애플릿에서 웹 서비스까지, 자바 제너럴 프로그래밍

자바와 .NET은 컴퓨팅 영역 전반에 걸쳐 컴퓨팅 패러다임을 적용할 수 있는 근간이 되는 개발 플랫폼이자 실행 플랫폼을 의미한다. 자바 플랫폼은 자바 언어, 자바 바이트코드(Bytecode), 그리고 자바 가상머신(Java Virtual Machine)을 기반으로 하고 있으며, .NET 플랫폼은 C#, MSIL(Microsoft Intermediate Language), CLR(Common Language Runtime)을 기반으로 하고 있다. 그리고, 중요한 한가지 사실은 이러한 자바와 .NET은 모두 동시대의 컴퓨팅 기술을 모두 반영하려 하고 있으며, 이들은 서로간의 협력을 필요로 하고 있다는 것이다. 다시 말해서, 이제 자바와 .NET은 질시와 반목을 넘어 화해와 협력을 통해, 지금까지 서로가 쌓아놓은 많은 독립적인 컴퓨팅 자산들을 하나로 묶어 보다 편리하고 풍요로운 서비스를 인간에게 제공해야 한다는 것이다. 이것이 바로 새로운 컴퓨팅 패럼다임의 시대를 맞는 자세가 아닐까 한다.

박용우 소빅 기술연구소 선임연구원

온라인 자바 사이트들의 협의회인 JCO(JavaCommunity.Org)의 회장직을 맡고 있으며, 컴퓨팅 아키텍트 클럽 RnDClub.net과 컴퓨팅 프로젝트 팀 RnDTeam.net에서 왕성한 활동을 벌이고 있다. 그리고, 자바스터디네트워크(www.javastudy.co.kr)에 자바 관련 컬럼을 기고하고 있다. 현재 소빅 기술연구소에서 선임연구원으로 일하고 있다.

나는 목수의 아들이다. 그래서, 어려서부터 아버지가 목수로서 일하는 모습을 보면서 자랐다. 톱을 이용하여 나무를 자르고, 대패와 끌을 이용하여 나무를 다듬고, 망치를 이용하여 못을 박고, 미장 도구를 이용하여 시멘트를 바르고, 그렇게 집을 완성해 가는 모습은 어린 나에게는 신기하게만 느껴졌다. 시골이라 그리 큰집을 짓지는 않았지만 그래도 거기에서 일하는 사람들은 그들만의 고유의 역할을 수행하면서도 서로 도와가며 함께 일을 하는 모습을 보면서 컸다. 그러나, 세월이 지날수록 지으려는 집의 규모가 커지고, 집을 짓기 위해 필요한 장비들 역기 거대해지기 시작했다. 막말로 포크레인 앞에서 삽질할 수는 없는 것 아닌가? 이렇게 갈수록 많은 기능을 요구하고 또한 보다 큰 집을 짓기 위해서는 보다 좋은 장비들도 필요했고, 보다 고급 기술자들도 필요하지만, 무엇보다도 지으려는 집에 대한 충분한 설계와 집을 짓는 사람들에 대한 완전한 역할 분담은 물론 집을 짓기 위한 전반의 프로세스를 잘 관리하여 각자가 맡은 역할에 따라 생성한 결과들을 잘 조합하는 것이 중요하다고 아버지께서는 항상 말씀하셨다. 또한, 이러한 방법과 기술들은 집을 짓건, 창고를 짓건, 빌라를 짓건, 아파트를 짓건, 빌딩을 짓건 항상 일률적이고 일관성있게 적용될 수 있어야 한다고 하셨다. 다시 말해서, 일관되고 통일성있는 기술을 이용하여, 충분한 설계와 서로 간의 완벽한 역할 분리, 그리고 집짓는 전반적인 과정에서의 이들간의 협력을 잘 유도해야 한다는

것이었다. 이는 좋은 도구를 사용하는 것보다 더 안정적이고 견고한 집을 지을 수 있도록 해 주고, 또한 전반적인 일의 진행을 보다 빠르고 효율적으로 유지할 수 있다는 것이다.

오늘따라, 항상 당신의 자리에서 묵묵히 당신의 역할을 다 하시던 아버지가 보고 싶다.

자바의 프로그래밍 철학

자바는 기본적으로 안정적이고 견고한 시스템을 구축하고, 생성된 코드를 재사용할 수 있도록 하고 있다. 또한, 자바를 이용하여 어떤 프로그램을 작성하더라고, 항상 똑같은 자바 문법을 이용하여 일관되고 일률적인 프로그래밍을 할 수 있도록 해 준다. 프로그래램의 확장 역시 단순히 패키지를 설치하여 바로 사용할 수 있도록 함으로써, 순전히 자신이 고민해야 할 비지니스 로직의 구현과 새로운 기술의 습득에 대해서만 신경 쓸 수 있도록 하고 있다.

자바의 이해

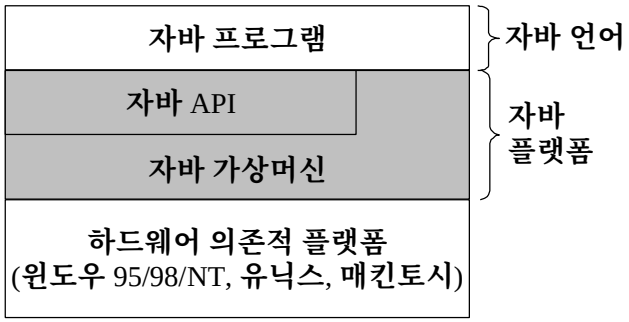
자바는 자바언어와 자바 플랫폼을 일컫는 말이다.

대부분의 사람들은 자바는 단순한 프로그래밍 언어라고 생각한다. 왜냐하면, 일반적으로 애플리케이션(Application), 애플릿(Applet), 서블릿(Servlet), JSP(JavaServer Pages), EJB(Enterprise JavaBeans), 미들릿(MIDlet) 등과 같은 프로그래밍 측면에서의 자바에 대해 많이 언급하기 때문이다. 그러나, 자바(Java)는 자바 가상머신 및 자바 API를 나타내는 자바 플랫폼과 자바 프로그램을 작성할 수 있도록 해 주는 자바 언어를 통칭하는 개념이다. <그림 1>에서는 이러한 자바에 대해 잘 보여주고 있다.

자바는 자바 플랫폼과 자바 언어로 구성되어 있으며, 각각에 대해 살펴보면 다음과 같다.

- 자바 플랫폼: 자바 프로그램을 실행시키기 위한 하드웨어 장치를 소프트웨어적으로 구현하여 제공해 주는 자바 가상머신(Java Virtual Machine)과 이 가상머신에서 프로그램을 작성할 때 사용할 수 있도록 제공되는 라이브러리 즉 자바 프로그래밍 API(Application Programming Interface) 등을 가리켜 자바 플랫폼(Platform)이라 한다. 자바 가상머신에서는 자바 언어로 작성되어 컴파일된 자바 바이트코드를 실행시켜 준다.
- 자바 언어: 자바 프로그램을 작성할 수 있도록 문법과 어휘를 제공해 주는 자바 프로그래밍 언어를 말한다. 자바 언어를 이용하여 자바 하드웨어 즉 자바 가상머신에서 실행하기 위한 프로그램을 작성할 수 있으며, 이 때 애플리케이션, 애플릿, 서블릿, JSP, EJB, 미들릿 등을 작성하기 위해서는 각 자바 프로그램에서 정의하고 있는 규약(Protocol)에 맞게 작성하면 된다. 예를 들어, 자바 애플리케이션을 작성하기 위해서는 반드시 public static void main(String args[]) 메소드를 갖는 클래스를 정의해 주어야 한다.

<그림 1> 자바(Java)



자바 언어를 이용하여 어떤 프로그램을 작성하더라도, 똑같은 자바 문법을 사용하여 .java 소스 파일을 작성하고, 똑같이 컴파일하며 바이트코드 형태의 .class 파일로 생성하고, 똑같이 자바 가상머신을 이용하여 실행하게 된다. 이 때, 어떤 프로그램을 작성하는냐에 따라 해당 프로그램에서 정의하고 있는 규약(또는 프로토콜)을 따르고, 자신이 구현해야 할 비지니스 로직에만 신경쓰면 된다. 다음은 자바 프로그램을 개발, 컴파일, 실행하기 위한 다음과 같은 과정을 잘 보여주고 있다. JSP 페이지에 대해 JSP 컨테이너가 내부적으로 자동으로 JSP 페이지 구현 클래스의 소스 코드를 생성하고, 컴파일해 주기는 하지만, 결과적으로는 다음과 같은 자바 프로그램의 개발 과정을 거치게 된다는 것이다.

1) 클래스 기반의 소스 코드 작성: 먼저, 자바 언어(Java Language)를 이용하여 자바 프로그램(Java Program)을 작성한다. 이 때, 자바 플랫폼에서 제공해 주는 자바 API를 사용할 수 있고, 또한, 작성하려는 자바 프로그램에 해당하는 규약에 맞게 작성해야 한다. 예를 들어, 자바 애플리케이션을 작성하기 위해서는 반드시 public static void main(String args[]) 메소드를 갖는 클래스를 정의해 주어야 한다. 이 때, 소스 파일 내에 정의되어 있는 클래스의 이름과 같은 이름의 .java 파일을 생성해 주어야 한다.
2) 컴파일: 자바 컴파일러(Java Compiler)를 이용하여 작성된 자바 프로그램을 자바 바이트코드(Java Bytecode) 형태로 컴파일된 .class 파일로 생성한다. 이 때, 컴파일 하는 파일 내에 정의되어 있는 각 클래스의 이름에 해당하는 이름을 갖는 .class 파일이 생성된다.
3) 실행: 자바 바이트코드 형태의 .class 파일을 각 플랫폼에 맞게 설치된 자바 가상머신에서 실행한다.

이렇게 어떤 자바 프로그램을 작성하더라고 기본적으로 자바 언어를 이용하여 일관되고 통일성 있게 작성한다는 것은 개발자에게 있어서 가장 큰 장점이라 할 수 있다. 또한, 자바 프로그램을 한 번 작성하여 컴파일하기만 하면 어떤 플랫폼에서도 실행가능하다는 것 역시 개발자에게 있어서 중요한 특징 중의 하나라 할 수 있다.

남을 위한 최소한의 배려, 자바 코드 규칙(Java Code Conventions)

자바에서는 기본적으로 자바 코드 규칙을 정의하고 있다. 다시 말해서, 자신이 작성한 프로그램을 다른 사람이 읽

기 쉽도록 하기 위해서는 프로그램을 작성하는 사람과 읽는 사람간에 약속이 있어야 하고, 이 때 "자바 코드 규칙(Java Code Conventions)"과 같은 관례가 필요하다. 이러한 기본적인 관례를 따라 프로그램을 작성함으로써 그 프로그램에 대한 가독성(Readability)을 높일 수 있다. 왜냐하면, 개발하려는 시스템이 커질수록 여러 사람들이 참여하게 되고, 이들간의 공통된 약속이 있어야 한다. 이렇게 참여자들간의 커뮤니케이션을 위한 기본적인 매카니즘이 정의되어 있다면, 여러 사람들이 각자의 역할에 충실하면서도 보다 빠르고 효율적으로 협업을 할 수 있을 것이다.

- 소프트웨어 비용의 80% 정도가 유지보수에 들어간다. 소프트웨어의 유지보수란 소프트웨어를 사용하면서 필요에 따라 소스 코드를 읽고, 수정하는 등의 작업을 말한다고 할 수 있다.
- 대부분의 경우 소프트웨어를 원래 개발한 사람이 끝까지 유지보수 할 수는 없다. 따라서, 유지보수 하는 사람은 다른 사람이 이미 작성해 놓은 소프트웨어를 읽고 이해해야 한다.
- 코드 규칙은 소프트웨어의 가독성(readability)을 높여 줌으로써, 개발자가 처음 접하는 새로운 소프트웨어를 보다 빠르고 완벽하게 이해할 수 있도록 해 준다.
- 소프트웨어를 상품으로서 판매할 경우, 소프트웨어 코드를 깔끔하게 작성해야 한다. 소프트웨어를 구입한 구매자의 경우 그 소프트웨어를 읽고 이해하여 새로운 기능을 구현하거나 또는 새로운 소프트웨어를 개발할 수도 있다.

이 때, 한 가지 중요한 것은 모든 사람이 코드 규칙을 만족하도록 프로그램을 작성해야 한다는 것이다. 코드 규칙이란 다른 사람을 위한 배려이고, 서로간의 약속이기 때문이다. 다음에 나오는 소스 코드에서는 자바 코드 규칙을 직접 적용한 예를 보여주고 있다.

```
/*
 * @(#)Blah.java 1.82 99/03/18
 *
 * Copyright (c) 1994-1999 Sun Microsystems, Inc.
 * 901 San Antonio Road, Palo Alto, California, 94303, U.S.A.
 * All Rights Reserved.
 *
 * This software is the confidential and proprietary information of Sun
 * Microsystems, Inc. ("Confidential Information"). You shall not
 * disclose such Confidential Information and shall use it only in
 * accordance with the terms of the license agreement you entered into
```

```
 * with Sun.
 */


package java.blah;

import java.blah.blahdy.BlahBlah;

/**
 * Class description goes here.
 *
 * @version 1.82 18 Mar 1999
 * @author Firstname Lastname
 */
public class Blah extends SomeClass {
    /* A class implementation comment can go here. */

    /** classVar1 documentation comment */
    public static int classVar1;

    /**
     * classVar2 documentation comment that happens to be
     * more than one line long
     */
    private static Object classVar2;

    /** instanceVar1 documentation comment */
    public Object instanceVar1;

    /** instanceVar2 documentation comment */
    protected int instanceVar2;
```

```
        /** instanceVar3 documentation comment */
        private Object[] instanceVar3;

        /**
         * ... constructor Blah documentation comment...
         */
        public Blah() {
            // ...implementation goes here...
        }

        /**
         * ... method doSomething documentation comment...
         */
        public void doSomething() {
            // ...implementation goes here...
        }

        /**
         * ...method doSomethingElse documentation comment...
         * @param someParam description
         */
        public void doSomethingElse(Object someParam) {
            // ...implementation goes here...
        }
    }
```

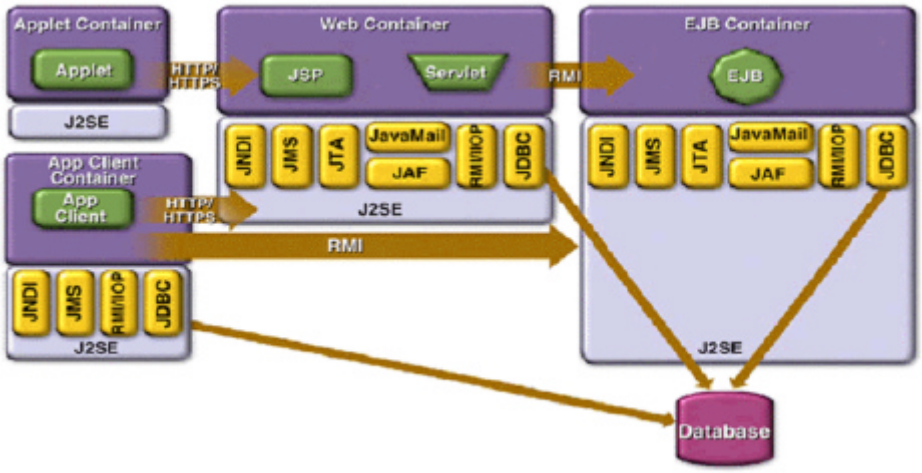
또한, 보다 자세한 내용은 다음의 사이트를 참고하기 바란다.

http://www.javastudy.co.kr/bbs/read.jsp?bbs=col_yoparkbbs&id=4

자바 프로그래밍 패러다임의 일관성

자바 프로그래밍 모델

<그림 2> 자바 프로그래밍 모델 (자바 프로그램 + 자바 API + 컨테이너 구조)



위의 그림에서는 자바를 이용하여 작성할 수 있는 프로그래밍 모델(또는 종류)에 대해 잘 보여주고 있다. 한 가지 재미있는 것은 <그림 2>에서 보여주고 있는 것과 같이 모든 자바 프로그램은 제공되는 자바 API를 이용하여 개발되고, 컨테이너 내에서 실행되도록 하는 구조를 갖고 있다는 것이다. 이렇게 컨테이너 내에서 각 프로그램이 실행되도록 함으로써, 보안, 분산 네트워킹, 라이프 사이클, 트랜잭션 처리 등과 같은 복잡한 작업을 프로그램의 개발자가 직접하지 않아도 된다는 것이다. 이러한 복잡한 작업은 컨테이너에서 하도록 하고, 개발자는 자신이 풀어야 할 비지니스 로직에만 신경쓸 수 있도록 해 준다는 것이다. 자, 이제부터 <그림 2>에서 보여주고 있는 각 프로그래밍에 대해서 살펴보도록 하겠다. 자바를 이용하여 할 수 있는 참으로 많고 다양한 프로그래밍 모델(또는 규약)이 존재하지만, 어떤 프로그램을 작성하더라도 해당 프로그램에서 정의하고 있는 규약만 따른다면, 일관성있고 통일된 프로그래밍을 할 수 있다는 것이다. 다시 말해서, 항상 똑같은 자바 프로그래밍 패러다임을 이용하여, 자신이 작성해야 하는 비지니스 로직에 대해서만 신경쓰면서 해당 프로그램 규약에 맞게 일관되고 통일성 있게 프로그램을 작성하면 된다는 것이다. <그림 2>에서 보여주고 있는 자바 프로그램을 살펴보면, 다음과 같다.

- 자바 애플리케이션 클라이언트와 애플리케이션 클라이언트 컨테이너
- 자바 애플릿과 애플릿 컨테이너
- 자바 서블릿과 웹 컨테이너

- JSP 페이지(구체적으로 JSP 페이지 구현 클래스)와 웹 컨테이너
- EJB와 EJB 컨테이너

요즘은, TCP/IP, HTTP, RMI, CORBA, RMI-IIOP, JNDI, JMS, JavaMail, JDBC 등을 이용한 분산 컴퓨팅 기반의 프로그래밍이 이루어지고 있는 추세이다. 이러한, 분산 환경은 기본적으로 프로그램(또는 객체, 컴포넌트) 간에 클라이언트/서버 구성을 갖게 한다. 한가지 재미있는 사실은 예전의 메인프레임을 이용한 머신 기반의 클라이언트/서버 구조에서와는 달리, 소프트웨어 기반의 클라이언트/서버 구조에서는 영원한 클라이언트도 영원한 서버도 존재하지 않는다는 것이다. 다시 말해서, 어떤 두 개의 프로그램 간에는 클라이언트/서버 구조를 갖지만, 어떤 경우에는 서로 역할이 뒤바뀌어 서버/클라이언트 구조가 될 수도 있다는 것이다. 또한, 다른 클라이언트에게 서비스를 제공하는 서버가 다시 다른 서버에게 서비스를 요청하기 위한 클라이언트로서 동작할 수도 있다는 것이다. <그림 2>에서는 클라이언트에 해당하는 모바일 프로그래밍에 대해서는 제외하고 있다. 왜냐하면, 앞으로 컴퓨팅 패러다임 자체가 클라이언트가 요청하는 정보를 제공하기 위한 모든 무거운 작업은 분산 컴퓨팅 환경을 기반으로 하는 서버에서 처리하여 결과 페이지까지 생성(랜더링)해 주고, 핸드폰 또는 PDA와 같은 클라이언트에서는 단지 서버에서 랜더링하여 제공한 페이지를 브라우징만 하면되도록 하고 있기 때문이다.

자바 기본 프로그래밍

자바를 이용하여 프로그램을 작성한다는 것은 자바 프로그램에서 사용할 클래스들을 하나씩 정의하는 것이다. 이러한 클래스는 다음과 같이 몇 가지로 나누어 생각해 볼 수 있다.

- 데이터 자체를 표현하기 위한 클래스: 프로그램에서 사용할 여러 가지 데이터를 클래스로 표현한다. 프로그램(또는 소프트웨어)의 가장 큰 목적 중의 하나는 바로 "데이터의 처리"이다.
- 데이터를 처리하는 로직을 표현하기 위한 클래스: 데이터를 처리하기 위한 로직을 표현하는 클래스를 작성한다. 물론, 경우에 따라서는 데이터와 로직이 같은 클래스 내에 나타날 수도 있을 것이다.
- 프로그램 규약 클래스: 데이터와 로직을 표현하기 위한 클래스를 모두 작성하였다면, 마지막으로 프로그램을 어떻게 서비스 해 줄것인지를 결정하고, 서비스할 프로그램의 종류에 해당하는 규약을 지키는 클래스를 작성한다. 예를 들어, 자바 애플리케이션으로 서비스 하고자 한다면 public static void main(String args[]) 메소드를 정의하는 클래스를 작성해야 할 것이다.

클래스의 작성과 선언 위치에 따른 구분

선언 위치에 따른 클래스의 분류

자바에서는 클래스의 선언 위치에 따라 다음과 같은 다섯 가지 클래스가 존재한다.

- 클래스: 우리가 일반적으로 말하는 클래스로서, 최상위 레벨에 선언되어 있는 클래스
- 중첩 클래스 (Nested Class): 클래스 내에 클래스 멤버(또는 스태틱 멤버)로서 선언되어 있는 클래스
- 내부 클래스 (Inner Class): 클래스 내에 인스턴스 멤버(또는 스태틱 멤버)로서 선언되어 있는 클래스
- 지역 클래스 (Local Class): 클래스 내의 인스턴스 메소드 내에 선언되어 있는 클래스
- 익명 클래스 (Anonymous Class): 인스턴스 메소드 내에 이름이 없이 선언되어 있는 클래스

일반 클래스 (Class)

자바에서 모든 프로그램의 단위는 클래스이다. 사용자에게 어떤 서비스를 제공하기 위한 자바 프로그램을 작성하기 위해서는, 먼저 프로그램에서 사용할 데이터를 표현하는 클래스를 정의해야 한다. 다음으로, 클래스로 표현되 데이터를 처리하기 위한 로직을 나타내는 클래스를 정의하게 될 것이다. 이 때, 데이터와 로직은 각각의 클래스로 나타날 수도 있고, 또는 하나의 클래스 내에 데이터와 로직이 함께 위치할 수도 있다. 예를 들어, 2차원 좌표 시스템에 존재하는 (x,y) 좌표를 표현하기 위한 Point 클래스를 정의하면 다음과 같다.

```
class Point {
    int x=0, y=0;

    public Point() {
    }

    public Point (int x, int y) {
        this.x = x;
        this.y = y;
    }

    public void setX(int x) {
        this.x = x;
    }

    public void setY(int y) {
```

```
        this.y = y;
    }

    public int getX() {
        return(x);
    }

    public int getY() {
        return(y);
    }
}
```

중첩 클래스(Nested Class)

(이달의 디스켓, NestingClass.java, NestingClass2.java)

클래스를 따로 선언하고 정의하기에는 간단하고 특정 클래스와 밀접한 관계를 갖는 경우, 또는 클래스로 정의해야 되지만 다른 클래스에서는 사용하지 않고 특정 클래스에서만 사용되는 클래스의 경우, 특정 클래스 내부에 static 키워드를 이용하여 클래스 멤버로서 선언할 수 있다. 이렇게 함으로써 클래스의 이름 짓는 것을 간단하게 하고 프로그램을 보다 알아보기 쉽게 할 수 있고, 프로그램의 구조를 보다 더 간단하고 알아보기 쉽게 할 수 있다. 그리고, 중첩 클래스가 포함하고 있는 메소드에서는 클래스 메소드에서처럼 인스턴스 변수 또는 인스턴스 메소드에 접근할 수 없지만, 클래스 변수 또는 클래스 메소드에는 접근할 수 있다. (이달의 디스켓, NestedClassTest.java)

클래스의 선언 지역	컴파일 했을 때 생성되는 파일	중첩 클래스에 대한 객체 생성

class A {
  static class B {
  ...
  }
  ...
}
	A.class
	A$B.class
		class NestingClass2 {
		  static class NestedClass2 {
		  ...
		  }
		  NestingClass2() {
		  NestedClass2 nested = new NestedClass2();
		  }
		}
		class NestedClassTest2 {
		  public static void main(String args[]) {
		  NestingClass2 nesting = new NestingClass2();
		  NestingClass2.NestedClass2 nested
		  = new NestingClass2.NestedClass2();
		  }

		}

내부 클래스 (Inner Class)

(이달의 디스켓, OuterClass.java, OuterClass2.java)

클래스 내부에 선언되는 클래스로서 중첩 클래스와는 달리 static 없이 인스턴스 멤버로서 선언된 클래스를 말한다. 중첩 클래스를 사용하는 것과 같이 내부 클래스를 사용하는 것 역시 프로그램의 구조를 더 좋게 할 수 있고, 특정 클래스와 관련 깊은 클래스를 그 클래스 외부에 따로 정의하지 않고 클래스의 내부에 정의함으로써, 프로그램을 보다 읽기 쉽고 알아보기 쉽게 해 준다. 이러한 내부 클래스는 이벤트 처리, 데이터 구조 클래스 등을 위해 자주 사용되며, 내부 클래스 객체는 생성 당시에 외부 클래스 객체에 대한 보이지 않는 참조값을 갖도록 초기화된다. 이를 통하여, 외부클래스 객체의 메소드와 필드가 내부 클래스 객체의 메소드와 필드인 것처럼 접근할 수 있게 해 준다.

클래스의 선언 지역	컴파일 했을 때 생성되는 파일	내부 클래스에 대한 객체 생성

class A {
  class B {
  ...
  }
  ...
}
 	
외부클래스의 변수 참조
  A.this.변수
외부클래스의 메소드 참조
  A.this.메소드
	A.class ✍ 외부클래스
	A$B.class ✍ 내부 클래스
		class OuterClass2 {
		  class InnerClass2 {
		  ...
		  }
		  OuterClass2() {
		  InnerClass2 nested = new InnerClass2();
		  }
		}
		class OuterClassTest2 {
		  public static void main(String args[]) {
		  OuterClass2 outer = new OuterClass2();
		  OuterClass2.InnerClass2 inner
		  = outer. new InnerClass2();
		  }
		}

지역 클래스(local class)

(이달의 디스켓, LocalClassTest.java, LocalClassTest2.java)

지역 클래스는 지역변수와 같이 메소드 내에 정의된 클래스를 가리킨다. 지역 클래스는 지역변수와 같이 메소드 내에서만 사용하고자 할 경우에 주로 사용한다. 이러한 지역 클래스는 메소드에서 자신을 포함하고 있는 클래스가 갖는 변수 및 메소드를 사용할 수 있듯이, 외부클래스가 포함하고 있는 변수 및 메소드를 사용할 수 있다. 그런데, 지역 클래스를 포함하고 있는 메소드가 리턴되고 난 후에도 지역 클래스 객체는 계속 존재하고, 계속 자신을 포함하고 있는 메소드의 지역변수에 접근해야 하기 때문에, 지역 클래스를 포함하고 있는 메소드의 지역변수 중 지역 클래스에서 참조하는 지역변수는 항상 final로 선언함으로써, 지역 클래스 객체가 언제라도 사용할 수 있도록 해 주어야 한다. 다시 말해서 지역변수는 메소드가 끝난 후에는 사라지게 되므로, 지역변수를 final로 선언함으로써 항상

메모리에 존재하도록 해 주어야 한다는 것이다. 또는, 지역변수로 선언하는 것이 아니라 클래스의 인스턴스 멤버변수로서 선언하는 것이 좋다.

클래스의 선언 지역	컴파일 했을 때 생성되는 파일	지역 클래스에 대한 객체 생성

class A {
public void m()
{
final int i=0;
class B {
...
}
...
}
...
}
	LocalClassTest2.class
	LocalClassTest2$1$PrimeChecker.class
		class LocalClassTest2 {
		Thread getPrimeChecker(final int number) {
		class PrimeChecker extends Thread {
		...
		}
		return(new PrimeChecker());
		}
		
		public static void main(String args[]) {
		LocalClassTest2 localClass = new LocalClassTest2();
		Thread primeChecker
		= localClass.getPrimeChecker(10);
		}
		}

익명 클래스 (Anonymous Class)

(이달의 디스켓, AnonymousClassTest.java)

단순한 클래스 또는 인터페이스를 정의하여 사용할 때, 여러 곳에서 사용하는 것이 아니고 단 한번만 정의하여 사용한다거나 할 경우 주로 익명 클래스를 사용할 수 있다. 익명 클래스는 클래스 또는 인터페이스에 대한 객체를 생성하면서 바로 클래스 또는 인터페이스를 정의하도록 되어 있습니다. 다시 말해서, new 수식이 있는 곳에서 바로 클래스 또는 인터페이스를 정의할 수 있으며, 이러한 익명 클래스는 주로 클래스 또는 인터페이스를 구현하여 바로 사용하고자 할 때 이용된다. 그리고, 익명 클래스를 사용할 때 한 가지 주의할 점은 익명 클래스는 new 수식의 연장이므로 끝에 꼭 세미콜론(;)을 붙여주어야 한다는 것이다. 다음 문장들과 마찬가지로 new 수식 역시 메소드 내에서만 사용되므로 이름이 없다는 것만 제외하고는 익명 클래스와 지역 클래스는 모든 점에서 같다. 따라서, 익명 클래스를 포함하고 있는 메소드의 지역변수 중 익명 클래스에서 참조할 수 있는 지역변수는 final로 선언된 지역변수만 가능하다.

클래스의 선언 지역	컴파일 했을 때 생성되는 파일	익명 클래스에 대한 객체 생성

class A {
public void m()
{
B b = new B
{
	AnonymousClassTest.class
	AnonymousClassTest$1.class
		class AnonymousClassTest {
		Thread getPrimeChecker(final int number) {
		return(new Thread() {
		public void run() {
		...

```
            ...
        }
        ...
    }
    ...
}
```

```
            }
        });
    }
    public static void main(String args[]) {
        Thread primeChecker
         =                                                    new
AnonymousClassTest().getPrimeChecker(10);
    }
}
```

자바 애플리케이션의 작성

프로그램을 자바 애플리케이션으로 서비스 한다는 것은, 현재 사용자가 앉아 있는 스탠드얼론 컴퓨터에서 프로그램을 직접 실행한다는 것을 말한다. 다시 말해서, 프로그램이 실행되는 위치와 서비스되는 위치(주로 사용자가 있는 위치)가 같다는 것을 말한다. 자바 애플리케이션으로 프로그램을 작성하고자 한다면, 반드시 자바 애플리케이션 규약(Protocol)에 해당하는 main() 메소드를 다음과 같이 정의하고 있는 프로그램 규약 클래스를 작성해 주어야 한다. 다음에 나오는 자바 프로그램은 자바 애플리케이션에 대한 예제로서, 특정 점을 생성한 후, 표준 출력에 점의 좌표를 출력해 주는 서비스를 제공하기 위한 프로그램이다.

```
public class PointDraw {
    public static void main(String args[]) {
        Point p = new Point(10, 20);

        System.out.println("p("+p.getX()+","+p.getY()+")");
    }
}
```

다음은 자바 AWT를 이용하여 점의 좌표값을 보여주기 위한 자바 애플리케이션 예제이다.

```
import java.awt.*;
import java.awt.event.*;

public class PointDrawAwt extends Frame {
    public PointDrawAwt(Point p) {
        Button b = new Button("p("+p.getX()+","+p.getY()+")");
```

```
        add(b);
        pack();

        b.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                System.exit(0);
            }
        });
    }

    public static void main(String args[]) {
        Point p = new Point(10, 20);

        PointDrawAwt f = new PointDrawAwt(p);
        f.setVisible(true);
    }
}
```

만약, 자바 스윙을 이용하여 점의 좌표값을 보여주고자 한다면 다음과 같이 해 주어야 할 것이다.

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class PointDrawSwing extends JFrame {
    public PointDrawSwing(Point p) {
        JButton b = new JButton("p("+p.getX()+","+p.getY()+")");
```

```
        getContentPane().add(b);
        pack();

        b.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                System.exit(0);
            }
        });
    }

    public static void main(String args[]) {
        Point p = new Point(10, 20);

        PointDrawSwing f = new PointDrawSwing(p);
        f.setVisible(true);
    }
}
```

<박스 1> 고급 UI 생성을 위한 스윙 컴포넌트와 MVC(Model, View, Controller) 모델

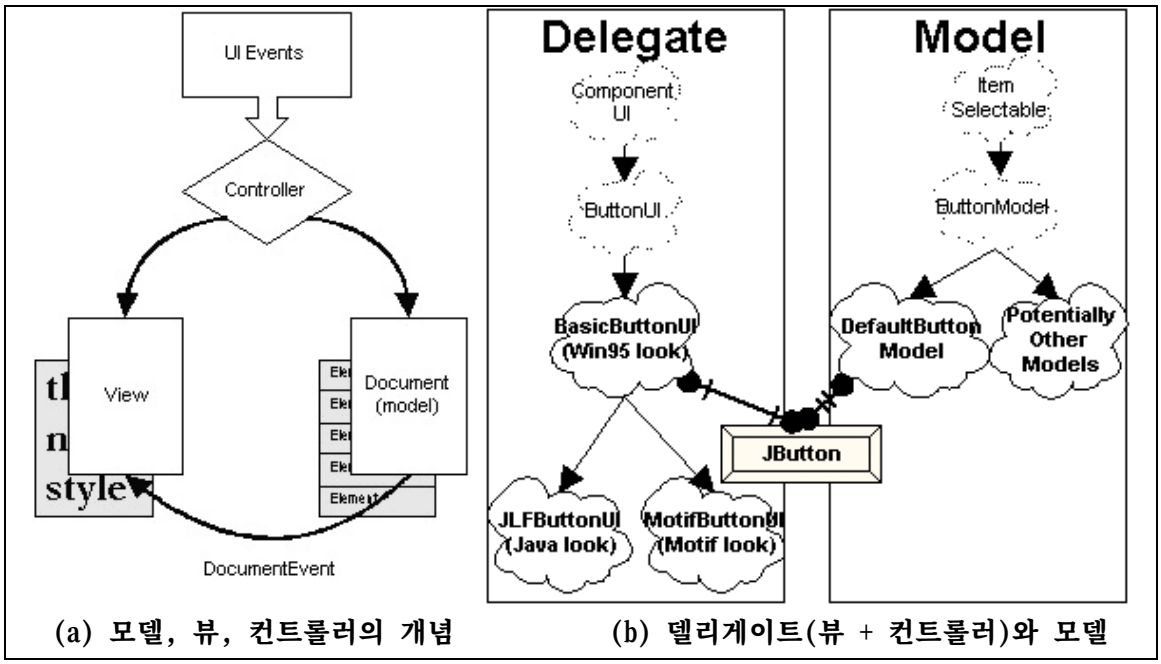
스윙 컴포넌트와 MVC 모델

스윙은 경량 컴포넌트(Light-weight component)이다. 기존의 AWT 컴포넌트가 실행되고 있는 플랫폼의 네이티브 코드를 호출하여 컴포넌트를 제공해 주는 피어(Peer) 구조를 가진 반면, 스윙은 모든 코드가 자바로 작성되었기 때문에 플랫폼 독립적인 동일한 인터페이스를 제공해 줄 수 있다. 이러한 스윙의 유연성을 가장 잘 보여주는 것이 바로 MVC(Model, View, Controller) 모델이다. MVC 모델은 유명한 객체지향언어인 Smalltalk에서 도입되었으며, JDK1.1 에서는 Observe/Observable 객체로 많이 사용되기도 했다. MVC 모델은 모델(Model)-뷰(View)-컨트롤러(Controller)로 구성된 구조이며, 뷰와 컨트롤러를 합쳐 델리게이트(Delegate)라 표현하기도 한다. 이렇게 부르는 이유는 어떻게 보여지고, 사용자 입력에 어떻게 반응하며, 데이터가 어떻게 표현되어지는지를 나타내기 때문이다. 다시 말해서, 사용자 인터페이스를 구축하기 위한 일종의 디자인 패턴(Design-Pattern)이라 할 수 있다. 모델, 뷰, 컨트롤러에 대해 살펴보면 다음과 같다.

- 모델(Model): 프로그램 상태에 대한 논리적인 표현으로서, 데이터가 변경되었을 경우에는 이를 뷰에게

통보한다.

- 뷰(View): 변하는 데이타에 대한 시각적인 표현을 제공하며 응답 메커니즘으로 컨트롤러를 사용한다.
- 컨톨로러(Controller): 사용자 입력을 어떻게 다뤄야 할지를 명세한다. 사용자와의 상호작용을 이끌어낸다.



(a) 모델, 뷰, 컨트롤러의 개념 (b) 델리게이트(뷰 + 컨트롤러)와 모델

이렇게 모델, 뷰, 컨트롤러로 분리함으로써 다음과 같은 장점을 제공해 준다.

- 같은 모델을 가지고 다양한 뷰를 생성할 수 있다. 즉, 하나의 테이터를 테이블(Table) 또는 차트(Chart)로 동시에 구현할 수 있다. 만약, 모델의 데이터를 수정하면 그 변경 내용이 뷰에 반영될 수 있도록 컨트롤러가 자동으로 반응하게 된다.
- 모델은 보여주기 위한 뷰와는 독립적으로 동작하기 때문에 뷰를 수정한다거나 새롭게 생성할 수 있다. 이 때, 뷰가 어떻게 변경이 된다고 하더라도 모델은 전혀 영향을 받지 않는다.

델리게이트(뷰 + 컨트롤러)는 뷰가 하는 역할(모델의 표현)과 컨트롤러의 역할(사용자 입력을 모델에 반영)을 수행한다. 이렇게 뷰와 컨트롤러를 하나로 합침으로써 컴포넌트의 디자인이 훨씬 쉬워질 수 있다. 예를 들어, Checkbox 컴포넌트가 갖는 상태값인 true 또는 false 값은 모델에 해당한다. 이러한 상태값을 화면상에 시각적으로 표현하는 부분은 델리게이트-뷰의 역할이다. Checkbox 컴포넌트에서 마우스를 클릭했을 때, 델리게이트-컨트롤러가 모델에 알리고, 이를 다시 델리게이션-뷰로하여금 변경된 모델을 반영하도록 한다. 위의 그림에서 보여주고 있는 것과 같이, 스윙에서 제공해 주고 있는 모든 위젯(Widget)은 JButton 컴포넌트처럼 Jcomponent 클래스의 하위클래스이다. 항상 JComponent 클래스는 하나의 모델과 관련 델리게이트를 갖는다. 하나의 클래스가 JButton의 모델로 행동하기 위해서는 ButtonModel 인터페이스를 구현해야 한다. 마찬가지로, 델리게이트는 Jcomponent 클래스에 해당하는 델리게이트 인터페이스(ButtonUI)를 구현해야 한다. JComponent 는 다양한 모델과 델리게이트로 구성되며, 사용자는 setModel() 메소드와 getModel() 메소드를 이용하여 모델에 접근할 수 있으며, setUI() 메소드와 getUI() 메소드를 이용하여 델리게이트를 액세스 할 수 있다.

<박스 2> 스윙에서 룩앤필 설정

룰앤픽 (Look and Feel)
스윙 컴포넌트는 경량 컴포넌트이다. 실행되고 있는 특정 플랫폼의 네이티브 메소드를 호출하는 중량 컴포넌트(AWT 컴포넌트 모델)와는 달리 스윙은 몇 가지를 제외하고는 대부분 자바로 순수 구현된 경량 컴포넌트라 할 수 있다. 이러한 경량 컴포넌트는 룩앤필 특성을 제공해 줄 수 있다. 다시 말해서, 다양한 외관을 가질 수 있으며, 런타임시에도 동적으로 외관을 바꿀수가 있다. 우선 Look&Feel(이하 l&f) 을 지정하기 위해서는 UIManager.setLookAndFeel() 을 사용할 수 있으며, 다음과 같이 변경할 수 있다.

```
UIManager.setLookAndFeel(UIManager.getCrossPlatformLookAndFeelClassName());
```

만약, 윈도우 룰앤픽을 적용하고 싶을 경우, 다음과 같이 할 수 있다.

```
UIManager.setLookAndFeel("com.sun.java.swing.plaf.windows.WindowsLookAndFeel")을 적용하면 된다.
```

setLookAndFeel 메소드의 매개변수는 완전한 클래스 이름을 나타내는 문자열이고, 컴포넌트가 생성되기 전에 프로그램이 룩앤필을 지정하면 UIManager는 특정한 룩앤필 클래스의 객체를 생성한다. 그리고, 성공적으로 수행하면 모든 컴포넌트는 설정된 룩앤필을 사용하게 될 것이다. 다음과 같은 세 가지 경우를 생각해 볼 수 있다.

- 만약 컴포넌트의 UI가 생성되기 전에 성공적으로 룩앤필을 명세하지 않으면, UIManager는 lib 디렉토리에 있는 swing.properties 라는 파일에서 사용자가 지정한 룩앤필이 있는지 검사한다.
- 사용자가 그 파일에 룩앤필을 지정하면 UIManager는 지정된 클래스를 초기화하고 객체화한다. 예를 들면, 다음과 같다.

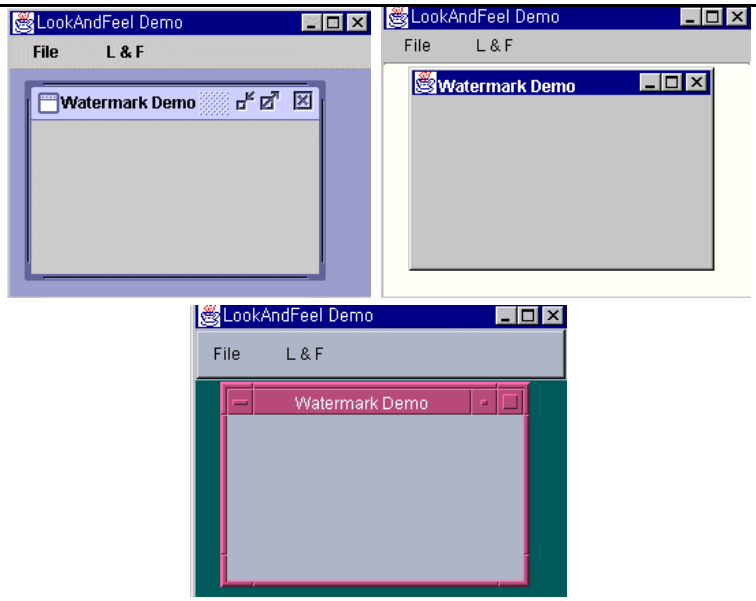
  ```
  # Swing properties
  swing.defaultlaf = com.sun.java.swing.plaf.motif.MotifLookAndFeel
  ```
- 위의 두 가지 경우가 아니라면, 프로그램은 디폴트인 Java l&f 을 사용하게 된다.

룩앤필을 얻기 위해서 다음과 같은 두 가지 메소드를 기본적으로 사용할 수 있다.

- UIManager.getCrossPlatformLookAndFeelClassName(): 플랫폼에 상관없이 사용할 수 있는 룩앤필을 리턴해 주는데, "javax.swing.plaf.metal.MetalLookAndFeel" 룩앤필을 리턴해 준다.
- UIManager.getSystemLookAndFeelClassName(): 현재 플랫폼에 맞는 룩앤필을 리턴해 준다. Win32 플랫폼에는 Windows 룩앤필을 Mac OS 에서는 Mac OS 룩앤필을 Sun 플랫폼에서는 CDE/Motif 룩앤필을 각각 리턴해 준다.

다음에 나오는 룩앤필을 왼쪽부터 순서대로 살펴보면, 각각 다음과 같다.

- CDE/Motif 룩앤필: "javax.swing.plaf.metal.MetalLookAndFeel"
- Windows 룩앤필: "com.sun.java.swing.plaf.windows.WindowsLookAndFeel"
- Mac OS 룩앤필: "javax.swing.plaf.mac.MacLookAndFeel"



(a) CDE/Motif 룩앤필 (b) Windows 룩앤필 (c) Mac OS 룩앤필

프로그램이 실행되고 있는 상태에서도 setLookAndFeel() 메소드를 이용하여 룩앤필을 변경할 수 있다. 이 때, 변경된 새로운 룩앤필을 컴포넌트에 적용하기 위해서는 반드시 SwingUtilities.updateComponentTreeUI() 메소드를 최상위 컨테이너에서 호출해 주어야 한다. 예를 들면, 다음과 같은 코드를 이용하여 룩앤필을 변경할 수 있다.

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import javax.swing.event.*;

public class LookAndFeel extends JFrame {
    static JFrame frame;
    JMenuBar bar;
    JTextArea textarea;
    ExitAction exit;
    LFAction lf;

    static String metal= "Metal";
    static String metalClassName = "javax.swing.plaf.metal.MetalLookAndFeel";

    static String motif = "Motif";
    static String motifClassName = "com.sun.java.swing.plaf.motif.MotifLookAndFeel";

    static String windows = "Windows";
    static String windowsClassName = "com.sun.java.swing.plaf.windows.WindowsLookAndFeel";

    public LookAndFeel() {
        super("LookAndFeel Demo");
```

```
        JDesktopPane desktop = new JDesktopPane();
        getContentPane().add(desktop);

        exit = new ExitAction();

        lf = new LFAction();

        bar = createMenuBar();

        JInternalFrame jif = new JInternalFrame("Watermark Demo", true, true, true, true);

        jif.setBounds(20,20,300,250);
        desktop.add(jif);

        setJMenuBar(bar);
    }

    JMenuBar createMenuBar() {
        JMenuItem m1,m2,m3;

        JMenuBar bar = new JMenuBar();
        bar.setOpaque(true);

        JMenu filemenu = new JMenu("File");
        filemenu.add(exit);

        JMenu lookMenu = new JMenu("L & F");

        m1 = new JMenuItem("Metal");
        m1.setActionCommand(metalClassName);
        lookMenu.add(m1);

        m2 = new JMenuItem("Motif");
        m2.setActionCommand(motifClassName);
        lookMenu.add(m2);

        m3 = new JMenuItem("Window");
        m3.setActionCommand(windowsClassName);
        lookMenu.add(m3);

        m1.addActionListener(lf);
        m2.addActionListener(lf);
        m3.addActionListener(lf);
```

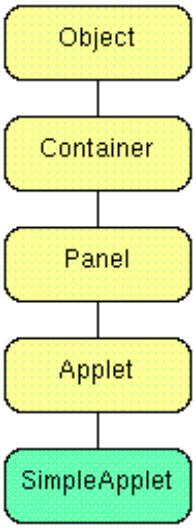
```
        bar.add(filemenu);
        bar.add(lookMenu);

        return bar;
    }

    public static void main(String args[]) {
        // set up frames, panels, text areas
        frame = new LookAndFeel();
        frame.addWindowListener(new WindowAdapter() {
            public void windowClosing(WindowEvent e) {
                Window win = e.getWindow();
                win.setVisible(false);

                System.exit(0);
            }
        });

        // make frame visible
        frame.setSize(600,450);
        frame.setVisible(true);
    }

    class LFAction extends AbstractAction {
        public void actionPerformed(ActionEvent e) {
            String lfs = e.getActionCommand();

            try {
                UIManager.setLookAndFeel(lfs);
                SwingUtilities.updateComponentTreeUI(frame);
            } catch(Exception e2) {
                System.err.println(e2);
                System.exit(0);
            }
        }
    }

    class ExitAction extends AbstractAction {
        public ExitAction() {
            super("나가기");
        }

        public void actionPerformed(ActionEvent e) {
            System.exit(0);
```

```
        }
    }
}
```

자바 애플릿의 작성

자바 애플릿(이하 애플릿)을 작성한다는 것은 애플릿 프로그램의 규약을 따르는 클래스를 작성한다는 것을 의미한다. 애플릿 프로그램의 규약이란, 애플릿과 애플릿 컨테이너(기존의 자바-호환 웹 브라우저 내에 내장되어 제공되는 자바 가상머신 또는 애플릿 뷰어) 간의 약속이라 할 수 있다. 다시 말해서, 애플릿 컨테이너가 애플릿을 실행하기 위해 정의하고 있는 라이프 사이클 메소드로서, 애플릿 컨테이너는 애플릿의 라이프 사이클 메소드를 규약에 따라 호출함으로써 애플릿이 실행될 수 있도록 하는 것이다. 자바 API에서 제공해 주고 있는 Applet 클래스는 Panel 클래스를 상속함으로써 애플릿의 화면 디자인과 애플릿의 라이프 사이클을 정의하는 메소드를 제공해 준다. 따라서, 애플릿의 화면 구성은 애플릿에서 상속하고 있는 Panel 컴포넌트에서 제공해 주는 기능을 이용할 수 있고, 애플릿의 라이프 사이클은 Applet 클래스에서 정의하고 있는 라이프 사이클 메소드를 재정의 함으로써 가능하다.

<그림 3> Applet 클래스의 상속 관계



Applet 클래스에서 정의하고 있는 애플릿의 라이프 사이클 메소드를 살펴보면, 다음과 같다.

- public void init() 메소드: init 메소드는 애플릿이 처음 생성됐거나 애플릿 컨테이너에 의해 메모리로 로드됐을때 호출된다. 이 메소드에서는 사용자 인터페이스의 생성이나 폰트 세팅과 같은 작업을 주로 수행한다.
- public void start() 메소드: start 메소드는 사용자가 애플릿을 포함한 웹페이지를 방문할 때 호출되는 메

소드이다. 애플릿이 시작되는 것을 알리기 위해 애플릿 컨테이너로부터 호출되는 메소드로서, 애니메이션의 시작이나 사운드 재생과 같은 애플릿의 시작에 필요한 작업이나 애플릿에서 사용하는 스레드를 시작하기 위한 작업을 주로 수행한다. start 메소드를 실행한 후에는 애플릿의 Panel에 그리기 위해 paint 메소드가 호출된다. 애플릿의 그리기 메소드인 paint 메소드는 항상 데디케이티드 드로윙(dedicated drawing)과 이벤트 핸들링(event-handling) 쓰레드로부터 호출된다.

- public void stop() 메소드: stop 메소드는 사용자가 다른 웹페이지로 이동할 때 애플릿을 중지할 수 있도록 하기 위해 호출되는 메소드이다. 일반적으로, stop 메소드에서는 애니메이션과 사운드를 정지하거나, 사용하고 있는 스레드를 정지하는 작업을 수행한다.
- public void destroy() 메소드: destroy 메소드는 브라우저가 종료될 때 호출된다. 이 메소드는 남아있는 쓰레드의 중지와 같은 마지막 청소 역활을 한다.

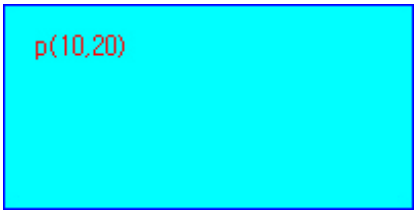
다음에 나오는 예제 애플릿에서는 필요에 따라 Applet 클래스의 init, paint 메소드를 제공하고 있다.

```
import java.awt.*;
import java.applet.Applet;

public class PointApplet extends Applet {
    Point p=null;

    public void init() {
        p = new Point(10, 20);
        setBackground(Color.cyan);
    }

    public void start() {
        System.out.println("starting...");
    }

    public void stop() {
        System.out.println("stopping...");
    }
```

```
    public void destroy() {
        System.out.println("preparing to unload...");
    }

    public void paint(Graphics g){
        System.out.println("Paint");
        g.setColor(Color.blue);
        g.drawRect(0, 0, getSize().width -1, getSize().height -1);
        g.setColor(Color.red);
        g.drawString("p("+p.getX()+","+p.getY()+")", 15, 25);
    }
}
```

위의 코드에서 보여주고 있는 것과 같이 애플릿을 위한 PointApplet 클래스는 public 권한으로 설정해 주어야 한다. 이는 로컬이 아닌 애플릿 컨테이너(브라우저 or 애플릿뷰어(appletviewer))에서 애플릿을 실행하기 의해 Applet 클래스를 참조할 수 있도록 하기 위한 것이다. 위와 같이 애플릿을 작성했을 경우에는 해당 애플릿을 실행시켜 주어야 한다. 애플릿은 HTML 페이지와 함께 웹 서버에 존재하고, 웹 브라우저에 의해 HTML 페이지와 함께 다운로드되어 해당 HTML 페이지 내에서 브라우저에 내장되어 있는 애플릿 컨테이너에 의해 실행된다. 이를 위해, 다음과 같이 애플릿을 위한 클래스를 HTML 페이지 내에 포함시켜 주어야 한다.

```
<HTML><BODY>
<APPLET CODE=PointApplet.class WIDTH=200 HEIGHT=100></APPLET>
</BODY></HTML>
```

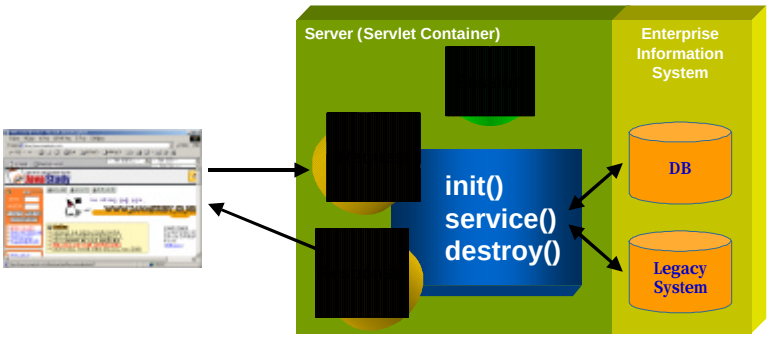


자바 서블릿의 작성

자바 서블릿은 서버의 기능을 향상시키기 위한 자바 서버측 기술이다. 일반적으로 서블릿은 동적인 웹페이지(또는 HTML 페이지)를 생성하기 위해 주로 사용된다. 자바 서블릿 기술은 <그림 4>에서 보여주고 있는 것과 같이 웹 클라이언트로부터 온 요청을 받아 자바 서블릿에서 처리한 후 그 결과를 응답으로서 되돌려주기 위한 자바 서버측

프로그래밍 기술입니다. 이러한 작업을 하기 위란 자바 서블릿은 자바 서버 즉 서블릿 컨테이너에서 실행되고, 클라이언트가 쇼핑한 물품 등과 같은 데이터를 유지하기 위해 세션을 사용할 수 있고, 웹 서버에 존재하면서 방대한 데이터를 저장하고 있는 기존의 데이터베이스 또는 레거시 시스템에 접근할 수도 있는 자바 프로그램입니다. 이러한 자바 서블릿은 자바 프로그램이기 때문에 하나의 클래스로서 정의되어야 하며, 서블릿 컨테이너에 의해 생성(인스턴스의 생성)되어 실행되므로 다음과 같은 생명주기(life-cycle) 메소드를 갖게 됩니다.

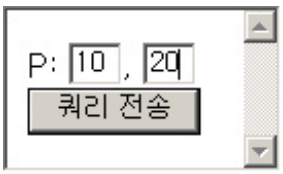
<그림 4> 자바 서블릿의 기본적인 구조



- public void init() 메소드: 서블릿 컨테이너는 요청을 처리할 서블릿의 인스턴스를 생성한 후, 서블릿이 초기화 작업을 수행할 수 있도록 init() 메소드를 내부적으로 자동으로 호출해 줍니다. 따라서, 서블릿에서 초기화시에 해야 할 작업이 있을 경우에는 서블릿 클래스의 init() 메소드 내에 정의해 주어야 합니다.
- public void service(...)메소드: 서블릿 컨테이너는 서블릿 인스턴스의 init() 메소드를 호출하여 서블릿이 초기화 작업을 수행할 수 있도록 해 줍니다. 그런 후에, 서블릿 인스턴스의 service(...) 메소드를 호출하여 서블릿에서 클라이언트로부터 온 요청을 받아 처리할 수 있도록 해 줍니다. 이 때, 서블릿이 요청과 관련된 작업을 할 수 있도록 하기 위한 ServletRequest 객체와 응답과 관련된 작업을 할 수 있도록 하기 위한 ServletResponse 객체를 service() 메소드를 호출하면서 매개변수로 전달해 줍니다. 따라서, 서블릿에서는 ServletRequest 객체를 이용하여 요청을 처리하고 응답에 해당하는 결과 페이지를 ServletResponse 객체에서 얻은 PrintWriter 객체를 통해서 출력해 줍니다.
- public void destroy()메소드: 서블릿이 요청과 관련된 작업을 모두 마치게 되면, 서블릿 컨테이너에 의해 스레드 풀에 들어가거나 또는 메모리에서 내려가게 됩니다. 스레드 풀에 들어갈 때는 나중에 다시 실행되기 때문에 상관없지만 메모리에서 내려갈 때는 서블릿이 쓰레기 수집(gabage collection)되는 것을 의미하기 때문에 서블릿이 마지막으로 마무리 작업을 하도록 해 주어야 합니다. 이렇게 서블릿이 죽기전에 유언을 할 수 있도록 기회를 주기 위해 서블릿 컨테이너에서는 서블릿 인스턴스의 destroy() 메소드를 호출해 주는 것입니다. 따라서, 서블릿이 메모리에서 내려가면서 마지막으로 수행해야 할 작업이 있다면 destroy() 메소드 내에 정의해 주어야 합니다.

만약, 클라이언트로부터 점의 (x, y) 좌표를 입력받아 Point 객체를 생성한 후, 해당 Point 객체가 표현하고 있는 좌표를 사용자에게 다시 되돌려 주어야 한다고 하자. 먼저, 다음과 같이 사용자로부터 점의 (x, y) 좌표를 입력받기 위한 HTML 폼을 제공해 주어야 한다.

```
<HTML><BODY>
<form action=/servlet/PointServlet method=POST>
P: <input type=text size=2 name=x>,
   <input type=text size=2 name=y><br>
   <input type=submit>
</form>
</BODY></HTML>
```



이 때, 위와 같은 HTML 페이지로부터 요청을 받아 해당 서비스를 제공하기 위한 서블릿의 소스 코드를 살펴보면, 다음과 같다.

```
import java.io.*;
import javax.servlet.*;

public class PointServlet extends GenericServlet {
   public void service(ServletRequest request,
                       ServletResponse response)
   throws ServletException, IOException {
      response.setContentType("text/html; charset=euc-kr");
      PrintWriter out = response.getWriter();

      int x = Integer.parseInt(request.getParameter("x"));
      int y = Integer.parseInt(request.getParameter("y"));
```

```
        Point p = new Point(x, y);

        out.println("<html>");
        out.println("<head>");
        out.println("p("+p.getX()+","+p.getY()+")");
        out.println("</body>");
        out.println("</html>");
    }
}
```

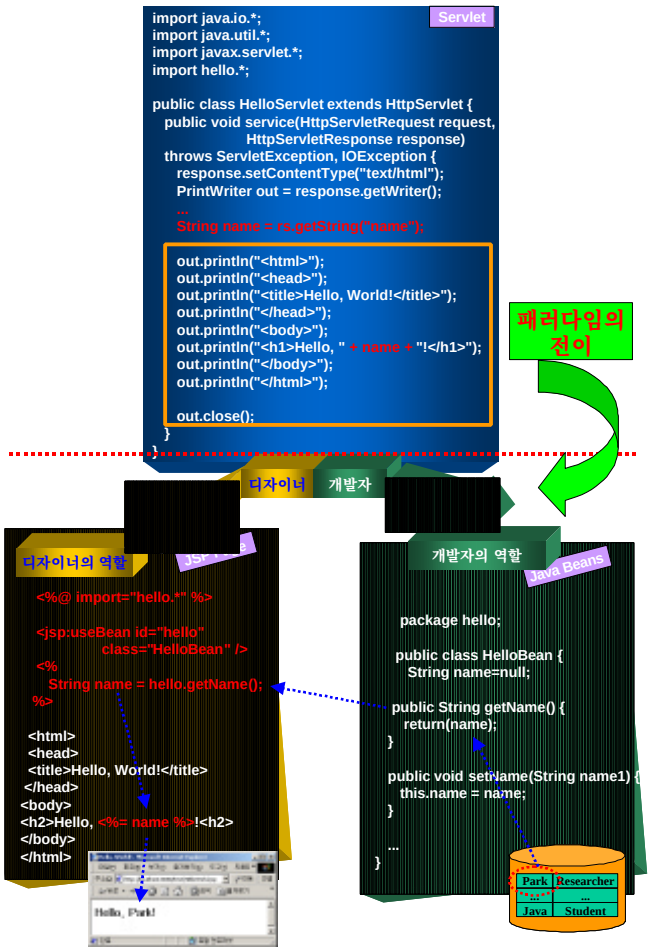
만약, HTTP 서블릿으로 작성하고자 한다면 다음과 같이 HttpService 클래스를 상속하도록 하고 service 메소드를 변경해 주어야 할 것이다.

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class PointHttpServlet extends HttpServlet {
    public void doPost(HttpServletRequest request, HttpServletResponse response)
    throws ServletException, IOException {
        response.setContentType("text/html; charset=euc-kr");
        PrintWriter out = response.getWriter();

        int x = Integer.parseInt(request.getParameter("x"));
        int y = Integer.parseInt(request.getParameter("y"));

        Point p = new Point(x, y);

        out.println("<html>");
```

```
            out.println("<head>");
            out.println("p("+p.getX()+","+p.getY()+")");
            out.println("</body>");
            out.println("</html>");
        }
    }
```

JSP 페이지의 작성

<그림 5> 개발자와 디자이너의 역할을 완전하게 분리하기 위한 새로운 패러다임의 출현



자바 서블릿 기술은 로직이 중요한 역할을 차지하는 웹 애플리케이션을 개발하기 위한 자바 서버측 기술이고, JSP 페이지는 프리젠테이션이 중요한 역할을 차지하는 웹 애플리케이션을 개발하기 위해 사용될 수 있는 자바 서버측 기술이다. JSP 기술에서는 디자이너와 개발자는 서로 협업을 통해 클라이언트에게 제공할 하나의 웹 애플리케이션을 개발할 수 있도록 하고 있다. 다시 말해서, 웹 애플리케이션의 개발에 참여하는 디자이너와 개발자는 다음과 같

은 서로의 역할에 따라 작업을 할 수 있다는 것이다.

- 디자이너의 역할: 디자이너는 웹 애플리케이션에서 제공하는 각종 서비스에 대한 결과 페이지에 해당하는 HTML 코드를 작성한다. 이렇게 디자이너가 작성한 HTML 코드는 어떤 서비스에 대한 요청에 맞게 결과 페이지를 동적으로 생성할 때 기본적인 골격으로 사용하게 될 것이며, 이러한 HTML 코드를 정적 템플릿(Static Template)이라 한다.
- 개발자의 역할: 개발자는 요청에 따라 그에 해당하는 결과 페이지를 생성하기 위한 서버측 프로그램을 작성한다. 이 때, 요청에 해당하는 동적인 결과 페이지를 생성하기 위해 디자이너가 디자인한 기본적인 HTML 코드(또는 정적 템플릿)를 사용하게 될 것이며, 사용자의 요청에 해당하는 동적 데이터를 생성한 후, 동적 데이터를 HTML 코드의 해당 위치에 추가해 주게 될 것이다.

위와 같이, 디자이너와 개발자의 역할을 완전히 분리할 수 있도록 하기 위해 JSP에서는 지시자 엘리먼트, 스크립트 엘리먼트, 액션 엘리먼트 등과 같은 태그 엘리먼트와 자바빈을 제공해 주고 있으며, 또한 사용자 정의 태그를 사용할 수 있도록 하기 위한 태그 라이브러리 기술을 제공해 주고 있다. 이러한 JSP 페이지에서 클라이언트로부터 점의 (x, y) 좌표를 입력받아 Location 객체를 생성한 후, 해당 Location 객체가 표현하고 있는 좌표를 사용자에게 다시 되돌려 주기위한 JSP 페이지를 작성하면 다음과 같다. JSP에서는 기본적으로 디자이너와 개발자의 역할을 분리함으로써 보다 작은 단위의 작업을 하도록 하고 있다. 따라서, 먼저 디자이너의 역할을 기반으로 하여 생성된 JSP 페이지를 살펴보면 다음과 같다.

```
<%@ page contentType="text/html; charset=euc-kr" %>
<%@ page import="test. Point" %>
<jsp:useBean id="p" class="Point" scope="page" />
<jsp:setProperty name="p" property="*" />
<html><head>
p(<jsp:getProperty name="p" property="x" />, <jsp:getProperty name="p" property="y" />)
</body></html>
```

다음으로, 프로그래머의 역할에 해당하는 자바빈을 살펴보면, 다음과 같다. 자바 빈을 작성할 때 한가지 주의할 점은 JSP 페이지 구현 클래스는 JSP 컨테이너에서 사용하는 작업 디렉토리에 생성된다. 그런데, 자바빈은 여러분이 등록한 자바빈 패키지 디렉토리에 존재할 것이다. 따라서, JSP 페이지에서 다른 패키지에 존재하는 자바빈을 액세스 하기 위해서는 반드시 자바빈 클래스를 public 권한으로 선언해 주어야 하며, 또한 getter 메소드 및 setter 메소드 역시 public 권한으로 선언해 주어야 한다. 마지막으로, 여러 JSP에서 사용하는 자바빈 클래스가 많아지면서

복잡해 질 수 있으므로 반드시 패키지 형태로 제공해 주기 위해 package 선언을 해 주어야 한다는 것이다. 위의 JSP 패키지에서 사용하고 있는 test.Location(test 패키지의 Location 클래스) 자바빈 클래스를 살펴보면 다음과 같다. 자바 AWT에서 제공해 주는 Point 클래스와 충돌이 발생할 수 있으므로 Location 클래스로 이름을 변경하였다.

```
package test;

public class Point {
    int x=0, y=0;

    public Point() {
    }

    public Point(int x, int y) {
        this.x = x;
        this.y = y;
    }

    public int getX() {
        return(x);
    }

    public int getY() {
        return(y);
    }

    public void setX(int x) {
        this.x = x;
    }

    public void setY(int y) {
        this.y = y;
```

```
    }
}
```

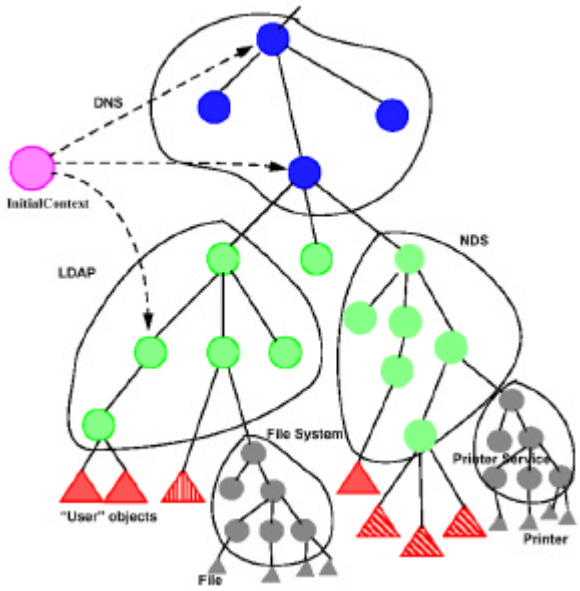
자바 분산 컴퓨팅 (Java Distributed Computing) 기술

분산 컴퓨팅 기술이란 어떤 서비스를 제공하기 위해 프로그램에서 제공하는 객체(또는 컴포넌트)가 두 대 이상의 머신(또는 플랫폼)에 분산되어 실행되도록 하기 위해 사용될 수 있는 컴퓨팅 기술을 말한다. 먼저, 하나의 서비스를 제공하는 프로그램을 구성하고 있는 객체(또는 컴포넌트)가 어떤 머신에 존재하는지를 찾아야 하는데, 이 때 사용할 수 있는 기술이 바로 룩업 서비스(네이밍/디렉토리 서비스)이다. 이렇게 서비스를 제공하는 객체를 찾은 후에는 객체에서 정의하고 있는 메소드를 호출하여 서비스를 제공받아야 한다.

룩업 서비스 (Lookup Service)

룩업 서비스의 가장 일반적인 예를 들자면, 인터넷 상에서 가장 일반적으로 사용하고 있는 DNS(Domain Name Service)가 있다. DNS는 IP(Internet Protocol) 주소를 호스트의 이름으로 맵핑해 주는 서비스를 말한다. 프로그램은 호스트의 이름에 해당하는 IP 주소를 찾기 위해 DNS 서비스를 사용하고 IP주소를 통하여 통신을 하게 된다. 네이밍 서비스에 이어, 몇몇 룩업 프로토콜은 디렉토리 서비스를 제공한다. LDAP(Lightweight Directory Access Protocol)와 선의 NIS+와 같은 디렉토리 서비스는 단순 네이밍 서비스보다 다른 정보와 서비스를 제공한다. 예를 들어, NIS+는 사용자 계정에 워크그룹(workgroup) 속성과 연관시키고, 이 속성은 워크그룹 속성을 가진 사용자만 머신을 액세스할 수 있도록 제한하는데 사용할 수 있다. 다음에 나오는 <그림 6>은 인터넷 상에서 찾고자 하는 자원을 빠르고 쉽게 찾을 수 있도록 하기 위한 룩업 서비스 간의 계층 구조를 보여주고 있다.

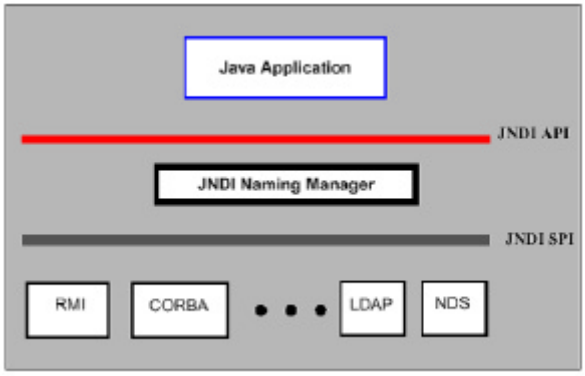
<그림 6> 인터넷 상의 네이밍 및 디렉토리 서비스의 구조



JNDI (Java Naming and Directory Interface)

JNDI API는 다양한 공급자가 자바 언어로 작성된 프로그램에 룩업 서비스의 접속을 쉽게 할 수 있도록 해 준다. 클라이언트와 서버가 같은 룩업 서비스를 사용하는한 클라이언트는 서버에 등록된 정보를 쉽게 찾을 수 있고 네트워킹할 수 있다. EJB 아키텍처에서도 분산되어 있는 플랫폼 상에서 세션빈(Session Bean) 또는 엔터티빈을 찾기위해 JNDI를 사용한다. 실제로 JNDI는 다음에 나오는 <그림 7>에서 보여주고 있는 것과 같이 아주 간단한 계층 구조를 갖고 있다.

<그림 7> JNDI 의 계층 구조



실제로, 클라이언트 애플리케이션, 서블릿, JSP, 세션빈 등과 같은 자바 프로램에서 사용하려는 객체를 찾기(룩업, Lookup) 위해, JNDI API를 사용할 수 있다. JNDI API에서는 각 서비스 제공자들이 제공해 주는 JNDI SPI를 이용

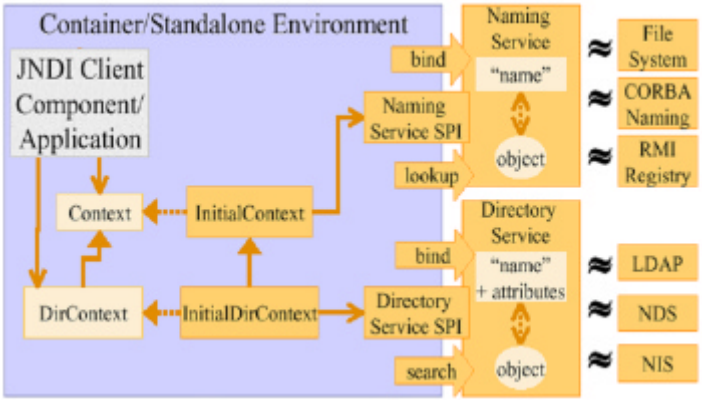
하여 사용자가 요청한 자원을 찾게 된다. 왜냐하면, 실제로 현재 사용되고 있는 룩업 서비스는 DNS(Domain Name System), COS(Common Object Services) 등의 네이밍 서비스 표준과 LDAP (Lightweight Directory Access Protocol), NDS(NetWare Directory Service), NIS(Netware Information System) 등의 표준이 존재하기 때문에 각 서비스 제공자에서 JNDI SPI를 제공해 주어야 한다. 물론, JNDI 애플리케이션의 개발자는 JNDI API만을 이용하여 프로그래밍 하면 되고, 실제로 클라이언트 애플리케이션에서 JNDI API를 이용하여 EJB 객체를 찾기 위해 다음과 같이 해 주면 된다.

```
import javax.naming.Context;
import javax.naming.InitialContext;
...
Context initial = new InitialContext();
Object objRef = initial.lookup("Hello");
```

JNDI는 기본적으로 다음과 같은 특징을 제공해 주고 있으며, 이는 다음에 나오는 <그림 8>에서 보여주고 있는 것과 같은 아키텍처를 갖기 때문이다.

- 네이밍과 디렉토리 서비스는 네트웍 상에 존재하는 거의 모든 요소의 정보들을 공유할 수 있도록 해 줌
- 자바로 작성된 응용 프로그램이 네이밍 및 디렉토리 서비스에 접근할 수 있도록 제공되는 API
- 자바 응용 프로그램이 어느 위치에 존재하든 필요한 자바 객체들을 검색할 수 있다
- 엔터프라이즈 자바빈즈 환경에서는 JNDI를 이용해 EJB Home 객체를 얻어낸 후 이를 이용해 자바빈즈 객체를 생성하거나 액세스 할 수 있다.
- DNS(Domain Name System), COS(Common Object Services) 등의 네이밍 서비스 표준과 LDAP (Lightweight Directory Access Protocol), NDS(NetWare Directory Service), NIS(Netware Information System) 등의 표준에 독립적인 API를 제공한다.
- 서비스를 제공하는 측이 제공하는 인터페이스(SPI, Service Provider Interface)를 통해 표준에 관계없이 디렉토리 서비스에 접근할 수 있다.

<그림 8> JNDI API & SPI 구성도



위의 <그림 8>에서 보여주고 있는 것과 같이, JNDI API에서 제공해 주는 네이밍 서비스를 사용할 경우에는 단지 InitailContext(또는 Context)에서 제공해 주는 기능에 대해서만, 디렉토리 서비스를 사용할 경우에는 단지 InitialDirContext(또는 DirContext)에서 제공해 주는 기능에 대해서만 알면 된다.

RMI (Remote Mothod Invocation)

RMI 네이밍 서비스는 다른 룩업 서비스나 네이밍 서비스와 유사하다. 실제로 Naming.lookup 메소드를 호출하면서 URL 매개변수를 전달함으로써 룩업을 수행한다. URL에는 머신이름, 객체가 동작중인지 알고 있는 RMI 네이밍 서버(rmiregistry)의 포트, 참조하고 메소드를 호출할 리모트 객체를 명시해야 한다. RMI는 RMI 네이밍 서버(rmiregistry)와 접속을 초기화하고 한번의 호출만으로 리모트 참조(객체)를 얻을 수 있다. 리모트 참조는 rmiregistry로부터 클라이언트까지 리스(lease)되는 것이다. 이 때, 클라이언트가 서버에게 "리모트 객체에 대한 참조가 여전히 필요하다"라고 알리지 않으면 리스(lease)는 해지되고, 메모리는 반환되게 된다. 이러한 리스는 사용자가 모르게 투명하게 진행되나, java.rmi.dgc.leaseValue의 값(millisecond)을 아래와 같이 서버 시작시 조정해 줄 수 있다.

RMI-IIOP (RMI Over Internet Inter-ORB Protocol)

RMI-IIOP는 OMG(Object Management Group)의 산업 표준 IIOP(Inter-Orb Protocol) 상에서 자바 RMI API를 제공해 준다. 다시 말해서, RMI-IIOP는 RMI에서 CORBA CosNaming 서비스를 사용하여 객체를 찾고 참조할 수 있도록 해 준다. RMI-IIOP를 이용하여 개발자는 클라이언트와 서버 간의 리모트 인터페이스를 작성할 수 있고, 이러한 인터페이스는 단순히 자바 기술과 자바 RMI API를 이용하여 구현할 수 있다. 이는 기존의 RMI 코드와 크게 다르지 않기 때문에 아키텍쳐 사이에 큰 상호운용성을 주고 있다. 이를 위해, rmic 컴파일러에서는 –iiop 옵션을 제공하여 RMI-IIOP에 필요한 스텁(stub)과 타이(tie) 클래스를 생성할 수 있도록 하고 있다. RMI-IIOP 룩업을 사용할 경우 RMI의 Naming.lookup 메소드 대신 InitialContext의 인스턴스를 사용해야 하며, 반환된 객체는

javax.rmi.PortableRemoteObject 클래스의 narrow 메소드를 사용하여 요청된 객체로 맵핑해 주어야 한다.

객체 직렬화 (Object Serialization)

자바 직렬화 매카니즘

자바 직렬화 메커니즘은 자바언어의 가장 뛰어난 특징중의 하나인 단순성과 유연성을 제공해 주고 있다. 직렬화는 휘발성을 갖는 메모리 장치에 존재하는 객체를 영속성을 갖는 디스크 장치(대표적으로 데이터베이스)에 저장함으로써, 객체에 대한 영속성을 부여하고자 하는 것이다. 다시 말해서, 객체에 대한 입출력을 가능하게 함으로써, 객체를 데이터베이스와 같은 영속성 장치에 저장하여, 시스템을 껐다 켜더라도 이전에 사용하던 객체의 상태를 그대로 복구하여 사용할 수 있다는 것이다. 이러한 객체 직렬화는 프로그램에 대한 영속성 역시 제공해 줄 수 있는 특징이다. 또한, 객체를 네트워크를 통해 입출력 함으로써, 객체가 플랫폼에 얽매이지 않고 시스템과 시스템 간을 네트워크를 통해 이동할 수 있도록 하는 것이다. 이러한 객체 직렬화가 중요한 이유는 객체지향 프로그램에서 클래스를 정의하여 사용하는 목적중의 하나가 실세계에 존재하는 데이터를 그대로 표현하는 것이기 때문이다. 다시 말래서, 객체라는 것은 실세계에 존재하는 데이터를 표현하고 있다는 것이다.

하지만, 이러한 객체의 직렬화가 어려울 수 있는 것은 객체는 단순한 데이터에 대한 표현으로서 문자 또는 숫자 값만을 포함하고 있는 것이 아니라, 참조형 변수를 포함하고 있기 때문이다. 다시 말해서, 하나의 객체에서 같은 시스템의 메모리 상에 존재하는 다른 객체를 참고하고 있기 때문에, 단순히 하나의 객체만을 입출력함으로써 나중에 그 객체의 상태를 그대로 복구할 수 있는 것이 아니다. 왜냐하면, 객체가 참조하고 있는 다른 객체 역시 복구해 주어야 하기 때문이다. 그러나, 한 가지 재미있는 것은 자바에서 제공해 주는 직렬화의 기본 메커니즘은 의외로 단순하면서도, 필요에 따라 기본 직렬화 매커니즘을 사용자 임의대로 커스터마이즈 할 수 있을 정도로 유연하다는 것이다. 먼저, 자바에서 제공해 주는 직렬화 매커니즘을 사용할 수 있는 방법과 이 메커니즘의 유연성을 이용할 수 있는 새로운 버젼의 클래스, protected 데이터의 보호, 전체 클래스의 재생성 등 세 가지 상황에 대해 살펴보도록 하자. 예를 들어, Person 클래스는 다음과 같이 어떤 사용자에 대한 데이터를 표현하고 있다.

```
import java.io.*;

public class Person implements Serializable {
    public String firstName;
    public String lastName;
    private String password;
    transient Thread worker;
```

```
    public Person(String firstName, String lastName, String password) {
        this.firstName = firstName;
        this.lastName = lastName;
        this.password = password;
    }

    public String toString() {
        return new String(lastName + ", " + firstName);
    }
}
```

Person 클래스는 영속적으로 만들고 싶은 데이터를 가지고 있는 클래스로서, 디스크에 저장하고 시스템을 재시작하거나 또는 다음 세션이 다시 시작될 때 다시 메모리에 적재시켜 사용하고자 한다. 자바에서는 아주 간단한게 이러한 문제를 해결할 수 있도록 해 주고 있는데, 단순히 Person 클래스가 java.io.Serializable 인터페이스를 구현한다고 선언해 주기면 하면 된다. Serializable 인터페이스는 아무 메소드도 없고, 단지 기본 직렬화 메커니즘을 사용하겠다고 자바 가상머신에게 알려주는 역할만을 한다. Person 클래스을 컴파일 한 후, 다음에 나오는 코드는 WritePerson.java(이달의 디스켓 참조) 프로그램의 일부로서 Person 객체를 바이트 스트림으로 변경하여 Person.ser 파일에 저장하는 작업을 하고 있다. (실제로, 생성된 Person.ser 파일 역시 이달의 디스켓으로 제공되므로 참조하지 바란다.)

```
import java.io.*;

class WritePerson {
    public static void main(String [] args) {
        Person p = new Person("Yongwoo", "Park", "park1234");
        ObjectOutputStream oos = null;
        try {
            oos = new ObjectOutputStream(new FileOutputStream("Person.ser"));
            oos.writeObject(p);
        } catch (Exception e) {
            e.printStackTrace();
        } finally {
```

```
            if (oos != null) {
                try {
                    oos.flush();
                } catch (IOException ioe) {
                }
                try {
                    oos.close();
                } catch (IOException ioe) {
                }
            }
        }
    }
```

다음에 나오는 코드는 ReadPerson.java(이달의 디스켓 참조) 프로그램의 일부로서 "Person.ser" 파일에 저장되어 있는 Person 객체를 파일로부터 읽어들이기 위한 코드이다. 이 때, "Person.ser" 파일에 대한 FileInputStream 스트림을 ObjectInputStream 스트림으로 변형하여 읽어들이야 한다.

```
import java.io.*;

class ReadPerson {
    public static void main(String [] args) {
        ObjectInputStream ois = null;
        try {
            ois = new ObjectInputStream(new FileInputStream("Person.ser"));
            Object o = ois.readObject();
            System.out.println("Read object " + o);
        } catch (Exception e) {
            e.printStackTrace();
        } finally {
            if (ois != null) {
                try {
```

```
                ois.close();
            } catch (IOException ioe) {
            }
        }
    }
}
```

지금까지 살펴본 자바에서 제공해 주는 직렬화 매커니즘을 이용한 객체의 입출력은 아주 광범위하고 다양한 상황에 적용가능한 예이다. 이렇게 한 객체를 직렬화 할 때, 모든 필드를 포함한 객체의 모든 상태를 저장하게 되는데, 심지어 Person 객체가 가지고 있는 private으로 선언되어 있는 password 필드까지 저장한다는 것을 의미한다. 그러나, 객체가 갖고 있는 어떤 필드에 대해서는 저장되지 않기를, 즉 영속성을 갖지 않을 필요가 있는 경우도 있다. 예를 들어, Person 객체가 가지고 있는 worker 필드의 경우 스레드 객체로서, 이 가상머신의 현재 세션에 대한 자원에 묶여 있다. 이러한 worker 스레드를 나중에 다시 사용하기 위해 직렬화 하는 것은 논리에 맞지 않는다고 할 수 있다. 이런 경우, 자바에서는 단순히 transient 키워드를 사용하여 필드를 선언함으로써 직렬화 하지 않도록 할 수 있다. 다시 말해서, transient로 선언되어 있는 필드는 객체가 직렬화되는 과정에서 저장되지 않는다는 것을 나타낸다.

직렬화와 클래스 버젼 결정

만약, 직렬화를 이미 수행하여 사용하고 있는 클래스를 변경했을 경우에는 어떻게 해야 할 것인가? 예를 들어, Person 클래스를 배포한 후, Person 클래스를 변경하여 나이를 나타내기 위한 age 필드를 추가했다고 하자. 이 때, Person 클래스와 WritePerson 클래스의 소스 코드가 다음과 같이 변경되었다.

```
import java.io.*;

public class Person implements Serializable {
    public String firstName;
    public String lastName;
    int age;
    private String password;
    transient Thread worker;
    static final long serialVersionUID = 4070409649129120458L;
```

```
//이것은 제대로 된 기호화 연산이 아니지만, 실행은 됩니다.
//하지만 조금 더 나은 것으로 대신해야 할 것입니다.
static String crypt(String input, int offset) {
    StringBuffer sb = new StringBuffer();
    for (int n=0; n<input.length(); n++) {
        sb.append((char)(offset+input.charAt(n)));
    }
    return sb.toString();
}

// 실제 응용프로그램에서는 패스워드로의 액세스를 동조하도록 하면서
// 페스워드를 System.out 로 출력해서는 안됩니다!
private void writeObject(ObjectOutputStream stream) throws IOException {
    password = crypt(password, 3);
    System.out.println("Password encyrpted as " + password);
    stream.defaultWriteObject();
    password = crypt(password, -3);
}

private void readObject(ObjectInputStream stream)
throws IOException, ClassNotFoundException {
    stream.defaultReadObject();
    password = crypt(password, -3);
    System.out.println("Password decrypted to " + password);
}

public Person(String firstName, String lastName, String password, int age) {
    this.firstName = firstName;
    this.lastName = lastName;
    this.password = password;
    this.age = age;
```

```
    }

    public String toString() {
        return new String(lastName + ", " + firstName + " age " + age);
    }
}

class WritePerson {
    public static void main(String [] args) {
        Person p = new Person("Yongwoo", "Park", "park1234", 29);
        ...
    }
}
```

위와 같이, Person 클래스가 변경되었지만, 이전에 저장되어 있는 Person.ser 파일을 읽어들이게 될 때 문제가 발생할 것이다. 왜냐하면, Person.ser 파일에는 이전 버전의 Person 객체가 저장되어 있기 때문에, 객체를 읽어들일 때 java.io.InvalidClassException 예외가 발생할 것이다. 이렇게, 자바 직렬화 메커니즘이 변경된 클래스에 대해 민감하기 때문에, 이미 직렬화되어 있는 객체의 클래스를 변경할 경우에는 항상 주의해야 한다. 자바에서는 한 클래스가 직렬화 되면, 클래스를 위한 serialVersionUID라 불리우는 64-bit "지문(fingerprint)"를 계산하여 생성하기 때문이다. serialVersionUID는 클래스 내에 직렬화 될 수 있는 모든 필드를 이용하여 계산된다. 따라서, 클래스를 변경하였을 경우(예를 들어, 새로운 필드(age)를 추가했을 경우)에는 클래스가 변경되기 이전의 serialVersionUID 값과 변경된 후의 serialVersionUID 값이 더 이상 일치하지 않게 되므로, 이전 버전의 Person 객체를 저장하고 있는 Person.ser 파일에서 객체를 읽어들일 수 없게 되는 것이다. 이러한 문제를 해결하기 위해서는 이번 버전의 Person 클래스와 현재 버전의 Person 클래스가 서로 호환된다는 것을 자바 가상머신이 알 수 있도록 해야 하고, 이를 위해 자바에서는 Person 클래스와 같은 직렬화하려는 객체의 클래스 내에 serialVersionUID 필드를 다음과 같이 고정시킬 수 있도록 하고 있습니다. 다시 말해서, 자바 가상머신에서 Person 클래스를 기반으로 하여 내부적으로 자동으로 계산하는 serialVersionUID 값을 직접 지정함으로서 클래스를 변경한 후에 다른 serialVersionUID 값으로 계산되는 것을 방지하자는 것이다.

```
static final long serialVersionUID = /* some long integer */;
```

그러나, 이 경우 Person.ser 파일 내에는 이전 버전의 Person 클래스를 이용하여 생성한 serialVersionUID 값이

저장되어 있기 때문에 아직까지는 문제가 된다. 다시 말해서, 새롭게 변경한 Person 클래스의 serialVersionUID 필드의 값을 이전 버전의 Person 클래스의 serialVersionUID 값으로 지정해 주는 것이 안전하다는 것이다. 이를 위해, JDK 1.2이상에서는 serialver 명령행 도구를 제공해 주고 있다. 다음과 같이 serialver 명령어를 사용하여 이전 버전의 Person 클래스의 serialVersionUID 값을 계산하여 구할 수 있다.

```
serialver Person
```

Person 클래스를 변경하기 전에 먼저 Person 클래스의 serialVersionUID 값을 구한 후, 이 serialVersionUID 값을 새롭게 변경하는 Person 클래스의 serialVersionUID 필드값으로 설정해 주면 되는 것이다. 예를 들어, 새롭게 변경되는 Person 클래스에는 다음과 같은 필드 선언이 추가될 것이다.

```
static final long serialVersionUID = 4070409649129120458L;
```

직렬화와 데이터의 보호를 위한 데이터 입출력 메소드의 커스터마이징

자바 직렬화는 private으로 선언된 필드에 대해서도 직렬화 작업을 수행한다. 왜냐하면, private 필드 역시 해당 클래스의 멤버이기 때문이다. 그러나, Person 클래스의 password 필드의 경우 중요한 정보로서 private으로 선언했지만, Person.ser 파일 내에 저장되어 있는 객체를 읽어들임으로써 어떤 사람도 password 값을 참조할 수 있게 된다. 심지어, 바이너리 데이터를 보여줄 수 있는 텍스트 편집기를 이용하여 Person.ser 파일을 열어보기만 해도 password 값을 읽을 수 있게 된다. 이러한 문제점을 해결하기 위해서는, 스트림에 직렬화된 객체를 출력할 때 중요한 필드의 경우 반드시 암호화해 주어야 한다. 이를 위해, 자바에서는 직렬화된 객체를 입력 또는 출력 작업을 수행할 때 사용자 정의 입출력 방식을 사용할 수 있도록 하기 위해 다음과 같은 두 개의 메소드를 제공해 주고 있다.

```
private void writeObject(ObjectOutputStream stream) throws IOException;
private void readObject(ObjectInputStream stream)   throws IOException, ClassNotFoundException;
```

따라서, Person 객체를 입출력 할 때 입력 및 출력 작업을 커스터마이즈 하기 위해 다음과 같이 Person 클래스에 writeObject 메소드와 readObject 메소드를 정의해 줄 수 있다.

```
import java.io.*;

public class Person implements Serializable {
```

```
public String firstName;
public String lastName;
int age;
private String password;
transient Thread worker;
static final long serialVersionUID = 4070409649129120458L;

//이것은 제대로 된 기호화 연산이 아니지만, 실행은 됩니다.
//하지만 조금 더 나은 것으로 대신해야 할 것입니다.
static String crypt(String input, int offset) {
    StringBuffer sb = new StringBuffer();
    for (int n=0; n<input.length(); n++) {
        sb.append((char)(offset+input.charAt(n)));
    }
    return sb.toString();
}

// 실제 응용프로그램에서는 패스워드로의 액세스를 동조하도록 하면서
// 페스워드를 System.out 로 출력해서는 안됩니다!
private void writeObject(ObjectOutputStream stream) throws IOException {
    password = crypt(password, 3);
    System.out.println("Password encyrpted as " + password);
    stream.defaultWriteObject();
    password = crypt(password, -3);
}

private void readObject(ObjectInputStream stream)
throws IOException, ClassNotFoundException {
    stream.defaultReadObject();
    password = crypt(password, -3);
    System.out.println("Password decrypted to " + password);
}
```

```
    public Person(String firstName, String lastName, String password, int age) {
        this.firstName = firstName;
        this.lastName = lastName;
        this.password = password;
        this.age = age;
    }

    public String toString() {
        return new String(lastName + ", " + firstName + " age " + age);
    }
}
```

writeObject 메소드에서 stream.defaulWriteObject 메소드를 호출하여 기본 직렬화 메카니즘을 실행하기 전에 패스워드를 함호화 해주어야 하고, reaObject 메서드에서는 그 과정을 반대로 수행go 주어야 한다는 것을 주의하기 바란다

직렬화와 전체 클래스의 재생성

만약, 다음에 나오는 것과 같이 Person 클래스의 lastName 필드와 firstName 필드를 하나의 fullName 필드를 합치려고 할 때 어떻게 해야 할까?

```
import java.io.*;

public class Person implements Serializable {
    public String fullName;
    int age;
    private String password;
    transient Thread worker;
    static final long serialVersionUID = 4070409649129120458L;

    //이것은 제대로 된 기호화 연산이 아니지만, 실행은 됩니다.
    //하지만 조금 더 나은 것으로 대신해야 할 것입니다.
```

```
    static String crypt(String input, int offset) {
        StringBuffer sb = new StringBuffer();
        for (int n=0; n<input.length(); n++) {
            sb.append((char)(offset+input.charAt(n)));
        }
        return sb.toString();
    }

    // 실제 응용프로그램에서는 패스워드로의 액세스를 동조하도록 하면서
    // 페스워드를 System.out 로 출력해서는 안됩니다!
    private void writeObject(ObjectOutputStream stream)
    throws IOException {
        password = crypt(password, 3);
        System.out.println("Password encyrpted as " + password);
        stream.defaultWriteObject();
        password = crypt(password, -3);
    }

    // readObject를 여기의 새 버젼으로 교체하십시오.
    private void readObject(ObjectInputStream ois)
    throws IOException, ClassNotFoundException {
        ObjectInputStream.GetField gf = ois.readFields();

        // 새로운 버젼을 가지고 있기를 바라며....
        fullName = (String) gf.get("fullName", null);
        if (fullName == null) {
            // 이런. 구버젼입니다. fullName을 만들어냅니다.
            String lastName = (String) gf.get("lastName", null);
            String firstName = (String) gf.get("firstName", null);
            fullName = lastName + ", " + firstName;
        }
        age = gf.get("age", 0);
```

```
        password = (String) gf.get("password", null);
        password = crypt(password, -3);
        System.out.println("Password decrypted to " + password);
    }

    public Person(String firstName, String lastName, String password, int age) {
        this.fullName = lastName + ", " + firstName;
        this.password = password;
        this.age = age;
    }

    public String toString() {
        return new String(fullName + " age " + age);
    }
}
```

위와 같이 Person 클래스를 변경한 후, 이번 버전의 Person 객체가 저장되어 있는 Person.ser 파일을 읽어들였을 때, serialVersionUID를 직접 지정하고 있기 때문에 에러가 나지는 않지만, 새로운 버전의 Person 클래스에는 Person.ser 파일에 저장되어 있는 lastName 필드와 firstName 필드기 없기 때문에 정상적으로 Person 객체를 생성할 수 없을 것이다. defaultReadObject를 사용하여 객체 직렬화를 수행할 때의 문제점은 스트림 필드의 값을 클래스의 필드의 이름을 이용하여 적용하고 있다는 것이다. 이러한 문제를 해결하기 위해서는 저장소로부터 필드를 읽는 방법을 관리해 주어야 하는데, 내포된 클래스인 ObjectInputStream.GetField 클래스를 이용하여 스트림에서 찾고자 하는 필드의 이름을 명시적으로 지정할 수 있다. 실제로, ObjectInputStream.GetField 클래스를 사용하여 Person 클래스의 readObject 메소드를 다음과 같이 변경해 주어야 한다.

```
private void readObject(ObjectInputStream ois)
throws IOException, ClassNotFoundException {
    ObjectInputStream.GetField gf = ois.readFields();

    fullName = (String) gf.get("fullName", null);
    if (fullName == null) {
        // 이전 버전의 객체일 경우 변경된 내용을 적당하게 반영
```

```
            String lastName = (String) gf.get("lastName", null);
            String firstName = (String) gf.get("firstName", null);
            fullName = lastName + ", " + firstName;
        }
        age = gf.get("age", 0);
        password = (String) gf.get("password", null);
        password = crypt(password, -3);
        System.out.println("Password decrypted to " + password);
    }
```

EXTERNALIZATION을 이용한 직렬화 성능의 향상

직렬화를 이용하면, 객체의 필드를 스트림으로 매빙하는 방법은 물론, 스트림에서 기대하지 않은 전혀 다른 필드를 발견했을 때의 처리도 할 수 있도록 한다. 이러한 유연성은 객체 직렬화의 장점이다. 객체를 직렬화 할 때 사용하는 직렬화 포맷에서는 객체의 필드 값뿐만 아니라, 클래스의 버전에 대한 메타데이터와 필드 이름, 필드의 형도 함께 저장할 수 있도록 하고 있다. 그러나, 자바에서 제공해 주는 객체 직렬화의 유연성은 성능을 저하시키는 요인중의 하나이다. 객체 직렬화의 성능을 개선하기 위해서는 표준 직렬화 포맷을 사용하지 않고, Externalizable 기법을 사용해야 한다. Externalizable 기법을 사용하는 예를 살펴보기 위해, TestSerialization 클래스의 주요 코드를 살펴보면 다음과 같다. (이달의 디스켓으로 제공되는 TestSerialization.java 파일을 참고하기 바란다.)

```
    Employee[] emps = new Employee[count];
    for (int n=0; n<count; n++) {
        // 직렬화 데이터를 랜덤하게 생성
        emps[n] = new Employee("LastName"+n, "FirstName"+n,
                               "222-33-"+n, 34000+n, n%10);
    }

    for (int outer=0; outer<tests; outer++) {
        ObjectOutputStream oos = null;
        FileOutputStream fos = null;
        BufferedOutputStream bos = null;
        long start = System.currentTimeMillis();
```

```
        try {
            fos = new FileOutputStream("Employee.ser");
            bos = new BufferedOutputStream(fos);
            oos = new ObjectOutputStream(bos);
            for (int n=0; n<count; n++) {
                oos.writeObject(emps[n]);
            }
            long end = System.currentTimeMillis();
            System.out.println("Serialization of " + count + " objects took " + (end-start) + " ms.");
        } finally {
            if (oos != null) oos.close();
            if (bos != null) bos.close();
            if (fos != null) fos.close();
        }
        new File("TestSerialization").delete();
    }
```

TestSerialization 클래스는 Employees를 OutputStream으로 기록하는데 걸리는 시간을 측정하기 위한 간단한 벤치마크용 프로그램이다. 위의 코드에서는 5000명의 종업원 데이터를 랜덤하게 생성하고, 하나의 파일에 기록하는 작업을 10번 반복한다. 약간의 차이는 있겠지만 필자의 PC에서 테스트 했을 때는 다음과 같은 결과가 나타났다.

```
Serialization of 5000 objects took 341 ms.
Serialization of 5000 objects took 210 ms.
Serialization of 5000 objects took 140 ms.
Serialization of 5000 objects took 141 ms.
Serialization of 5000 objects took 140 ms.
```

Employee 클래스는 Serializable 인터페이스를 구현함으로써 간단한 직렬화 기능을 사용하고 있다. Serializable 인터페이스를 구현하면 Employee 객체를 ObjectOutputStream의 writeObject 메소드에 전달하는 것만으로 객체를 직렬화 할 수 있다. ObjectOutputStream은 Employee 클래스의 메타데이터와 인스턴스 필드를 스트림으로 기록하는 과정을 자동화해 준다. 다시 말해서, 모든 직렬화 작업을 내부적으로 자동으로 수행한다는 뜻이다. 이 때, 보다 빠른 성능을 얻기 위해서는 Employee 클래스가 Serializable 대신 Externalizable을 구현하도록 선언하여 객

체 직렬화 작업을 직접 수행해 줄 수 있다.이 때, 한가지 주의할 것은 반드시 Employee 클래스에 매개변수를 갖지 않는 public 생성자를 선언해야 한다는 것이다. 객체를 Externalizable로 선언하면 개발자는 객체의 상태를 스트림으로 기록할 모든 책임을 지게되며, ObjectOutputStream클래스는 더이상 클래스의 메타 데이터와 인스턴스의 필드 값을 자동으로 스트림으로 기록하지 않게된다. 대신, readExternal 메소드와 writeExternal 메소드를 이용하여 스트림을 직접 다뤄야 한다. 예를 들어, Employee 클래스에 다음과 같이 readExternal 메소드와 writeExternal 메소드를 추가함으로써, 보다 빠른 성능을 얻을 수 있다.

```
public void readExternal(java.io.ObjectInput s)
throws ClassNotFoundException, IOException {
    lastName = s.readUTF();
    firstName = s.readUTF();
    ssn = s.readUTF();
    salary = s.readInt();
    level = s.readInt();
}
public void writeExternal(java.io.ObjectOutput s)
throws IOException {
    s.writeUTF(lastName);
    s.writeUTF(firstName);
    s.writeUTF(ssn);
    s.writeInt(salary);
    s.writeInt(level);
}
```

이 때, ObjectInput과 ObjectOutput 인터페이스는 각각 DataInput과 DataOUtput 인터페이스를 상속받기 때문에, readInt()와 writeInt()와 같은 메소드를 이용하여 기본 데이터타입을 읽거나 쓸 수 있으며, readUTF()와 writeUTF와 같은 메소드를 이용하여 문자열을 읽거나 쓸 수 있다. Externalization을 이용한 버전(이달의 디스켓으로 제공되는 TestSerialization2.java 파일을 참고하기 바란다.) 그러나, 약간의 성능 향상을 가져다 주기는 했지만, Externalizable을 구현하는 클래스에서는 위에서 살펴본 것과 같이 보다 많은 코드를 작성해야 한다는 것을 의미한다. 그리고 코드가 많아지면 버그도 더 많아질 수 있다. 예를 들어, 필드 하나를 기록하는 것을 잊거나, 기록할때와 다른 순서로 필드들을 읽어들인다면, Externalization는 수행되지 않는다. Serializable 인터페이스를 이용하면, 이러한 문제들은 ObjectOutputStream에 의해 해결된다. 아마도 Externalizable 객체의 가장 큰 단점은 스트림을 해석

하기 위한 클래스가 반드시 있어야 한다는 것이다. 불행히도, 자바의 직렬화 메커니즘은 재생성을 위한 코드는 제공하지 않고 있다. 따라서, 이런 일을 하기 위한 코드는 별도로 작성해야 하고, 다음의 사이트에 예제가 있으니 참고하기 바란다.

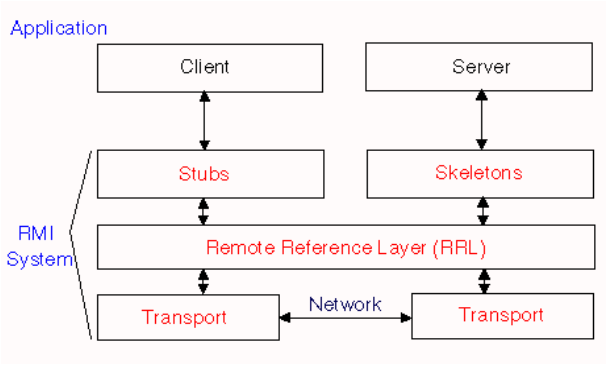
http://staff.develop.com/halloway/JavaTools.html

물론, 성능이 가장 큰 관건이라면 Externalizable 객체를 이용할 수 밖에 없을 것이다. 만일 코드가 LAN 안에서 많은 수의 이벤트를 관리하면서 실시간에 가까운 성능을 얻고자 한다면, 아마도 해당 이벤트를 Externalizable 객체로 모델링해야 할 것이다.

분산되어 있는 객체들을 하나로, 리모트 메소드 호출 (RMI; Remote Method Invocation)

RMI 아키텍쳐

<그림 9> RMI 아키텍쳐



RMI는 기존의 객체 모델이나 언어에 관계없이 분산 객체 모델을 자바로 통합하고 리모트 객체를 로컬 객체와 같이 쉽게 다루기 위한 기술이다. 예를 들어, 리모트 객체의 변수값의 설정하거나, 리모트 객체의 메소드를 호출하면서 매개변수를 넘겨주거나, 또는 리모트 객체의 메소드에서 리턴하는 값을 받는 등과 같이, 로컬 객체를 사용하는 것과 거의 동일한 방법으로 리모트 객체를 사용할 수 있다. 게다가, 값 또는 레퍼런스에 의해서 객체의 부분이나 객체 전체를 넘겨주는 등의 리모트 객체를 호출하기 위한 정교한 메커니즘을 포함할뿐만 아니라, 리모트 작업시에 일어날 수도 있는 네트워크 에러에 대한 부수적인 예외처리 메커니즘도 포함하고 있다. 이러한 RMI는 다음에 나오는 그림에서와 같이 그 목적에 따라 몇가지 계층으로 나뉘어지는데 하나의 메소드 호출은 이 많은 계층들을 거치게 된다.

먼저, 클라이언트측에 존재하는 스텁(stub) 계층과 서버측에 존재하는 스켈레톤(skeleton) 계층은 클라이언트와 서버 각각에서 실제적인 클래스의 구현으로부터 메소드 호출이 리모트 환경이라는 것을 숨기면서 대행자 역할을 한다. 다시 말해서, 클라이언트 애플리게이션에서 로컬 메소드를 호출하는 것과 같은 방법으로 리모트 메소드를 호출할 수 있도록 해 준다는 것이다. 스텁 객체는 리모트 객체를 로컬에 있는 것처럼 호출할 수 있도록 해주는 대행자이다. 다음으로, 리모트 레퍼런스 계층은 메소드 호출과 메소드의 매개변수 그리고 반환값을 네트워크를 통해 전송될 수 있도록 묶어주는 부분이다. 마지막으로, 전송 계층은 하나의 시스템에서 다른 시스템으로 네트워크 연결을 하는 부분이다.

스텁 (Stub)

RMI에서는 리모트로 메쏘드를 호출하는 객체를 클라이언트라 하고, 호출되는 메소드를 제공하는 객체를 서버라고 할 수 있다. 클라이언트의 객체가 서버에 존재하는 객체의 메쏘드를 리모트로 호출하려고 할 때, 객체가 가지고 있는 메쏘드의 존재를 표현하고 있는 것이 바로 스텁이다. 실제로, 스텁은 서버에 존재하는 객체가 리모트로 호출할 수 있는 메쏘드를 공개하는 인터페이스로서 일반적으로 서버에 저장되어 있다가, 클라이언트 객체가 서버에 있는 메쏘드를 호출할 때 클라이언트로 전달된다. 스텁 객체는 원래 서버의 객체를 기반으로 rmi 컴파일러에 의해 생성되지만, 클라이언트에서 서버의 메쏘드를 호출할 수 있도록 하기 위해 클라이언트에 존재한다. 이러한 스텁은 다음과 같은 두 가지 일을 한다.

- 서버에 존재하는 객체의 메쏘드를 호출해 준다.
- 메쏘드를 호출하면서 매개변수를 서버에 전달할 수 있도록 일정한 포맷으로 변환한다. 이러한 변환 작업을 매개변수 마샬링(Parameter Marshalling)이라 한다.

스켈레톤 (Skeleton)

이러한 매개변수 마샬링 작업이 끝났으면, 스텁은 서버에 존재하는 객체의 메쏘드를 호출한다. 이 때, 스텁이 서버에 보내는 정보는 서버 객체의 아이디, 실행해야 할 메쏘드, 그리고 변환된 매개변수 등이다. 클라이언트 측에서 스텁이 대리인의 역할을 하는 것과 마찬가지로, 서버 측에서 대리인 역할을 하는 것이 바로 스켈레톤(skeleton)이다. 스켈레톤을 수신자(receiver)라 하기도 한다. 스켈레톤에서 하는 역할은 스텁의 반대이다. 네트워크를 통해 전달받은 매개변수들을 서버에 맞게 변형한 후, 스텁이 보낸 아이디의 객체가 갖고 있는 메쏘드를 실행시켜 주는 것이다. 그리고, 메소드의 실행이 끝났을 때, 메소드에서 리턴한 값을 다시 매개변수 마샬링을 이용하여 변환한 후, 다시 스텁에게 되돌려 주는 것이다. 스텁은 이 결과를 받아서 다시 시스템에 맞게 변환시킨 뒤, 원래 메쏘드를 불렀던 객체에게 결과를 넘겨준다. 이 때, 서버 객체로부터 발생된 예외를 던질 수도 있다.

RMI 서버 프로그래밍

먼저, 클라이언트 객체가 서버 객체를 호출할 수 있도록 하기 위해, 서버 객체에 대한 인터페이스를 정의해 주어야 한다. 왜냐하면, 클라이언트 객체가 서버 객체의 메소드를 호출할 때, 이 인터페이스를 참조하기 때문이다. 이 인터페이스는 서버와 클라이언트 양쪽에 존재한다. 지금까지 살펴본 점을 찍는 작업을 RMI를 이용하여 서비스를 제공하기 위해, 먼저 다음과 같이 리모트 인터페이스를 정의할 수 있다.

```
import java.rmi.Remote;
import java.rmi.RemoteException;

public interface RemotePoint extends Remote {
    public void setX(int x) throws RemoteException;
    public void setY(int y) throws RemoteException;
    public int getX() throws RemoteException;
    public int getY() throws RemoteException;
}
```

다음으로, 위의 리모트 인터페이스를 모두 구현하고 있는 서버 객체 클래스를 다음과 같이 정의해야 한다.

```
import java.rmi.RemoteException;
import java.rmi.server.UnicastRemoteObject;

class RemotePointImpl extends UnicastRemoteObject implements RemotePoint {
    int x=0, y=0;

    public RemotePointImpl() throws RemoteException {
    }

    public void setX(int x) throws RemoteException {
        this.x = x;
    }

    public void setY(int y) throws RemoteException {
        this.y = y;
```

```
    }

    public int getX() throws RemoteException {
        return(x);
    }

    public int getY() throws RemoteException {
        return(y);
    }
}
```

여기서 한가지 주의할 점은 서버 객체를 위한 클래스를 선언할 때는 반드시 UnicastRemoteObject 클래스를 상속하고, 리모트 인터페이스를 모두 구현해 주어야 한다는 것이다. 이렇게 RemotePoint 인터페이스와 RemotePointImpl 클래스를 작성하였다면, 다음으로 객체의 이름을 RMI 레지스트리(Registry)에 등록해 주어야 한다. JNDI (Java Naming and Directory Interface)의 일환이기도 한데 클라이언트에 있는 객체에게 서버 객체의 존재를 알려주는 역할을 한다. 즉, 서버객체를 어떠한 이름하에 등록하여 클라이언트의 객체가 그 이름으로 리모트 객체를 찾을 수 있도록 해 주는 것을 말한다. 클라이언트는 이 이름으로 서버 객체를 찾으며, 찾았을 경우 그 이름에 해당하는 서버 객체의 스텁 클래스가 클라이언트로 전송되어 서버객체의 메쏘드를 호출할 수 있도록 해 주는 것이다. 이렇게, 서버 객체를 RMI 레지스트리에 등록하기 위해 다음과 같이 자바 애플리케이션 클래스를 작성해 주어야 한다.

```
import java.rmi.Naming;

public class RemotePointServer {
    public static void main(String args[])throws Exception {
        RemotePointImpl remotePointRef = new RemotePointImpl();

        Naming.rebind("RemotePoint",remotePointRef);
    }
}
```

RMI 클라이언트 프로그래밍

위와 같이 서버측에서 서비스를 제공해 주기 위한 클래스들을 모두 작성하였다면, 다음으로 클라이언트에서 서버측의 서비스를 사용하는 클래스들을 작성해 주어야 한다. 클라이언트측에서 서버측의 서비스를 사용하기 위해서는 반드시 서버측 객체가 제공해 주는 메소드들에 대해 정의하고 있는 인터페이스를 가져와 가지고 있어야 한다. 클라이언트 프로그램에서는 단지 서버에 존재하는 객체를 인터페이스를 이용하여 가져와 로컬에 존재하는 객체인 것처럼 메소드를 호출할 수 있다. 다음은 클라이언트 애플리케이션에 해당하는 클래스를 정의하고 있다.

```
import java.rmi.Naming;

public class RemotePointClient {
    public static void main(String[] args) throws Exception {
        if(args.length < 3) {
            System.out.println("Usage:java RemotePointClient <server> x y "+args.length);
            return;
        }

        int x = Integer.parseInt(args[1]);
        int y = Integer.parseInt(args[2]);

        String url="rmi://"+args[0]+"/RemotePoint";

        RemotePoint p = (RemotePoint)Naming.lookup(url);

        p.setX(x);
        p.setY(y);

        System.out.println("p("+p.getX()+","+p.getY()+")");
    }
}
```

RMI 서버 프로그램의 실행

위와 같이, RMI를 이용하여 서비스를 제공하기 위한 서버측과 클라이언트측 클래스들을 모두 작성하였다면, RMI를 이용하여 서버 애플리케이션과 클라이언트 애플리케이션을 각각 실행시켜주어야 한다. 이렇게 RMI 서버를 실행하

기 위한 과정을 살펴보면, 다음과 같다.

- 서버 객체 클래스와 인터페이스를 모두 컴파일한다. 이 때, 각 클래스와 인터페이스에 대한 .class 파일이 생성된다.

```
javac *.java
```

- rmic 컴파일러를 이용하여 서버 객체에 대한 스텁 클래스와 스켈레톤 클래스를 생성한다. 이 때, RemotePointImpl_Skel.class, RemotePointImpl_Stub.class 파일등 스텁 클래스와 스켈레톤 클래스가 생성된다.

```
rmic RemotePointImpl
```

- rmiregistry 실행한다.

```
start rmiregistry
```

- 서버 프로그램 실행한다.

```
start java RemotePointServer
```

RMI 클라이언트 프로그램 실행

RMI 서버와 마찬가지로 RMI 클라이언트 객체의 클래스와 인터페이스를 모두 컴파일해 주어야 하낟. 여기서 인터페이스는 서버 객체에서 서비스를 제공하기 위해 작성한 인터페이스(또는 리모트 인터페이스) 파일을 의미한다. 성공적으로 컴파일 하였다면, 다음으로 보안정책을 정의하는 파일을 생성해 주어야 한다. 이러한 작업을 모두 마쳤다면 마지막으로 RMI 클라이언트 프로그램을 실행시켜 주어야 한다. 이러한 과정을 살펴보면, 다음과 같다.

- 클라이언트 애플리케이션 프로그램과 인터페이스를 모두 컴파일한다.

```
javac *.java
```

- 보안정책 파일을 만든다. 어떤 파일 이름을 사용해도 상관없다. 이 예제에서는 보안정책 파일의 이름을 RemotePointClient.policy 라고 하겠다. 위에서 자바 보안매니저를 실행시킨 것을 기억할 것이다. 자바 보안매니저는 원래 어떠한 네트웍 연결도 허용하지 않는다. 그러므로 이러한 보안정책을 설정해 주어야 하는 것이다. 보안정책 파일엔 다음과 같이 해주면 되겠다. RMI 포트는 기본적으로 1099 번이지만, 다음에 나오는 보안 설정에서는 1024에서 65535 사이의 포트번호를 사용할 수 있도록 하고 있다. 보안정책 파일엔 이외에 다른 많은 정보를 설정할 수 있다.

```
grant {
    permission java.net.SocketPermission "*:1024-65535", "connect";
};
```

- 클라이언트 애플리케이션 프로그램을 보안정책과 함께 실행시켜 준다.

```
java RemotePointClient localhost 10 20 -Djava.security.policy=RemotePointClient.policy
```

앞에서도, 언급한 것처럼 먼저 클라이언트 프로그램을 시작하기 전에 반드시 자바 보안매니저를 실행시켜주어야 한다. 만약, 클라이언트가 자바 애플릿일 경우엔 자동으로 보안이 실행되므로 상관없지만, 클라이언트 프로그램이 자바 애플리케이션이라면 반드시 실행시켜 주어야 한다. 그리고, 클라이언트 프로그램에서 서버 객체의 이름을 찾을 때 반드시 "rmi://서버이름/등록된 객체이름"와 같은 형식의 URL을 사용하기 바란다. 또한, RMI가 네트워크를 통해서 이루어기 때문에 어떤 예외가 발생할지 모르므로, 반드시 예외처리 하기 바란다.

동적 클래스로딩에 관해 주의할점

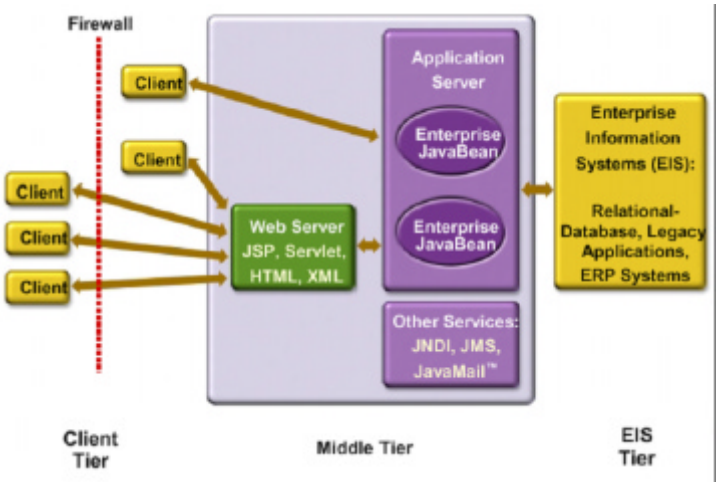
앞에서 언급했던 것처럼 동적 클래스로딩은 클라이언트가 필요로 하는 스텁 클래스를 필요할 때마다 서버로부터 가져오는 것을 말한다. 실제로 RMI 프로그램을 개발할 때 이러한 동적 클래스로딩이 몇 가지를 문제를 가지므로 주의해야 한다.

클라이언트 객체는 리모트 인터페이스(스텁 클래스)를 이용하여 서버 객체의 메쏘드를 호출한다. 다시 말해서, 서버 객체가 다른 여러가지 메쏘드를 정의하고 있더라도, 클라이언트에서는 리모트 인터페이스에 정의되어 있는 메소드만 호출할 수 있다는 것이다. 예를 들어, 클라이언트 객체가 서버 객체의 어떠한 메쏘드를 호출했는데, 그 결과 리모트 객체를 상속한 객체가 리턴되었지만, 리모트 객체를 상속한 객체는 리모트 인터페이스를 구현하지 않는다고 하자. 이 때, 그렇다면, 클라이언트 객체가 전달받은 객체의 메소드를 호출할 수 없게 될 것이다. 만약, 리모트 인터페이스를 구현하지 않는 다른 클래스의 객체가 리턴되었을 경우에는 어떻게 할까. 예를 들어, 서버 객체의 메쏘드를 호출할 때 String 객체를 매개변수로 전달되었을 경우에는, String 객체가 Remote 인터페이스를 구현하거 있지 않으므로 객체 자체가 복사되어 서버에 보내지게 된다. 만약 서버가 String 객체의 어떠한 메쏘드를 사용한다고 해도 객체 자체가 복사가 되어 보내졌으므로 문제가 없다. 간단히 말하면, 리모트 인터페이스를 구현하는 객체가 매개변수 또는 메쏘드의 리턴값으로 전달될 경우, 스텁클래스만이 보내지고 리모트 인터페이스를 구현하지 않는 객체를 보낼 경우 객체 자체가 복사되어 보내진다는 것이다.

진정한 분산 컴포넌트 아키텍처, EJB (Enterprise JavaBeans)

다층 웹 애플리케이션의 구조

<그림 10> 다층 웹 애플리케이션의 구조



위의 그림에서는 J2EE 플랫폼을 이용한 다층(Multi-tier) 기업 시스템의 아키텍처로서, 컴포넌트 기반의 개발 환경을 통해 개발 시간 단축과 유지보수의 용이성 향상을 불러오고, 기업 내부에 존재하는 변화하기 힘든 기존 시스템과의 연동 또한 비교적 쉽게 이루어질 수 있도록 해주는 구조이다. 각 층에 대해 간단히 살펴보면 다음과 같다. 실제로, J2EE 플랫폼은 기존의 자바 기술들을 총망라하는 기술의 총아라 할 수 있고, 그 기반이 되는 플랫폼은 바로 J2SE이다. 따라서, 모든 자바 기술들은 통일성 있고 일관되게 연결되고 있다는 것을 알 수 있다.

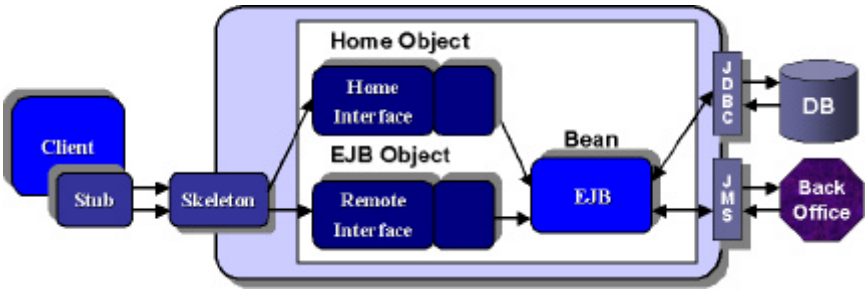
- 클라이언트(Client) 층 : 클라이언트는 웹 브라우저, 모바일 폰(Mobile Phone) 또는 PDA 등과 같이 브라우저를 통해 마크업 언어를 보여주거나 또는 AWT/SWING 컴포넌트를 사용하여 UI를 제공해 주는 애플릿 또는 자바 애플리케이션 일 수 있다.
- 프리젠테이션(Presentation) 층 : 주로 서블릿 또는 JSP 등과 같은 자바 서버측 기술로서, XSLT 기술을 이용하여 클라이언트에 맞게 HTML, XML, WML, DHTML 등의 페이지를 동적으로 생성하여 전달해 준다.
- 애플리케이션(Application) 층 : 비즈니스 로직을 포함하고 있는 엔터프라이즈 자바 빈즈(EJB; Enterprise Java Beans)를 이용하여 사용자의 요청에 해당하는 작업을 수행하거나, 또는 기업 정보 시스템 또는 레거시 시스템과 연동한다.
- EIS (Enterprise Information System) 층: 기업 정보 시스템, 관계형 데이터베잇, 레거시 시스템, ERP 시스템 등과 같은 기업의 정보 시스템에 해당한다.

EJB 아키텍처 (Enterprise JavaBeans Architecture)

처음 자바빈즈(Java Beans)는 클라이언트 측의 재사용 가능한 컴포넌트를 개발하기 위해 탄생되었지만, 오히려 엔터프라이즈 컴퓨팅 환경에 맞게 다시 설계된 엔터프라이즈 자바빈, 즉 EJB가 더 널리 알려지게 되었다. 다층 클라

이언트/서버 구조와 컴포넌트 기반의 개발이 엔터프라이즈 컴퓨팅 환경 구축의 기반 환경이 되면서, 점차 제어(Control) 로직과 비즈니스 로직을 분리하는 것이 일반화되었다. 즉 구매, 티켓발매, 입출금 등의 비즈니스 로직 단위를 하나의 컴포넌트로 구성하고 이를 클라이언트 측에서 자유로이 조합하고 이용함으로써 하나의 시스템을 구성하는 방식이다. 컴포넌트는 그 자체가 완벽한 소프트웨어가 아니라 소프트웨어를 구성하는 요소로서 직접 개발하지 않은 개발자도 쉽게 사용할 수 있도록 Self-Scribing해야 한다. 더구나 서버측 컴포넌트라면 컴포넌트들 간의 트랜잭션, 검색(Naming and Directory Service) 서비스들을 제공하는 컨테이너와 함께 제공되어야 한다. 엔터프라이즈 자바빈즈는 이와 같이 다층 클라이언트/서버 구조에서 웹 서버와 같이 요청을 받아들이고 분배하는 리스너(Listener) 계층과 데이터베이스 계층 사이에 존재하는 비즈니스 로직을 컴포넌트화하여 재사용 가능하도록 해 주며, 또한 여기서 발생하는 트랜잭션 등의 복잡한 작업을 단순하게 해 준다. 다음에 나오는 엔터프라이즈 자바빈의 구조를 잘 보여주고 있으며, 주요 구성요소에 대해 살펴보면 다음과 같다.

- ✍✍ 엔터프라이즈 빈 (Enterprise Bean) : 비즈니스 로직을 포함하고 있는 컴포넌트로서 클라이언트에 직접 노출되지 않는다. 즉 비즈니스 로직 자체를 제외한 어떠한 제어(트랜잭션, 지속성 관리 등) 로직도 포함되어 있지 않다.
- ✍✍ 홈 인터페이스 (EJB Home 또는 Home Object) : JNDI 등의 디렉토리 서비스를 통해 네트웍 상에서 이 빈이 검색가능하도록 해 주며 Bean인스턴스를 새로이 생성하거나 제거하는 임무를 가지고 있다. 일반적으로, 홈 인터페이스라 하며, 클라이언트는 홈 인터페이스를 이용하여 엔터프라이즈 빈을 생성할 수 있다.
- ✍✍ 리모트 인터페이스 (EJB Object) : 클라이언트들의 메쏘드 호출을 가로채 트랜잭션과 보안, 상태(State) 관리, 지속성 관리 등의 문제를 구현해 적용한다. 일반적으로, 리모트 인터페이스라 하며, 클라이언트는 엔터프라이즈 빈의 리모트 인터페이스를 이용하여 비지니스 로직을 사용할 수 있다.



EJB 프로그래밍 개요

EJB를 이용하여 클라이언트에게 서비스를 제공하도록 프로그램을 작성할 수 있다. 이 때, EJB에서 제공해 주는 기능을 사용하는 클라이언트는 서블릿, JSP 페이지, 엔터프라이즈 빈, RMI 호환 애플릿/애플리케이션 클라이언트 등 다양하게 나타날 수 있다. 여기서는 간단하게 서비스를 제공하기 위한 세션빈을 작성하고, 세션빈에서 제공해 주는

서비스를 이용하기 위한 클라이언트 애플리케이션을 작성하는 방법에 대해 살펴봄으로써, 프로그램 작성 규약은 바뀌었지만 기존의 자바 프로그래밍과 같이 일관되고 통일성 있게 작성된다는 것을 살펴보도록 하겠다. 먼저, 엔터프라이즈 자바빈을 작성하는 순서를 살펴보면 다음과 같다.

- 하나의 엔터프라이즈 빈을 작성하기 위해서는 홈 인터페이스, 리모트 인터페이스, 빈 클래스 등 세 개의 소스 코드를 작성해야 한다.
- 엔터프라이즈 빈을 위한 세 개의 소스 코드를 컴파일 한다.
- EJB 컨테이너에서 제공해 주는 디플로이먼트 툴을 이용하여 EJB 컨테이너에 배치한다.
- 클라이언트 프로그램을 작성, 컴파일, 그리고 실행한다.

먼저, 홈 인터페이스는 JNDI를 이용하여 엔터프라이즈 빈을 찾은 후, 빈의 인스턴스를 생성할 수 있도록 하기 위해 제공되는 인터페이스이다. 홈 인테페이스에서는 엔터프라이즈 빈의 생성, 찾기, 제거 등과 같은 빈의 관리를 제공해 준다. 이 때, 빈 클래스에서는 홈 인터페이스에서 선언하고 있는 메소드와 연관된 이벤트 처리 메소드를 정의해 주어야 한다.

```
import java.rmi.*;
import javax.ejb.*;

public interface HelloHome extends EJBHome {
    public Hello create() throws CreateException, RemoteException;
}
```

다음으로, 리모트 인터페이스를 작성해야 하는데, 클라이언트는 리모트 인터페이스를 통해서만 엔터프라이즈 빈의 메소드를 사용할 수 있다. 다시 말해서, 엔터프라이즈 빈에서 클라이언트에게 제공할 메소드를 선언하고 있는 것이 바로 리모트 인터페이스이다. 리모트 인터페이스의 예를 들면 다음과 같다. 이 때, 빈 클래스에서는 리모트 인터페이스에서 선언하고 있는 메소드를 반드시 구현해 주어야 한다.

```
import java.rmi.*;
import javax.ejb.*;

public interface Hello extends EJBObject {
    public String getHello() throws RemoteException;
```

```
    public void setHello(String hello) throws RemoteException;
}
```

마지막으로, 홈 인터페이스에서 선언하고 있는 생성, 찾기, 제거와 연관된 이벤트 메소드는 물론 리모트 인터페이스에서 선언하고 있는 메소드를 반드시 구현해 주어야 한다. 따라서, 위와 같이 홈 인터페이스와 리모트 인터페이스를 정의한 경우에, 빈 클래스에는 반드시 홈 인터페이스에 선언되어 있는 create 메소드와 연관된 ejbCreate() 메소드와 리모트 인터페이스에서 선언하고 있는 getHello 메소드 및 setHello() 메소드를 구현해 주어야 한다. 빈 클래스의 소스 코드를 살펴보면, 다음과 같다.

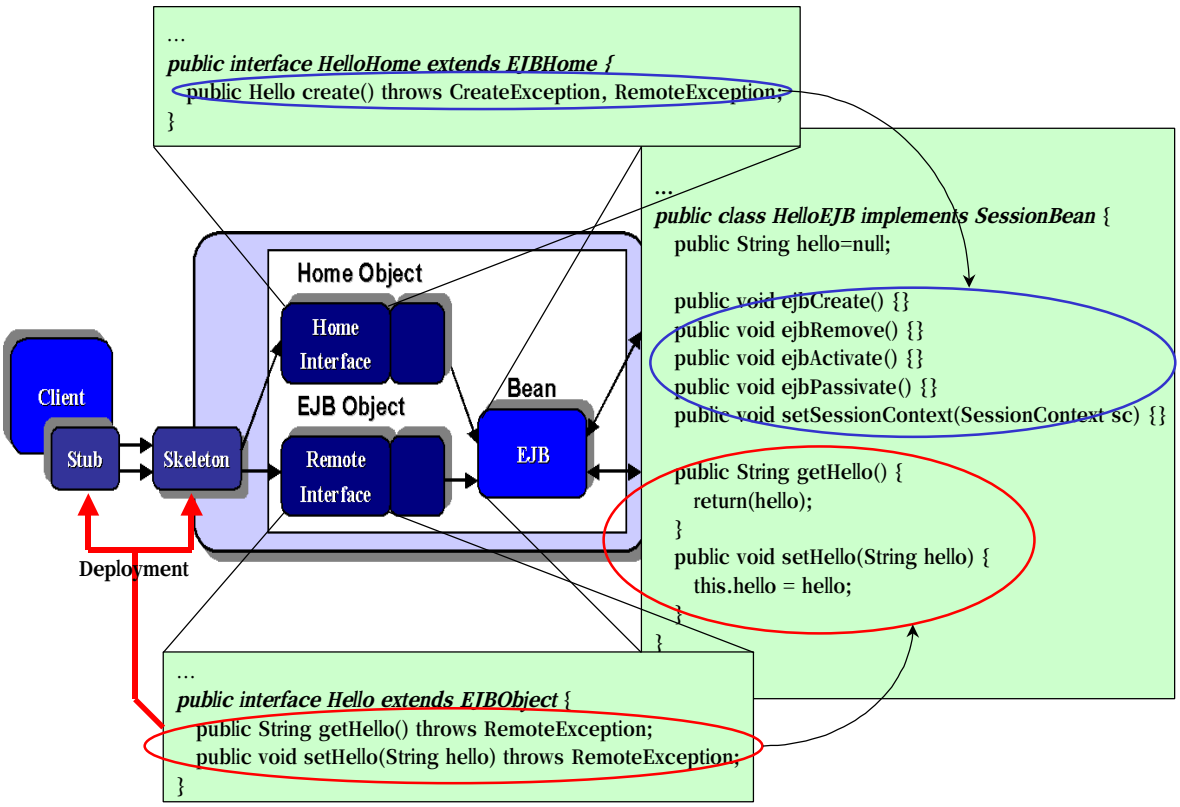
```
import java.util.*;
import javax.ejb.*;

public class HelloEJB implements SessionBean {
    public String hello=null;

    public String getHello() {
        return(hello);
    }

    public void setHello(String hello) {
        this.hello = hello;
    }

    public HelloEJB() {
    }

    public void ejbCreate() {}
    public void ejbRemove() {}
    public void ejbActivate() {}
    public void ejbPassivate() {}
    public void setSessionContext(SessionContext sc) {}
}
```

이렇게 엔터프라이즈 빈을 위한 홈 인터페이스, 리모트 인터페이스, 빈 클래스 등을 모두 작성하였다면, 컴파일 한

후 EJB 컨테이너에서 제공해 주는 디플로이툴을 이용하여 디플로이먼트시켜 주어야 한다. 다음으로, 엔터프라이즈 빈에서 제공해 주는 서비스를 사용하기 위한 클라이언트를 작성할 수 있는데, 클라이언트는 필요에 따라 AWT/SWING을 사용하는 애플릿/애플리케이션 RMI 클라이언트, 서블릿, JSP, 엔터프라이즈 빈 등이 될 수 있다. 여기서는 간단한 RMI 클라이언트 애플리케이션에 대한 소스 코드를 살펴보면, 다음과 같다.

```
import javax.naming.Context;
import javax.naming.InitialContext;
import javax.rmi.PortableRemoteObject;

public class HelloClient {
    public static void main(String[] args) {
        try {
            Context initial = new InitialContext();
            Object objRef = initial.lookup("Hello");

            HelloHome home = (HelloHome)PortableRemoteObject.narrow(objRef, HelloHome.class);

            Hello hello1 = home.create();
            Hello hello2 = home.create();

            hello1.setHello("Welcome to Park's Park");
System.out.println(" 1 : " + hello1.getHello());
            hello2.setHello("Hello, World!");
System.out.println(" 2 : " + hello2.getHello());
System.out.println(" 3 : " + hello1.getHello());
        } catch (Exception ex) {
            System.err.println("Caught an unexpected exception!");
            ex.printStackTrace();
        }
    }
}
```

지금까지 작성한 EJB 컴포넌트를 위한 홈 인터페이스, 리모트 인터페이스, 엔터프라이즈 빈 클래스 등과 EJB 아키텍처와의 관계에 대해 그림으로 살펴보면, 다음과 같다.

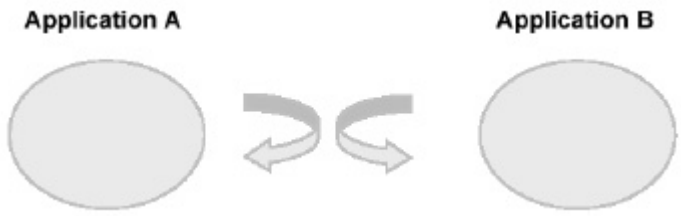


본 고에서 EJB 프로그램 예제로서 보여준 홈 인터페이스, 리모트 인터페이스, 빈 클래스, 클라이언트 애플리케이션에 대한 소스 코드와 컴파일, 디플로이먼트, 실행 등 자세한 과정에 대해서는 이달의 디스켓으로 제공되는 EJB.PPT 파일을 참고하기 바란다. 참고로, EJB.PPT 파일의 내용은 J2EE SDK 1.3을 기준으로 작성한 것이다.

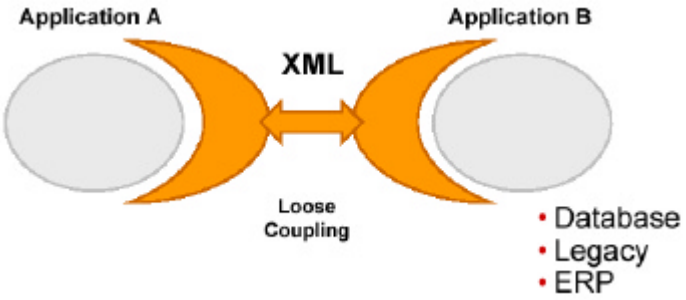
신인류의 신바벨탑, 웹 서비스 (Web Service)

XML을 기반으로 기존의 자바 기술을 재배치

웹 서비스는 차세대 분산 컴퓨팅 기술이라 할 수 있다. XML을 이용하여 이기종 컴퓨팅 환경으로 구성되어 있는 웹 상에서 정보를 공유할 수 있었고, 이제 XML은 정보가 아닌 프로세스를 공유할 수 있는 아주 단순한 방법을 제공해 주고 있다. 다시 말해서, 정보에 대한 구조적 표현으로부터 어플리케이션 간의 메시징에 대한 구조적 표현으로 XML 어플리케이션이 자연스럽게 발전한 것이다. 웹 서비스가 나타나기 전까지는 다음에 나오는 그림에서 보여주고 있는 것과 같이 조직 내에서 사용되는 프로그래밍 언어와 미들웨어가 서로 다르기 때문에 엔터프라이즈 어플리케이션을 통합하는 것이 매우 어려웠다. 실제로, 기업 내에서 사용하고 있는 애플리케이션은 서로 상이한 환경에서 독립적으로 실행되도록 구축되어 왔다.

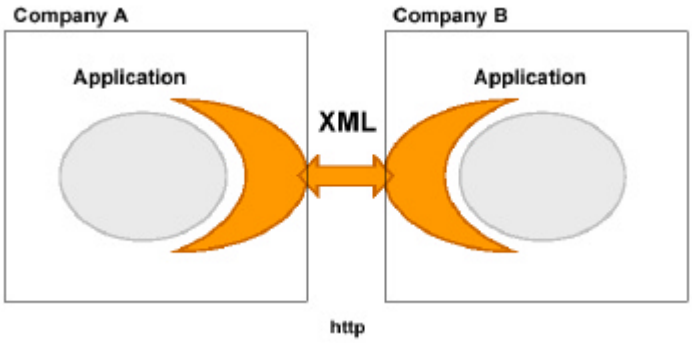


그러나, 웹 서비스를 이용하면 인터넷이 가능한한 어떤 어플리케이션도 웹 상에서 통합될 수 있다. 웹 서비스를 제공할 수 있는 기본 기술은 HTTP와 같은 표준 웹 프로토콜 상에서 동작하는 XML 메시징을 이용하여 상이한 환경에서 동작하고 있는 애플리케이션 간에 Loose Coupling을 제공하는 것이다. 다시 말해서, 기존에 이미 구축되어 있는 애플리케이션의 앞단에 XML 메시징을 해석하여 두 개의 애플리케이션을 연동할 수 있도록 SOAP 기반의 XML 모듈을 추가하는 것이다.



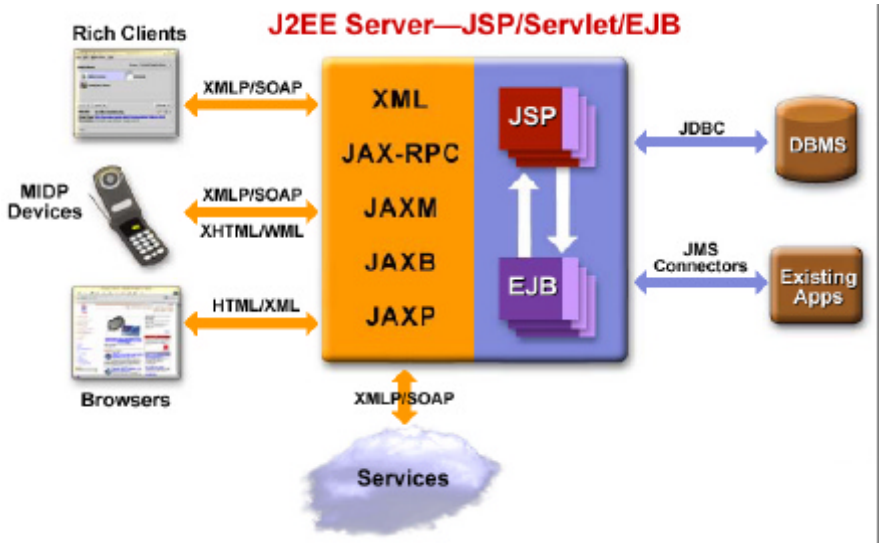
이렇게 SOAP과 같은 XML 기반의 프로토콜을 사용함으로써 다음에 나오는 그림에서와 같이 두 회사 간에 가지고 있는 애플리케이션에서 제공해 주고 있는 서비스를 연도할 수 있게 할 수 있는 것이다. 이러한 XML 메시징은 어떤 프로그래밍 언어, 어떤 미들웨어, 또는 어떤 플랫폼도 모두 참여할 수 있는 초경량 통신 방식으로서 상호운용성을 쉽게 극대화 시킬 수 있기 때문이다.

<그림 11> 이기종에서의 애플리케이션과 XML 기반의 웹 서비스



<그림 11>을 살펴보면, 먼저 JSP/서블릿/EJB등으로 구성되어 있는 기존의 시스템의 앞단에 XML 기술을 전진배치 시킨다는 것을 알 수 있다. 다시 말해서, 웹 서비스를 위한 전혀 새로운 기술이 새롭게 나타난 것이 아니고, 기존의 엔터프라이즈 시스템에서 XML 기반의 SOAP 메시지를 처리할 수 있도록 한꺼풀 덧씌운다고 할 수 있다. 이렇게 XML 기반의 웹 서비스를 제공해 줌으로써, 클라이언트에 맞는 결과 페이지를 동적으로 랜더링 할 수 있다는 것이다.

<그림 12> 자바 XML 기반의 웹 서비스



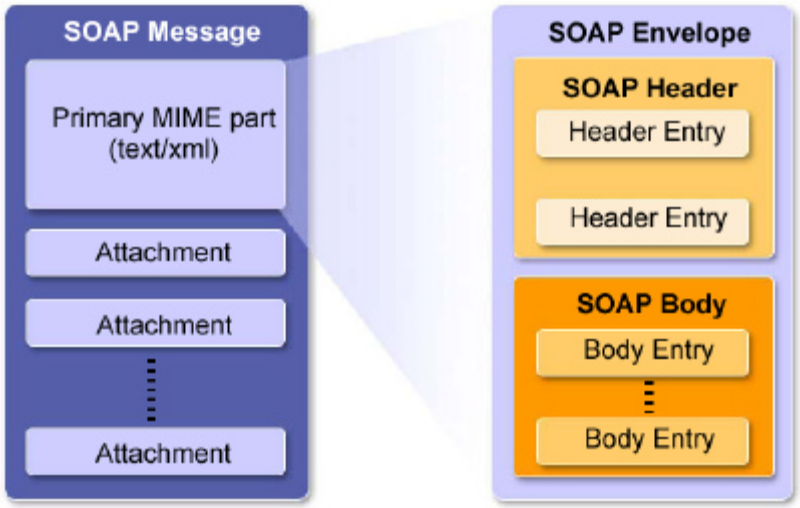
다시 말해서, <그림 12>에서 보여주고 있는 것과 같이 스윙 클라이언트와 같은 고급 사양의 클라이언트에 맞는 결과 페이지를 생성해 주고, MIDP 장치가 연결되었을 때는 MIDP 장치 내의 브라우저가 볼 수 있는 형태의 마크업 언어로 랜더링 해 준다는 것이다. 물론, 웹 브라우저가 접속했을 때는 웹 서비스에 대한 결과 페이지로서 XML/HTML 페이지를 생성하여 전달해 줄 수 있다는 것이다. 웹 서비스는 비즈니스, 애플리케이션, 시스템 서비스에 대한 XML 기반의 인터페이스를 이용하여 간단하게 구현될 수 있다.

SOAP(Simple Object Access Protocol)

SOAP는 비즈니스 파트너가 UDDI를 이용하여 WSDL 기술을 찾을 때, 웹 서비스 상에 있는 하나 이상의 연산을 호출하기 위해 사용할 수 있다. SOAP은 비지니스 메소드 요청을 XML 도큐먼트로서 수행하기 위한 명세이고, HTTP(S) 또는 SMTP 등과 같은 다양한 저수준 프로토콜을 지원할 수 있다. 이를 위해, 프로그래밍 언어적 중립성, 확장성, 강력한 산업체 지원 등의 이유 때문에 XML이 사용된다. 인터넷 상의 어떤 시스템도 소켓을 통해서 통신을 할 수 있고, 어떤 시스템과도 상호운용 할 수 있는 단순한 프로토콜이며, 디폴트 액세스 포트인 80 포트를 사용하

여 방화벽(firewall)을 항해할 수 있으므로 HTTP가 사용된다.

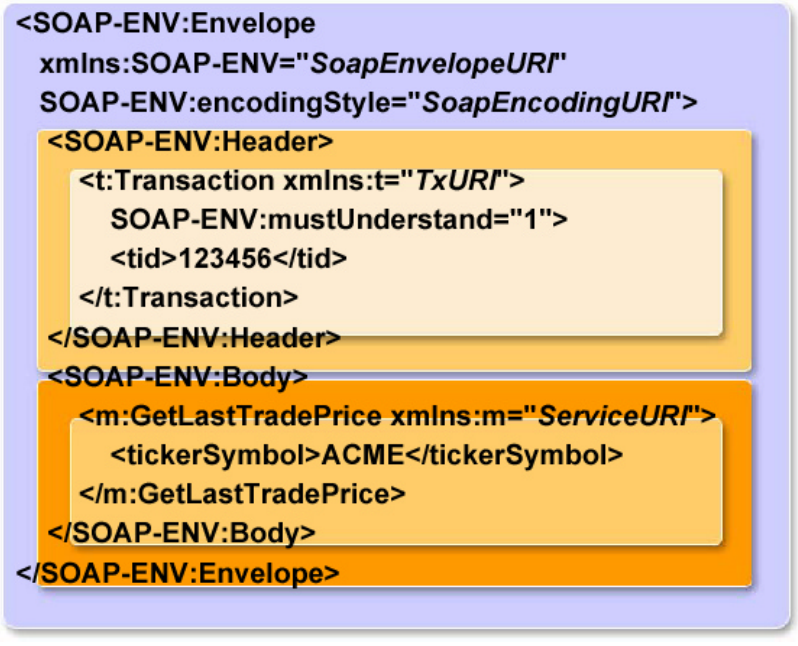
<그림 13> SOAP 메시지의 구조



각 비지니스에서 제공해 주는 웹 서비스를 이용하기 위해 SOAP 메시지를 이용한다. 이러한 SOAP 메시지는 기본적으로 기존의 HTTP 프로토콜을 이용하여 전송된다. 다시 말해서, HTTP 메시지의 바디에 SOAP 메시지가 첨부되어 웹 서비스에 대한 요청과 응답이 이루어진다는 것이다. <그림 13>에서는 이러한 SOAP 메시지의 구조를 잘 보여주고 있다. HTTP 프로토콜을 이용할 때도 마찬가지지만 GET 방식 또는 POST 방식으로 HTTP 헤더에 관련 정보를 보내고 HTTP 바디에는 간단한 name=value 형식으로 데이터를 전달한다. 이 때, 파일을 업로드 할 경우에는 해당 파일을 전달해 주어야 한다. SOAP 메시지 역시 단순한 요청과 함께 데이터를 전달하기 위해 XML 형식의 SOAP 메시지를 보낼 수 있고, 필요에 따라 Attachment를 이용하여 여러 가지 종류의 여러 파일을 첨부하여 SOAP 메시지를 전달할 수 있도록 하는 구조를 갖고 있다는 것을 알 수 있다.

<그림 13>의 왼쪽에서 보여주고 있는 SOAP 메시지 중에서 XML 형식의 기본적인 SOAP 메시지의 구조에 대해 <그림 13>의 오른쪽에서 자세히 보여주고 있다. HTTP 요청 또는 HTTP 응답 메시지의 구조와 마찬가지로 SOAP 메시지 역시 SOAP 헤더와 SOAP 바디로 구성되어 있다는 것을 볼 수 있다. 한 가지 재미있는 것은, 웹 서비스를 요청하기 위한 SOAP 요청 메시지는 물론 웹 서비스의 수행 결과를 요청한 측에 다시 전달하기 위한 SOAP 응답 메시지 모두 <그림 13>에서 보여주고 있는 SOAP 메시지 구조를 갖는다는 것이다.

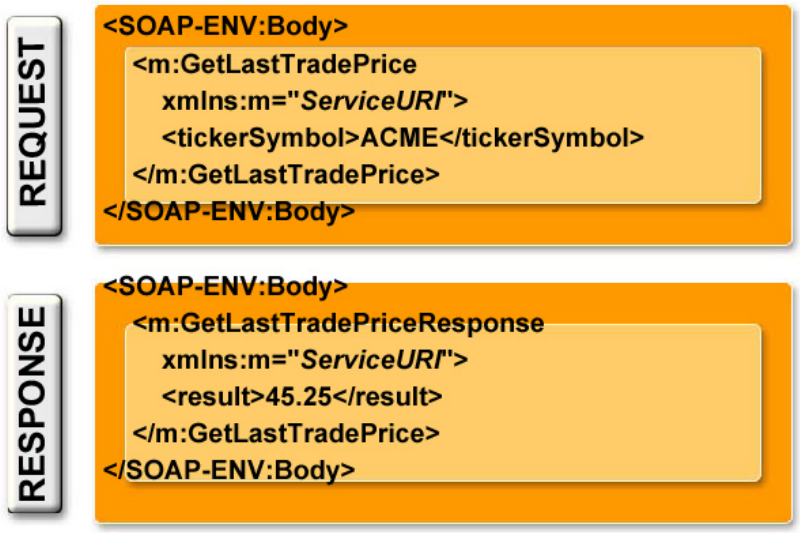
<그림 14> SOAP Envelope 예



이 때, SOAP 헤더와 SOAP 바디를 하나로 묶어 SOAP 메시지를 전달하게 되는데, 이렇게 SOAP 헤더와 SOAP 바디를 하나의 SOAP 메시지로 봉합하기 위해 SOAP 봉투(SOAP Envelope)를 사용하도록 하고 있다. 다시 말해서, SOAP 메시지를 주고 받는 것을 편지를 부치는 것에 비유하고 있는 것이다. <그림 14>에서는 실제로 SOAP 메시지를 보내기 위해 SOAP 헤더와 SOAP 바디를 하나로 묶는 XML 기반의 SOAP Envelop의 예를 보여주고 있다. 이렇게 SOAP 메시지를 보내기 위해서는 항상 XML 기반의 SOAP Envelope 메시지를 생성해 주어야 하고, 이러한 SOAP Envelope 메시지를 HTTP 메시지의 바디에 붙혀 HTTP 프로토콜을 이용하여 주고 받게 된다.

<그림 15>에서는 SOAP 요청 메시지와 SOAP 응답 메시지를 위한 SOAP Envelope 중에서 SOAP 바디에 해당하는 XML 코드를 보여주고 있다. <그림 15>에서 보여주고 있는 SOAP 요청 메시지와 SOAP 응답 메시지를 살펴보면, 먼저 SOAP 요청 메시지는 GetLastTradePrice 서비스(실질적으로는 메소드)를 요청하면서 tickerSymbol 값으로 "ACME"를 전달하고 있다는 것을 알 수 있다. 이 때, "ServiceURI" 부분에는 실제 웹 서비스를 제공할 때의 URI가 나타날 것이다. 다음으로, SOAP 응답 메시지를 살펴보면, GetLastTradePriceResponse라고 나타냄으로써 GetLastTradePrice 서비스(실질적으로는 메소드)에 대한 응답 메시지임을 나타내고 있으며, 이 서비스의 결과값으로서 42.25 값을 되돌려 주고 있음을 알 수 있다. SOAP 요청 메시지에서와 마찬가지로 "ServiceURI" 부분에는 실제 웹 서비스를 제공할 때의 URI가 나타날 것이다.

<그림 15> SOAP 요청 메시지와 SOAP 응답 메시지 예



<그림 15>에서 보여주고 있는 SOAP 요청 메시지와 SOAP 응답 메시지는 <그림 16>에서 보여주고 있는 것과 같이 HTTP 요청 메시지와 HTTP 응답 메시지의 바디 부분에 포함되어 메시지가 전달된다. 실제로, <그림 16>에서는 웹 서비스를 요청하기 위해 HTTP 요청 메시지의 바디 부분에 SOAP 요청 메시지를 포함하고 있는 것을 보여주고 있다. 이 때, <그림 16>에서는 "BEAS" 라는 심볼에 대한 GetLastStockQuote 웹 서비스를 요청하고 있는 예를 보여주고 있다.

<그림 16> SOAP 요청 메시지 예

```
POST /StockQuote HTTP/1.1
Host: www.sample.com
Content-Type: text/xml; charset="utf-8"
Content-Length: nnnn
SOAPAction: "Some-URI"

<SOAP-ENV:Envelope
   xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
         SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
     <SOAP-ENV:Body>
          <m:GetLastStockQuote xmlns:m="Some-URI">
               <symbol>BEAS</symbol>
          </m:GetLastStockQuote>
     </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

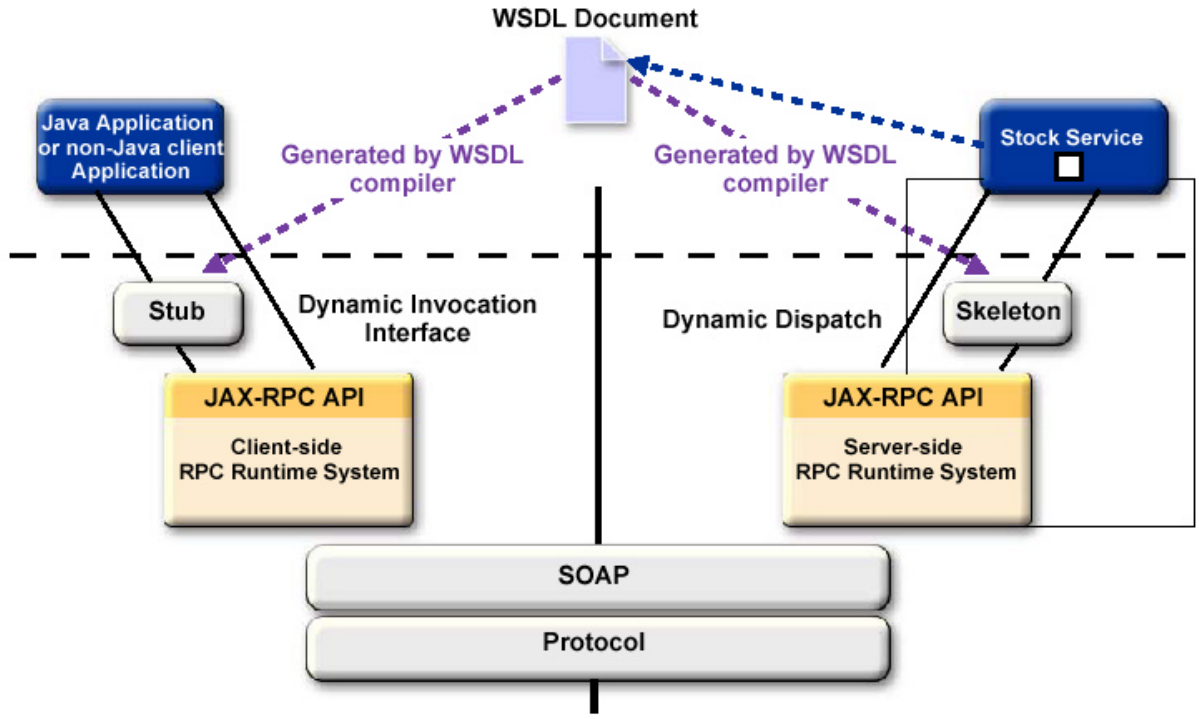
따라서, 실제로, 웹 서비스를 제공하는 측에서는 HTTP 요청 메시지의 바디 부분에 있는 SOAP Envlope 메시지를 해석하여 웹 서비스를 제공해 주어야 하는데, 이러한 작업은 대부분의 웹 서비스 툴에 의해 제공되고 있으므로, 실제로 신경 쓸 부분은 그리 많지 않다. 마찬가지로, 웹 서비스를 요청한 측에서도 웹 서비스 제공자로부터 온 HTTP 응답 메시지의 바디에 포함되어 있는 SOAP 응답 메시지를 해석하여 웹 서비스의 결과를 참조해야 한다.

WSDL(Web Services Description Language)

WSDL 명세는 웹 서비스 호출에 대한 인터페이스, 의미론(semantics) 등을 기술하고 있는 XML 도큐먼트이다. 비즈니스가 사용하고자 하는 서비스를 찾기 위해서는, 실질적으로 호출하기 전에 서비스를 호출하기 위한 문법(syntax)과 의미론(semantics) 등에 대해 미리 이해해야 할 필요가 있다. 이 때, WSDL은 간단한 서비스를 빠르고 쉽게 기술하고 문서화 할 수 있도록 해 준다.

실제로, 웹 서비스를 제공하는 웹 서비슨 제공자는 자신이 제공하는 웹 서비스에 대한 명세를 제공해 주어야 한다. 우리가 오프라인에서 사용하는 서비스 등록증(실제로 있는지는 모르겠지만.. 말을 하자면...) 정도라고 보면 될 것이다. 이러한 서비스 등록증인 WSDL은 서비스 명과 함께 서비스를 사용하기 위해 필요한 매개변수 등, 등록된 웹 서비스에 대한 여러 가지 정보를 포함하고 있는 XML 문서라고 할 수 있다.

<그림 17> 웹 서비스와 WSDL 개념도



먼저, <그림 17>에서는 XML-RPC를 이용하여 웹 서비스를 구현하고 있는 구성을 보여주고 있다. <그림 17>을 살펴보면, 왼쪽이 웹 서비스를 이용하는 클라이언트 측이고 오른쪽이 웹 서비스를 제공하는 서버측이라 할 수 있다. <그림 17>에 보여주고 있는 웹 서비스 제공자는 “Stock Service”를 만든 후, 이 서비스를 웹 서비스로서 제공하기 위해 “Stock Service”에 대한 웹 서비스 명세인 XML 기반의 WSDL 문서를 생성한다. 이러한 WSDL 문서는 일반적으로 자동화 툴을 이용하여 생성하도록 제공되고 있다. 이렇게 생성된 WSDL 문서를 컴파일하여 XML-RPC 기

반의 웹 서비스를 호출할 수 있도록 하기 위해, 웹 서비스 호출을 위한 스텁과 스켈레톤을 자동으로 생성한다. 왜냐하면, 네트워크 상에 분산되어 있는 객체에서 제공해 주는 메소드를 호출함으로써 웹 서비스를 이용할 수 있도록 구현할 수 있기 때문이다.

이러한 웹 서비스 구현 방식은 자바와 XML을 기반으로 하고 있는 XML-RPC를 이용하여 제공할 수 있도록 하고 있다. 따라서, 웹 서비스 클라이언트는 SOAP Envelope 메시지를 생성한 후, 웹 서비스 호출을 위한 스텀을 이용하여 HTTP 프로토콜을 이용하여 전송하게 된다. 이렇게 전송된 메시지를 웹 서비스 제공 서버 측에서는 스켈레톤을 이용하여 받게 되고, SOAP Envelope 메시지를 해석하여 해당 서비스를 수행한 후, 그 결과를 다시 클라이언트 측에 전달해 주게 되는 것이다. 실제로, 웹 서비스를 제공하기 위해 생성해야 하는 WSDL 문서와 이 WSDL 문서를 기반으로 실제로 웹 서비스를 리모트로 호출할 수 있도록 하기 위한 스텁과 스켈레턴은 자동화 도구에 의해 생성하도록 하고 있다. <그림 18>에서는 WSDL 문서에 대한 예를 잘 보여 주고 있다.

<그림 18> WSDL 문서 예

```
<?xml version="1.0">
<definitions name="StockService"
    targetNamespace=http://example.com/stockservice.wsdl
    xmlns:tns=http://example.com/stockservice.wsdl
    xmlns:xsdl=http://example.com/stockservice.xsd
    xmlns:soap=http://schemas.xmlsoap.org/wsdl/soap/
    xmlsns=http://schemas.xmlsoap.org/wsdl/>
<message name="GetLastTradePriceInput"><part name="tickerSymbol" element="xsd:string"/> </message>
<message name="GetLastTradePriceOutput"><part name="result" element="xsd:float"></message>
<portType name="StockServicePortType">
    <operation name="GetLastTradePrice">
        <input message="tns:GetLastTradePriceInput"/>
        <output message="tns:GetLastTradePriceOutput"/>
    </operation>
</portType>
<binding name="StockServiceSoapBinding" type="tns:StockServicePortType">
<soap:binding style="rpc" transport="http:/schemas.xmlsoap.org/soap/http">
<operation name="GetLastTradePrice">
    <soap:operation soapAction="http://example.com/GetLastTradePrice"/>
      <input><soap:body use="encoded" namespace=http://example.com/stockquote"
            encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"></input>
      <output><soap:body use="encoded" namespace="http://example.com/stockquote"
            encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"></output>
</operation>  </binding>
<service name="StockService">
<port name="StockServicePort" binding="tns:StockServiceSoapBinding">
<soap:address location= http://example.com/StockService/> </port>
</service>
</definitions>
```

먼저, WSDL 문서에서는 XML 스키마 데이터 타입 명세를 이용하여 데이터 타입을 정의할 수 있는데, 예를 들면 다음과 같다.

```
<types>
   <simpleType name=”Color” base=”xsd:string”>
      <enumeration value=”Green” />
      <enumeration value=”Blue” />
   </simpleType>
</types>
```

다음으로, <그림 18>에서 보여주고 있는 WSDL 문서에서는 포트 타입, 연산(operation) 및 메시지(프로토콜과 메시지 포맷) 등을 나타내기 위해 다음과 같은 코드를 사용하고 있음을 알 수 있다. 이 때, 정확히 하나의 프로토콜을 명세해야 하고, 어드레스 정보를 명세해서는 안된다. 실제로, 다음에 나오는 코드는 SOAP 1.1 바인딩 예를 보여주고 있다.

```
<soap:binding style="rpc" transport="http:/schemas.xmlsoap.org/soap/http">
<operation name="GetLastTradePrice">
    <soap:operation soapAction="http://example.com/GetLastTradePrice"/>
      <input><soap:body use="encoded" namespace=http://example.com/stockquote"
             encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"></input>
      <output><soap:body use="encoded" namespace="http://example.com/stockquote"
             encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"></output>
</operation>  </binding>
```

SOAP 메시지를 위한 프로토콜 바인딩은 다음과 같으며, SOAP의 HTTP 바인딩을 위한 SOAPAction 헤더는 URI로 명세해야 한다. 그리고, soap:operation 태그에서는 style 속성은”rpc” 또는 “documnet” 값을 가질 수 있다. 다시 말해서, style=”rpc|document” 이어야 한다. 또한, soap:body 태그를 이용하여 메시지 부분 부분이 SOAP 바디 엘리먼트 내에서 어떻게 나타날 지를 명세할 수 있다.

- HTTP
- FTP
- SMTP

Port는 바인딩을 위한 어드레스를 명세하여 엔드포인트를 명세한다. Service는 관련된 포트 셋을 그룹화하기 위해 사용된다. 서비스에서 나타내는 그룹은 다음과 같은 특성을 갖는다.

- 구룹으로 묶인 어떤 포트도 각각 통신하지 않는다.
- 하나의 서비스 내에 있는 포트는 portType을 공유하지만, 서로 다른 바인딩을 사용한다.

<그림 18>에서 보여주고 있는 WSDL 문서에서 포트와 서비스를 명세하는 부분을 살펴보면 다음과 같은데, 하나의 포트를 그룹으로 묶어 제공하는 서비스를 명세하고 있는 것을 알 수 있다.

```
<service name="StockService">
<port name="StockServicePort" binding="tns:StockServiceSoapBinding">
<soap:address location= http://example.com/StockService/> </port>
</service>
```

이 때, "StockService" 서비스에는 "StockServicePort" 포트 하나를 포함하고 있다. 이 때, "StockServicePort" 포트는 "StockServiceSoapBinding"에 바인딩되어 있음을 알 수 있다. 실제로, "StockServiceSoapBinding" 바인딩을 살펴보면, 다음과 같다.

```
<binding name="StockServiceSoapBinding" type="tns:StockServicePortType">
```

이 때, "StockServiceSoapBinding" 바인딩은 다시 "StockServicePortType" 타입을 갖는데, 실제로 "StockServicePortType" 타입은 다음과 같이 명세되어 있다.

```
<portType name="StockServicePortType">
    <operation name="GetLastTradePrice">
        <input message="tns:GetLastTradePriceInput"/>
        <output message="tns:GetLastTradePriceOutput"/>
    </operation>
</portType>
```

현재, JCP에서는 WSDL을 지원하기 위한 JWSDL(Java API for WSDL) 명세에 대한 작업을 진행 하고 있다.

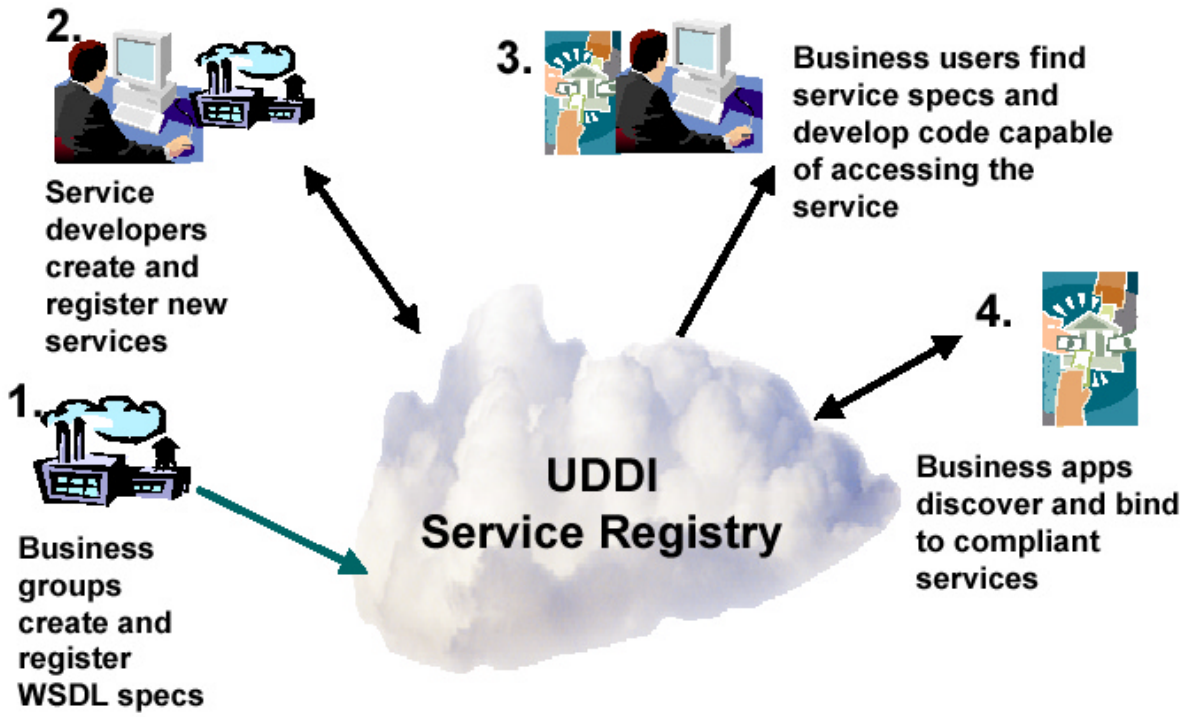
UDDI(Universal Description, Discovery, and Integration)

UDDI는 웹 서비스 제공자와 탐색자 모두를 위해 중요한 새로운 프로젝트이다. UDDI 프로젝트의 멤버는 UBR(UDDI Business Registry)이라 불리는 웹 서비스를 운영할 것이다. 이 때, UBR은 비즈니스 및 서비스에 대한 전역적이고 공개된 디렉토리라 할 수 있다. 웹 서비스 제공자는 UBR 내에 자신의 서비스를 등록하고 기술할 수 있다. 사용자는 웹 서비스를 찾고 해당 서비스와 상호작용 할 필요가 있는 정보를 위치시키기 위해 UBR에 대한 쿼리를 수행할 수 있다. UDDI는 어떤 서비스를 찾고 있는 시스템을 해당 서비스를 기술하고 있는 문서와 직접 연결시키기 위한 방법이라 할 수 있다. UDDI는 "white pages"-type 비즈니스 서치와 "yellow pages"-type 토픽 서치 뿐만 아니라 "green pages"-type 서비스 타입 서치 등을 표준으로 포함하고 있다. 이 때, 개발자는 "green pages"-type 서비스 타입 서치를 이용하여 특정 서비스 타입에 해당하는 모든 서비스를 찾을 수 있다.

UDDI는 레지스트리 내에 있는 정보에 대한 출판(publishing), 편집(editing), 보기(browsing), 검색(searching) 등

을 위해 SOAP 메시징(일반적으로 XML/HTTP)을 사용한다. 또한, 레지스트리 서비스로 리턴되거나 또는 보내질 다양한 종류의 데이터를 캡슐화 하기 위한 XML 스키마를 포함하고 있다. 향후 자바 플랫폼에서 UDDI 기능을 제공해 주기 위해, JAXR(Java APIs for XML Registries) 에서는 레지스트리를 액세스하는데 사용할 수 있는 API 명세를 제공할 것이다. 한 가지 주의할 것은 아직까지는 웹 서비스를 개발하기 위해 필요하지 않다는 것이다. 대신, 프로토콜과 직접 상호작용 할 수 있는 보다 일반적인 XML API를 사용할 수 있다는 것이다.

<그림 19> UDDI 를 이용한 웹 서비스의 등록 및 조회



<그림 19>에서는 UDDI를 이용하여 서비스를 등록하고 등록된 서비스를 찾아 사용하는 과정에 대해 자세히 보여주고 있다. <그림 19>에서 보여주고 있는 과정에 대해 살펴보면, 다음과 같다.

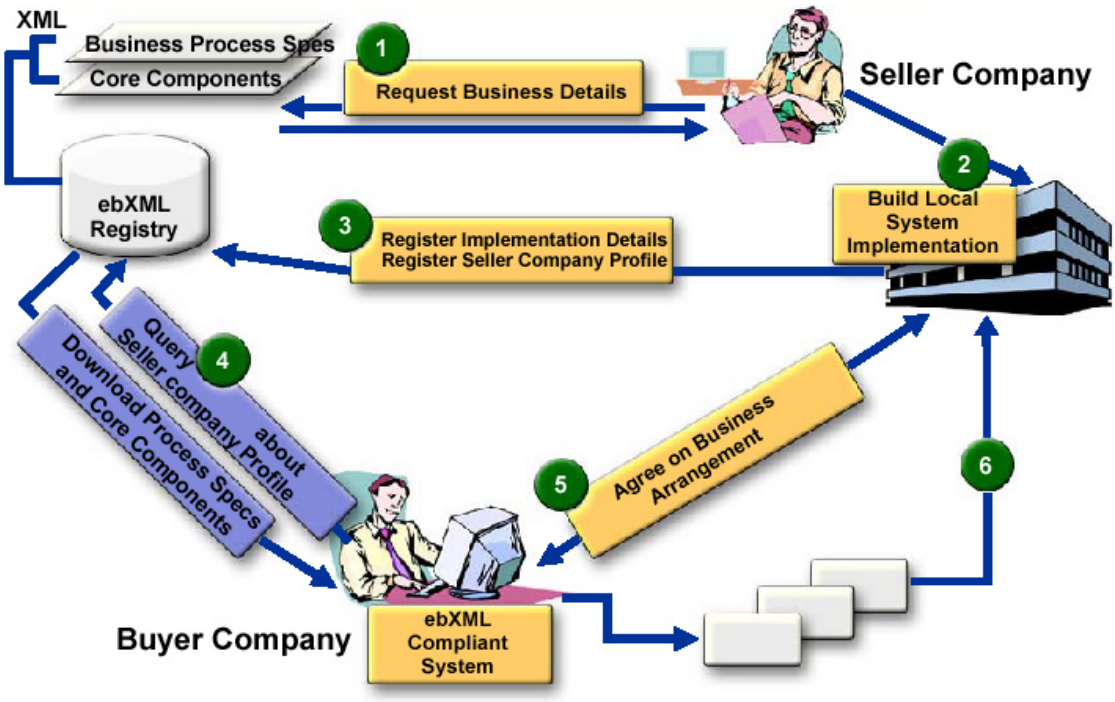
1. 비지니스 그룹에서 WSDL 스펙을 생성하고 등록한다.
2. 서비스 개발자는 서비스를 생성하고 등록한다.
3. 비지니스 사용자는 서비스 스펙을 찾고 서비스를 액세스 하기 위한 코드를 작성한다.
4. 비지니스 애플리케이션에서는 서비스를 찾아 바인딩 하여 사용한다.

ebXML

비즈니스 트랜잭션, 다중-요청 트랜잭션, 스키마, 도큐먼트 플로우 등을 가능하도록 하는 구조를 필요로 하는 확장

된 비즈니스 익스체인지에 대해, 어플리케이션 요구사항이 순수하게 SOAP 기반의 구현의 한계를 늘렸다. SOAP는 이러한 확장된 비즈니스 인스체인지를 최상위층에서 개발할 수 있도록 저수준 구조를 제공하는 반면, 이러한 이슈를 이미 해결한 보다 향상된 프레임워크가 필요하게 되었다. 이러한 이유로 ebXML이 나오게 되었고, ebXML은 XML 명세, B2B 통합을 위한 e-인프라스트럭처를 제공하기 위해 설계된 관련 프로세스와 행동 등의 세트이다.

<그림 20> ebXML



<그림 20>에서 보여주고 있는 과정에 대해 살펴보면, 다음과 같다.

1. 판매 회사에서는 비지니스 관련 상세 정보(비지니스 프로세스 스펙)를 요청한다.
2. 판매 회사에서는 ebXML에 기반하여 자사 시스템을 구현한다.
3. 판매 회사에서는 구현 명세 정보 및 자사 프로파일을 ebXML 레지스트리에 등록한다,
4. 구매 회사에서는 판매 회사의 프로파일을 검색하고, 검색한 프로세스 스펙과 핵심 컴포넌트를 다운받는다.
5. 거래 약정에 대한 제안 및 승인한다.

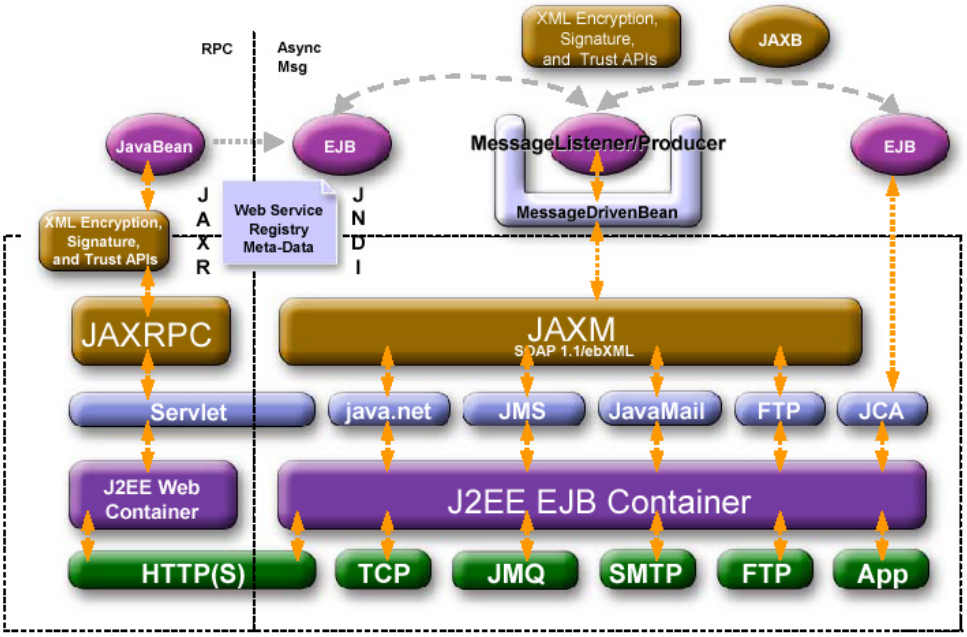
웹 서비스 제공을 위한 서비스 팩

썬에서는 웹 서비스 팩을 통해서 웹 서비스 기능을 제공하기 위한 API와 아키텍처 셋을 제공해 주고 있다. 현재, 웹 서비스 팩은 다음과 같은 자바 기술 관련 기능들을 포함할 계획이다.

- The JavaServerTM Faces API : 현재 JCP 프로그램에서 개발중인 자바 기술로서, 자바 웹 어플리케이션 GUI를 생성하기 위한 표준 API이고, GUI 하부구조를 설계하고 유지보수 하는데 드는 개발자의 수고를 제거하기 위한 자바 기술이다.
- Tomcat : JSP와 자바 서블릿에 대한 오픈-소스 구현으로서, 아파치 소프트웨어 파운데이션 내의 자카르타 프로젝트에 의해 개발되었다. JSP 기술은 사용자 인터페이스와 컨텐츠 생성을 분리하고 있으며, 디자이너가 컨텐츠의 변경없이 전체 페이지 레이아웃을 변경할 수 있도록 해 준다. 자바 서블릿 기술은 웹 개발자가 기존의 웹 서버의 기능을 확장하거나 기존의 비즈니스 시스템을 액세스 할 수 있도록 간단하고 일관성있는 방법을 제공해 준다.
- JAX Pack : JAXP(Java API for XML Processing), JAXR(Java API for XML Registries), JAXM(Java API for XML Messaging), JAXB(Java Architecture for XML Binding), JAX-RPC(Java API for XML-based Remote Procedure Call) 등과 같은 XML 기반의 자바 API와 아키텍처 기술의 모음이라 할 수 있다.

J2EE 기반의 웹 서비스 시스템

다음에 나오는 그림은 J2EE에 기반하고 있는 웹 서비스 시스템의 주요 구조를 개괄적으로 보여주고 있다. 다음에 나오는 그림에서는 파싱하고 메시징하기 위해 사용될 수 있는 많은 API를 생략하였고, 대신 J2EE에 기반하고 있는 웹 서비스 배치시에 나타나는 표준, 프로토콜, 그리고 주요 서버시스템 등에 대해서는 자세히 보여주고 있다.



흩어져 있는 컴퓨팅 파워를 하나로, JXTA P2P

JXTA는 "juxta"와 같이 발음할 수 있으며, 실제 P2P가 존재하는 것과 같이 서로 바로 다음에 병렬 배치시키는 것을 뜻하는 "juxtapose"라는 단어의 약자라고 할 수 있다. P2P는 다른 피어(peer)들과 순간적으로 연계할 수 있는 시스템 기능을 가져야 하고, 하나의 그룹으로 합쳐지고, 잠시동안 병렬 배치되고, 그리고 나서 흩어질 수 있도록 해 주어야 한다. JXTA는 자바 기술 및 지니 기술과 함께 할 것이며, JXTA 프로젝트의 네 가지 주요 관심사에 대해 살펴보면, 다음과 같다.

- 피어와 피어 사이의 “pipe” 기능 : 분산 환경에서 네트워크를 통해서 한 피어의 표준 입력과 다른 피어의 표준 출력을 연결하는 파이프라인 기능을 제공해 주어야 한다. 파이프라인 기능은 표준 유닉스 통신 프로토콜이다.
- 그룹 개념(grouping notion) : 동적이고, 유동적이며, 유연한 환경에서 그룹을 생성할 수 있는 기능을 제공해 주어야 하며, 피어들이 서로를 인식하거나 다른 피어를 믿게 하기 위한 서로 다른 방법을 제공해 주어야 한다. 이 때, 한 가지 이슈는 피어들을 어떻게 하나의 그룹으로 묶고, 논리적이고 응집력이 있도록 어떻게 합칠 것인가 하는 것이다.
- 모니터 및 측정 기능 (ability to monitor and meter) : 어떤 일이 진행 중인지 알거나, 피어들에 대한 제어 정책을 생성할 수 있는 기능을 제공해 주어야 한다.
- 보안 계층 : 피어와 피어가 서로 믿고 병렬배치 될 수 있도록 신뢰성을 제공해 주어야 하며, 이를 위한 보안 계층을 제공해 주어야 한다.

2001년 2월 22일 샌프란시스코에서 있었던 오렐리(O'Reilly)의 첫번째 P2P(peer-to-peer) 컨퍼런스에서 썬마이크로시스템즈의 빌조이(Bill Joy)가 JXTA를 발표하였다. JXTA는 분산 컴퓨팅의 새로운 스타일을 제시하기 위한 자바 기술로서, P2P의 주도권을 잡기 위한 포석이라 할 수 있다. 향후, JXTA는 P2P 어플리케이션을 위한 하부구조 서비스를 제공해 줄 것이다.

특집을 마치며...

본 특집에서는 자바를 이용하여 할 수 있는 대부분의 프로그래밍에 대해 살펴보려고 하였다. 그러나, 그 방대한 프로그래밍을 약간의 소스코드와 함께 소개하려다 보니, 초반에는 거의 200페이지 이상의 분량이 되었다. 기껏해야 25페이지 안팎으로 원고를 써야하는데 말이다. 너무 무모했다 싶기도 하고, 또한 이렇게 방대한 내용을 한 회의 글로서 마물리하려 했었던 우리들의 무식함(?)에 감탄을 금치못했다. 필자가 본 특집에서 얘기하고자 하는 것을 다음과 같이 요약할 수 있을 것이다.

“자바는 기본적으로 안정적이고 견고한 시스템을 구축하고, 생성된 코드를 재사용할 수 있도록 하고 있다. 또

한, 자바를 이용하여 어떤 프로그램을 작성하더라고, 항상 똑같은 자바 문법을 이용하여 일관되고 일률적인 프로그래밍을 할 수 있도록 해 준다. 프로그래램의 확장 역시 단순히 패키지를 설치하여 바로 사용할 수 있도록 함으로써, 순전히 자신이 고민해야 할 비지니스 로직의 구현과 새로운 기술의 습득에 대해서만 신경 쓸 수 있도록 하고 있다."

바야흐로, 지금까지 구축된 모든 인포메이션 시스템들이 웹 서비스라는 새로운 패러다임에 의해 하나의 가상의 인포메이션 시스템으로 묶이게 되고, 방대한 인포메이션을 제공할 수 있게 된다. 신인류의 신바벨탑이 탄생하고 있는 것이다. 이렇게 웹 서비스를 이용하여 하나로 묶인 방대한 인포메이션 시스템에서 클라이언트에 맞게 랜더링하여 제공해 주는 페이지를 클라이언트는 단순히 브라우징만 하면 되도록 하고 있다. 이 얼마나 풍요롭고 편리한 세상인가. 또한, 개발자는 누군가가 제공해 주는 마술봉과 같이 매직을 발휘하는 단 하나의 툴만 써서도 이 세상의 모든 일을 할 수 있는 슈퍼맨이 된 것 같다. 그러나, 그 이전에 시스템에 대한 충분한 분석 설계와 개발 공정의 확립을 통한 각자의 역할 분담, 그 다음에 정말로 멋진 자시만의 툴을 써서 개발 생산성을 높힐 수 있으면 어떨까? 혼자서 모든 것을 할 수 있는 그러한 시대는 가고, 이제 수많은 사람들이 모여 서로의 역할을 충실히 하면서 시스템을 구축하기 위한 전 공정에서 서로간의 협업을 통해 보다 빠르고 효율적으로 작업을 해 나가야 할 것이다.

필자 연락처 : 박용우(ywoopark@sovik.co.kr)

참고자료

Java Web Application Programming Bible, 박용우, 영진닷컴

J2ME MID Profile, Roger Riggs et al., Sun Microsystems

WAP and STK Interactions, Fernando Liobregat et al., Oberthur Card Systems

자바 언어 스펙, http://java.sun.com/docs/books/jls/second_edition/html/j.title.doc.html

자바 언어 스펙 한글화 프로젝트, http://www.javaline.co.kr/javaLangSpec/

자바 가상머신 스펙, http://java.sun.com/docs/books/vmspec/index.html

자바 튜토리얼, http://java.sun.com/docs/books/tutorial/index.html

자바 2 SDK, SE v.1.4 도큐먼트: http://java.sun.com/j2se/1.4/docs/index.html

J2EE 튜토리얼, http://java.sun.com/j2ee/tutorial/1_3-fcs/index.html

JAR 파일, http://java.sun.com/docs/books/tutorial/jar/

http://javastudy.co.kr/docs/b612/swing/mvc.html,

http://javastudy.co.kr/docs/b612/swing/lookandfeel.html

http://www.javastudy.co.kr/docs/webdox/Rmi.htm

http://java.sun.com/docs/books/tutorial/uiswing/index.html

Java Programming Language Basics Part 1, http://www.javastudy.co.kr/docs/jhan/javabasic1/index.html

Java Programming Language Basics Part 2, http://www.javastudy.co.kr/docs/jhan/javabasic2/index.html

Writing Advanced Application for the Java Platform, http://www.javastudy.co.kr/docs/jhan/javaadvance/index.html

Funcamental Of Corba Programming, http://www.javastudy.co.kr/docs/jhan/fun_corba/corba.html

Fundamental Of Java Servlet Programming - Servlet API, http://www.javastudy.co.kr/docs/jhan/fun_servlet/servlet.htm

Java2 Platform Enterprise Edition 개요 및 설명, http://www.javastudy.co.kr/docs/jhan/j2ee/overview.html

Web Services and JavaTM Technology - A Natural Fit, James R. Russell, JavaOne 2001

The J2EETM Platform vs. Microsoft .NET - A Comparison of Web Services Architectures, Ed Roman, JavaOne 2001

SOAP 1.1 in the JavaTM Platform: Introducing the JAX-RPC Techonolgy, Roberto Chinnici, Rahul Sharma, JavaOne 2001

Java TM API for XML Messaging (JAXM), Nicholas Kassem and Anil Vijendran, JavaOne 2001

B2B-SOAP Performance and the J2EETM Platform, Harvey W. Gunther, JavaOne 2001

The Web Services Revolution, James Tauber, JavaOne 2001

Creating Web Services With J2EETM: uPlanetTM and SUN ONETM, Laurence P. G. Cable, JavaOne 2001

Overview of JavaTM API for XML Registries (JAXR), Farrukh Najmi, JavaOne 2001

Developing and Dploying Web Services With Oracle9i and Java, Alok Srivastava, JavaOne 2001