio.jsp

```
  <%@ page  language="java" import="java.sql.*" %>
<%
String inputVariable=request.getParameter("name");
String[] radioVariable=request.getParameterValues("className");
%>
<html>
<head><title>Query String in JSP</title></head>
```

```
<body>
<%
try{
   out.print("<br>Field 1 :"+inputVariable);
 for(int i=0;i<=radioVariable.length;i++){
    out.print("<br>Check box Field "+i+" :"+radioVariable[i]);
   }
}
catch(Exception e)
{
 e.printStackTrace();
}
%>
<br>
<br>
<form method="get" name="form1">
Name <input type="text" name="name"> <br><br>
 class 1
 <input type="radio" name="className" value="c1">
 class 2 <input type="radio" name="className" value="c2"><br><br>
<input type="submit" name="submit" value="Submit">
</form>
</body>
</html>
```

Cookies

Cookies use request object to get values through in cookie file. It uses request.getCookies() method to get cookie and return in cookie[] array.

SERVER VARIABLE

Server variable give server information or client information. Suppose we need to get remote client IP address, browser, operating system, Context path, Host name any thing related to server can be get using request object in JSP.

Server variable example in JSP

serverVariable.jsp

```
  <%@ page contentType="text/html; charset=utf-8" language="java" %>
<html>
<head>
<title>Server Variable in JSP</title>
</head>
<body>
Character Encoding : <b><%=request.getCharacterEncoding()%></b>
<br />
Context Path : <strong><%=request.getContextPath()%></strong><br />
```

```
Path Info : <strong><%=request.getPathInfo()%></strong><br />
Protocol with version: <strong><%=request.getProtocol()%></strong><br />
Absolute Path file:<b><%=request.getRealPath("serverVariable.jsp")%></b><br/>
Client Address in form of IP Address:<b><%=request.getRemoteAddr()%></b><br/>
Client Host : <strong><%=request.getRemoteHost()%></strong><br />
URI : <strong><%=request.getRequestURI()%></strong><br />
Scheme in form of protocol
<strong><%=request.getScheme()%></strong><br />
Server Name : <strong><%=request.getServerName()%></strong><br />
Server Port no : <strong><%=request.getServerPort()%></strong><br />
Servlet path current File name : <b><%=request.getServletPath()%></b><br />
</body>
</html>
```

Properties and method of Request objects

Method	Return	Description
equals(Object obj)	Objects	Matching objects
getAttribute(String str)	Objects	Returns the String associated with name in the request scope
getAttributeNames()	Enumeration	an Enumeration of attribute names
getAuthType()	String	
getCharacterEncoding()	String	Returns character encoding for page in request MIME type charset=iso-8859-1
getContentLength()	int	get length of content body in request
getContentType()	String	
getContextPath()	String	
getCookies()	Cookie[]	
getDateHeader(String str)	long	get request header with given name and date value
getHeader(String str)	String	get request header with given name and value
getHeaderNames()	Enumeration	
getHeaders(String str)	Enumeration	
getInputStream()	ServletInputStream	
getIntHeader(String str)	int	
getLocalAddr(String str)	String	
getLocale(String str)	Locale	
getLocales()	Enumeration	
getLocalName()	String	
getLocalPort()	int	
getMethod()	String	
getParameter(String str)	String	
getParameterMap()	Map	
getParameterNames()	Enumeration	
getParameterValues(String str)	String	
getPathInfo()	String	

getPathTranslated()	String	
getProtocol()	String	
getQueryString()	String	
getReader()	BufferedReader	
getRemoteAddr()	String	
getRemoteHost()	String	
getRemotePort()	int	
getRemoteUser()	String	
getRequestDispatcher(String str)	RequestDispatcher	
getRequestedSessionId()	String	
getRequestURI()	String	
getRequestURL()	StringBuffer	
getReader()	BufferedReader	
getRemoteAddr()	String	
getRemoteHost()	String	
getRemotePort()	int	
getRemoteUser()	String	
getRequestDispatcher(String str)	RequestDispatcher	
getRequestedSessionId()	String	
getRequestURI()	String	
getRequestURL()	StringBuffer	
getScheme()	String	
getServerName()	String	
getServerPort()	int	
getServletPath()	String	
getSession()	HttpSession	
getSession(boolean true|false)	HttpSession	
getUserPrincipal()	Principal	
hashCode()	int	
isRequestedSessionIdFromCookie()	boolean	
isRequestedSessionIdFromURL()	boolean	
isRequestedSessionIdValid()	boolean	
isSecure()	boolean	
isUserInRole()	boolean	
removeAttribute(String str)	void	
setAttribute(String str, Object obj)	void	
setCharacterEncoding(String str)	void	Set a character encoding for send to client request body charset=iso-8859-1
toString()	String	
getServerPort()	int	
getServletPath()	String	
getSession()	HttpSession	
getSession(boolean true|false)	HttpSession	

JSP SESSION OBJECT

Session object is medium to interact with client and server. Session is a connection between user and server, involving exchange of information between user’s computer and server. Server knows information about each other by these session object. Web server put information of user in session object and whenever it needs information gets it from these session objects. This session object stores information in key value pair, just like hashtable. Today programming without session cannot thinkable. Most of web application is user based, somewhat use transaction (credit card, database transaction), shopping cart, email, and this needs session. Web server should know who is doing this transaction. This session object helps to differentiate users with each other, and increase application’s security. Every user have unique session, server exchange information with session objects until it get expired or destroyed by web server.

When JSP Session use

Mostly session is work with user base application, when login screen is used then set session if login is successful. It set for maximum time provided by web server or defined by application.

When JSP Session destroys

When user has done their work and want to close browser, it should be expire or destroy forcefully by user, ensure no other person can use his session.

JSP Store and Retrieve Session Variables

JSP Example of Creating session and retrieving session

session.jsp

```
<%@ page contentType="text/html; charset=iso-8859-1" language="java" %>
<html>
<body>
<form name="frm" method="get" action="sessionSetRetrieve.jsp">
<table width="100%" border="0" cellspacing="0" cellpadding="0">
  <tr>
    <td width="22%">&nbsp;</td>
    <td width="78%">&nbsp;</td>
    </tr>
  <tr>
    <td>Session value Set </td>
    <td><input type="text" name="sessionVariable" /></td>
  </tr>
  <tr>
    <td>&nbsp;</td>
    <td>&nbsp;</td>
  </tr>
  <tr>
    <td>&nbsp;</td>
    <td><input type="submit" name="submit" value="Submit"></td>
    </tr>
  <tr>
    <td>&nbsp;</td>
    <td>&nbsp;</td>
    </tr>
</table>
</form>
</body>
</html>
```

sessionSetRetrieve.jsp

```
  <%@ page language="java" import="java.util.*"%>
<%
String sessionSet=request.getParameter("sessionVariable");
session.setAttribute("MySession",sessionSet);
/// String getSessionValue= (String)session.getAttribute("sessionSet");
//this is use for session value in String data
%>
<html>
<head><title>Cookie Create Example</title></head>
<body>
Session :    <%=(String)session.getAttribute("MySession")%>
</body>
</html>
```

session.setAttribute("MySession",sessionSet) this is use to set new session variable. If we need to retrieve session variable, have to use session.getAttribute. In this we have to get it by session variable name here we are using **MySession** is session variable as key.

session.setMaxInactiveInterval(**2700**);

session.setMaxInactiveInterval(2700), this use to set maximum session time. 2700 is time in number. In this period, if user don't do anything session get expired automatically.

Remove JSP Session Variables or Expire JSP Session

When session is no long needed, should be removed forcefully by user. This can be done by calling session method of invalidate method session.invalidate();

session.invalidate();

This method expires session for current user, who request for log out. New session can be find by isNew() method of session, when first time session is created, that is new session.

session.isNew();

JSP COOKIES

Cookie a small data file reside in user's system. Cookie is made by web server to identify users. When user do any request send to web server and web server know information of user by these cookies.

After process request, web server response back to request by knowing through this cookie. JSP provides cookie classes, **javax.servlet.http.Cookie**, by this classes we can create and retrieve cookie. Once we set value in cookie, it lived until cookie gets expired. Cookie plays a big role in session, because maximum session is tracked by cookie in JSP.

JSP Example of creating Cookie

```
 <%@ page contentType="text/html; charset=iso-8859-1" language="java" %>
<html>
<body>
<form name="frm" method="get" action="createNewCookie.jsp">
<table width="100%" border="0" cellspacing="0" cellpadding="0">
  <tr>
    <td width="22%">&nbsp;</td>
    <td width="78%">&nbsp;</td>
    </tr>
  <tr>
    <td>Cookie value Set </td>
    <td><input type="text" name="cookieSet" /></td>
  </tr>
  <tr>
    <td>&nbsp;</td>
    <td>&nbsp;</td>
  </tr>
  <tr>
    <td>&nbsp;</td>
    <td><input type="submit" name="submit" value="Submit"></td>
    </tr>
  <tr>
    <td>&nbsp;</td>
    <td>&nbsp;</td>
    </tr>
</table>
</form>
</body>
</html>
```

createNewCookie.jsp

```
  <%@ page language="java" import="java.util.*"%>
<%
String cookieSet=request.getParameter("cookieSet");
Cookie cookie = new Cookie ("cookieSet",cookieSet);
response.addCookie(cookie);
%>
<html>
<head>
```

```
<title>Cookie Create Example</title></head>
<body>
   <%
  Cookie[] cookies = request.getCookies();
       for (int i=0; i<cookies.length; i++) {
     out.println(cookies[i].getName()+" : "+cookies[i].getValue()+"<br/>");
   }
   %>
</body></html>
```

```
Cookie cookie = new Cookie ("cookieSet",cookieSet);
```

We are creating new object of cookie class.

Here **new Cookie("Key", "value");**

Then we are adding in cookie of response object.

response.addCookie(cookie); This is adding in cookie object new data.

Expire JSP cookie

```
Cookie cookie = new Cookie ("cookieSet",cookieSet);
cookie setMaxAge(0);
```

This will expire cookie.,with 0 second. We can set cookie age who long is reside for user.

Retrieve cookie from JSP

Cookie is in array object, we need to get in request.getCookies() of array.

```
Cookie[] cookies = request.getCookies();
for (int i=0; i<cookies.length; i++) {
     out.println(cookies[i].getName()+" : "+cookies[i].getValue()+"<br/>");
}
```

AN OVERVIEW OF RMI APPLICATIONS

A j2EE applications runs with in a JVM, however objects used by a j2EE application do not need to run on the same JVM as the j2EE application. This is because a J2EE application and its components can invoke objects located on a different JVM by using the Java Remote Method Invocation (RMI) system. RMI is used for remote communication between java applications and components, both of which must be written in the Java programming language.

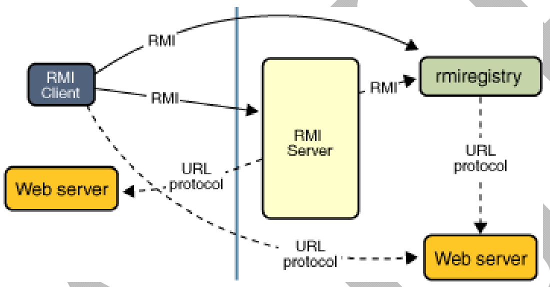
RMI applications often comprise two separate programs, a server and a client. A typical server program creates some remote objects, makes references to these objects accessible, and waits for clients to invoke methods on these objects. A typical client program obtains a remote reference to one or more remote objects on a server and then invokes methods on them. RMI provides the mechanism by which the server and the client communicate and pass information back and forth. Such an application is sometimes referred to as a distributed object application.

Distributed object applications need to do the following:

- **Locate remote objects.** Applications can use various mechanisms to obtain references to remote objects. For example, an application can register its remote objects with RMI's simple naming facility, the RMI registry. Alternatively, an application can pass and return remote object references as part of other remote invocations.

- **Communicate with remote objects.** Details of communication between remote objects are handled by RMI. To the programmer, remote communication looks similar to regular Java method invocations.
- **Load class definitions for objects that are passed around.** Because RMI enables objects to be passed back and forth, it provides mechanisms for loading an object's class definitions as well as for transmitting an object's data.

The following illustration depicts an RMI distributed application that uses the RMI registry to obtain a reference to a remote object. The server calls the registry to associate (or bind) a name with a remote object. The client looks up the remote object by its name in the server's registry and then invokes a method on it. The illustration also shows that the RMI system uses an existing web server to load class definitions, from server to client and from client to server, for objects when needed.



There are three processes that participate in supporting remote method invocation.

1. The *Client* is the process that is invoking a method on a remote object.
2. The *Server* is the process that owns the remote object. The remote object is an ordinary object in the address space of the server process.
3. The *Object Registry* is a name server that relates objects with names. Objects are *registered* with the Object Registry. Once an object has been registered, one can use the Object Registry to obtain access to a remote object using the name of the object.

In this tutorial, we will give an example of a Client and a Server that solve the classical "Hello, world!" problem. You should try extracting the code that is presented and running it on your own computer.

There are two kinds of classes that can be used in Java RMI.

1. A *Remote* class is one whose instances can be used remotely. An object of such a class can be referenced in two different ways:
    1. Within the address space where the object was constructed, the object is an ordinary object which can be used like any other object.
    2. Within other address spaces, the object can be referenced using an *object handle*. While there are limitations on how one can use an object handle compared to an object, for the most part one can use object handles in the same way as an ordinary object.

    For simplicity, an instance of a Remote class will be called a *remote object*.
2. A *Serializable* class is one whose instances can be copied from one address space to another. An instance of a Serializable class will be called a *serializable object*. In

other words, a serializable object is one that can be marshaled. Note that this concept has no connection to the concept of serializability in database management systems.

If a serializable object is passed as a parameter (or return value) of a remote method invocation, then the value of the object will be copied from one address space to the other. By contrast if a remote object is passed as a parameter (or return value), then the object handle will be copied from one address space to the other.

Serializable Classes

We now consider how to design Remote and Serializable classes. The easier of the two is a Serializable class. A class is Serializable if it implements the java.io.Serializable interface. Subclasses of a Serializable class are also Serializable. Many of the standard classes are Serializable, so a subclass of one of these is automatically also Serializable. Normally, any data within a Serializable class should also be Serializable. Although there are ways to include non-serializable objects within a serializable objects, it is awkward to do so. See the documentation of java.io.Serializable for more information about this.

Using a serializable object in a remote method invocation is straightforward. One simply passes the object using a parameter or as the return value. The type of the parameter or return value is the Serializable class. Note that both the Client and Server programs must have access to the definition of any Serializable class that is being used. If the Client and Server programs are on different machines, then class definitions of Serializable classes may have to be downloaded from one machine to the other. Such a download could violate system security. This problem is discussed in the Security section.

The only Serializable class that will be used in the "Hello, world!" example is the String class, so no problems with security arise.

Remote Classes and Interfaces

Next consider how to define a Remote class. This is more difficult than defining a Serializable class. A Remote class has two parts: the interface and the class itself. The Remote interface must have the following properties:

1. The interface must be public.
2. The interface must extend the interface java.rmi.Remote.
3. Every method in the interface must declare that it throws java.rmi.RemoteException. Other exceptions may also be thrown.

The Remote class itself has the following properties:

1. It must implement a Remote interface.
2. It should extend the java.rmi.server.UnicastRemoteObject class. Objects of such a class exist in the address space of the server and can be invoked remotely. While there are other ways to define a Remote class, this is the simplest way to ensure that objects of a class can be used as remote objects. See the documentation of the java.rmi.server package for more information.
3. It can have methods that are not in its Remote interface. These can only be invoked locally.

Unlike the case of a Serializable class, it is not necessary for both the Client and the Server to have access to the definition of the Remote class. The Server requires the definition of both the Remote class and the Remote interface, but the Client only uses the Remote interface. Roughly speaking, the Remote interface represents the type of an object handle, while the Remote class represents the type of an object. If a remote object is being used remotely, its type must be declared to be the type of the Remote interface, not the type of the Remote class.

In the example program, we need a Remote class and its corresponding Remote interface. We call these Hello and HelloInterface, respectively. Here is the file HelloInterface.java:

```
import java.rmi.*;
/**
 * Remote Interface for the "Hello, world!" example.
 */
public interface HelloInterface extends Remote {
  /**
   * Remotely invocable method.
   * @return the message of the remote object, such as "Hello, world!".
   * @exception RemoteException if the remote invocation fails.
   */
  public String say() throws RemoteException;
}
```

Here is the file Hello.java:

```
import java.rmi.*;
import java.rmi.server.*;
/**
 * Remote Class for the "Hello, world!" example.
 */
public class Hello extends UnicastRemoteObject implements HelloInterface {
  private String message;
  /**
   * Construct a remote object
   * @param msg the message of the remote object, such as "Hello, world!".
   * @exception RemoteException if the object handle cannot be constructed.
   */
  public Hello (String msg) throws RemoteException {
    message = msg;
  }
  /**
   * Implementation of the remotely invocable method.
   * @return the message of the remote object, such as "Hello, world!".
   * @exception RemoteException if the remote invocation fails.
   */
  public String say() throws RemoteException {
    return message;
  }
}
```

All of the Remote interfaces and classes should be compiled using javac. Once this has been completed, the stubs and skeletons for the Remote interfaces should be compiled by using the rmic stub compiler. The stub and skeleton of the example Remote interface are compiled with the command:

```
  rmic Hello
```

The only problem one might encounter with this command is that rmic might not be able to find the files Hello.class and HelloInterface.class even though they are in the same directory where rmic is being executed. If this happens to you, then try setting the CLASSPATH environment variable to the current directory, as in the following command:

```
  setenv CLASSPATH .
```

If your CLASSPATH variable already has some directories in it, then you might want to add the current directory to the others.

Programming a Client

Having described how to define Remote and Serializable classes, we now discuss how to program the Client and Server. The Client itself is just a Java program. It need not be part of a Remote or Serializable class, although it will use Remote and Serializable classes.

A remote method invocation can return a remote object as its return value, but one must have a remote object in order to perform a remote method invocation. So to obtain a remote object one must already have one. Accordingly, there must be a separate mechanism for obtaining the first remote object. The Object Registry fulfills this requirement. It allows one to obtain a remote object using only the name of the remote object.

The name of a remote object includes the following information:

1. The Internet name (or address) of the machine that is running the Object Registry with which the remote object is being registered. If the Object Registry is running on the same machine as the one that is making the request, then the name of the machine can be omitted.
2. The port to which the Object Registry is listening. If the Object Registry is listening to the default port, 1099, then this does not have to be included in the name.
3. The local name of the remote object within the Object Registry.

Here is the example Client program:

```
 /**
  * Client program for the "Hello, world!" example.
  * @param argv The command line arguments which are ignored.
  */
 public static void main (String[] argv) {
        try {
    HelloInterface hello =
      (HelloInterface) Naming.lookup ("//ortles.ccs.neu.edu/Hello");
    System.out.println (hello.say());
```

```
   } catch (Exception e) {
     System.out.println ("HelloClient exception: " + e); }
  }
```

The Naming.lookup method obtains an object handle from the Object Registry running on ortles.ccs.neu.edu and listening to the default port. Note that the result of Naming.lookup must be cast to the type of the Remote interface.

The remote method invocation in the example Client is hello.say(). It returns a String which is then printed. A remote method invocation can return a String object because String is a Serializable class.

The code for the Client can be placed in any convenient class. In the example Client, it was placed in a class HelloClient that contains only the program above.

Programming a Server

The Server itself is just a Java program. It need not be a Remote or Serializable class, although it will use them. The Server does have some responsibilities:

1. If class definitions for Serializable classes need to be downloaded from another machine, then the security policy of your program must be modified. Java provides a security manager class called RMISecurityManager for this purpose. The RMISecurityManager defines a security policy that allows the downloading of Serializable classes from another machine. The "Hello, World!" example does not need such downloads, since the only Serializable class it uses is String. As a result it isn't necessary to modify the security policy for the example program. If your program defines Serializable classes that need to be downloaded to another machine, then insert the statement System.setSecurityManager (new RMISecurityManager()); as the first statement in the main program below.
2. At least one remote object must be registered with the Object Registry. The statement for this is: Naming.rebind (objectName, object); where object is the remote object being registered, and objectName is the String that names the remote object.

POSSIBLE QUESTIONS

PART-B

(Each Question carries 6 Marks)

1. What is JSP? Explain evolution of dynamic content technologies in detail
2. Write a program to insert an applet into JSP page.
3. Develop a web page for online exam using Java Script.
4. What is Java Server Pages? Elaborate the evolution of Dynamic Content Technologies.
5. What are JSP directives? List out its types and explain.
6. Explain about the Relationships in JSP.
7. Discuss the flow of control in java server pages with example.
8. Write a JSP program to store and retrieve cookie information.
9. Discuss about RMI concept in detail.
10. Write a program to stream contents a file using JSP

PART-C

(One Compulsory Question carries 10 Marks)

1. Discuss about Java Server Pages
2. Write a JSP program to add applet to a JSP page
3. Elaborate RMI concept in detail with an example.
4. Write a JSP program to stream contents a text file using JSP

J2EE (17CSP301)
Unit V- Multiple Choice Questions

S.no	Question	Option 1	Option 2	Option 3	Option 4	Answer
1	A ______________ is called by a client to provide a web service, the nature of which depends on the J2EE application.	servlet	JSP classes	EJB	EIS	JSP classes
2	There are ____________ methods that are automatically called when a JSP is requested and when the JSP terminates normally.	2	3	4	5	3
3	The ___________ method is called first when the JSP is requested and is used to initialize objects and variables that are used throughout the life of the JSP.	jspInt()	jspDestroy	service()	request()	jspInt()
4	The ___________ method is automatically called when the JSP terminates normally.	jspInt()	jspDestroy	service()	request()	jspDestroy
5	The ______________ method is automatically called and retrieves a connection to HTTP.	jspInt()	jspDestroy	service()	request()	service()
6	There are ____________ factors that we must address when installing a JSP.	2	3	4	5	3
7	__________ tags define java code that is to be executed before the output of the JSP program is sent to the browser.	JSP	HTML	XML	DHTML	JSP
8	A JSP tag begins with a ___________.	</	<*	<%	<!	<%
9	A JSP tag ends with a ____________.	/>	*>	%>	!>	%>
10	There are ____________ types of JSP tags.	2	3	4	5	5
11	A _____________ tag opens with <%-- and closes with --%>.	comment	declaration statement	directive	expression	comment
12	A ___________ tag opens with <%!.	comment	declaration statement	directive	expression	declaration statement
13	A ___________ tag opens with <%@.	comment	declaration statement	directive	expression	directive

14	A __________ tag opens with <%=.	comment	declaration statement	directive	expression	expression
15	A __________ tag opens with <%.	comment	declaration statement	directive	scriptlet	scriptlet
16	There are __________ kinds of loops commonly used in a JSP program.	2	3	4	5	3
17	The _____________ loop repeats usually a specified number of times.	for	while	do...while	do… until	for
18	The _________ loop executes continuously as long as a specified condition remains true.	for	while	do...while	do… until	while
19	The __________ loop executes atleast once.	for	while	do...while	do… until	do...while
20	The __________ is the method used to parse a value of a specific field.	getParameter()	getParameterValues()	jspInit()	jspService()	getParameter()
21	There are _____________ predefined implicit objects that are in every JSP program.	2	3	4	5	4
22	There are __________ commonly used methods to track a session.	2	3	4	5	3
23	A JSP database system is able to share information among JSP programs within a ___________ by using a session object.	servlet	session	EJB	EIS	session
24	There are __________ steps necessary to make an object available to remote clients.	2	3	4	5	3
25	Method invoked by the client is called _______________.	server method	client method	RMI method	Remote method	client method
26	In addition to the methods that can be invoked by remote clients, the developer must also define other methods that support the processing of client-invoked methods. They are referred as _____________.	server method	client method	RMI method	Remote method	server method
27	In RMI, port number ___________ is the default port.	1099	1199	1299	1399	1099
28	The __________ method is used to locate the remote object.	myMethod()	lookup()	catch()	getMessage()	lookup()

29	The ______________ method returns a String object that is passed to the println() method.	myMethod()	lookup()	catch()	getMessage()	myMethod()
30	Any exceptions that are thrown while the client-side program runs are trapped by the ___________ block.	myMethod()	lookup()	catch()	getMessage()	catch()
31	The __________ calls the getMessage() method to retrieve the error message that is associated with the exception	myMethod()	lookup()	getMessage()	catch()	catch()
32	The __________ is at the center of every remote object because the remote interface defines how the client views the object.	API	remote interface	server program	client program	remote interface
33	RMI handles transmission of requests and provides the facility to load the object's bytecode, which is referred to as ___________.	static code loading	dynamic code loading	object code loading	bytecode loading	dynamic code loading
34	The _________ method registers the remote object with the RMI remote object registry or with another naming service.	rebind()	bind ()	unbind()	binder()	rebind()
35	A __________ serves as a firewall and grants or rejects downloaded code access to the local file system and similar privileged operations.	server program	client program	security manager	web browser	server program
36	Reference to a remote object can be __________.	bound	unbound	rebound	bound, unbound, and rebound	bound, unbound, and rebound
37	A JSP is called by a ____________.	server	client	web service	EJB	client
38	Once a _________ is created, it must be placed in the same directory as HTML pages. the root element of the deployment descriptor.	servlet	JSP	EJB	EIS	JSP
39	Once a __________ is created, it must be placed in a particular directory that is included in the CLASSPATH	servlet	JSP	EJB	EIS	servlet

40	There are ____________factors one must address when installing a JSP.	2	3	4	5	3
41	A JSP program consists of a combination of ____________.	servlets and HTML tags	servlets and EJB tags	HTML tags and JSP tags	servlets and JSP tags	HTML tags and JSP tags
42	A powerful feature available in __________ is the ability to change the flow of the program to truly create dynamic content for a web page based on conditions received from the browser.	servlet	JSP	EJB	EIS	JSP
43	The __________ statement in JSP is divided into several JSP tags.- beans> element.	IF	WHILE	DO…WHILE	SWITCH	SWITCH
44	A pair of HTML table data cell tags _________ are placed inside the FOR loop along with a JSP tag that contains an element of the array.	<TB>	<TD>	<TR>	<TC>	<TD>
45	JSP virtual machine runs on a _____________ .	web browser	web server	windows	DOS	web server
46	TOMCAT is one of the most popular JSP ____________.	web browser	client program	virtual machine	web server	virtual machine
47	Java Beans works on __________.	JDK	BDK	SDK	FDK	BDK
48	One of the following can be downloaded freely from the net.	Tomcat	Java	BDK	All of the above	4
49	The request string sent to the JSP by the browser is divided into _____________ general components that are separated by the question mark.	2	3	4	5	2
50	The secured version of HTTP is __________.	SHTTP	SVHTTP	HTTPS	HTTPSV	HTTPS
51	The __________ enables JSP programs to track multiple sessions simultaneously while maintaining data integrity of each session.	unique password	unique ID	unique username	unique name	unique ID
52	__________ attributes can be retrieved and modified each time the JSP program runs.	Servlet	JSP	Session	EJB	Session

53	A session object stores _____________.	implicit data	explicit data	attributes	hidden fields	attributes
54	One of the _________ syntax given below removes a page scope from the stack.	abstract Map peekPageScope()	abstract Map popPageScope()	abstract Map pushPageScope().	none of the above	abstract Map peekPageScope()